1 2 3 Idee per il tuo futuro

Fiorenzo Formichi
Giorgio Meini

Corso di informatica

per Informatica

Algoritmi e linguaggio C++
Pagine web

TECNOLOGIA ZANICHELLI

Fiorenzo Formichi
Giorgio Meini

Corso di informatica

per Informatica

Algoritmi e linguaggio C++
Pagine web

TECNOLOGIA ZANICHELLI

Realizzazione editoriale:
– Coordinamento redazionale: Matteo Fornesi
– Segreteria di redazione: Deborah Lorenzini
– Progetto grafico: Editta Gelsomini
– Collaborazione redazionale, impaginazione, disegni e indice analitico: Stilgraf, Bologna

Contributi:
– I capitoli della sezione B sono a cura di Federico Meini
– Rilettura dei testi in inglese: Roger Loughney

Copertina:
– Progetto grafico: Miguel Sal & C., Bologna
– Realizzazione: Roberto Marchetti
– Immagine di copertina: valdis torms/Shutterstock; Artwork Miguel Sal & C.

Prima edizione: gennaio 2012

L'impegno a mantenere invariato il contenuto di questo volume per un quinquennio (art. 5 legge n. 169/2008) è comunicato nel catalogo Zanichelli, disponibile anche online sul sito www.zanichelli.it, ai sensi del DM 41 dell'8 aprile 2009, All. 1/B.

File per diversamente abili
L'editore mette a disposizione degli studenti non vedenti, ipovedenti, disabili motori o con disturbi specifici di apprendimento i file pdf in cui sono memorizzate le pagine di questo libro. Il formato del file permette l'ingrandimento dei caratteri del testo e la lettura mediante software screen reader. Le informazioni su come ottenere i file sono sul sito www.zanichelli.it/diversamenteabili

Suggerimenti e segnalazione degli errori
Realizzare un libro è un'operazione complessa, che richiede numerosi controlli: sul testo, sulle immagini e sulle relazioni che si stabiliscono tra essi. L'esperienza suggerisce che è praticamente impossibile pubblicare un libro privo di errori. Saremo quindi grati ai lettori che vorranno segnalarceli.
Per segnalazioni o suggerimenti relativi a questo libro scrivere al seguente indirizzo:

lineazeta@zanichelli.it

Le correzioni di eventuali errori presenti nel testo sono pubblicati nel sito www.online.zanichelli.it/aggiornamenti

Zanichelli editore S.p.A. opera con sistema qualità
certificato CertiCarGraf n. 477
secondo la norma UNI EN ISO 9001:2008

Indice

SEZIONE A

Algoritmi e linguaggio C++

A1 Informatica e informazione

A2 Algoritmi

A3 Linguaggi di programmazione

A4 Il linguaggio di programmazione C++

A5 Le funzioni in C++

A6 Gli *array* in C++

A7 Le strutture in C++

A8 Ordinamento e ricerca

A9 La ricorsione

A10 I file

A11 Introduzione alla programmazione orientata agli oggetti

A12 Strumenti di sviluppo in ambiente Linux e Windows

Il capitolo A12 è disponibile,
con chiave di attivazione, all'indirizzo:
www.online.zanichelli.it/formichimeinicorsoinformatica

SEZIONE
B

Pagine web

B1 Il linguaggio HTML

B2 CSS (*Cascading Style Sheet*) per pagine web

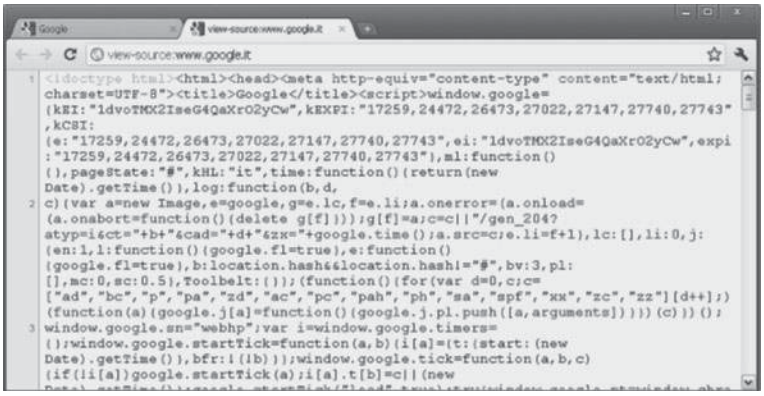
SEZIONE

A Algoritmi e linguaggio C++

A1 Informatica e informazione

Milioni di persone ogni giorno si connettono al *web* per cercare informazioni di vario genere. Per far questo spesso si affidano a sistemi automatici che, analizzando alcune parole chiave fornite dall'utente, interrogano un insieme di dati in precedenza raccolti e organizzati e restituiscono un indice dei contenuti disponibili classificandoli in base a criteri che ne stabiliscono il grado di rilevanza con quanto specificato dall'utente. L'immagine che segue è relativa alla *home-page* di uno dei più noti motori di ricerca e costituisce l'interfaccia tramite la quale i cibernauti possono avviare le loro ricerche:

Nascosta dietro questa semplice pagina *web* esiste un'infrastruttura molto complessa, resa dai suoi progettisti di semplice utilizzo affinché il maggior numero di persone possa fruire in maniera semplice delle sue funzionalità. Senza scendere nel dettaglio dell'architettura generale di questo motore di ricerca, possiamo vederne nella figura che segue il primo strato nascosto, con l'avvertenza che questo, in qualche modo, è un po' come sollevare solo la buccia più esterna di una cipolla:



Questa vista, limitata a una sola parte del **software** che dal punto di vista grafico realizza la *home-page* del sito, si ottiene con il comando del browser che consente di visualizzare il codice sorgente della pagina.

OSSERVAZIONE Da quanto appena visto è facile intuire come nell'universo informatico vi siano almeno due categorie di attori: gli **utilizzatori** e gli **sviluppatori**.
I primi sono coloro che si siedono davanti a un computer e utilizzano le sue funzionalità per gli scopi più disparati: lavoro, studio, gioco ecc.
I secondi sono gli informatici di professione, cioè coloro che mettono il loro lavoro al servizio dei primi sviluppando i sistemi che altri o loro stessi utilizzeranno.

In modo un po' scherzoso potremmo dire che, rispetto a ciò che appare sullo schermo, i primi sono coloro che in qualche modo stanno «davanti», mentre i secondi sono quelli che stanno «dietro». Ovviamente, mentre occorrono notevoli conoscenze tecniche per appartenere alla categoria degli informatici di professione, per appartenere a quella degli utenti finali ne occorrono decisamente di meno – talvolta solo il minimo indispensabile – e questo grazie al lavoro dei primi, costantemente orientato a rendere sempre più semplice l'uso del computer per tutti.
Nel corso del tempo quella dell'**informatico** è divenuta una professione multiforme, in grado di coprire una vasta gamma di mansioni e di affrontare sempre nuove sfide, tra cui la ricerca di soluzioni a molti problemi della società contemporanea, utilizzando la tecnologia informatica disponibile, e la produzione di nuovi strumenti e nuova conoscenza, facendo avanzare lo stato dell'arte della ricerca tecnologica e scientifica.
In linea di principio, un buon informatico dovrebbe:

- possedere conoscenze e competenze, circa le scienze e le tecnologie dell'informazione e della comunicazione, tali da poter essere utilizzate nella progettazione, nello sviluppo e nella gestione di sistemi informatici;
- avere la capacità di affrontare e analizzare problemi per lo sviluppo di sistemi informatici che ne permettano la soluzione;
- riuscire ad acquisire metodologie di indagine per applicarle a situazioni concrete tramite la conoscenza di vari tipi di strumenti (logici, matematici ecc.) di supporto alle competenze informatiche;
- essere capace di lavorare in gruppo o a livello autonomo in attività inerenti la progettazione, l'organizzazione, la gestione e la manutenzione di sistemi informatici (con specifico riguardo ai requisiti di affidabilità, prestazioni e sicurezza), al fine di potersi inserire adeguatamente in vari ambienti di lavoro quali:
  - aziende produttrici di software (*software house*);
  - aziende ed enti per la ricerca e lo sviluppo di soluzioni nell'ambito delle tecnologie dell'informazione e della comunicazione (ICT, *Information and Communication Technology*);
  - centri di calcolo pubblici e privati;

Software e hardware

Il termine *software* viene generalmente utilizzato per indicare i programmi in grado di funzionare su computer o comunque su un qualsiasi dispositivo con capacità di elaborazione (telefono cellulare, console per videogiochi, navigatore satellitare ecc.).
Il termine deriva dalla lingua inglese come contrazione di due vocaboli, *soft* (morbido) e *ware* (oggetto, prodotto). Tradizionalmente esso si contrappone ad *hardware*, ovvero l'insieme delle componenti fisiche (elettroniche, elettriche, ottiche e meccaniche) di un sistema di elaborazione.

Il computer

Il computer moderno è una macchina basata sul concetto di **programma memorizzato**. Le origini di tale idea non sono chiare, alcuni ritengono che si debba a von Neumann, altri a Turing, che si erano conosciuti nel 1935. In ogni caso, von Neumann fu il primo che la realizzò praticamente. Il punto chiave consistette nell'inserire i programmi dentro la macchina, non sotto la tradizionale forma di connessioni tra cavi, ma come cariche elettriche o impulsi elettrici, in modo tale da controllare e modificare le operazioni della macchina senza la necessità di riposizionare cavi o interruttori.
Riuscire a immaginare – come fecero von Neumann e Turing – una macchina controllata dall'interno richiese un notevole salto concettuale, perché ▶

– aziende fornitrici di servizi informatici e gestionali;
– enti pubblici e privati con esigenze di gestione di grandi basi di dati;
– piccole e medie aziende in ogni settore produttivo con esigenze di gestione informatica e reti di computer.

Ma l'informatica, considerata una delle scienze moderne per eccellenza, in realtà affonda le sue radici nel passato; padri dell'informatica sono infatti solitamente considerati due matematici del secolo scorso la cui attività è stata al culmine negli anni immediatamente successivi alla seconda guerra mondiale:

- **John von Neumann** (1903-1957), matematico e informatico ungherese naturalizzato statunitense, ideò l'architettura della «macchina» che porta il suo nome, sulla base della quale sono progettati e realizzati i computer moderni;
- **Alan Turing** (1912-1954), matematico, logico e crittoanalista britannico, concepì il modello di «macchina» che porta il suo nome, fondamento teorico di ogni moderno sistema programmabile.

L'introduzione del **PC** (*Personal Computer*), e il conseguente ingresso dell'informatica nelle case di tutti, è invece un evento molto più recente: il primo PC fu costruito intorno alla metà degli anni '70. Nello stesso periodo ha origine la più grande rivoluzione informatica, e cioè la diffusione dell'accesso a Internet; le funzionalità del protocollo **TCP/IP**, sul quale si basa la trasmissione delle informazioni sulla rete, sono state inizialmente dimostrate su reti composte da poche decine di computer. Oggi Internet connette centinaia di milioni di computer.

1 Dati e informazioni

Spesso nella pratica di tutti i giorni si tende a rendere interscambiabili i termini «dato» e «informazione» e in effetti non è sempre facile fare una netta distinzione tra questi.
Consideriamo le seguenti definizioni e proviamo a fare qualche esempio.

▶ Un **dato** (dal latino *datum*, «fatto») è la misura primaria di un fenomeno che siamo interessati a osservare.

ESEMPIO

Si può utilizzare un metro per misurare l'altezza di una persona, oppure utilizzare un termometro per misurare la sua temperatura corporea. Supponendo di aver misurato nel primo caso 178 cm e nel secondo caso 39 °C, abbiamo ottenuto due **dati** distinti rispetto ad altrettanti fenomeni osservati.

▶ L'**informazione** è ciò che si ottiene dall'elaborazione di uno o più dati e che è in grado di accrescere il nostro stato di conoscenza rispetto a un fenomeno con il quale siamo interessati a interagire.

ESEMPIO

Nel caso della temperatura corporea a cui si è fatto riferimento, una sua opportuna elaborazione ci può portare a dedurre l'**informazione** che il soggetto osservato ha qualche problema di salute: infatti alcune minime conoscenze di medicina ci dicono che una temperatura superiore a 37 °C è indice di una alterazione febbrile in corso (nel caso specifico la nostra elaborazione si è limitata al confronto del dato rilevato con un valore di soglia standard).

ESEMPIO

Sempre relativamente al concetto di informazione è possibile pensare alla differenza funzionale che c'è tra un lampione e un semaforo: entrambi utilizzano l'energia elettrica in ingresso per produrre energia luminosa in uscita, ma mentre la luce del lampione ha il solo scopo di illuminare, i tre diversi colori della luce del semaforo hanno una importante valenza informativa nel controllo dei flussi di traffico a un incrocio.

In generale, da un punto di vista puramente intuitivo, possiamo affermare che, maggiore è la quantità di dati di cui si dispone rispetto a un fenomeno da analizzare, migliore sarà la **qualità dell'informazione** che, a partire da una corretta elaborazione, si riesce a conseguire per approfondire la sua conoscenza.

ESEMPIO

Un medico incrocia tra loro i sintomi e i risultati delle analisi (dati) per formulare una corretta diagnosi dei problemi di salute del paziente (informazione). Allo stesso modo un economista analizza i dati di una certa azienda per determinarne l'andamento economico ed eventualmente individuare i correttivi idonei affinché la sua redditività si mantenga su livelli soddisfacenti.

Non sempre, osservando lo scenario di un fenomeno, tutti i dati analizzabili risultano utili alla sintesi dell'informazione. Per esempio, esistono situazioni in cui il paziente presenta dei sintomi che non solo non sono utili alla formulazione di una corretta diagnosi, ma rischiano di portare il medico fuori strada.

OSSERVAZIONE Possiamo affermare che, in generale, il conseguimento di buone informazioni discende da un aspetto sia **quantitativo** sia **qualitativo** dei dati. Ovvero è necessario raccogliere più dati possibile circa il fenomeno oggetto della nostra analisi, ma anche essere in grado di scartare quelli che non sono necessari ai nostri scopi.
Una corretta elaborazione di tali dati ci condurrà quindi a informazioni utili sia alla comprensione sia a una corretta interazione con il fenomeno studiato.

Esaminiamo ora il lavoro del nostro medico (o economista): egli è in grado di raccogliere molti dati circa il paziente (o azienda) che sta analizzando, di scartare con precisione i dati inutili, di correlare in maniera corretta i dati e di elaborarli fino a formulare una corretta diagnosi e individuare la cura appropriata, ma... il malato muore (o l'azienda fallisce)!

► fino ad allora le macchine erano sempre state controllate dall'esterno mediante leve, pulsanti e dispositivi vari.
In particolare von Neumann decise che le funzioni di base di un computer dovessero essere presenti nella macchina come parte integrante della sua struttura fisica, potendone variare a piacere ordine e combinazione di esecuzione, così da poter lavorare su problemi diversi senza dover modificare la macchina ma semplicemente cambiando le istruzioni. «Una volta fornite le istruzioni al dispositivo», spiegava von Neumann, questo le avrebbe «svolte completamente e senza bisogno di ulteriori interventi dell'intelligenza umana.»
Sulla base di queste idee von Neumann contribuì nel 1945 alla progettazione del computer EDVAC, mentre Turing realizzò il computer ACE nel 1950.

Informatica e velocità

La storia dell'umanità vista attraverso l'ottica della velocità dimostra che solo chi controlla le «macchine» che permettono di agire velocemente produce ricchezza e detiene il potere. Dove per «macchina» si può intendere la fionda che lancia la pietra, il jet supersonico che trasporta un missile come pure il pacifico fax. Quello che conta è guadagnare tempo sul tempo, superare i limiti dello spazio.

PAUL VIRILIO

La rivoluzione industriale, che pure aveva prodotto un'accelerazione notevole dei mezzi di trasporto, produzione e comunicazione, era tuttavia rimasta a una velocità «relativa»: ►

▶ la dimensione spazio-tempo del mondo cambiava ma continuava a esistere. Oggi l'informatica tende verso la velocità «assoluta»: le tele-tecnologie mirano ad annullare le distanze e in genere viene introdotto un nuovo concetto di spazio-tempo che individua una realtà che tende a essere sempre più «virtuale».

Dov'è che ha sbagliato? Probabilmente non ha tenuto conto di un ulteriore elemento critico, il **fattore tempo**: ha impiegato troppo tempo nel processo di indagine/formulazione della diagnosi.

OSSERVAZIONE Spesso nella vita reale le informazioni hanno una validità limitata nel tempo, basta pensare all'andamento dei titoli in borsa: nel giro di pochi minuti la loro valutazione può cambiare notevolmente.

Una delle funzioni dell'informatica è quella di mettere a disposizione dell'utente di un sistema informatico procedure e tecnologie in grado di supportare il processo che porta dalla raccolta, selezione, elaborazione dei dati al conseguimento delle informazioni in tempi rapidi.

2 La codifica delle informazioni

Da sempre la società umana ha fatto della comunicazione delle informazioni uno degli strumenti alla base della propria evoluzione: tramite essa gli individui si sono scambiati e tramandati concetti e significati. Da un certo momento la comunicazione ha smesso di essere solo verbale e ha iniziato a essere rappresentata tramite simboli, a partire dai graffiti rupestri fino ad arrivare alle forme di scrittura così come le conosciamo oggi.
Una costante di questo processo è sicuramente stata la scelta della simbologia adottata: la scrittura cuneiforme dei Sumeri, i geroglifici egiziani, gli ideogrammi cinesi, le cifre romane e quelle arabe sono altrettante forme di scrittura, ciascuna delle quali ha adottato un proprio alfabeto.

▶ Un **alfabeto** è un qualunque insieme finito di simboli che ragionevolmente devono essere facilmente distinguibili e producibili.

▶ Chiameremo **configurazioni (o stringhe) «ben formate»** sequenze più o meno lunghe di simboli definite su un certo alfabeto, distinguibili tra di loro e che, in base a precise regole di composizione, rappresentano enunciati sintatticamente corretti.

ESEMPIO

Facendo riferimento ai numeri (alfabeto composto dalle cifre arabe e da alcuni simboli speciali come «+» e «–» per rappresentare i numeri positivi e negativi, il separatore delle cifre decimali «,» e quello delle migliaia «.») possiamo dire che le stringhe «100,52» e «23.712» sono ben formate, mentre «100,5,2» e «237.12» non lo sono.

▶ Un **codice** è un insieme di regole che stabilisce una corrispondenza biunivoca tra un insieme di configurazioni ben formate, definite su un certo alfabeto, e un insieme di significati.

In pratica un codice associa ogni configurazione a una particolare entità di informazione: la configurazione diventa un modo per rappresentare l'entità informativa. Condizione necessaria per l'esistenza di un codice è che vi siano almeno tante configurazioni quanti sono i dati distinti da rappresentare.
Le regole di composizione delle configurazioni ben formate costituiscono l'aspetto di correttezza sintattica della codifica, mentre il codice stesso ne stabilisce la semantica, cioè l'attribuzione dei significati (FIGURA **1**).

FIGURA **1**

OSSERVAZIONE Si noti la differenza che c'è tra **simbologia** e **significato associato**. Essa è particolarmente evidente quando si considerano le nozioni di «cifra» e di «numero»: la prima è un simbolo, mentre il secondo è una nozione astratta. Per esempio, se pensiamo al numero cinque, facciamo riferimento a un concetto che esprime una precisa quantità. Il simbolo «5» è quello che secondo il sistema di numerazione decimale viene utilizzato per rappresentare concretamente tale concetto; parimenti, secondo la numerazione romana, è possibile rappresentare lo stesso concetto utilizzando il simbolo «V».
La rappresentazione o codifica è la condizione che stabilisce la corrispondenza tra simboli e significati: tale convenzione è essenziale perché la comunicazione possa avere luogo.

ESEMPIO

Uno stesso dato può essere codificato in modi diversi (FIGURA **2**).

FIGURA **2**

Ovviamente, a parità di alfabeto, cambiando codice è possibile avere una stessa configurazione associata a significati differenti (FIGURA **3**).

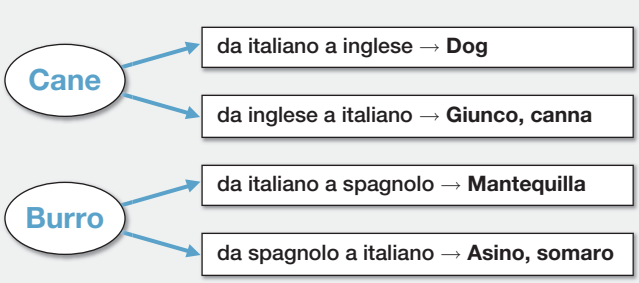


FIGURA **3**

OSSERVAZIONE Sistemi di rappresentazione diversi possono usare uno stesso alfabeto per esprimere significati diversi. Per esempio la notazione italiana e quella anglosassone utilizzano la virgola e il punto nella rappresentazione dei numeri con un significato molto diverso. Nel primo caso, infatti, la virgola viene utilizzata come separatore della parte intera da quella decimale e il punto come separatore delle migliaia, mentre nel secondo caso avviene il contrario:

NOTAZIONE ITALIANA

$$123{,}456 = 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 + 4 \times 10^{-1} + 5 \times 10^{-2} + 6 \times 10^{-3}$$

NOTAZIONE ANGLOSASSONE

$$123{,}456 = 1 \times 10^5 + 2 \times 10^4 + 3 \times 10^3 + 4 \times 10^2 + 5 \times 10^1 + 6 \times 10^0$$

I dati gestiti dai sistemi di elaborazione devono essere codificati per poter essere memorizzati, elaborati, trasmessi, ed essendo i computer costituiti da componenti fisici (circuiti elettronici, dischi magnetici ed ottici ecc.) si sfruttano per la codifica le loro caratteristiche intrinseche. La cosa più semplice da fare con i circuiti è distinguere due stati diversi come una tensione bassa/alta, o l'assenza/presenza di corrente. Lo stesso vale per le unità di memoria, dove è possibile individuare due stati di magnetizzazione delle particelle che costituiscono il rivestimento dei dischi magnetici, oppure la successione di *pits* e *lands* presenti sulla superficie di un supporto ottico letto mediante un raggio laser.

Associando rispettivamente ai due diversi stati i simboli dell'alfabeto costituito dalle sole cifre binarie **«0»** e **«1»** è possibile utilizzare come codici configurazioni più o meno lunghe di **cifre binarie** (***binary digit*: bit**). A ciascuna di queste configurazioni verrà associato, mediante un'opportuna codifica, uno specifico significato.

Nell'uso comune la scelta del sistema decimale è probabilmente derivata dal numero totale di dita delle nostre mani, ma – senza voler entrare nel dettaglio – si può affermare che il sistema binario è tanto efficace quanto quello decimale, o come un qualsiasi altro sistema che utilizzi un diverso numero di cifre.

OSSERVAZIONE L'uso del minor numero possibile di simboli consente di introdurre una semplificazione dei componenti fisici che devono memorizzare ed elaborare le informazioni. Il ricorso a due soli simboli riduce la probabilità di errore nel loro riconoscimento, infatti, essendo i simboli associati a grandezze fisiche, tanti più se ne devono distinguere e tanto meno la loro rilevazione sarà affidabile in presenza di **rumore** (con *rumore* si intende un qualsiasi elemento aleatorio che disturba la qualità e l'affidabilità del dato).

Il **bit** costituisce l'unità elementare di informazione rappresentabile e memorizzabile; ma, dato che la quantità di informazione che può essere contenuta in un singolo bit è minima (0/1), per poter codificare dati complessi è necessario utilizzare gruppi di bit.

Tutto è numero

Pitagora di Samo, uno dei matematici più noti, è anche una delle figure più misteriose e controverse dell'antichità. Visse nel VI secolo a.C., ma non esiste nessun trattato scritto di suo pugno e la sua storia rimane spesso sospesa tra leggenda e verità. Egli creò il Sodalizio Pitagorico: più che una scuola, fu quasi una setta di «filosofi» (termine coniato da Pitagora stesso). Le scoperte della scuola erano segrete, il linguaggio cifrato e le comunicazioni solo orali. L'attività si basava sullo sviluppo delle conoscenze matematiche e aritmetiche, sullo studio dei numeri interi e dei numeri razionali, ottenuti dal rapporto (*ratio* in latino) tra numeri interi.

Pitagora ebbe l'intuizione di riconoscere il numero come entità astratta, non solo come strumento di calcolo, ma anche come elemento fondamentale della realtà e principio assoluto che pervade l'universo, giungendo alla conclusione che: «**Tutto è numero**».

L'idea che il cosmo fosse rappresentabile solo per mezzo di numeri naturali e razionali, per ironia della sorte, entrò in crisi in seguito a una scoperta degli stessi pitagorici: l'esistenza di numeri irrazionali, come la misura della diagonale di un quadrato di lato unitario; per questo motivo la scoperta venne tenuta gelosamente segreta, almeno finché Ippaso di Metaponto non la rivelò all'esterno. La reazione dei pitagorici fu durissima: egli fu bandito dalla scuola e gli fu costruito, sebbene ancora in vita, un monumento funebre; Ippaso morì poco tempo dopo in un naufragio dovuto, secondo la leggenda, alla collera di Zeus.

Un gruppo di 8 bit viene detto **byte** e consente di codificare 256 (2^8) simboli o dati elementari distinti; per esempio un byte può essere utilizzato per codificare tutti i simboli (caratteri) presenti in un testo: lettere maiuscole e minuscole, cifre numeriche e segni di interpunzione.

I computer utilizzano **codici numerici** per rappresentare qualsiasi tipo di informazione: un numero vero e proprio, un testo, un'immagine, un filmato. Ogni dato viene quindi trasformato in una successione di cifre binarie per poter essere memorizzato ed elaborato mediante un computer.

ESEMPIO

La memoria utilizzata per codificare una pagina di testo è di qualche migliaio di byte, quella usata per un'immagine è di alcuni milioni di byte, mentre un lungo filmato può richiedere miliardi di byte per essere memorizzato.

3 L'informatica e i suoi ambiti

Nella terminologia anglosassone l'informatica viene generalmente indicata con il termine ***Computer science***; pensare l'informatica come la scienza dei computer fornisce però una visione riduttiva di questa disciplina, per la quale il computer è solo uno strumento. In italiano si usa spesso l'espressione **Scienze dell'informazione**, ma anche questa non è del tutto adeguata e per molto tempo è stata interpretata in maniera distorta, addirittura associandola al giornalismo.
Volendo dare una definizione possiamo dire che:

▶ L'**informatica** è la scienza che studia i modi e i mezzi per rappresentare ed elaborare informazione, ovvero le modalità e gli strumenti adatti a raccogliere, organizzare, conservare, trasmettere l'informazione, assieme ai mezzi per la sua utilizzazione, in un contesto di procedimenti che risolvono problemi o che ricavano risultati che a loro volta costituiscono nuova informazione.

In particolare l'informatica ha ricevuto grande interesse, sviluppo e diffusione quando si sono realizzate le possibilità di trattare l'informazione con strumenti rapidi e automatici basati sull'uso di computer costruiti con tecnologia elettronica. Il termine **informatica** si riferisce infatti al trattamento automatico dell'informazione: esso è un neologismo derivato dalla lingua francese dove costituisce una contrazione dei termini *informazione* e *automatica*.

OSSERVAZIONE Da quanto detto risulta chiaro che l'informatica non si limita affatto allo studio del computer e del suo utilizzo e, anche se prima dell'avvento del computer non era, come disciplina, ancora autonoma dalle altre scienze, né evidenziata nelle sue caratteristiche fondamentali, l'informatica non è nata con la diffusione dei calcolatori elettronici, si può anzi dire che ne ha permesso la progettazione e realizzazione.
L'informatica affonda le proprie radici nella matematica, anche se ormai è una scienza ben distinta.

ICT

L'acronimo ICT (*Information and Communication Technology*, in italiano Tecnologia dell'Informazione e della Comunicazione) viene utilizzato per indicare l'insieme delle tecnologie che consentono l'elaborazione e la comunicazione dell'informazione in formato digitale (cioè rappresentata mediante codici numerici). Rientrano in questo ambito lo studio, la progettazione, lo sviluppo, la realizzazione e la gestione dei sistemi informativi e di telecomunicazione computerizzati, con particolare attenzione alle applicazioni software e alle componenti hardware che le eseguono.
Il fine ultimo dell'ICT è la gestione dei dati e delle informazioni; l'ICT è ormai da considerarsi una risorsa essenziale per le organizzazioni pubbliche e private, per le quali è sempre più importante riuscire a gestire in maniera rapida, efficace ed efficiente i volumi crescenti di informazioni che le caratterizzano.

ESEMPIO

Per capire quali sono i rapporti tra matematica e informatica e quali sono invece le peculiarità dell'informatica come scienza, è utile fare un esempio.
Consideriamo un'operazione matematica semplicissima: la moltiplicazione a due cifre. In prima istanza la matematica e l'informatica hanno due punti di vista diversi sul problema della moltiplicazione a due cifre: la matematica si occupa di scoprire un procedimento per risolvere il problema, mentre l'informatica si occupa di codificare questo procedimento in un linguaggio eseguibile da una macchina.

OSSERVAZIONE Se è noto un procedimento, è possibile spiegarlo a un'altra persona o a una macchina; l'informatica si occupa proprio di automatizzare la procedura matematica in maniera da renderla utilizzabile tramite un computer e questo perché:

- la macchina può eseguire tale procedura molto più rapidamente dell'uomo;
- l'uomo può commettere errori nell'eseguire la procedura, mentre la macchina la applica sempre in maniera corretta;
- lasciando i compiti più lunghi e ripetitivi alla macchina, l'uomo può dedicarsi ad attività concettualmente più utili e interessanti.

Dal punto di vista dell'utente l'informatica offre una serie di strumenti e funzionalità basate sull'uso del computer corredato da «applicazioni» software che cercano di guidare l'utente nelle sue attività, tenendo conto del suo livello di capacità e di iniziativa.
Il mercato offre svariati strumenti software orientati ad aumentare la produttività di varie categorie di utenti: elaborazione di testi, fogli elettronici, gestione di basi di dati, posta elettronica, applicazioni di matematica e statistica, CAD (*Computer Aided Design*) e così via.
Dal punto di vista della classificazione dei settori in cui si articola l'informatica professionale riportiamo di seguito un estratto della classificazione della aree disciplinari suggerita dalla ACM (*Association for Computing Machinery*).

1. **Algoritmi e strutture dati**: studia i procedimenti per trattare l'informazione, le loro potenzialità e limiti, le loro caratteristiche, complessità e difficoltà di effettuazione; studia inoltre come l'informazione può essere rappresentata e strutturata ai fini del suo miglior trattamento.
2. **Linguaggi di programmazione**: studia i linguaggi per esprimere tali procedimenti, le loro caratteristiche, le possibilità di traduzione in linguaggi eseguibili in modo automatico e le tecniche di programmazione nei vari linguaggi.
3. **Architetture degli elaboratori**: studia la struttura di computer capaci di eseguire programmi in maniera sempre più veloce, affidabile e intelligente.
4. **Architetture di rete**: studia le tecniche di interconnessione tra computer e i protocolli di comunicazione per lo scambio di informazioni su rete.
5. **Sistemi operativi**: studia come dotare i computer di un programma di base che ne renda possibile e agevole l'uso e ne sfrutti in modo ottimale le prestazioni.

6. **Ingegneria del software**: studia strumenti e metodi per ottimizzare il lavoro degli sviluppatori e per ottenere programmi più efficienti e più usabili in un'ottica di ottimizzazione del lavoro umano.
7. **Computazione numerica e simbolica**: studia la costruzione e l'uso di modelli per la rappresentazione dei problemi da risolvere con programmi di calcolo numerico o simbolico.
8. **Basi di dati e sistemi per il reperimento dell'informazione**: studia il trattamento di grandi masse di dati che devono essere facili da accedere e da modificare, gestiti e conservati con sicurezza, protetti da intrusioni.
9. **Intelligenza artificiale**: studia come usare i computer per compiti sempre più complessi e tradizionalmente demandati a operatori umani intelligenti (riconoscimento del linguaggio naturale, traduzioni, sistemi esperti di supporto alle decisioni ecc.).
10. **Robotica e visione**: studia la gestione e il controllo di dispositivi automatici in grado di compiere operazioni complesse; tra le funzioni necessarie a tale scopo è importante la visione e il riconoscimento di forme e di oggetti fisici.

Questi sono gli ambiti disciplinari dal punto di vista degli addetti ai lavori, i tecnici capaci di usare queste conoscenze per creare strumenti informatici o servizi informatici, e sono indipendenti dalla capacità di una persona di servirsi di strumenti o servizi propri dell'informatica.

Sintesi

Dato. È la misura primaria di un fenomeno che siamo interessati a osservare.

Elaborazione. Insieme di attività orientate alla raccolta, organizzazione, correlazione, comunicazione e interpretazione finalizzata dei dati.

Informazione. È il risultato dell'elaborazione di uno o più dati che è in grado di accrescere il nostro stato di conoscenza rispetto a un fenomeno con il quale siamo interessati a interagire.

Alfabeto. Insieme finito di simboli facilmente distinguibili e riproducibili.

Configurazioni «ben formate». Sequenze di simboli definite su un certo alfabeto, distinguibili tra di loro che, in base a regole di composizione, rappresentano enunciati sintatticamente corretti.

Codice. Insieme di regole che stabilisce una corrispondenza biunivoca tra un insieme di configurazioni ben formate definite su un certo alfabeto e un insieme di significati.

Sistema binario. Sistema numerico utilizzato nella tecnologia informatica per rappresentare dati (qualsiasi tipo di dato viene rappresentato in formato numerico).

Bit. Quantità minima di informazione rappresentabile: può assumere i due valori «0» e «1».

Informatica. Scienza che studia i modi e i mezzi per rappresentare ed elaborare informazione; con questo termine ci si riferisce al trattamento automatico dell'informazione: è un neologismo derivato dal francese e costituisce una contrazione dei termini *informazione* e *automatica*.

Ambiti dell'informatica. Tra le varie aree disciplinari dell'informatica, alcune delle più importanti sono le seguenti: Algoritmi e strutture dati, Linguaggi di programmazione, Architetture degli elaboratori, Architetture di rete, Sistemi operativi, Ingegneria del software, Computazione numerica e simbolica, Basi di dati, Intelligenza artificiale, Robotica e visione.

A2 Algoritmi

Qual è il giorno della settimana che corrisponde a una certa data? Questo è un piccolo problema che, nella vita di tutti i giorni, capita spesso di dover risolvere. In genere ci si affida a una agenda, cartacea o elettronica che sia, ma la questione può essere risolta con i semplici calcoli descritti di seguito:

- si prenda in considerazione una data, per esempio 12/01/1998;
- si chiami Y il valore dell'anno ed M il valore del mese diminuito di 2, nel nostro esempio $Y = 1998$ ed $M = 1 - 2 = -1$;
- se $M <= 2$ si sottragga 1 da Y e si aggiunga 12 a M, nel nostro esempio $Y = 1998 - 1 = 1997$ ed $M = -1 + 12 = 11$.

Per le date del calendario giuliano (in vigore fino al 4 ottobre 1582) si procede come indicato di seguito (***INT***(…) indica la sola parte intera dell'espressione tra parentesi):

$$D = 5 + \text{giorno} + Y + \textbf{\textit{INT}}(Y : 4) + \textbf{\textit{INT}}\,(31 \times M : 12)$$

Per le date del calendario gregoriano (il nostro calendario in vigore dal 15 ottobre 1582) si procede invece nel modo seguente:

$$D = \text{giorno} + Y + \textbf{\textit{INT}}(Y : 4) - \textbf{\textit{INT}}(Y : 100) + \textbf{\textit{INT}}(Y : 400) + \textbf{\textit{INT}}(31 \times M : 12)$$

Nel nostro esempio:

$$D = 12 + 1997 + 499 - 19 + 4 + 28 = 2521$$

In entrambi i casi si divide D per 7 e si prende il resto di tale divisione: se il valore ottenuto è 0 il giorno è domenica, se 1 è lunedì, se 2 è martedì e così via. Nel nostro esempio:

$$2521 - \textbf{\textit{INT}}(2521 : 7) \times 7 = 2521 - 2520 = 1 \text{ ovvero lunedì}$$

OSSERVAZIONE Le «variabili» letterali D, M, Y sono state utilizzate per rappresentare dati e risultati, come si è soliti fare con le formule delle scienze applicate o con le espressioni algebriche.

OSSERVAZIONE Il resto della divisione di D per 7 è stato calcolato sottraendo al valore di D la parte intera della divisione moltiplicata per 7. È questo un metodo generale: volendo, per esempio, calcolare il resto della divisione di 12 per 5, si determina il risultato della divisione $(12 : 5 = 2{,}4)$, si prende la parte intera $(\textbf{\textit{INT}}(2{,}4) = 2)$, la si moltiplica per 5 $(2 \times 5 = 10)$ e si sottrae il risultato da 12 $(12 - 10 = 2)$ ottenendo il resto.

Calendari giuliano e gregoriano

Il **calendario giuliano** è un calendario solare, ovvero basato sul ciclo delle stagioni.
Fu promulgato da Giulio Cesare (da cui prende il nome), nel 46 a.C. Esso fu da allora il calendario ufficiale di Roma e dei suoi domini.
Nel 1582 è stato sostituito dal **calendario gregoriano** per decreto di papa Gregorio XIII; diverse nazioni tuttavia hanno continuato a utilizzare il calendario giuliano, adeguandosi poi in tempi diversi tra il XVIII e il XX secolo.
Il calendario gregoriano è il calendario ufficiale della maggior parte dei paesi del mondo ed è un calendario solare derivato dal calendario giuliano.
L'anno si compone di 12 mesi di durate diverse (da 28 a 31 giorni), per un totale di 365 o 366 giorni. Gli anni di 366 giorni sono detti *bisestili*: è bisestile un anno ogni quattro, esclusi gli anni la cui numerazione è multipla di 100, ma non di 400.
L'introduzione dell'anno bisestile si rese necessaria per accordare la durata media dell'anno del calendario con quella dell'*anno solare*, che è di circa 365,24 giorni.
Per compensare le differenze con il ciclo solare accumulate dal calendario giuliano al momento della sua introduzione furono «saltati» 11 giorni: dal 4 ottobre al 15 ottobre del 1582.

1 Dal problema all'algoritmo

Anche se non è facile dare una definizione esatta del termine **problema**, si può affermare in prima approssimazione che, almeno nell'uso comune, viene indicata con questa parola una qualsiasi situazione reale che necessiti di un'azione per poter essere risolta, ovvero trasformata in uno stato finale che si valuta migliore o comunque preferibile a quello di partenza. Risolvere un problema significa quindi ottenere un risultato partendo da una determinata situazione iniziale e seguendo un opportuno procedimento.

OSSERVAZIONE È importante evidenziare come le soluzioni debbano essere espresse in modo comprensibile e composte da operazioni che siano alla portata, in termini sia interpretativi sia operativi, di chi dovrà applicare il procedimento risolutivo.
In genere si dice che un problema è ben formulato quando contiene tutte le informazioni necessarie per ottenere i risultati voluti applicando un metodo risolutivo.

Come abbiamo visto nell'esempio in apertura del capitolo, è possibile studiare e proporre procedure (ovvero metodi e procedimenti sistematici) finalizzate a effettuare una classe di operazioni, ovvero a risolvere una classe di problemi.

Intuitivamente chiameremo **algoritmo** un insieme di istruzioni la cui esecuzione porta a risolvere un problema o comunque a conseguire un determinato risultato.
Il concetto di algoritmo è fondamentale nell'informatica ed è considerato primitivo, cioè non formalmente definibile in termini di altri concetti.

Il procedimento per determinare il giorno della settimana può essere sinteticamente descritto tramite una sequenza di formule (l'operazione n ***MOD*** m determina il resto della divisione tra n ed m ed equivale all'espressione $n - \textbf{\textit{INT}}(n : m) \times m$):

$Y \leftarrow anno$
$M \leftarrow mese - 2$
Se $M <= 2$ ***allora*** $Y \leftarrow Y - 1$ e $M \leftarrow M + 12$
$D \leftarrow ((giorno + Y + \textbf{\textit{INT}}(Y : 4) - \textbf{\textit{INT}}(Y : 100) + \textbf{\textit{INT}}(Y : 400) +$
$+ \textbf{\textit{INT}}(31 \times M : 12))$ ***MOD*** $7)$

OSSERVAZIONI

- L'ordine temporale di valutazione delle formule è fondamentale per la correttezza del calcolo (il valore assegnato precedentemente a una variabile è spesso successivamente utilizzato nella valutazione di una nuova espressione).

Il termine *algoritmo*

La parola *algoritmo* è entrata in uso negli anni '50 del secolo scorso per sostituire la parola *algorismo* che, dal medioevo, designava il processo di calcolare con i numeri arabi.
Nel medioevo si credeva (come tuttora molti studenti!) che derivasse dal greco *algiros* «doloroso» e *arithmos* «numero», ma in realtà deriva dal nome di Al-Khwarizmi, un grande matematico arabo del IX secolo, famoso per avere formalizzato il sistema di numerazione posizionale che tutti oggi utilizziamo. Il suo libro sui numeri «indiani» fu tradotto in latino, anche se la diffusione nel mondo occidentale della numerazione araba è da attribuirsi soprattutto all'opera di Leonardo da Pisa, noto come Fibonacci.
In ogni caso le procedure che permettevano di effettuare i calcoli divennero note come *algorismi* o *algoritmi*, e lo stesso termine fu in seguito usato in generale per definire le procedure di calcolo necessarie per ottenere un determinato risultato.

- Si è fatto ricorso a variabili letterali per la necessità di rappresentare quantità il cui valore numerico non è noto perché ancora da determinare, oppure perché è variabile in relazione a diversi usi del procedimento.
- Il simbolo «←» è un operatore di assegnamento col significato che la variabile che sta alla sua sinistra assume il risultato della valutazione dell'espressione che sta alla sua destra (in questo senso le variabili permettono la «memorizzazione» di valori numerici: si noti come espressioni del tipo $M \leftarrow M + 12$ o $Y \leftarrow Y - 1$ permettano l'aggiornamento del valore di una variabile).
- L'assegnazione del valore a una variabile può essere condizionato dal fatto che essa stessa, o altre variabili, assuma/no o meno determinati valori (***Se*** … ***allora ...***).

▶ Una **variabile** è una coppia (**nome,valore**) dove:

- il **nome** è l'identificatore associato alla variabile (esso rimane costante durante l'esecuzione dell'algoritmo);
- il **valore** è il contenuto associato alla variabile in un certo istante dell'esecuzione dell'algoritmo (esso può variare durante l'esecuzione dell'algoritmo).

OSSERVAZIONE Nella gestione delle variabili si è fatto uso del metacarattere «←» come operatore di assegnamento o modifica del valore di una variabile. Questo operatore è **distruttivo**, nel senso che con la sua applicazione si ha la modifica del contenuto nella variabile che sta alla sua sinistra e di conseguenza la perdita del precedente valore contenuto in essa.
Per esempio, supponendo che M contenga il valore 2, dopo l'esecuzione dell'istruzione $M \leftarrow M + 12$ il suo valore sarà 14; infatti l'ordine di esecuzione dell'assegnamento prevede prima la valutazione dell'espressione $M + 12$ $(2 + 12 = 14)$ e successivamente la memorizzazione del risultato nella variabile M.

OSSERVAZIONE In informatica l'operatore di moltiplicazione «×» viene in generale sostituito dal simbolo «*» con lo stesso significato. Questo, storicamente, è dovuto alla necessità di utilizzare una simbologia che non generi confusione, differenziandolo chiaramente dalla lettera «x». Nello stesso modo per la divisione viene utilizzato il simbolo «/» in sostituzione di «:».

2 Algoritmi ed esecutori

▶ Un **algoritmo** è l'esplicitazione dei passi elementari necessari a risolvere un determinato problema, o una classe di problemi simili. Esso, generalmente, opera su dei dati di ingresso (**input**) per fornire dei risultati in uscita (**output**).

Come abbiamo già osservato, un problema non necessariamente deve essere di natura matematica: per esempio anche cucinare una pietanza potrebbe essere un problema al quale è necessario trovare una soluzione. In questo caso l'algoritmo risolutivo è costituito dalla relativa ricetta che descrive (esplicita) passo per passo quali sono le azioni da intraprendere per ottenere il risultato finale. I dati di ingresso (input) sono in questo caso gli ingredienti da utilizzare, mentre l'uscita (output) è rappresentata dal piatto pronto per essere consumato. Chi esegue l'algoritmo, in questo caso il processo di realizzazione della pietanza, in genere viene detto **esecutore**.

Le seguenti proprietà rappresentano le caratteristiche fondamentali di un algoritmo.

- 1. **Finitezza**: un algoritmo è composto da un numero finito di istruzioni (passi elementari).
- 2. **Terminazione**: dopo l'esecuzione di un numero finito di passi l'algoritmo deve terminare.

ESEMPIO

Esaminiamo il seguente procedimento:

- prendere un numero naturale N;
- aggiungere 1 a N;
- ripetere il passo precedente.

Questo procedimento non è un algoritmo perché è in contrasto con la proprietà di terminazione, in quanto prevede l'esecuzione di alcune istruzioni un numero infinito di volte.

OSSERVAZIONE In effetti esistono algoritmi che non terminano mai; generalmente essi vengono indicati con il termine *metodo di computazione*. Un esempio è rappresentato da un procedimento che calcola le successive cifre decimali di π.
Nella nostra trattazione ci limiteremo a trattare algoritmi che rispettano la proprietà di terminazione.

- 3. **Determinatezza**: a ogni passo dell'algoritmo deve essere specificata l'azione da intraprendere senza che vi siano ambiguità.

ESEMPIO

Consideriamo il seguente procedimento:

- prendere due numeri M ed N;
- se M è molto maggiore di N allora calcolare $M - N$, altrimenti calcolare $M + N$;
- fornire il risultato ottenuto.

Il procedimento non è un algoritmo perché il secondo passo è ambiguo, in quanto il concetto di «molto maggiore» non è determinato e sarebbe pertanto necessario definire un criterio per valutare quando un numero è molto maggiore di un altro.

- 4. **Effettività**: l'azione specificata in ogni passo dell'algoritmo deve essere effettivamente eseguibile dall'esecutore preposto per l'esecuzione dell'algoritmo stesso; non avrebbe infatti senso prevedere in un algoritmo azioni che l'esecutore designato non è in grado di svolgere.
- 5. **Generalità**: un algoritmo dovrebbe essere progettato per risolvere non tanto uno specifico problema, quanto una classe di problemi simili.

ESEMPIO

A proposito della proprietà di generalità, un algoritmo per determinare il giorno della settimana della data 12/01/1998 sarebbe di scarso interesse, mentre è senz'altro utile un algoritmo che consente di determinare il giorno della settimana di una qualsiasi data.

Nella proprietà di effettività si è fatto riferimento al termine *esecutore*, identificando come esecutore «chi» è preposto all'esecuzione dei singoli passi specificati in un algoritmo. In genere si classificano gli esecutori in due categorie fondamentali:

- **esecutori intelligenti**;
- **esecutori automatici**.

Alla prima categoria appartengono gli esecutori che, nell'affrontare la risoluzione di uno specifico problema, sono in grado di applicare le conoscenze derivanti dalle esperienze acquisite nel corso della propria attività. Un esecutore intelligente è quindi caratterizzato dalla capacità di accrescere le proprie conoscenze mano a mano che affronta nuovi tipi di problemi: l'esempio più evidente di un esecutore intelligente è l'uomo (con questo termine indichiamo la nostra specie, senza ovviamente fare distinzioni di genere).

Alla seconda categoria appartengono invece esecutori che riescono a risolvere un determinato problema solo eseguendo meccanicamente i singoli passi di un algoritmo predefinito, generalmente senza apprendere nulla nel corso della propria attività. L'unico vantaggio che questo tipo di esecutori presenta rispetto a quello intelligente è la possibilità di eseguire i passi di un algoritmo con precisione e velocità.

ESEMPIO

La lavatrice espleta la sua attività in funzione del programma di lavaggio che è stato impostato.
Il computer esegue programmi che implementano algoritmi per la risoluzione di vari tipi di problemi.

Nell'eseguire un algoritmo si può supporre che l'esecutore predisponga una tabella che permetta di tenere traccia dei valori assunti dalle singole variabili nel corso della valutazione delle espressioni.

ESEMPIO

Per la determinazione del giorno della settimana della data 12/01/1998 la **tabella di traccia** sarebbe la TABELLA **1**.

TABELLA **1**

Istruzione	Anno	Mese	Giorno	*Y*	*M*	*D*
Y ← *anno*	1998	1	12	1998		
M ← *mese* – 2	1998	1	12	1998	–1	
Y ← *Y* – 1	1998	1	12	1997	–1	
M ← *M* + 12	1998	1	12	1997	11	
D ← ((*giorno* + *Y* + *INT*(*Y*/4) – *INT*(*Y*/100) + *INT*(*Y*/400) + *INT*(31 * *M*/12)) *MOD* 7)	1998	1	12	1997	11	1

Analizziamo due semplici algoritmi espressi in un linguaggio simile al linguaggio naturale, ma con una struttura che ne facilita l'interpretazione e la non ambiguità delle indicazioni.

Algoritmo 1

```
inizio procedimento
    leggi il valore di m e di n
    c ← 1
    finché (c < m) e (c <= n) esegui
        inizio operazione
            se (c divide m) e (c divide n)
                allora r ← c
            c ← c + 1
        fine operazione
    scrivi il valore di r
fine procedimento
```

Algoritmo 2

```
inizio procedimento
    leggi il valore di m e di n
        finché (m <> n) esegui
            inizio operazione
            se (m > n)
                allora m ← m – n
                altrimenti n ← n – m
            fine operazione
    scrivi il valore di n
fine procedimento
```

Applicando i due algoritmi ai valori di input $m = 12$ ed $n = 9$ otteniamo le due tabelle di traccia riportate nelle TABELLE **2** e **3**.

TABELLA **2**

Algoritmo 1					
Istruzione	***m***	***n***	***c***	***r***	**Condizione**
leggi il valore di *m* e di *n*	12	6			
c ← 1	12	6	1		
(*c* < *m*) **e** (*c* <= *n*)	12	6	1		Vero
(*c* **divide** *m*) **e** (*c* **divide** *n*)	12	6	1		Vero
r ← *c*	12	6	1	1	
c ← *c* + 1	12	6	2	1	
(*c* < *m*) **e** (*c* <= *n*)	12	6	2	1	Vero
(*c* **divide** *m*) **e** (*c* **divide** *n*)	12	6	2	1	Vero
r ← *c*	12	6	2	2	

▶ TABELLA **2**

Algoritmo 1					
Istruzione	***m***	***n***	***c***	***r***	**Condizione**
$c \leftarrow c + 1$	12	6	3	2	
($c < m$) **e** ($c <= n$)	12	6	3	2	Vero
(c **divide** m) **e** (c **divide** n)	12	6	3	2	Vero
$r \leftarrow c$	12	6	3	3	
$c \leftarrow c + 1$	12	6	4	3	
($c < m$) **e** ($c <= n$)	12	6	4	3	Vero
(c **divide** m) **e** (c **divide** n)	12	6	4	3	Falso
$c \leftarrow c + 1$	12	6	6	3	
($c < m$) **e** ($c <= n$)	12	6	6	3	Vero
(c **divide** m) **e** (c **divide** n)	12	6	6	3	Falso
$c \leftarrow c + 1$	12	6	7	3	Vero
($c < m$) **e** ($c <= n$)	12	6	7	3	Falso
scrivi il valore di r	12	6	7	**3**	

TABELLA **3**

Algoritmo 2			
Istruzione	***m***	***n***	**Condizione**
leggi il valore di m e di n	12	9	
($m <> n$)	12	9	Vero
($m > n$)	12	9	Vero
$m \leftarrow m - n$	3	6	
($m <> n$)	3	6	Vero
($m > n$)	3	6	Falso
$n \leftarrow n - m$	3	3	
($m <> n$)	3	3	Falso
scrivi il valore di n	3	**3**	

OSSERVAZIONE Entrambi gli algoritmi con input $m = 12$ e $n = 9$ producono in output il medesimo risultato (3), ma il secondo algoritmo, per giungere al risultato, impiega un minor numero di passi e di variabili rispetto al primo.

È sensato chiedersi che cosa rappresentino i valori calcolati dai due algoritmi e se questi, a parità dei valori di input, producano sempre lo stesso risultato. Di fatto non esiste un criterio generale per ricavare da un algoritmo la relazione che sussiste tra i dati di input e i risultati di output: nonostante la semplicità di questi due algoritmi non è così immediato riconoscere che entrambi servono a calcolare il massimo comun divisore (MCD) tra due numeri.

In generale si può affermare che non esiste un algoritmo univoco per risolvere un determinato problema.

Ulteriori esempi in questo senso sono forniti dai due successivi procedimenti (l'algoritmo 3 rappresenta il procedimento di calcolo del massimo comun divisore che generalmente viene insegnato a scuola).

Algoritmo 3

inizio procedimento
  scomponi *m* ed *n* in fattori primi
  moltiplica i fattori primi comuni, col minimo esponente
  scrivi il risultato della moltiplicazione
fine procedimento

Algoritmo 4

inizio procedimento
  trova tutti i divisori di *m* e tutti i divisori di *n*
  scrivi il più grande tra i divisori comuni
fine procedimento

I quattro algoritmi illustrati forniscono tutti lo stesso risultato (il MCD tra due numeri), anche se non è immediato dimostrarlo.

OSSERVAZIONE I quattro diversi procedimenti sono adatti a esecutori con differenti capacità di comprensione e di esecuzione; in particolare i primi due usano semplici operazioni aritmetiche, come sottrazioni e divisioni, da ripetere più volte; il terzo e il quarto, invece, indicano operazioni più complesse («trovare tutti i fattori primi», oppure «tutti i divisori») eseguite una sola volta. Possiamo quindi affermare che i primi due sono più adatti a un esecutore automatico e veloce, gli altri due sono pensati per un esecutore intelligente, eventualmente più lento.

Anche se nella definizione di algoritmo si fa riferimento a un'idea generale di esecutore, questo si identifica spesso con il computer. Quindi un algoritmo deve essere definito in modo che possa essere interpretato ed eseguito correttamente dall'elaboratore elettronico; per questo motivo l'algoritmo deve essere opportunamente codificato secondo le regole di uno specifico linguaggio di programmazione. Esistono diversi linguaggi di programmazione, ognuno con le proprie caratteristiche particolari, ma tutti con una caratteristica comune: quella di rendere disponibili istruzioni che permettono di tradurre l'algoritmo in un programma eseguibile dal computer: si parla di **implementazione** dell'algoritmo.
Lo schema di FIGURA **1** rappresenta sinteticamente il percorso logico che, a partire da un problema, porta all'implementazione di un programma che lo risolve.

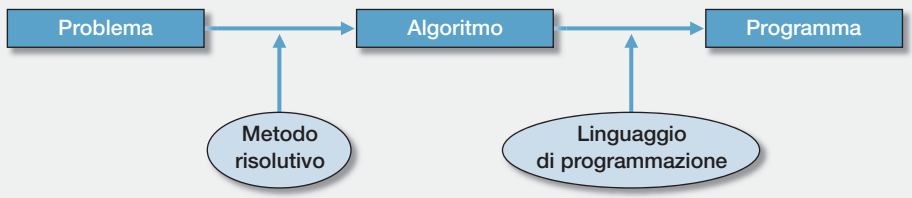


FIGURA **1**

OSSERVAZIONE Nella formulazione delle condizioni negli esempi di algoritmi proposti sono stati utilizzati degli operatori per il «confronto» dei valori contenuti nelle variabili: «<» (minore), «>» (maggiore), «<=» (minore o uguale), «>=» (maggiore o uguale), «<>» (diverso). In particolare per gli operatori formati da due simboli è importante rispettare l'ordine in cui questi sono scritti; «>=» non è la stessa cosa di «=>»: il primo è un operatore valido mentre il secondo non lo è.

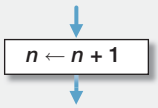
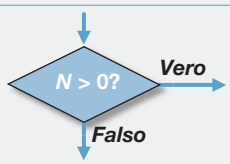
3 La rappresentazione degli algoritmi

3.1 Diagrammi a blocchi e notazione lineare strutturata

Nel corso degli ultimi cinquant'anni sono state utilizzate diverse tecniche di rappresentazione degli algoritmi, sempre con l'obiettivo di definire metodologie efficaci per descrivere gli algoritmi stessi, senza ricorrere a descrizioni discorsive non formalizzate, troppo generiche e appesantite da dettagli talvolta inutili.

Una tradizionale tecnica di rappresentazione è quella del **diagramma a blocchi** (**DAB**). Questa tecnica – nota anche come **diagramma di flusso** o ***flow-chart*** secondo la terminologia anglosassone – si basa su una rappresentazione grafica della sequenza di operazioni dell'algoritmo. Un algoritmo viene strutturato in blocchi di istruzioni ciascuno dei quali, a seconda della propria funzione, viene rappresentato mediante una particolare simbologia: linee orientate (connettori di flusso) collegano tra di loro i vari blocchi per indicare l'ordine di esecuzione delle operazioni dell'algoritmo. Esistono cinque tipi base di blocchi elementari descritti e rappresentati graficamente nella TABELLA **4**.

TABELLA **4**

Blocco di inizio Indica il punto iniziale dell'algoritmo	
Blocco di fine Indica il punto finale dell'algoritmo	
Azione basica Indica un'operazione elementare, per esempio un assegnamento	
Input/Output Rappresenta un'operazione di ingresso (Leggi) o uscita (Scrivi) di dati	
Controllo basico (o selezione) Rappresenta il punto di scelta tra due rami dell'algoritmo in funzione del risultato dell'espressione logica in esso contenuta	

ESEMPIO

Il DAB dell'algoritmo relativo alla determinazione del giorno della settimana di una data fornita come input è mostrato in FIGURA **2**.

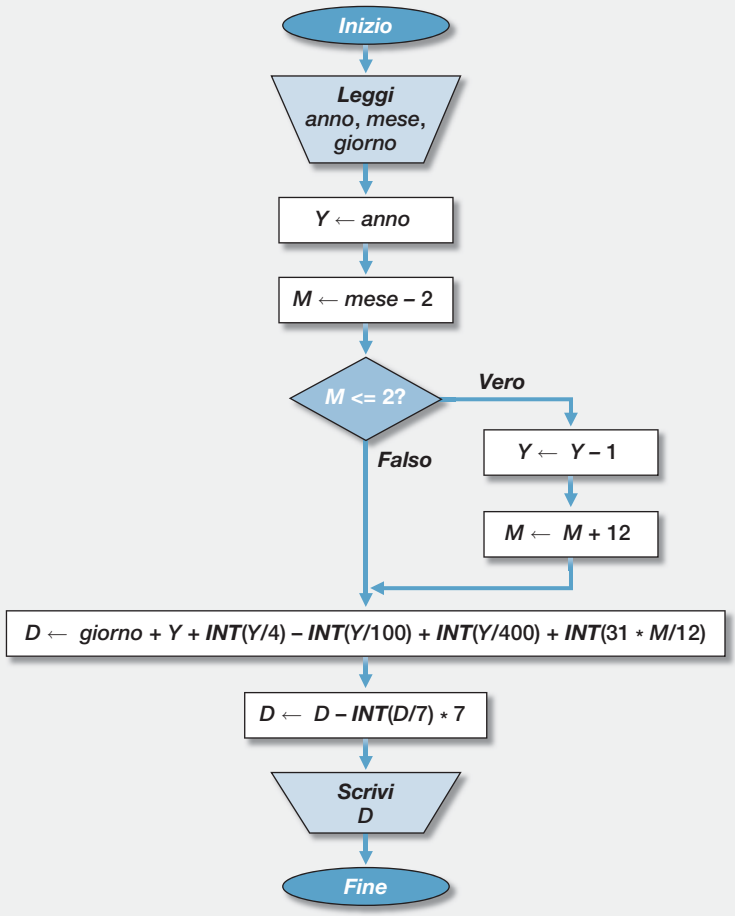


FIGURA **2**

Una tecnica di rappresentazione degli algoritmi alternativa è quella nota come **notazione lineare strutturata** (**NLS**), in cui i simboli grafici dei DAB sono sostituiti da parole chiave (TABELLA **5**).

TABELLA **5**

Blocco di inizio	***Inizio***
Blocco di fine	***Fine***
Azione basica	$n \leftarrow n + 1$
Input/Output	***Leggi/Scrivi*** *n*
Controllo basico (la parte «**altrimenti**» è stata posta tra parentesi quadre per indicare la sua facoltatività)	***Se*** ($n > 0$) ***allora*** …. [***altrimenti*** …]

ESEMPIO

La rappresentazione dell'algoritmo per determinare il giorno della settimana in NLS è la seguente:

```
Inizio
   Leggi anno, mese, giorno
   Y ← anno
   M ← mese – 2
   Se (M <= 2)
      Allora
         Inizio
            Y ← Y – 1
            M ← M + 12
         Fine
   D ← giorno + Y + INT(Y/4) – INT(Y/100) + INT(Y/400) + INT(31 * M/12)
   D ← D – INT(D/7) * 7
   Scrivi D
Fine
```

OSSERVAZIONE Si noti come nella descrizione appena fornita non tutte le istruzioni abbiano il margine sinistro allineato: il margine sinistro più o meno rientrato viene utilizzato per migliorare la leggibilità della rappresentazione dell'algoritmo, in questo modo è possibile evidenziare la dipendenza di alcune istruzioni da altre. Per esempio l'esecuzione delle due istruzioni $Y \leftarrow Y - 1$ e $M \leftarrow M + 12$ dipende dal verificarsi o meno della condizione $M <= 2$; inoltre se la suddetta condizione è vera, le due istruzioni devono essere eseguite entrambe, come se fossero un unico blocco, e pertanto sono inserite in un costrutto del tipo ***Inizio…Fine***.

3.2 Schemi di composizione delle operazioni di un algoritmo

In generale in un algoritmo si possono distinguere almeno **3 schemi fondamentali di composizione delle operazioni**.

- **Sequenza**: una o più azioni basiche sono eseguite una di seguito all'altra nell'ordine in cui sono scritte (nell'esempio precedente le due assegnazioni di valori a Y ed M sono eseguite in sequenza).
- **Selezione**: l'esecutore sceglie tra l'eseguire una o più operazioni in alternativa ad altre, in funzione del verificarsi o meno di una certa condizione logica, selezionando il «percorso» da seguire in dipendenza da quanto eseguito in precedenza nell'avanzamento del procedimento di soluzione (nell'esempio dato la verifica della condizione $M <= 2$ stabilisce se modificare o meno i valori delle variabili Y ed M).
- **Ripetizione** o **iterazione**: l'esecutore ripete più volte una o più operazioni in funzione del verificarsi o meno di una condizione logica che controlla il numero di volte per cui tali istruzioni devono essere ripetute (questo schema è comunemente denominato **ciclo**).

ESEMPIO

Consideriamo l'algoritmo 2 per il calcolo del MCD. La sua descrizione mediante NLS è la seguente:

```
Inizio
   Leggi m,n
      Finché (m <> n) esegui
         Inizio
            Se (m > n)
            Allora m ← m – n
            Altrimenti n ← n – m
         Fine
   Scrivi n
Fine
```

La rappresentazione grafica con DAB, invece, è quella di FIGURA 3.

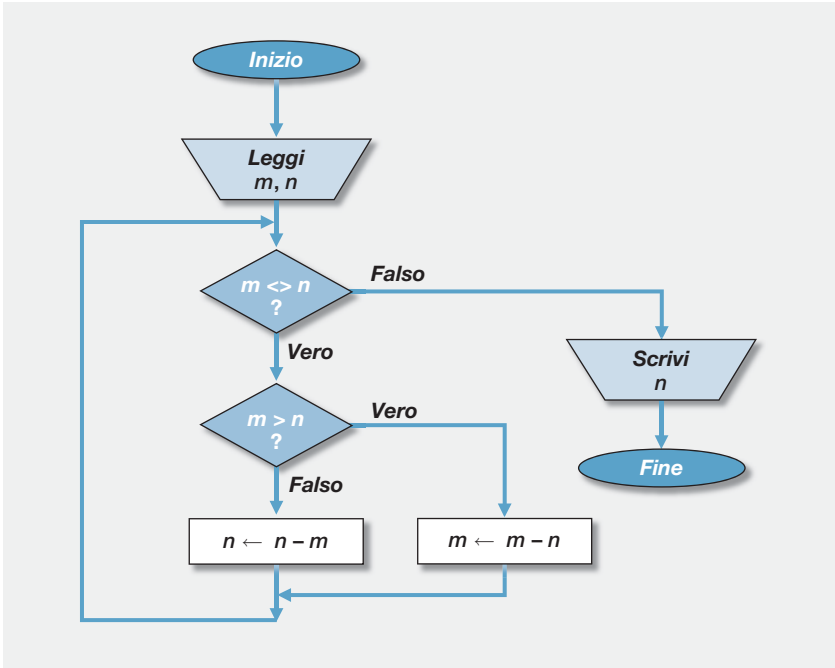


FIGURA 3

In questo algoritmo sono presenti sia una selezione sia una ripetizione: nella NLS sono entrambe evidenziate dall'uso dei costrutti «**Se ... allora ... altrimenti ...**» e «**Finché ... esegui ...**», per indicare rispettivamente la selezione e la ripetizione. Nel DAB si può notare come il primo blocco di selezione sia relativo al controllo della ripetizione, mentre il secondo blocco di controllo è relativo alla selezione.
In generale si può individuare la presenza di un ciclo in corrispondenza di un blocco di selezione e un connettore di flusso che torna «indietro».

Teorema di Böhm-Jacopini

Il teorema di Böhm-Jacopini, enunciato nel 1966 dagli informatici Corrado Böhm e Giuseppe Jacopini, afferma che qualunque algoritmo può essere implementato utilizzando tre soli schemi – la sequenza, la selezione e il ciclo – da applicare ricorsivamente alla composizione di blocchi di istruzioni elementari.
Questo teorema ha un interesse soprattutto teorico, dato che spesso i linguaggi di programmazione prevedono costrutti semplici da usare, anche se internamente complessi, per evitare ai programmatori di doversi occupare di operazioni dispersive rispetto alle finalità implementative degli algoritmi.

Nel seguito presentiamo alcuni esempi di algoritmi che implementano gli schemi fondamentali di composizione utilizzando la NLS, lasciando al lettore l'esercizio di rappresentare gli algoritmi mediante DAB.

3.3 Esempi di algoritmi basati sullo schema di «sequenza»

Calcolo dell'area di un triangolo

```
Inizio
    Leggi base,altezza
    x ← base * altezza
    area ← x/2
    Scrivi area
Fine
```

Scambio del contenuto di due variabili *A* e *B*

```
Inizio
    Leggi A,B
    C ← A
    A ← B
    B ← C
    Scrivi A,B
Fine
```

OSSERVAZIONE Si noti come nell'algoritmo che scambia il contenuto di due variabili, in conseguenza del fatto che l'operatore di assegnamento è distruttivo e che una variabile può contenere un solo valore alla volta, sia necessario, per la risoluzione del problema, utilizzare una terza variabile *C*.

ESEMPIO

La tabella di traccia dell'esecuzione dell'algoritmo per il calcolo dell'area del triangolo con valori di base e altezza rispettivamente uguali a 3 e 6 è riportata nella TABELLA **6**.

TABELLA **6**

Istruzione	Base	Altezza	x	Area
Leggi *base,altezza*	3	6		
*x ← base * altezza*	3	6	18	
area ← x/2	3	6	18	9
Scrivi *area*	3	6	18	**9**

ESEMPIO

Di seguito vediamo la tabella di traccia dell'esecuzione dell'algoritmo per lo scambio del contenuto di due variabili con valori iniziali rispettivamente di 2 e 3 (TABELLA **7**).

TABELLA **7**

Istruzione	A	B	C
Leggi *A,B*	2	3	
C ← A	2	3	2
A ← B	3	3	2
B ← C	3	2	2
Scrivi *A,B*	**3**	**2**	2

3.4 Esempi di algoritmi basati sullo schema di «selezione»

Stabilire se il numero *n* è pari o dispari

```
Inizio
    Leggi n
    resto ← n – INT(n/2) * 2
    Se (resto = 0)
        Allora
            Scrivi "pari"
        Altrimenti
            Scrivi "dispari"
Fine
```

Stabilire se il numero *n* è positivo

```
Inizio
    Leggi n
    Se (n > 0)
        Allora
            Scrivi "positivo"
Fine
```

OSSERVAZIONE Esaminando i due esempi precedenti è possibile vedere come esistano due diversi tipi di selezione:

- a due vie: ***Se ... allora ... altrimenti ...***
- a una via: ***Se ... allora ...***

ESEMPIO

La tabella di traccia dell'esecuzione dell'algoritmo per determinare se un numero è pari o dispari nel caso *n* uguale a 5 è quella della TABELLA **8**.

TABELLA **8**

Istruzione	*n*	Resto	Condizione
Leggi *n*	5		
resto ← *n* – ***INT***(*n*/2) * 2	5	1	
(*resto* = 0)	5	1	Falso
Scrivi "*dispari*"	5	1	

ESEMPIO

La TABELLA **9** riporta la tabella di traccia relativa all'esecuzione dell'algoritmo per determinare se un numero è positivo nel caso *n* uguale a –8.

TABELLA **9**

Istruzione	*n*	Condizione
Leggi *n*	8	
(*n* > 0)	8	Falso

Vi sono situazioni che necessariamente richiedono un utilizzo articolato della selezione con la conseguente strutturazione del procedimento risolutivo in problemi e sottoproblemi. A questo proposito, prendiamo in esame il seguente algoritmo che ha lo scopo di determinare il più grande tra tre numeri (come vedremo esistono anche altri modi per approcciare a una corretta soluzione di questo problema, quello proposto di seguito costituisce un classico esempio di composizione di selezioni):

```
Inizio
   Leggi A,B,C
   (1)  Se (A > B)
        (2)  Allora
                Se (A > C)
                   Allora
                      Scrivi A
                   Altrimenti
                      Scrivi C
        (3)  Altrimenti
                Se (B > C)
                   Allora
                      Scrivi B
                   Altrimenti
                      Scrivi C
Fine
```

Nello specifico, le azioni da intraprendere per risolvere il problema sono condizionate da più strutture di selezione «nidificate» una all'interno dell'altra. In casi come questo l'algoritmo può essere pensato come una scomposizione di un problema in sottoproblemi; in particolare, nell'esempio appena visto, può essere individuata la seguente gerarchia di sottoproblemi:

Problema: determinazione del massimo tra tre numeri *A*, *B* e *C*
Sottoproblema 1: determinazione del massimo tra *A* e *B*
Sottoproblema 2: nel caso $A > B$: determinazione del massimo tra *A* e *C*
Sottoproblema 3: nel caso $A <= B$: determinazione del massimo tra *B* e *C*

ESEMPIO

La tabella di traccia dell'esecuzione dell'algoritmo con *A*, *B*, *C* rispettivamente uguali a 2, 5, 3 è riportata nella TABELLA **10**.

TABELLA **10**

Istruzione	A	B	C	Condizione
Leggi *A,B,C*	2	5	3	
(*A* > *B*)	2	5	3	Falso
(*B* > *C*)	2	5	3	Vero
Scrivi *B*	2	**5**	3	

Il DAB dell'algoritmo evidenzia la nidificazione dello schema di selezione (FIGURA **4**).

OSSERVAZIONE Come si può notare entrambe le rappresentazioni (NLS e DAB), anche se corrette, risultano essere di comprensione non immediata, pur trattandosi di soli tre numeri. Applicare questo approccio risolutivo a un numero maggiore di valori porterebbe a situazioni molto complesse da rappresentare. Una soluzione più semplice ed elegante a questo problema è la seguente:

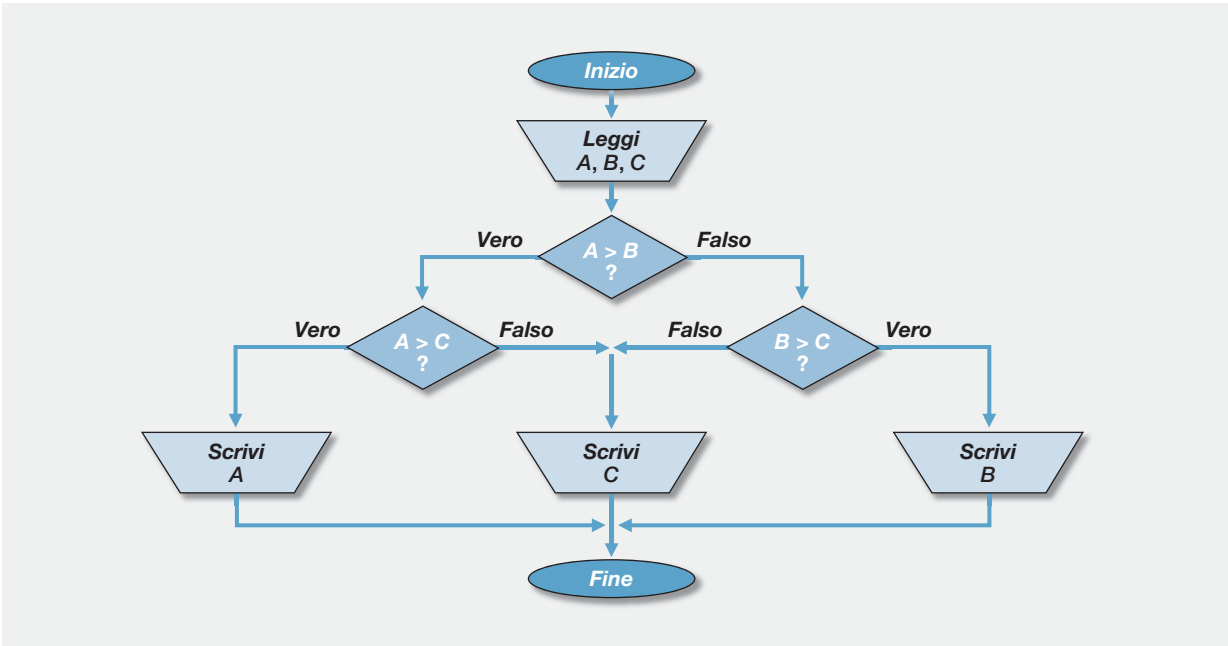


FIGURA **4**

Inizio
  Leggi *A*
    max ← *A*
  Leggi *B*
  Se (*B* > *max*)
    Allora *max* ← *B*
  Leggi *C*
  Se (*C* > *max*)
    Allora *max* ← *C*
  Scrivi *max*
Fine

La variabile *max* alla fine conterrà il valore più grande dei tre: essa viene posta inizialmente uguale ad *A* e il suo valore viene aggiornato se all'atto delle letture di *B* e di *C* si incontra un valore più grande del suo contento corrente.
Nel caso $A = 2, B = 5$ e $C = 3$ la tabella di traccia è quella della TABELLA **11**.

TABELLA **11**

Istruzione	A	B	C	Max	Condizione
Leggi *A*	2				
max ← *A*	2			2	
(*B* > *max*)	2	5	3	3	Vero
max ← *B*	2	5	3	5	
(*C* > *max*)	2	5	3	5	Falso
Scrivi *max*	2	5	3	**5**	

Inoltre questo schema di soluzione può essere facilmente esteso alla determinazione del massimo tra n numeri utilizzando un ciclo:

```
Inizio
   Leggi n
   Leggi A
   max ← A
   n ← n – 1
   Finché (n > 0) esegui
      Inizio
         Leggi A
         Se (A > max)
            Allora max ← A
         n ← n – 1
      Fine
   Scrivi max
Fine
```

La variabile n viene utilizzata per il numero di valori da esaminare e, di conseguenza, per controllare l'esecuzione del ciclo (viene infatti decrementata fino ad arrivare a 0), mentre la sola variabile A conterrà – uno alla volta – i valori da esaminare; alla fine nella variabile *max* si otterrà il valore maggiore. Si noti che la lettura del primo valore avviene fuori dal ciclo, in modo da inizializzare con questo valore la variabile *max*.

3.5 Esempi di algoritmi basati sullo schema di «ripetizione»

Redazione di un elenco di libri (1)

```
Inizio
   Finché (ci sono libri da esaminare)
      esegui
         Inizio
            Prendi il libro
            Trascrivi il titolo
         Fine
Fine
```

Redazione di un elenco di libri (2)

```
Inizio
   Esegui
      Inizio
         Prendi il libro
         Trascrivi il titolo
      Fine
   Finché (ci sono libri da esaminare)
Fine
```

I due esempi riportati sono chiaramente equivalenti; infatti in ambedue i casi si richiede all'esecutore di esaminare un libro alla volta e di trascriverne il titolo. Schematicamente le due situazioni sono così:

Ciclo con controllo in «testa»

Ciclo con controllo in «coda»

In entrambi i casi l'esecuzione delle operazioni è ripetuta se la condizione è **vera** e il ciclo si arresta se la condizione è **falsa**.

OSSERVAZIONE Nel secondo caso la condizione di terminazione viene valutata alla fine delle istruzioni del ciclo: questo fatto comporta che *almeno una volta* le stesse sono eseguite. Al contrario, nel primo caso la condizione è valutata prima delle istruzioni del ciclo, pertanto se questa risulta immediatamente falsa l'intero ciclo non viene eseguito *nemmeno una volta*.

ESEMPIO

Per meglio capire le due distinte tecniche di iterazione e come, in situazioni specifiche, possano condurre a risultati diversi, prendiamo in esame il seguente problema. Dato un numero *N*, determinare quante volte occorre sommare a *N* il valore 2 per ottenere un valore superiore a 100. L'algoritmo risolutivo può essere realizzato con i due tipi di ciclo seguenti (in entrambi gli esempi la variabile *C* ha la funzione di contatore del numero di ripetizioni del ciclo e rappresenta il risultato dell'algoritmo):

Algoritmo 1: controllo in «testa»

```
Inizio
    C ← 0
    Leggi N
    Finché (N <= 100) esegui
        Inizio
            N ← N + 2
            C ← C + 1
        Fine
    Scrivi C
Fine
```

Algoritmo 2: controllo in «coda»

```
Inizio
    C ← 0
    Leggi N
    Esegui
        Inizio
            N ← N + 2
            C ← C + 1
        Fine
    Finché (N <= 100)
    Scrivi C
Fine
```

Valutiamo le due seguenti possibilità:

- il valore iniziale *N* è minore di 100: le due soluzioni sono equivalenti;
- il valore iniziale *N* è maggiore di 100: con il primo algoritmo *C* assume il valore 1 (errato), con il secondo il valore 0 (corretto).

A conferma dell'affermazione fatta riportiamo le tabelle di traccia nei due casi per *N* uguale a 99 (TABELLE **12** e **13**) e per *N* uguale a 101 (TABELLE **14** e **15**).

TABELLA **12**

Algoritmo 1: controllo in «testa»			
Istruzione	***N***	***C***	**Condizione**
C ← 0		0	
Leggi *N*	99	0	
(*N* <= 100)	99	0	Vero
N ← *N* + 2	101	0	
C ← *C* + 1	101	1	
(*N* <= 100)	101	1	Falso
Scrivi *C*	101	**1**	

TABELLA **13**

Algoritmo 2: controllo in «coda»			
Istruzione	***N***	***C***	**Condizione**
C ← 0		0	
Leggi *N*	99	0	
N ← *N* + 2	101	0	
C ← *C* + 1	101	1	
(*N* <= 100)	101	1	Falso
Scrivi *C*	101	**1**	

▶

TABELLA **14**

Algoritmo 1: controllo in «testa»			
Istruzione	***N***	***C***	**Condizione**
C ← 0		0	
Leggi *N*	101	0	
(*N* <= 100)	101	0	Falso
Scrivi *C*	101	**0**	

TABELLA **15**

Algoritmo 2: controllo in «coda»			
Istruzione	***N***	***C***	**Condizione**
C ← 0		0	
Leggi *N*	101	0	
N ← *N* + 2	103	0	
C ← *C* + 1	103	1	
(*N* <= 100)	103	1	Falso
Scrivi *C*	103	**1**	

Torniamo all'esempio introduttivo relativo alla redazione di un elenco di libri. Entrambi i tipi di ciclo utilizzati per la risoluzione sono **indeterminati**, nel senso che l'esecutore non è in grado di stabilire a priori quante volte dovrà iterare il ciclo, in quanto esso deve essere ripetuto fintantoché vi sono libri da esaminare. Disponendo come ulteriore dato di ingresso del numero preciso dei libri da esaminare – per esempio 100 – il problema potrebbe essere risolto come segue:

```
Inizio
   Per 100 volte esegui
      Inizio
         Prendi il libro
         Trascrivi il titolo
      Fine
Fine
```

In questo caso in cui l'esecutore conosce a priori il numero di volte che dovrà ripetere le operazioni del ciclo si tratta di un ciclo **determinato**.

OSSERVAZIONE

Effettuare spese in giro per negozi

```
Inizio
   Finché hai soldi esegui
      Inizio
         Entra in un negozio
         Compra un oggetto
         Paga
      Fine
Fine
```

Comprare 10 oggetti in 10 negozi

```
Inizio
   Per 10 volte esegui
      Inizio
         Entra in un negozio
         Compra un oggetto
         Paga
      Fine
Fine
```

Nel caso di un ciclo determinato si conosce precisamente il momento in cui la ripetizione avrà termine, mentre nel caso di un ciclo indeterminato questo momento non è noto, poiché la terminazione dipende dal verificarsi o meno della condizione che ne controlla l'arresto; come caso limite, se una condizione di fine ciclo non si verifica mai, esso viene ripetuto infinite volte senza terminare.

ESEMPIO

```
Inizio
   n ← 1
   Finché (n > 0) esegui
      Inizio
         n ← n + 1
      Fine
Fine
```

In questo caso la condizione specificata per il controllo del ciclo è sempre verificata, in quanto *n* inizialmente viene posto uguale a 1 e nel corso del ciclo viene sempre incrementato di una unità; di conseguenza il valore di *n* risulta sempre maggiore di 0 e pertanto il ciclo non terminerà mai!

OSSERVAZIONE

Perché un **ciclo indeterminato** sia ben costruito è necessario che le istruzioni eseguite all'interno del ciclo stesso garantiscano che la condizione di controllo divenga falsa causandone la terminazione.
Nel caso di un **ciclo determinato** la terminazione è garantita dalla struttura stessa del ciclo.

ESEMPIO

```
Inizio
   n ← 1
   Finché (n < 100) esegui
      Inizio
         n ← n + 1
      Fine
Fine
```

Il ciclo termina dopo essere stato eseguito esattamente 99 volte. Infatti dopo aver aggiunto 99 volte un'unità al valore della variabile *n*, inizialmente uguale a 1, questa assume il valore 100 e la condizione «*n* < 100» risulta falsa, causando la terminazione del ciclo.

Infine il seguente algoritmo rappresentato in NLS determina la somma di *n* numeri, ciascuno fornito come valore di input:

```
Inizio
   Leggi n
   somma ← 0
   Finché (n > 0) esegui
      Inizio
         Leggi numero
         somma ← somma + numero
         n ← n – 1
      Fine
   Scrivi somma
Fine
```

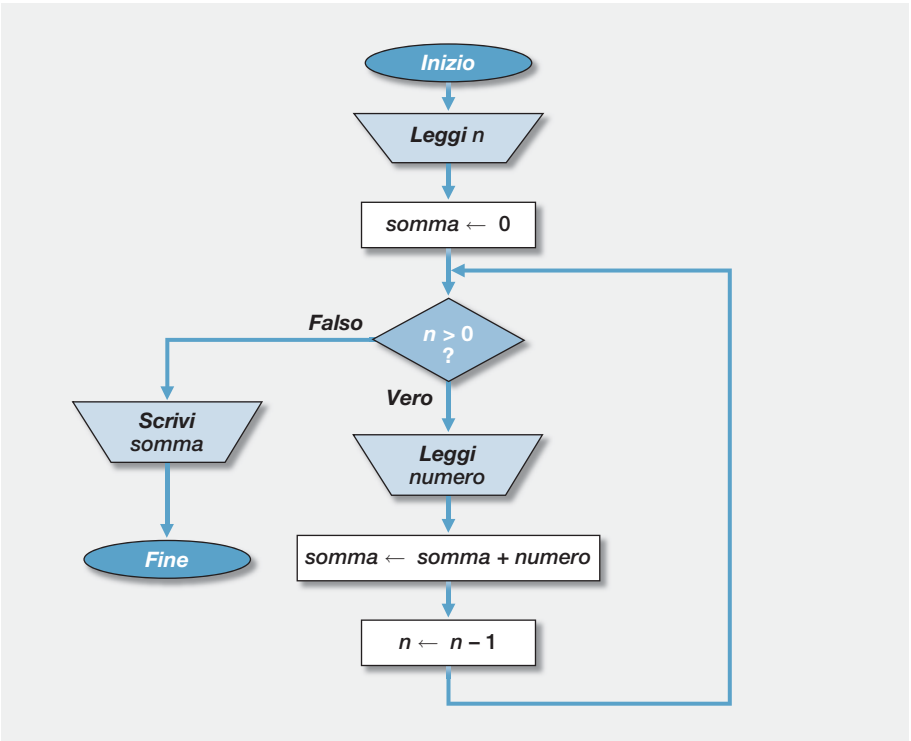


FIGURA **5**

Il DAB corrispondente è quello di FIGURA **5**, mentre la tabella di traccia per i tre valori 8, 4 e 7 è la TABELLA **16**.

TABELLA **16**

Istruzione	*n*	Numero	Somma	Condizione
***Leggi** n*	3			
somma ← 0	3		0	
(n > 0)	3		0	Vero
***Leggi** numero*	3	8	0	
somma ← somma + numero	3	8	8	
n ← n – 1	2	8	8	
(n > 0)	2	8	8	Vero
***Leggi** numero*	2	4	8	
somma ← somma + numero	2	4	12	
n ← n – 1	1	4	12	
(n > 0)	1	4	12	Vero
***Leggi** numero*	1	7	12	
somma ← somma + numero	1	7	19	
n ← n – 1	0	7	19	
(n > 0)	0	7	19	Falso
***Scrivi** somma*	0	7	**19**	

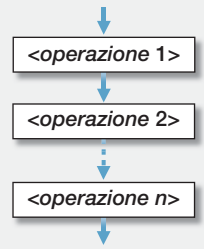
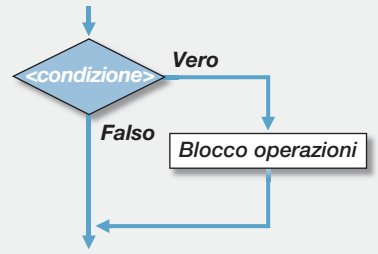
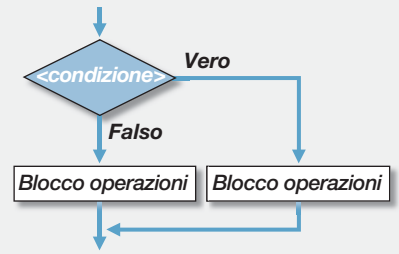
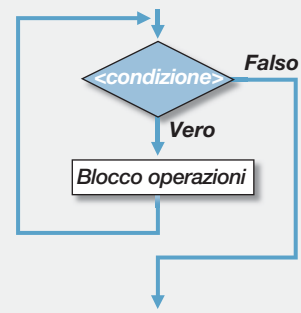
OSSERVAZIONE Esaminando in dettaglio il precedente algoritmo, analizziamo l'uso che viene fatto delle variabili *n*, *numero* e *somma*:

- *n*: indica quanti numeri dovranno essere elaborati e viene utilizzata per controllare l'esecuzione del ciclo; viene pertanto utilizzata come **contatore** (in questo caso «a scalare»[1]);
- *numero*: è la variabile utilizzata per la lettura dei singoli numeri da sommare (uno alla volta);
- *somma*: è la variabile che alla fine del procedimento conterrà il risultato; è posta inizialmente uguale a 0 e successivamente sommata ripetutamente al valore *numero*; variabili utilizzate in questo modo sono definite **accumulatori**.

1. Un uso alternativo, ma altrettanto classico, della variabile contatore consiste nell'incrementarne il valore di una unità a ogni iterazione.

La TABELLA **17** riassume le corrispondenze significative tra NLS e DAB.

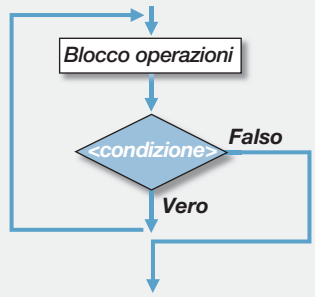
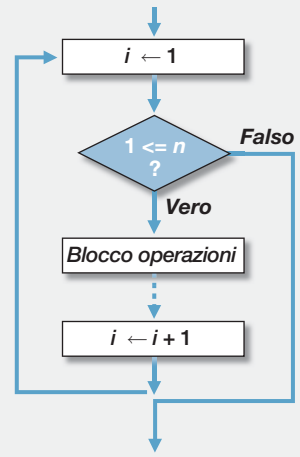
TABELLA **17**

Schema	NLS	DAB
Sequenza	*<operazione 1>* *<operazione 2>* … *<operazione n>*	
Selezione a una via	***Se*** *<condizione>* ***allora …***	
Selezione a due vie	***Se*** *<condizione>* ***allora …*** ***altrimenti …***	
Ciclo indeterminato con controllo in testa	***Finché*** *<condizione>* ***esegui*** ***Inizio*** ***…*** ***Fine***	

Algoritmi e fattore umano

Nella risoluzione di un problema ciò che fa la differenza è la qualità della strategia risolutiva adottata e la sua descrizione algoritmica.
Ovviamente questa attività è riservata all'uomo, anche se una branca dell'informatica è dedicata all'intelligenza artificiale, che tenta di programmare i computer per attività di questo tipo.
Allo stato attuale nella definizione degli algoritmi l'attività umana riveste ancora un ruolo centrale: è quindi il prodotto dell'attività umana a essere eseguito dalle macchine in modo automatico.
Al di là della conoscenza più o meno approfondita di tecniche e di strumenti utilizzabili per la progettazione e l'implementazione degli algoritmi, è la capacità e la competenza a fare la differenza.

▶ TABELLA **17**

Schema	NLS	DAB
Ciclo indeterminato con controllo in coda	***Esegui*** ***Inizio*** ***...*** ***Fine*** ***Finché*** *<condizione>*	
Ciclo determinato	***Per** n **volte esegui*** ***Inizio*** ***...*** ***Fine***	

OSSERVAZIONE Nella versione DAB del ciclo determinato viene usata la variabile contatore «*i*» per contare il numero delle iterazioni; nella versione NLS questa variabile è implicita nel costrutto.

4 Analisi di problemi e sintesi di algoritmi

La professione di informatico richiede spesso l'ideazione e la verifica di un algoritmo che risolva con un certo grado di genericità un dato problema: è un'attività che si apprende e si perfeziona con il tempo e con l'esperienza, commettendo errori e correggendo gli errori commessi.

OSSERVAZIONE L'attività di analisi di un problema e di sintesi di un algoritmo può essere rappresentata mediante un diagramma di flusso come quello di FIGURA **6**.
Si tratta di un tipico procedimento iterativo per «tentativi ed errori», in cui la verifica assume il ruolo cruciale di validazione dell'algoritmo e che richiede normalmente, anche a informatici esperti, vari passaggi di perfezionamento della prima soluzione.

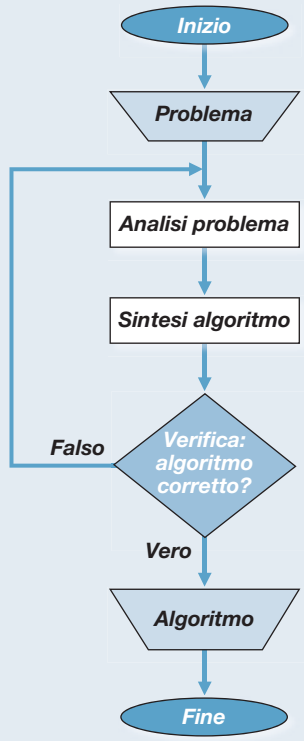


FIGURA 6

Nel seguito sono presentati alcuni esempi di analisi di problemi e di sintesi e verifica dei relativi algoritmi risolutivi; ma ricordiamo che la competenza si costruisce solo cimentandosi con gli esercizi proposti.

4.1 Problema 1: determinazione della data della Pasqua

La TABELLA **18** descrive un algoritmo per la determinazione della data della Pasqua (giorno p del mese n) a partire dall'anno x. Per esempio, utilizzando la tabella per determinare la Pasqua dell'anno 2010, risulta $p = 4$ e $n = 4$, cioè la data del 4 aprile (TABELLA **19**).
Il calcolo manuale permette di verificare che non ci sono selezioni o iterazioni, per cui l'algoritmo risulta essere una semplice successione di assegnamenti:

Inizio
  Leggi X
    $B \leftarrow$ ***Int***$(X/100)$
    $C \leftarrow X -$ ***Int***$(X : 100) * 100$
    $A \leftarrow (5 * B + C) -$ ***Int***$((5 * B + C) : 19) * 19$
    $R \leftarrow$ ***Int***$((3 * B + 75)/4)$
    $S \leftarrow (3 * B + 75) -$ ***Int***$((3 * B + 75)/4) * 4$
    $T \leftarrow$ ***Int***$((8 * B + 88)/25)$
    $H \leftarrow (19 * A + R - T) -$ ***Int***$((19 * A + R - T)/30) * 30$ ▶

$G \leftarrow$ ***Int***$((A + 11 * H)/319)$
$J \leftarrow$ ***Int***$((300 - 60 * S + C)/4)$
$K \leftarrow (300 - 60 * S + C) -$ ***Int***$((300 - 60 * S + C)/4) * 4$
$M \leftarrow (2 * J - K - H + G) -$ ***Int***$((2 * J - K - H + G)/7) * 7$
$N \leftarrow$ ***Int***$((H - G + M + 110)/30)$
$Q \leftarrow (H - G + M + 110) -$ ***Int***$((H - G + M + 110)/30) * 30$
$P \leftarrow (Q + 5 - N) -$ ***Int***$((Q + 5 - N)/32) * 32$
Scrivi P, N
Fine

TABELLA **18**

Passo	Numero da...	...dividere per	Risultato	
			Quoziente intero	Resto
1	x (anno)	100	b	c
2	$5b + c$	19	–	a
3	$3(b + 25)$	4	r	s
4	$8(b + 11)$	25	t	–
5	$19a + r - t$	30	–	h
6	$a + 11h$	319	g	–
7	$60(5 - s) + c$	4	j	k
8	$2j - k - h + g$	7	–	m
9	$h - g + m + 110$	30	n	q
10	$q + 5 - n$	32	–	p

TABELLA **19**

Passo	Numero da...	...dividere per	Risultato	
			Quoziente intero	Resto
1	$x = 2010$	100	$b = 20$	$c = 10$
2	$5b + c = 110$	19	–	$a = 15$
3	$3(b + 25) = 135$	4	$r = 33$	$s = 3$
4	$8(b + 11) = 248$	25	$t = 5$	–
5	$19a + r - t = 313$	30	–	$h = 13$
6	$a + 11h = 158$	319	$g = 0$	–
7	$60(5 - s) + c = 130$	4	$j = 32$	$k = 2$
8	$2j - k - h + g = 49$	7	–	$m = 0$
9	$h - g + m + 110 = 123$	30	$n = 4$	$q = 3$
10	$q + 5 - n = 4$	32	–	$p = 4$

L'algoritmo deve essere verificato mediante una tabella di traccia (scegliamo nuovamente l'anno 2010 come dato) (TABELLA **20**).

TABELLA **20**

Istruzione	X	B	C	A	R	S	T	H	G	J	K	M	N	Q	P
Leggi *X*	2010														
B ← ***Int***(*X*/100)	2010	20													
C ← *X* – ***Int***(*X* : 100) * 100	2010	20	10												
A ← (5 * *B* + *C*) – ***Int***((5 * *B* + *C*) : 19) * 19	2010	20	10	15											
R ← ***Int***((3 * *B* + 75)/4)	2010	20	10	15	33										
S ← (3 * *B* + 75) – ***Int***((3 * *B* + 75)/4) * 4	2010	20	10	15	33	3									
T ← ***Int***((8 * *B* + 88)/25)	2010	20	10	15	33	3	5								
H ← (19 * *A* + *R* – *T*) – ***Int***((19 * *A* + *R* – *T*)/30) * 30	2010	20	10	15	33	3	5	13							
G ← ***Int***((*A* + 11 * *H*)/319)	2010	20	10	15	33	3	5	13	0						
J ← ***Int***((300 – 60 * *S* + *C*)/4)	2010	20	10	15	33	3	5	13	0	32					
K ← (300 – 60 * *S* + *C*) – ***Int***((300 – 60 * *S* + *C*)/4) * 4	2010	20	10	15	33	3	5	13	0	32	2				
M ← (2 * *J* – *K* – *H* + *G*) – ***Int***((2 * *J* – *K* – *H* + *G*)/7) * 7	2010	20	10	15	33	3	5	13	0	32	2	0			
N ← ***Int***((*H* – *G* + *M* + 110)/30)	2010	20	10	15	33	3	5	13	0	32	2	0	4		
Q ← (*H* – *G* + *M* + 110) – ***Int***((*H* – *G* + *M* + 110)/30) * 30	2010	20	10	15	33	3	5	13	0	32	2	0	4	3	
P ← (*Q* + 5 – *N*) – ***Int***((*Q* + 5 – *N*)/32) * 32	2010	20	10	15	33	3	5	13	0	32	2	0	4	3	4
Scrivi *P*, *N*	2010	20	10	15	33	3	5	13	0	32	2	0	4	3	4

OSSERVAZIONE Una singola tabella di traccia corrispondente a un solo dato non è normalmente sufficiente per stabilire la correttezza di un algoritmo: è buona norma realizzare più tabelle di traccia relative a dati diversi.

4.2 Problema 2: calcolo del costo di una spedizione

Un'agenzia di spedizioni computa il costo relativo al trasporto di un pacco in funzione del peso, della distanza da percorrere e dell'urgenza («normale» o «urgente»), secondo lo schema riportato nella TABELLA **21**.

TABELLA **21**

Classe	0-100 km	100-500 km	Oltre 500 km
NORMALE	1 €/kg	1,5 €/kg	2 €/kg
URGENTE	1,5 €/kg	2 €/kg	3 €/kg

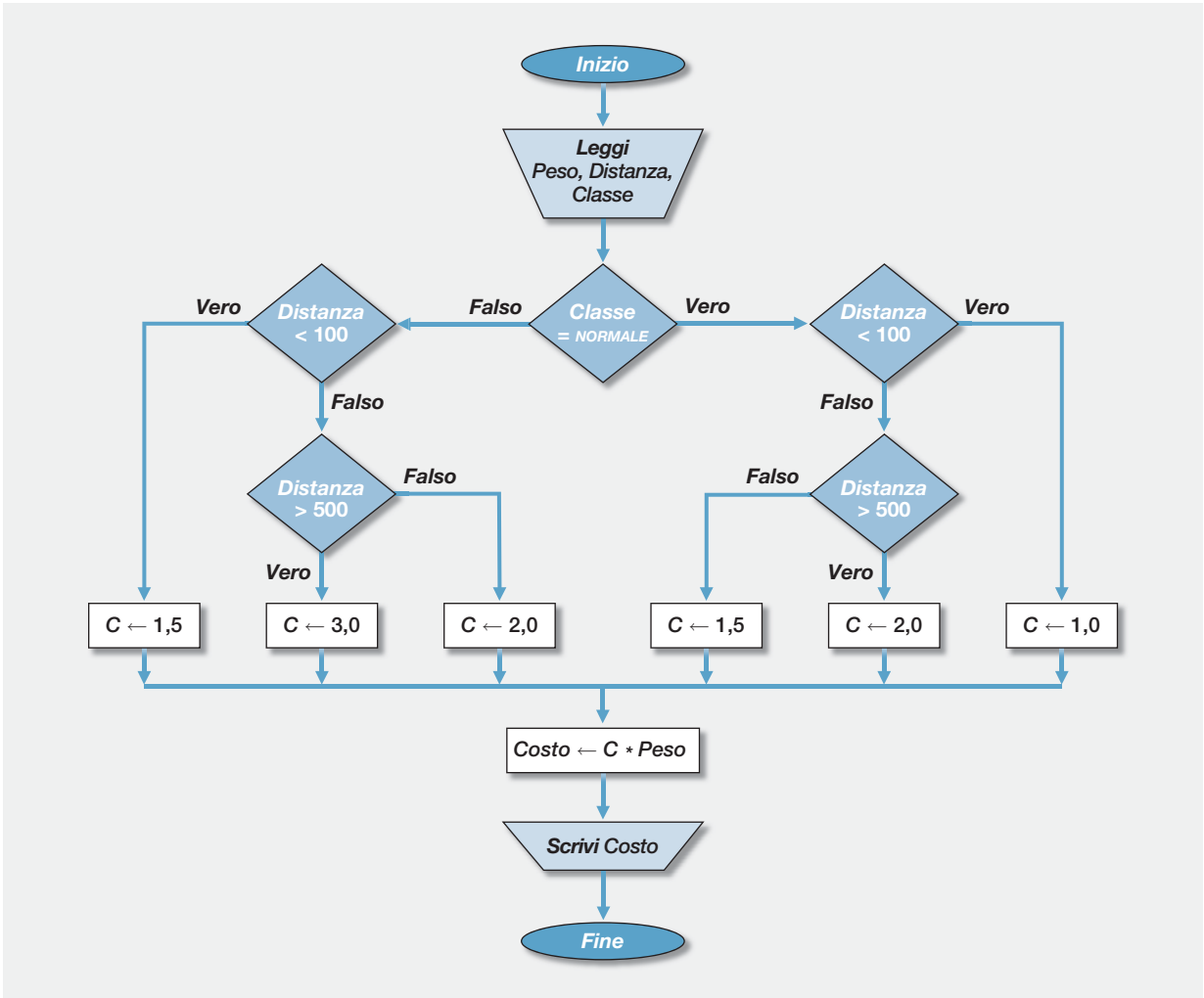


FIGURA **7**

Anche in questo caso iniziamo da un'analisi del problema calcolando alcuni costi:

- pacco «normale» di 50 kg con distanza 200 km: $1{,}5 \times 50 = 75$ €;
- pacco «urgente» di 10 kg con distanza 800 km: $3 \times 10 = 30$ €.

È subito evidente come l'algoritmo debba «selezionare» nella tabella il corretto coefficiente per lo svolgimento del calcolo, per cui un algoritmo risolutivo potrebbe essere quello di FIGURA **7**.
Per verificare l'algoritmo compiliamo le tabelle di traccia dei due esempi calcolati in precedenza (TABELLE **22** e **23**).

TABELLA **22**

Blocco	Peso	Distanza	Classe	C	Costo	Condizione
Leggi *Peso*, *Distanza*, *Classe*	50	200	NORMALE			
(*Classe* = NORMALE)	50	200	NORMALE			Vero
(*Distanza* < 100)	50	200	NORMALE			Falso ▶

▶ TABELLA **22**

Blocco	Peso	Distanza	Classe	C	Costo	Condizione
(*Distanza* > 500)	50	200	NORMALE			Falso
C ← 1,5	50	200	NORMALE	1,5		
Costo ← *C* * *Peso*	50	200	NORMALE	1,5	75	
Scrivi *Costo*	50	200	NORMALE	1,5	75	

TABELLA **23**

Blocco	Peso	Distanza	Classe	C	Costo	Condizione
Leggi *Peso*, *Distanza*, *Classe*	10	800	URGENTE			
(*Classe* = NORMALE)	10	800	URGENTE			Falso
(*Distanza* < 100)	10	800	URGENTE			Falso
(*Distanza* > 500)	10	800	URGENTE			Vero
C ← 3,0	10	800	URGENTE	3,0		
Costo ← *C* * *Peso*	10	800	URGENTE	3,0	30	
Scrivi *Costo*	10	800	URGENTE	3,0	30	

4.3 Problema 3: rimbalzi di una pallina di gomma

Una pallina di gomma lasciata cadere rimbalza ogni volta fino a un'altezza che è l'80% dell'altezza da cui cade (la prima volta è l'altezza iniziale, le volte successive è l'altezza del rimbalzo precedente). Lasciando cadere la pallina da 1 m i rimbalzi successivi raggiungeranno le seguenti altezze da terra:

1° rimbalzo: $1 \times 0{,}8 = 0{,}8$ m
2° rimbalzo: $0{,}8 \times 0{,}8 = 0{,}64$ m
3° rimbalzo: $0{,}64 \times 0{,}8 = 0{,}512$ m
4° rimbalzo: $0{,}512 \times 0{,}8 = 0{,}4096$ m

Volendo determinare mediante un algoritmo l'altezza dell'*n*-esimo rimbalzo, è intuitivo che dobbiamo «ripetere» il calcolo per *N* volte (con *N* numero dei rimbalzi) come nel diagramma di flusso di FIGURA **8**.
Verifichiamo l'algoritmo compiliamo la tabella di traccia relativa all'esempio calcolato in precedenza (TABELLA **24**).

TABELLA **24**

Blocco	Altezza	*N*	Condizione
Leggi *altezza*, *N*	1	4	
altezza ← *altezza* * 0,8	0,8	4	
N ← *N* – 1	0,8	3	
(*N* > 0)	0,8	3	Vero ▶

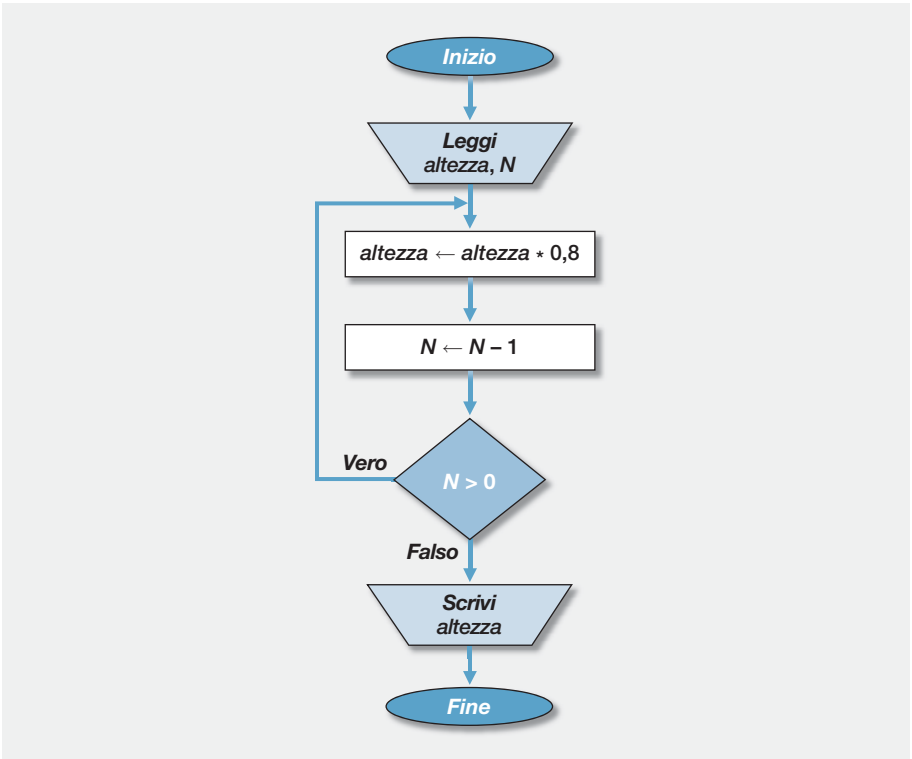


FIGURA **8**

▶ TABELLA **24**

Blocco	Altezza	N	Condizione
altezza ← *altezza* * 0,8	0,64	3	
N ← *N* – 1	0,64	2	
(*N* > 0)	0,64	2	Vero
altezza ← *altezza* * 0,8	0,512	2	
N ← *N* – 1	0,512	1	
(*N* > 0)	0,512	1	Vero
altezza ← *altezza* * 0,8	0,4096	1	
N ← *N* – 1	0,4096	0	
(*N* > 0)	0,4096	0	Falso
Scrivi *altezza*	0,4096	0	

OSSERVAZIONE L'uso di una variabile – in questo caso *N* – successivamente decrementata da un valore iniziale fino a 0 per contare il numero di volte che, mediante un ciclo, si ripete un blocco di istruzioni è una tipica tecnica di costruzione degli algoritmi iterativi.

OSSERVAZIONE Il semplice spostamento del blocco di uscita (corrispondente all'istruzione ***Scrivi***) all'interno del ciclo consente di avere un algoritmo che visualizza le altezze dei rimbalzi successivi:

```
Inizio
   Leggi altezza, N
   Esegui
      Inizio
         altezza ← altezza * 0,8
         Scrivi altezza
         N ← N – 1
      Fine
   Finché (n > 0)
Fine
```

Sappiamo che la pallina si ferma quando effettua un rimbalzo la cui altezza è inferiore a 1 cm (0,01 m). Il seguente algoritmo è una variante del precedente e determina il numero di rimbalzi che la pallina effettua quando viene lasciata cadere da una certa altezza iniziale:

```
Inizio
   Leggi altezza
   N = 0
   Esegui
      Inizio
         altezza ← altezza * 0,8
         N ← N + 1
      Fine
   Finché (altezza > 0,01)
   Scrivi N
Fine
```

OSSERVAZIONE Anche in questo caso la variabile N – inizializzata a 0 e successivamente incrementata per ogni ripetizione del ciclo di istruzioni – è utilizzata con una modalità tipica, come «contatore».

Verifichiamo questa variante dell'algoritmo mediante una tabella di traccia per l'altezza iniziale di 5 cm (0,05 m) (TABELLA **25**).

TABELLA **25**

Istruzione	Altezza	N	Condizione
Leggi *altezza*	0,05		
$N \leftarrow 0$	0,05	0	
altezza ← *altezza* * 0,8	0,04	0	
$N \leftarrow N + 1$	0,04	1	
(*altezza* > 0,01)	0,04	1	Vero
altezza ← *altezza* * 0,8	0,032	1	
$N \leftarrow N + 1$	0,032	2	▶

▶ TABELLA **25**

Istruzione	Altezza	*N*	Condizione
(*altezza* > 0,01)	0,032	2	Vero
altezza ← *altezza* * 0,8	0,0256	2	
N ← *N* + 1	0,0256	3	
(*altezza* > 0,01)	0,0256	3	Vero
altezza ← *altezza* * 0,8	0,02048	3	
N ← *N* + 1	0,02048	4	
(*altezza* > 0,01)	0,02048	4	Vero
altezza ← *altezza* * 0,8	0,016384	4	
N ← *N* + 1	0,016384	5	
(*altezza* > 0,01)	0,016384	5	Vero
altezza ← *altezza* * 0,8	0,0131072	5	
N ← *N* + 1	0,0131072	6	
(*altezza* > 0,01)	0,0131072	6	Vero
altezza ← *altezza* * 0,8	0,01048576	6	
N ← *N* + 1	0,01048576	7	
(*altezza* > 0,01)	0,01048576	7	Vero
altezza ← *altezza* * 0,8	0,008388608	7	
N ← *N* + 1	0,008388608	8	
(*altezza* > 0,01)	0,008388608	8	Falso
Scrivi *N*	0,008388608	8	

4.4 Problema 4: sequenza dei numeri di Fibonacci

Un qualsiasi numero di Fibonacci successivo al secondo è uguale alla somma dei due numeri di Fibonacci che lo precedono nella sequenza (il primo numero di Fibonacci è 1, il secondo numero di Fibonacci è 1). È molto semplice costruire la sequenza dei primi numeri di Fibonacci seguendo questa regola:

1) 1
2) 1
3) 2 (1 + 1)
4) 3 (2 + 1)
5) 5 (3 + 2)
6) 8 (5 + 3)
7) 13 (8 + 5)
8) 21 (13 + 8)
9) 34 (21 + 13)
10) 55 (34 + 21)

Anche in questo caso è evidente che il calcolo segue uno schema ripetitivo che in un algoritmo, per determinare l'*n*-esimo numero di Fibonacci, dovrà essere reso mediante un ciclo; in questo caso però dobbiamo utilizzare due variabili per «ricordare» i due precedenti valori della sequenza mano a mano che si procede nella sua costruzione:

```
Inizio
   Leggi N
   F1 ← 1
   F2 ← 1
   Esegui
      Inizio
         F ← F1 + F2
         N ← N – 1
         F2 ← F1
         F1 ← F
      Fine
   Finché (N > 0)
   Scrivi F
Fine
```

Verifichiamo mediante una tabella di traccia il funzionamento dell'algoritmo per $N = 5$ (TABELLA **26**).

TABELLA **26**

Istruzione	F1	F2	F	N	Condizione
Leggi N				5	
$F1 \leftarrow 1$	1			5	
$F2 \leftarrow 1$	1	1		5	
$F \leftarrow F1 + F2$	1	1	2	5	
$N \leftarrow N - 1$	1	1	2	4	
$F2 \leftarrow F1$	1	1	2	4	
$F1 \leftarrow F$	2	1	2	4	
$(N > 0)$	2	1	2	4	Vero
$F \leftarrow F1 + F2$	2	1	3	4	
$N \leftarrow N - 1$	2	1	3	3	
$F2 \leftarrow F1$	2	2	3	3	
$F1 \leftarrow F$	3	2	3	3	
$(N > 0)$	3	2	3	3	Vero
$F \leftarrow F1 + F2$	3	2	5	3	
$N \leftarrow N - 1$	3	2	5	2	
$F2 \leftarrow F1$	3	3	5	2	

▶ TABELLA **26**

Istruzione	*F1*	*F2*	*F*	*N*	Condizione
$F1 \leftarrow F$	5	3	5	2	
$(N > 0)$	5	3	5	2	Vero
$F \leftarrow F1 + F2$	5	3	8	2	
$N \leftarrow N - 1$	5	3	8	1	
$F2 \leftarrow F1$	5	5	8	1	
$F1 \leftarrow F$	8	5	8	1	
$(N > 0)$	8	5	8	1	Vero
$F \leftarrow F1 + F2$	8	5	13	1	
$N \leftarrow N - 1$	8	5	13	0	
$F2 \leftarrow F1$	8	8	13	0	
$F1 \leftarrow F$	13	8	13	0	
$(N > 0)$	13	8	13	0	Falso
Scrivi *F*	13	8	13	0	

Ora il quinto numero della sequenza di Fibonacci che abbiamo calcolato in precedenza è 5 e non 13, per cui l'algoritmo non è corretto, anche se è evidente dalla tabella di traccia che calcola effettivamente i numeri della sequenza sommando i due precedenti: ciò che è errato è che non abbiamo considerato che i primi due numeri della sequenza non devono essere calcolati, per cui la variabile *N* deve essere inizialmente decrementata di 2:

```
Inizio
   Leggi N
      F1 ← 1
      F2 ← 1
      N ← N – 2
      Esegui
         Inizio
            F ← F1 + F2
            N ← N – 1
            F2 ← F1
            F1 ← F
         Fine
      Finché (N > 0)
   Scrivi F
Fine
```

La tabella di traccia riportata nella TABELLA **27** dimostra che la correzione apportata rende l'algoritmo corretto.

TABELLA **27**

Istruzione	*F1*	*F2*	*F*	*N*	Condizione
Leggi *N*				5	
F1 ← 1	1			5	
F2 ← 1	1	1		5	
N ← *N* – 2	1	1		3	
F ← *F1* + *F2*	1	1	2	3	
N ← *N* – 1	1	1	2	2	
F2 ← *F1*	1	1	2	2	
F1 ← *F*	2	1	2	2	
(*N* > 0)	2	1	2	2	Vero
F ← *F1* + *F2*	2	1	3	2	
N ← *N* – 1	2	1	3	1	
F2 ← *F1*	2	2	3	1	
F1 ← *F*	3	2	3	1	
(*N* > 0)	3	2	3	1	Vero
F ← *F1* + *F2*	3	2	5	1	
N ← *N* – 1	3	2	5	0	
F2 ← *F1*	3	3	5	0	
F1 ← *F*	5	3	5	0	
(*N* > 0)	5	3	5	0	Falso
Scrivi *F*	5	3	5	0	

OSSERVAZIONE Anche in questo caso un diverso posizionamento delle istruzioni ***Scrivi*** consente di produrre come risultato dell'algoritmo la sequenza dei numeri di Fibonacci anziché solo l'*n*-esimo numero:

```
Inizio
   Leggi N
   F1 ← 1
   Scrivi F1
   F2 ← 1
   Scrivi F2
   N ← N – 2
   Esegui
      Inizio
         F ← F1 + F2
         Scrivi F
         N ← N – 1
         F2 ← F1
         F1 ← F
      Fine
   Finché (N > 0)
Fine
```

5 Un esempio di algoritmo numerico: il calcolo della radice quadrata

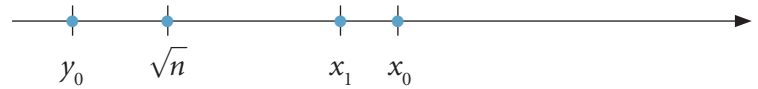
Media geometrica e media aritmetica

La **media geometrica** di due numeri è definita come la radice quadrata del loro prodotto.
La **media aritmetica** di due numeri è invece la metà della loro somma.

Esiste un algoritmo relativamente semplice – attribuito al padre della Fisica e della Matematica moderne, lo scienziato inglese Isac Newton – anche per il calcolo approssimato della radice quadrata di un numero positivo n.
Esso ha inizio con la scelta casuale di un numero positivo x_0 da considerarsi una prima approssimazione del valore $\sqrt{n}$ (il numero x_0 è comunque scelto in modo che risulti maggiore di $\sqrt{n}$: a questo scopo è sufficiente che il suo quadrato sia $x_0^2 > n$; quindi il numero x_0 costituisce un valore approssimato per eccesso di $\sqrt{n}$).
Calcolando $y_0 = \frac{n}{x_0}$ si ha che $x_0 \cdot y_0 = n$; il valore $\sqrt{n}$ che intendiamo approssimare è quindi la media geometrica di x_0 e y_0, e y_0 è un valore approssimato per difetto di $\sqrt{n}$.
La media aritmetica di x_0 e y_0 è:

$$x_1 = \frac{x_0 + y_0}{2} = \frac{1}{2}\left(x_0 + \frac{n}{x_0}\right)$$

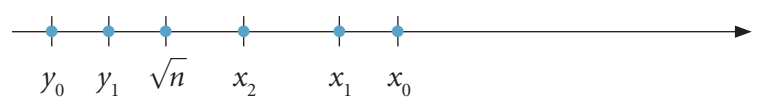
Ricordando che la media aritmetica fra due numeri è maggiore o uguale alla loro media geometrica[2], x_1 risulterà compreso fra $\sqrt{n}$ e x_0, quindi x_1 sarà un nuovo valore approssimato per eccesso di $\sqrt{n}$ e sarà un'approssimazione migliore, cioè più prossima a $\sqrt{n}$ di quanto non fosse x_0 (infatti $x_0 > x_1 > \sqrt{n}$):



Determinando $y_1 = \frac{n}{x_1}$, ed essendo $x_1 > \sqrt{n}$, si ha $x_1 \cdot y_1 = n$, e $\sqrt{n}$ è la media geometrica di x_1 e y_1; y_1 è quindi un nuovo valore approssimato per difetto di $\sqrt{n}$. Inoltre, essendo $x_0 < x_1$, si ha che $\frac{n}{x_1} > \frac{n}{x_0}$, cioè $y_1 > y_0$; anche y_1 è un valore approssimato per difetto di $\sqrt{n}$ e rappresenta un'approssimazione migliore rispetto a y_0.
Ripetendo i passaggi svolti in precedenza si calcola la media aritmetica di x_1 e y_1:

$$x_2 = \frac{x_1 + y_1}{2} = \frac{1}{2}\left(x_1 + \frac{n}{x_1}\right)$$

Essa rappresenta un nuovo valore approssimato per eccesso di $\sqrt{n}$ ed è una approssimazione migliore, più prossima a $\sqrt{n}$ di quanto non fosse x_1:



2. Infatti, supponendo $a, b \geq 0$, si ha:

$\frac{1}{2}(a+b) \geq \sqrt{ab}$

$\frac{1}{4}(a+b)^2 \geq ab$

$a^2 + b^2 + 2ab \geq 4ab$

$a^2 + b^2 - 2ab \geq 0$

$(a-b)^2 \geq 0$

che è vera per ogni valore di a e b.

Calcolando $y_2 = \frac{n}{x_2}$ si ottiene un nuovo valore approssimato per eccesso di $\sqrt{n}$, approssimazione migliore rispetto a y_1.
Proseguendo in questo modo risulta definito un **algoritmo numerico iterativo** che, partendo da un numero arbitrario $x_0 > \sqrt{n}$, produce la successione di numeri $x_1, x_2, x_3, \ldots$, definiti dalla formula

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{n}{x_i}\right)$$

in cui ogni valore della successione è espresso in funzione del valore precedente. Inoltre tutti i valori della successione sono maggiori o uguali a $\sqrt{n}$ e ogni valore approssima $\sqrt{n}$ meglio del precedente.
La formula definisce un algoritmo per il calcolo approssimato di $\sqrt{n}$ che ha come input il numero n e il valore iniziale x_0:

```
Inizio
   Leggi n, x0
   x ← x0
   Finché (x * x <> n)
      Inizio
         x ← 0,5 * (x + n/x)
      Fine
   Scrivi x
Fine
```

Il cuore dell'algoritmo è rappresentato dal ciclo iterativo che calcola il nuovo valore della variabile x a partire dal precedente valore assunto dalla stessa variabile, partendo dal valore iniziale x_0. Il ciclo ha termine quando il valore approssimato della radice elevato al quadrato (in realtà moltiplicato per sé stesso) è esattamente uguale al valore n.

OSSERVAZIONE La convenzione di fissare il valore iniziale x_0 (secondo valore di input) al valore di cui si richiede l'approssimazione della radice (primo valore di input) nel caso che questo sia maggiore di 1, o di fissarlo a 1 altrimenti, consente di rispettare l'ipotesi di costruzione dell'algoritmo stesso ($x_0 > \sqrt{n}$); questa soluzione può essere implementata realizzando un algoritmo che ha come unico input il valore del quale restituisce la radice quadrata:

```
Inizio
   Leggi n
   Se (n > 1)
      Allora x ← n
      Altrimenti x ← 1
   Finché (x * x <> n)
      Inizio
         x ← 0,5 * (x + n/x)
      Fine
   Scrivi x
Fine
```

È possibile verificare il comportamento dell'algoritmo di Newton mediante la sua tabella di traccia (TABELLA **28**).

TABELLA **28**

Istruzione	*x*	*n*	Condizione
Leggi *n*		2	
(*n* > 1)		2	Vero
x ← *n*	2	2	
(*x* * *x* <> *n*)	2	2	Vero
x ← 0,5 * (*x* + *n*/*x*)	1,5	2	
(*x* * *x* <> *n*)	1,5	2	Vero
x ← 0,5 * (*x* + *n*/*x*)	1,416666667	2	
(*x* * *x* <> *n*)	1,416666667	2	Vero
x ← 0,5 * (*x* + *n*/*x*)	1,414215686	2	
(*x* * *x* <> *n*)	1,414215686	2	Vero
x ← 0,5 * (*x* + *n*/*x*)	1,414213562	2	
(*x* * *x* <> *n*)	1,414213562	2	Falso
Scrivi *x*	**1,414213562**	2	

OSSERVAZIONE La condizione di terminazione del ciclo dell'algoritmo per il calcolo della radice quadrata è molto incerta; infatti, se gli errori di calcolo che purtroppo anche il computer commette utilizzando numeri non interi (ogni macchina ha una propria precisione) impediscono la determinazione di un risultato esatto, il ciclo si ripete senza fine a causa dell'insufficiente precisione che impedisce il verificarsi della condizione di terminazione.

6 La «macchina» di Turing come esecutore di algoritmi

Nel 1936 – molto prima della progettazione e realizzazione del primo elaboratore elettronico – il matematico inglese Alan Turing ideò una «macchina» immaginaria capace di eseguire ogni tipo di calcolo su numeri e simboli. Essa, di fatto, costituisce un modello formale per la rappresentazione e l'esecuzione di algoritmi.

Nella sua formulazione classica la **macchina di Turing** (nel seguito **MDT**) prevede:

- un insieme finito di simboli detto **alfabeto**, che comprende anche un simbolo vuoto, lo **spazio** o ***blank*** (nel seguito rappresentato dal simbolo «–»);

- un **nastro** di lunghezza illimitata suddiviso in caselle di cui solo una sua parte finita contiene inizialmente simboli diversi dal *blank*;
- una **testina** che scorre sopra il nastro capace di leggere o di scrivere da/in una casella del nastro un simbolo dell'alfabeto;
- un **dispositivo di controllo** dotato di uno stato interno che determina il comportamento della macchina in base al suo stato e al contenuto della casella del nastro attualmente sotto la testina di lettura/scrittura (FIGURA **9**).

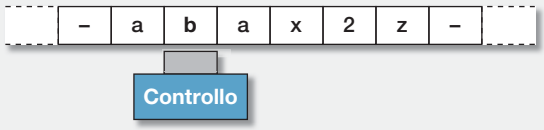


FIGURA **9**

La macchina analizza il contenuto delle caselle del nastro una cella alla volta, iniziando dalla cella contenente un simbolo diverso da *blank* che si trova più a sinistra sul nastro. Il funzionamento avviene per **passi**: a ogni passo la macchina legge un simbolo sul nastro e, in base al suo stato interno:

- modifica il suo stato interno;
- scrive un simbolo dell'alfabeto sul nastro;
- sposta la testina a sinistra (<) o a destra (>) di una casella.

Lo **stato interno** di una MDT definisce il contesto in cui una decisione viene presa; una MDT ha solo un numero finito di stati.

Il **comportamento** di una MDT può essere opportunamente programmato definendo un insieme di regole del tipo:

(**stato-corrente**, **simbolo-letto**, **nuovo-stato**, **simbolo-scritto**, **direzione**)

Essendo composta da cinque elementi, la singola regola viene chiamata **quintupla**; essa stabilisce il nuovo stato interno, il simbolo da scrivere sul nastro nella posizione attuale e la direzione di spostamento a partire dallo stato interno attuale e del simbolo presente sul nastro nella posizione corrente della testina. Il «programma» costituito dalle quintuple che possono essere applicate a una MDT viene detto **matrice funzionale**.

Il funzionamento di una MDT avviene nel modo seguente.

- La macchina determina la regola da applicare in base allo stato interno e al simbolo corrente (quello contenuto nella cella del nastro sopra la quale si trova la testina).

Alan Turing

Questo grande matematico inglese è morto suicida nel 1954 all'età di quarantadue anni. È stato uno dei pionieri dello studio della logica di funzionamento dei computer, così come li conosciamo oggi, e il primo a interessarsi all'intelligenza artificiale.

Durante la seconda guerra mondiale Turing fu al servizio del governo inglese per decifrare i messaggi in codice delle comunicazioni tedesche. La marina tedesca aveva sviluppato una macchina, denominata «Enigma», capace di generare un codice segreto variabile. Turing e i suoi collaboratori riuscirono a decifrare i messaggi di «Enigma» per la marina inglese, che ne trasse importanti vantaggi militari.

Dopo la guerra Turing tentò inutilmente di far costruire in Inghilterra il primo elaboratore elettronico universale, basandosi sugli studi teorici che aveva effettuato da studente e che avevano portato all'ideazione di quella che oggi è nota come «macchina» di Turing.

Negli primi anni '50 Turing fu processato per la sua omosessualità, che a quel tempo era un reato, e condannato a una cura ormonale che gli creò gravi problemi psicofisici: due anni dopo Turing pose fine alla sua esistenza.

Funzioni e MDT

Una **funzione** è una relazione che associa gli elementi di due insiemi. Sia definita la funzione $f : A \rightarrow B$; scriveremo $f(a) = b$ se $a \in A$ e $b \in B$ e gli elementi a e b sono in relazione secondo f.
Proprietà fondamentale di una funzione è quella per cui, se $f(a) = b$ e $f(a) = c$ allora si ha che $b = c$.
In effetti ogni **MDT** calcola una funzione che, applicata a un input (argomento), produce al termine della computazione un output (risultato o valore della funzione). Formalmente si può affermare che, dato un nastro con configurazione iniziale x e una configurazione finale y, una MDT esegue una elaborazione del tipo $y = f(x)$, dove f altro non è che la matrice funzionale della MDT.

- Se esiste una tale regola cambia lo stato, scrive il simbolo nella cella sopra la quale si trova la testina e sposta la testina di una cella a destra o a sinistra come indicato dalla regola.
- Se non esiste una regola applicabile, l'esecuzione termina.

OSSERVAZIONE Poiché la computazione deve essere deterministica, a partire da una singola configurazione non devono essere applicabili regole diverse: non possono esistere più regole per lo stesso stato e lo stesso simbolo corrente, ovvero nella matrice funzionale non devono essere presenti due o più quintuple con le prime due componenti uguali.

OSSERVAZIONE Al contrario di un algoritmo rappresentato mediante DAB o NLS l'ordine in cui le regole compaiono nella matrice funzionale di una MDT non ha nessuna importanza; le regole sono selezionate in base allo stato attuale e al simbolo corrente e sono sempre ricercate in tutta la matrice funzionale della MDT.

OSSERVAZIONE Affinché il calcolo possa terminare è necessario che ad alcune delle configurazioni possibili non corrisponda alcuna regola; in caso contrario, per qualunque risultato di una singola operazione esisterebbe sempre un'altra regola da applicare per eseguire una nuova operazione: tali configurazioni sono chiamate **configurazioni finali**.

ESEMPIO

Progettiamo una semplice MDT con alfabeto $\alpha \equiv \{X, Y, -\}$ che, disponendo di un nastro che contiene inizialmente una sequenza di simboli di α, scambi tutte le occorrenze del simbolo «X» con «Y» e viceversa.
Assumendo che la testina sia inizialmente posizionata sul simbolo più a sinistra della sequenza, dobbiamo cambiare una «X» in «Y» o, viceversa, una «Y» in «X» e successivamente spostare la testina sulla cella immediatamente a destra.
La matrice funzionale di questa MDT sarà:

Stato corrente	Simbolo letto	Nuovo stato	Simbolo scritto	Spostamento
0	X	0	Y	>
0	Y	0	X	>

In questo esempio si è fatto uso di un solo stato e la testina si sposta solo da sinistra verso destra, fino a che non si incontra un simbolo *blank* che indica la fine della sequenza di simboli. Non essendo presente alcuna regola per questo simbolo, il procedimento termina.

Supponendo di avere inizialmente sul nastro la sequenza «XXYX» otterremo le trasformazioni mostrate in FIGURA **10**. ▶

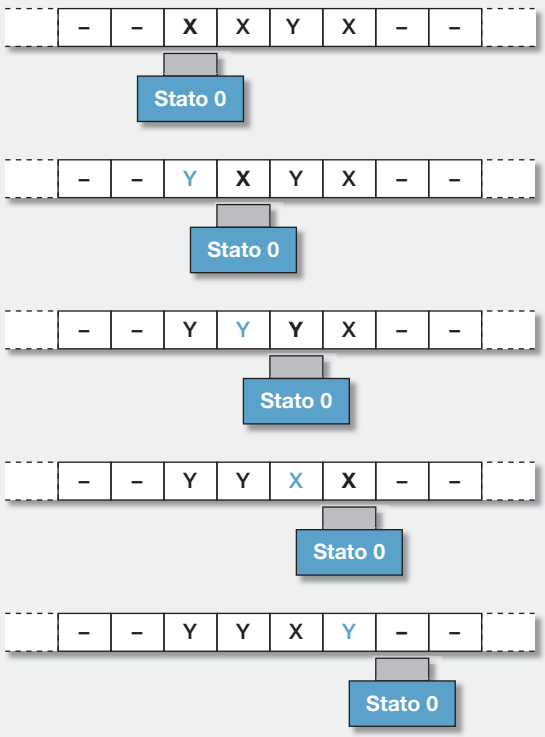


FIGURA **10**

3. Una sequenza di caratteri è palindroma se la sua lettura da destra verso sinistra è identica alla sua lettura da sinistra verso destra.

ESEMPIO

Una classica MDT è quella che riconosce una sequenza di caratteri palindroma[3]: la MDT dovrà scrivere il carattere «S» in caso affermativo, «N» altrimenti (utilizziamo lo stesso alfabeto α dell'esempio precedente). Ovviamente è necessario trovare una strategia risolutiva – un algoritmo – come la seguente:

«cancellare un carattere all'estremità sinistra della sequenza e il corrispondente all'estremità destra se essi sono uguali; se ripetendo questo procedimento si riescono a eliminare tutti i simboli inizialmente presenti sul nastro si conclude con successo, se si incontrano due simboli estremi diversi tra loro si cancellano i simboli rimanenti sul nastro e si conclude con insuccesso».

La matrice funzionale di una MDT che implementa l'algoritmo descritto è sviluppata nella TABELLA **29**.

TABELLA **29**

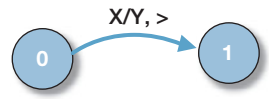
Stato corrente	Simbolo letto	Nuovo stato	Simbolo scritto	Spostamento	Descrizione
0	X	1	–	>	Nello stato 0 si legge l'estremo sinistro e lo si cancella: lo stato 1 codifica il simbolo X
0	Y	2	–	>	Nello stato 0 si legge l'estremo sinistro e lo si cancella: lo stato 2 codifica il simbolo Y
0	–	7	S	<	Tutti i simboli sono stati eliminati, il procedimento termina con successo: si scrive il simbolo S e si passa allo stato finale 7
1	X	1	X	>	Si ricerca l'estremo destro per verificare che sia un simbolo X: scorrendo si riscrivono i simboli letti nelle singole celle per non alterare il contenuto del nastro
1	Y	1	Y	>	

▶ TABELLA **29**

Stato corrente	Simbolo letto	Nuovo stato	Simbolo scritto	Spostamento	Descrizione
1	–	3	–	<	Il simbolo *blank* significa che è stato superato l'estremo destro: nello stato 3, che codifica il simbolo X, si torna indietro di una cella per posizionarsi sull'estremo destro per verificare se è o meno una X
2	X	2	X	>	Si ricerca l'estremo destro per verificare che sia un simbolo Y: scorrendo si riscrivono i simboli letti nelle singole celle per non alterare il contenuto del nastro
2	Y	2	Y	>	
2	–	4	–	<	Il simbolo *blank* significa che è stato superato l'estremo destro: nello stato 4, che codifica il simbolo Y, si torna indietro di una cella per posizionarsi sull'estremo destro per verificare se è o meno una Y
3	X	5	–	<	L'estremo destro è il simbolo X e coincide con il sinistro: si elimina il simbolo e si ritorna all'estremo sinistro utilizzando lo stato 5
3	Y	6	–	<	L'estremo destro è il simbolo Y e non coincide con il sinistro: si utilizza lo stato 6 per eliminare tutti i simboli rimanenti procedendo verso sinistra
3	–	0	–	<	Sono stati eliminati tutti i simboli con successo: si ritorna nello stato 0 per scrivere il simbolo S
4	Y	5	–	<	L'estremo destro è il simbolo Y e coincide con il sinistro: si elimina il simbolo e si ritorna all'estremo sinistro utilizzando lo stato 5
4	X	6	–	<	L'estremo destro è il simbolo X e non coincide con il sinistro: si utilizza lo stato 6 per eliminare tutti i simboli rimanenti procedendo verso sinistra
4	–	0	–	<	Sono stati eliminati tutti i simboli con successo: si ritorna nello stato 0 per scrivere il simbolo S
5	X	5	X	<	Nello stato 5 si scorrono le celle del nastro verso sinistra riscrivendo i simboli che contengono fino a raggiungere l'estremo sinistro
5	Y	5	X	<	
5	–	0	–	>	È stato raggiunto l'estremo sinistro: si ritorna allo stato 0 per continuare il procedimento
6	X	6	–	<	Nello stato 6 si scorrono le celle del nastro verso sinistra cancellando i simboli che contengono fino a raggiungere l'estremo sinistro
6	Y	6	–	<	
6	–	7	N	<	È stato raggiunto l'estremo sinistro: si conclude scrivendo il simbolo N e si passa allo stato finale 7

Si noti come, non essendoci alcuna regola che contempli il 7 come stato corrente, questo costituisca lo stato finale in cui la computazione termina.

In alternativa alla rappresentazione tramite la matrice funzionale è possibile utilizzare un grafo che fornisce una notazione più compatta. Per esempio la regola (0, X, 1, Y, >) può essere espressa come segue:



ESEMPIO

Rappresentando la matrice funzionale dell'esempio precedente con questa notazione si ottiene il grafo di FIGURA **11**.

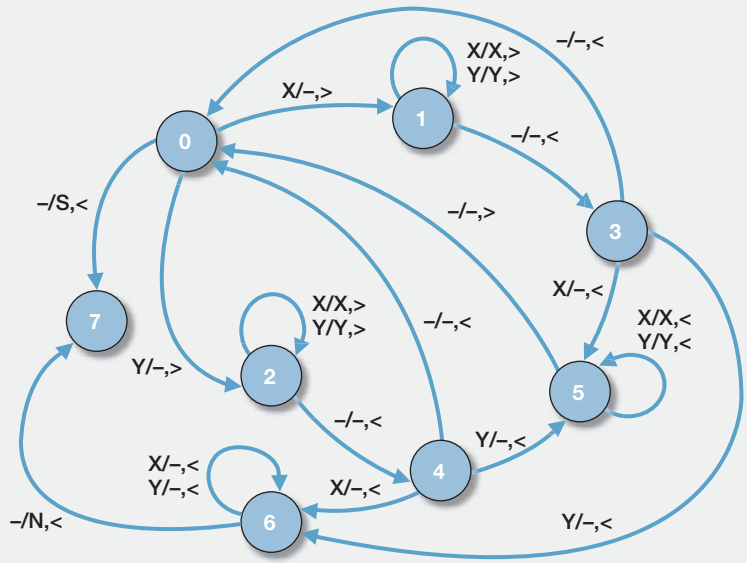


FIGURA **11**

Ipotizzando un nastro iniziale con la sequenza palindroma «YXY», si ottiene la computazione illustrata in FIGURA **12**.

ESEMPIO

Un ulteriore esempio è dato dalla MDT che, dato un numero espresso mediante le cifre in base 10, ne computa il successivo: l'alfabeto su cui si basa questa macchina è ovviamente $\alpha \equiv \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –\}$.
Una possibile strategia risolutiva è la seguente:

«a partire dall'ultima cifra del numero, se essa è minore di 9 la si sostituisce con la cifra successiva e si termina, altrimenti si pone questa uguale a 0 e si passa a esaminare la cifra immediatamente a sinistra ripetendo il procedimento».

Supponendo che inizialmente la testina di lettura/scrittura sia posizionata sulla cifra più significativa del numero, la matrice funzionale della MDT è quella descritta nella TABELLA **30**.
Supponendo di avere un nastro iniziale con il numero «129» (costituito dalla sequenza di cifre «1», «2», «9») si ottengono le trasformazioni illustrate in FIGURA **13**. ▶

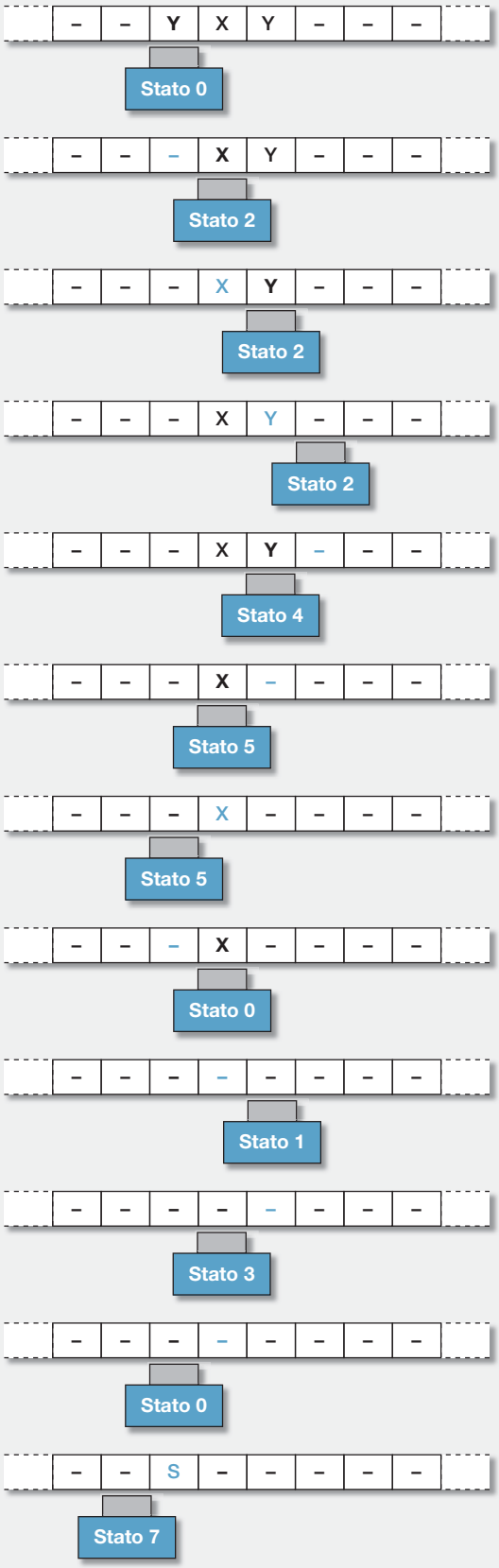


FIGURA **12**

ESEMPIO

TABELLA **30**

Stato corrente	Simbolo letto	Nuovo stato	Simbolo scritto	Spostamento	Descrizione
0	0	0	0	>	Spostamento verso destra per raggiungere la fine del numero: nello stato 0 si riscrivono i simboli letti per non alterare il contenuto del nastro
0	1	0	1	>	
0	2	0	2	>	
0	3	0	3	>	
0	4	0	4	>	
0	5	0	5	>	
0	6	0	6	>	
0	7	0	7	>	
0	8	0	8	>	
0	9	0	9	>	
0	–	1	–	<	Se si raggiunge il simbolo *blank* si utilizza lo stato 1 per posizionarsi sull'ultima cifra
1	0	2	1	<	Nello stato 1 si sostituisce il simbolo della cifra corrente con il simbolo della cifra successiva se essa è inferiore a 9: lo stato 2 è lo stato finale privo di regole da applicare
1	1	2	2	<	
1	2	2	3	<	
1	3	2	4	<	
1	4	2	5	<	
1	5	2	6	<	
1	6	2	7	<	
1	7	2	8	<	
1	8	2	9	<	
1	9	1	0	<	Se la cifra corrente è 9 si scrive 0 e, generando un riporto, si sposta la testina nella cella immediatamente a sinistra mantenendo lo stato 1
1	–	2	1	<	Se nello stato 1 si raggiunge l'estremo sinistro del numero, ciò sta a significare che il numero era composto da tutti 9 (che sono stati trasformati in 0): si scrive il simbolo 1 e si passa allo stato finale 2

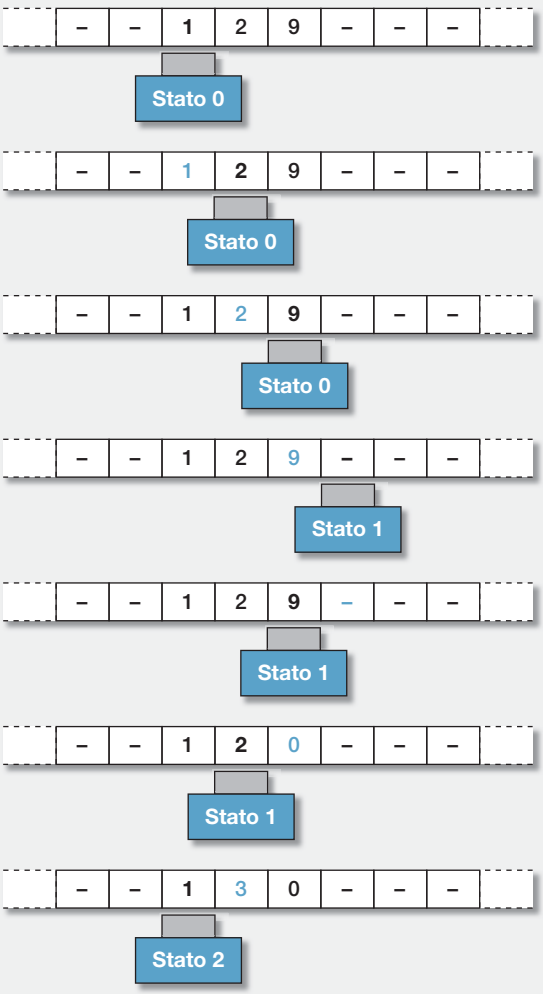


FIGURA **13**

ESEMPIO

Consideriamo il problema di programmare una MDT che, dato un nastro iniziale contenente una sequenza di simboli «*» consecutivi, termina la sua esecuzione lasciando sul nastro una sequenza composta dalla metà dei simboli iniziali (in particolare, se inizialmente sono presenti sul nastro N simboli, se N è pari alla fine dovranno essere presenti sul nastro $N/2$ simboli, se N è dispari $(N – 1)/2$ simboli. Per la MDT si può adottare la strategia che prevede di ripetere i seguenti passi:

- scorrere la sequenza di ingresso partendo dal simbolo più a sinistra;
- se la sequenza di ingresso contiene un solo * si elimina e termina;
- se la sequenza di ingresso contiene almeno due * si eliminano entrambi e si aggiunge un * alla sequenza di uscita, che viene costruita nella parte del nastro che si trova a destra della sequenza di ingresso separata da questa dal simbolo $.

L'alfabeto su cui si basa la nostra macchina sarà $\alpha \equiv \{*, \$, –\}$; la matrice funzionale descritta nella TABELLA **31** implementa la strategia risolutiva illustrata. Ipotizzando la sequenza iniziale «***» si avrà la computazione illustrata in FIGURA **14**. ▶

TABELLA **31**

Stato corrente	Simbolo letto	Nuovo stato	Simbolo scritto	Spostamento	Descrizione
0	*	0	*	>	Nello stato 0 si scorre il nastro verso destra per trovare la fine della sequenza e posizionare il simbolo separatore $: si passa allo stato 1
0	–	1	$	<	
1	*	1	*	<	Nello stato 1 si ritorna indietro copiando i simboli * trovati: si passa allo stato 2 se si raggiunge l'estremo sinistro
1	–	2	–	>	
1	$	5	$	<	Lo stato 1 è utilizzato per il ritorno all'estremo sinistro: se si trova il simbolo $ si usa lo stato 5 per «saltare» il carattere separatore e raggiungere l'estremo sinistro
2	*	3	–	>	Se nello stato 2 si trova il simbolo * si elimina e si passa allo stato 3, per eliminare eventualmente anche il secondo *
3	*	4	–	>	Se nello stato 3 si trova il simbolo * si elimina e si passa allo stato 4, per raggiungere l'estremo destro
3	$	7	–	>	Se nello stato 3 si trova il simbolo $ significa che non c'è un secondo simbolo *: si elimina il simbolo separatore e si passa allo stato finale 7
4	*	4	*	>	Lo stato 4 è utilizzato per raggiungere l'estremo destro e aggiungere un simbolo * in fondo alla sequenza di uscita: i simboli trovati sono riscritti per non alterare il contenuto del nastro
4	$	4	$	>	
4	–	1	*	<	Raggiunto l'estremo destro si aggiunge un simbolo * e si ritorna nello stato 1 per ripetere il procedimento
5	–	6	–	>	Nel percorso verso l'estremo sinistro si è appena superato il simbolo separatore: se si trova il simbolo *blank* significa che la sequenza di ingresso è terminata, per cui si passa allo stato 6 per eliminare il simbolo $
5	*	1	*	<	Si è appena superato il simbolo separatore nel percorso per raggiungere l'estremo sinistro: si passa allo stato 1 per continuare copiando i simboli * trovati allo scopo di non alterare il contenuto del nastro
6	$	7	–	>	Si elimina il simbolo separatore e si passa allo stato finale

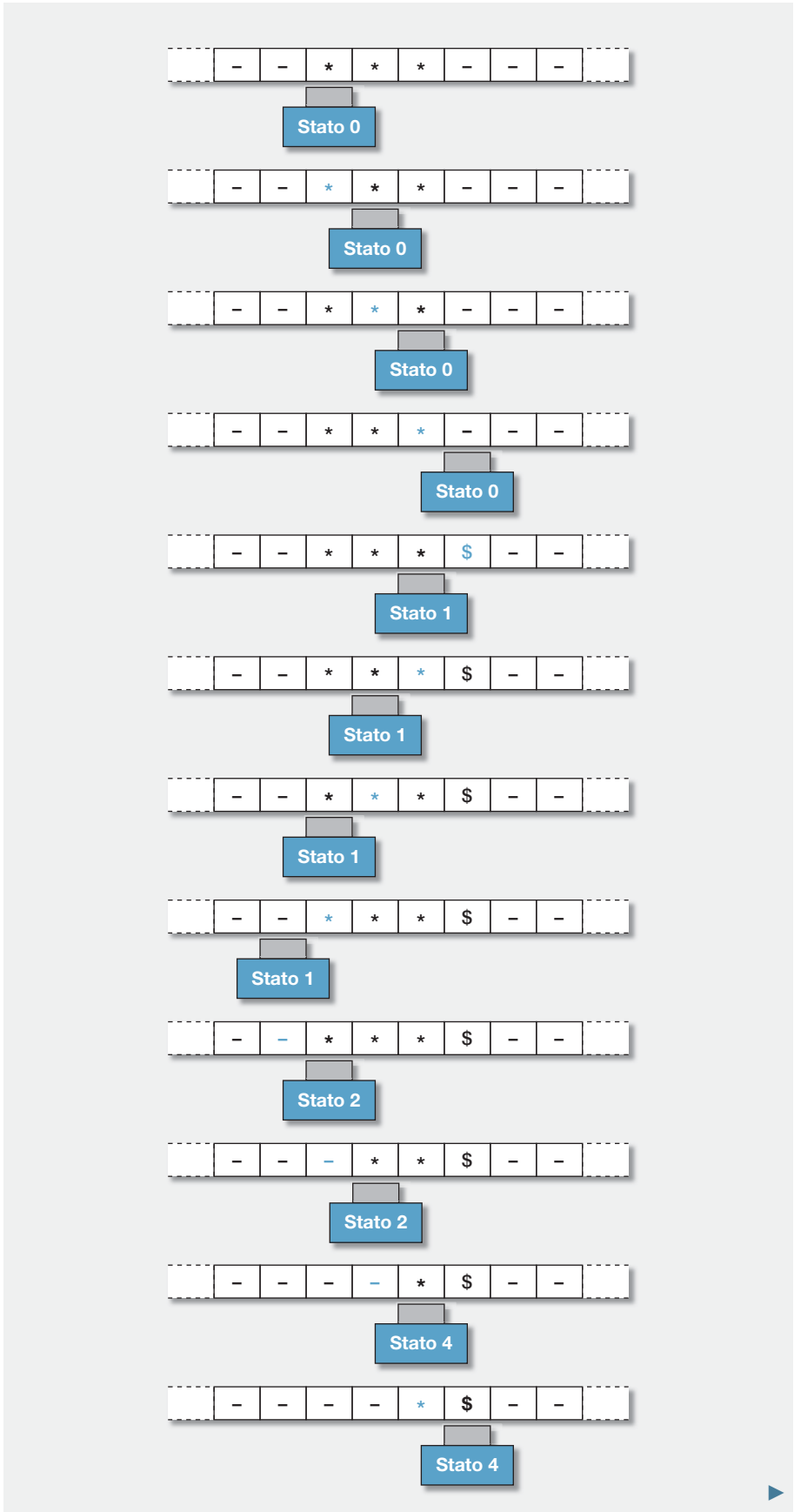


FIGURA **14** (prima parte)

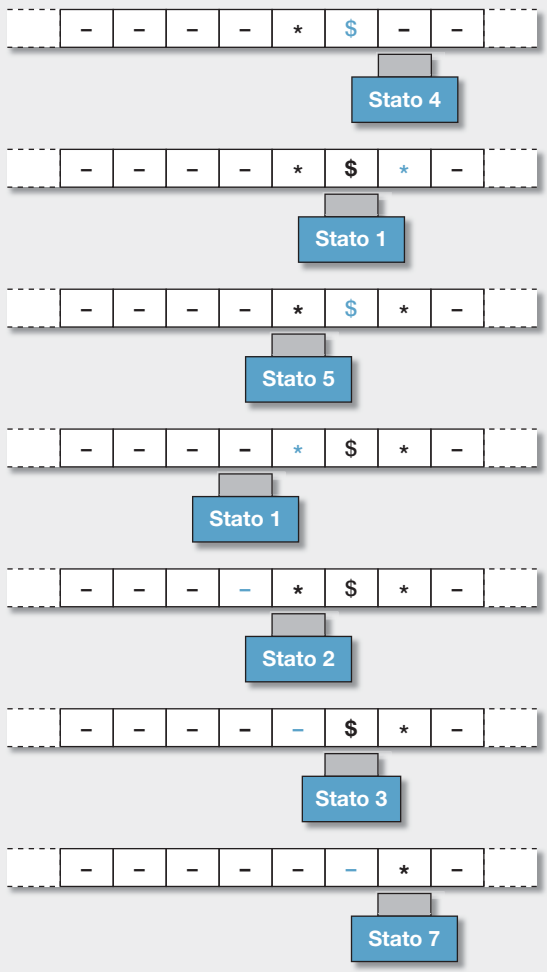


FIGURA **14** (seconda parte)

OSSERVAZIONE Sarebbe stato possibile fare a meno del simbolo «$» e gestire le due sequenze iniziale e finale separate da un simbolo *blank*; in questo caso però avremmo dovuto distinguere, mediante un ulteriore stato specifico, il simbolo *blank* di separazione delle due sequenze da quello che precede l'estremo sinistro della sequenza di ingresso e da quello che segue l'estremo destro della sequenza di uscita.

Dagli esempi prodotti è evidente che la macchina di Turing esegue gli algoritmi impiegando operazioni estremamente elementari, per cui anche le computazioni più semplici sono molto lunghe e complesse; si ritiene comunque che la MDT sia in grado di eseguire qualsiasi algoritmo, in particolare qualsiasi algoritmo che risulti eseguibile mediante un moderno computer.

MDT universale

La **tesi di Church**, dal nome del matematico Alonzo Church, afferma che «**per ogni algoritmo esiste una MDT che lo implementa**».

È inoltre possibile programmare una MDT in una MDT: questa MDT è nota come la **macchina di Turing universale** (MTU).

La MTU prende come input la specifica di una MDT e il suo nastro iniziale e calcola la funzione della macchina avuta in ingresso, sul nastro specificato. Si può pensare la MTU come una MDT con due nastri, di cui uno dedicato a contenere la matrice funzionale della MDT da simulare e l'altro l'input su cui eseguire la simulazione.

La MTU rappresenta un risultato logico fondamentale: **la macchina che calcola funzioni è a sua volta una funzione**.

Un moderno elaboratore elettronico può essere visto come la realizzazione di una MTU che esegue un programma sui dati di input: sia il programma sia i dati sono residenti nella memoria che, diversamente dal nastro immaginario, ha una dimensione finita.

7 Cenni sulla valutazione della complessità degli algoritmi

È intuitivo che possono esistere più algoritmi che risolvono uno stesso problema, basta pensare al problema di raggiungere un certo luogo della città partendo da uno specifico punto di partenza: sicuramente è possibile seguire itinerari diversi che portano alla meta. Si tratta però di valutare l'efficienza dei diversi procedimenti. In generale la **complessità computazionale**, o efficienza, viene valutata rispetto all'uso di risorse come lo **spazio**, cioè la quantità di memoria necessaria, e il **tempo** richiesto per l'esecuzione dell'algoritmo; tra le due il tempo è la risorsa più critica (facendo riferimento all'esempio precedente, l'itinerario più veloce sarà probabilmente da preferire).
Il tempo di esecuzione di un algoritmo viene normalmente espresso come una funzione della dimensione dei dati in ingresso e indicato come di seguito:

$$T(n)$$

ESEMPIO

Volendo cercare il massimo di un insieme di valori numerici, il tempo necessario al completamento dell'algoritmo è una funzione del numero dei valori dell'insieme. Utilizzando l'algoritmo già illustrato per l'individuazione del massimo tra *n* numeri:

```
Inizio
    Leggi n
    Leggi valore
    max ← valore
    n ← n – 1
    Finché (n > 0) esegui
        Inizio
            Leggi valore
            Se (valore > max)
                allora max ← valore
            n ← n – 1
        Fine
    Scrivi max
Fine
```

risulta immediatamente chiaro come per 2 soli numeri il ciclo e le istruzioni in esso contenute siano eseguiti solo una volta, mentre nel caso di 1000 numeri 999 volte.

Nel caso in cui dovessimo **ordinare** i valori dell'insieme, per esempio dal più grande al più piccolo, vi sono poi altre considerazioni da fare. Intuitivamente un qualsiasi algoritmo che risolva questo problema può trovare due casi limite:

- caso migliore: l'insieme è già ordinato;
- caso peggiore: l'insieme è ordinato in senso inverso a quello che vogliamo ottenere.

In questo caso il tempo di esecuzione non dipende solo dalla quantità di dati forniti in input all'algoritmo, ma anche dalla loro disposizione iniziale: in generale si tende a valutare la complessità di questi algoritmi come la media del tempo impiegato per il caso migliore e per il caso peggiore.

OSSERVAZIONE Il tempo di esecuzione di un algoritmo dipende ovviamente anche dalla velocità dell'esecutore utilizzato; per questo motivo la misurazione della complessità computazionale si basa sul numero di operazioni compiute durante l'esecuzione dell'algoritmo.

Ipotizzando che le istruzioni di un algoritmo abbiano tutte un tempo di esecuzione unitario (in realtà, essendo le istruzioni di natura diversa, questo non necessariamente è vero) una stima del tempo totale T di esecuzione di un algoritmo è esprimibile come:

$$T = n_0 + n_1 + ... + n_n$$

dove n_i rappresenta il numero di volte per cui viene eseguita l'i-esima istruzione dell'algoritmo.
Riprendendo l'esempio precedente relativo alla determinazione del più grande tra n numeri e ignorando le istruzioni *Leggi/Scrivi* nel caso di applicazione a un insieme di 100 valori si ha la situazione sintetizzata nella TABELLA **32**.

TABELLA **32**

Riga NLS	Tipo	Numero di esecuzioni
4	Assegnamento	1
5	Decremento	1
6	Selezione	99
6	Salto	98
9	Selezione	99
10	Assegnamento	nel caso migliore 0 nel caso peggiore 99 **caso medio 99 : 2 = 49**
11	Decremento	99
	Costo medio totale	**446**

ESEMPIO

Consideriamo il problema del calcolo della somma dei numeri naturali che vanno da 1 a N.
Questo problema può essere risolto in due modi diversi:

- utilizzando la formula di Gauss per cui la somma dei primi N numeri naturali è data da

$$\frac{n \cdot (n + 1)}{2}$$

```
Inizio
   Leggi N
   somma ← N * (N + 1)/2
   Scrivi somma
Fine
```

- oppure tramite un algoritmo iterativo:

```
Inizio
   Leggi N
   somma ← 0
   Finché (N > 0) esegui
      Inizio
         somma ← somma + N
         N ← N – 1
      Fine
   Scrivi somma
Fine
```

Il primo metodo ha un costo indipendente dal valore di *N*: una somma, una moltiplicazione, una divisione e un assegnamento per un costo totale costante uguale a 4.
Con il secondo metodo si hanno $2N + 1$ assegnamenti, *N* somme, *N* decrementi, *N* confronti ed *N* ripetizioni del ciclo, per un costo totale variabile in funzione di *N* uguale a $6N + 1$.
Nel caso di 1000 valori si ha un costo con il secondo metodo pari a 6001 rispetto al costo 4 del primo metodo!

ESEMPIO

Vediamo ora due distinti algoritmi per la visualizzazione dei primi *N* termini di una progressione geometrica[4] di ragione 2 (cioè la sequenza: 2, 4, 8, 16, …).
Il primo algoritmo utilizza un ciclo con una variabile contatore *i* per i termini visualizzati e una variabile accumulatore *a* per il calcolo dei singoli termini; la strategia è quella di calcolare e visualizzare ogni singolo termine moltiplicando la ragione per il termine precedente:

```
Inizio
   Leggi N
      i ← 1
      a ← 1
   Finché (i <= N) esegui
      Inizio
         a ← a * 2
         i ← i + 1
      Scrivi a
   Fine
Fine
```

Il secondo algoritmo utilizza una strategia diversa: ogni volta che viene calcolato e visualizzato un termine si riparte dall'inizio per il calcolo del termine successivo. In questo caso vengono utilizzati due cicli, uno interno all'altro: quello esterno (con contatore *i*) è relativo ai termini visualizzati (fino all'*N*-esimo), mentre quello interno (con contatore *j*) effettua di volta in volta il calcolo del termine corrente:

```
Inizio
   Leggi N
      i ← 1
   Finché (i <= N) esegui
      Inizio
         j ← 0
         a ← 1
```

▶

4. Una progressione geometrica è una sequenza di valori numerici in cui ogni valore è ottenuto dal precedente moltiplicandolo per un valore costante (ragione della progressione).

```
    Finché (j < i) esegui
      Inizio
        a ← a * 2
        j ← j + 1
      Fine
      Scrivi a
      i ← i + 1
    Fine
Fine
```

Come si può notare nel primo caso abbiamo:

- assegnamenti: $2N + 2$
- somme: $2N$

mentre nel secondo si hanno:

- assegnamenti: $1 + 2N + N * (N - 1) + N =$
  $= 1 + N * (2 + N - 1 + 1) = 1 + N * (N + 2) = N^2 + 2N + 1$
- somme: $N * (N - 1) + N = N * (N - 1 + 1) = N^2$.

La complessità computazionale del primo algoritmo è proporzionale al valore di N, mentre quella del secondo è proporzionale al quadrato del valore di N. Per N uguale a 1000 il costo del primo metodo è di circa 4000 operazioni elementari, mentre il costo del secondo metodo è di oltre 2 000 000 di operazioni!
Entrambi gli algoritmi sono corretti, e quindi ugualmente efficaci nel risolvere il problema dato, ma la loro efficienza è estremamente diversa e fa di gran lunga preferire il primo al secondo.

OSSERVAZIONE Anche se la tecnologia informatica realizza di anno in anno computer sempre più veloci, il progettista di algoritmi non può limitarsi a realizzare algoritmi corretti, ma deve valutarne l'efficienza in termini di costo di esecuzione.

L'algoritmo di Shlemiel il pittore

In relazione all'efficienza degli algoritmi su uno dei blog di programmazione più noti (*Joel on software* di J. Spolsky) è riportata la seguente istruttiva storiella.

«Shlemiel trovò lavoro come pittore delle strisce al centro delle strade: il primo giorno di lavoro prese un secchio di vernice e completò 300 metri di strada meritandosi un premio da parte del datore di lavoro per la sua velocità. Il secondo giorno Shlemiel riuscì a completare solo 150 metri di strada, ma in ogni caso il datore di lavoro si complimentò con lui. Il terzo giorno non riuscì a fare più di 30 metri e il datore di lavoro, molto deluso, gli chiese che cosa era accaduto dopo il primo giorno. Shlemiel rispose che non poteva farci nulla se il secchio della vernice era ogni giorno più lontano!»

Sintesi

Problema. Una qualsiasi situazione reale che necessiti di specifiche azioni per essere risolta, ovvero trasformata in uno stato finale che si valuta migliore, o comunque preferibile, a quello di partenza.

Algoritmo. Un algoritmo è l'esplicitazione dei passi elementari necessari a risolvere un determinato problema o un classe di problemi simili; esso generalmente opera su dei dati di ingresso (input) per fornire dei risultati in uscita (output).

Proprietà di un algoritmo

FINITEZZA: un algoritmo è composto da un numero finito di istruzioni.

TERMINAZIONE: dopo l'esecuzione di un numero finito di passi l'algoritmo deve terminare.

DETERMINATEZZA: a ogni passo dell'algoritmo deve essere specificata l'azione da intraprendere senza ambiguità.

EFFETTIVITÀ: l'azione specificata in ogni passo dell'algoritmo deve essere effettivamente eseguibile dall'esecutore prescelto per l'esecuzione dell'algoritmo.

GENERALITÀ: un algoritmo dovrebbe essere progettato per risolvere una classe di problemi simili.

Esecutore. Entità preposta all'esecuzione dei passi specificati in un algoritmo. In genere si usa classificare gli esecutori in *esecutori intelligenti*, che risolvono problemi applicando le conoscenze acquisite nel corso della propria attività, ed *esecutori automatici*, che risolvono problemi eseguendo meccanicamente i passi di un algoritmo senza acquisire nuove conoscenze.

Rappresentazione degli algoritmi. Due delle tecniche più diffuse per la rappresentazione di algoritmi sono: il *diagramma a blocchi* (DAB), che prevede una rappresentazione grafica che evidenzia visivamente la sequenza delle operazioni e le strutture che compongono l'algoritmo, e la *notazione lineare strutturata* (NLS), un linguaggio simile al linguaggio naturale, ma privo di ambiguità, che usa parole chiave predefinite per rappresentare le strutture di esecuzione di un algoritmo.

Schemi fondamentali di composizione. In un algoritmo si possono individuare tre schemi fondamentali di composizione.

SEQUENZA: una o più istruzioni che sono eseguite una di seguito all'altra nell'ordine in cui sono state scritte.

SELEZIONE: l'esecutore esegue una di due distinte istruzioni o sequenze di istruzioni al verificarsi o meno di una certa condizione.

RIPETIZIONE (O ITERAZIONE): l'esecutore deve ripetere più volte una istruzione o un gruppo di istruzioni fino a che continua a verificarsi una specifica condizione.

Tipi di iterazione. Si distinguono due tipi di ripetizione: i *cicli determinati*, per i quali si conosce a priori il numero delle volte che il ciclo verrà ripetuto, e i *cicli indeterminati*, per i quali il numero di cicli ripetuti dipende dal verificarsi o meno della condizione che ne controlla l'arresto. La condizione può essere espressa in testa o in coda al ciclo: nel primo caso il ciclo potrebbe anche non essere mai eseguito (se la condizione è inizialmente falsa), mentre nel secondo caso il ciclo sarà eseguito almeno una volta.

Contatori e accumulatori. Si tratta di variabili il cui contenuto viene rispettivamente incrementato di una quantità fissa e generalmente unitaria o di una quantità variabile; nel caso dei contatori, questi vengono detti *a scalare* se la quantità è negativa. Una tipica istruzione che implementa un contatore è $c \leftarrow c + 1$, mentre $a \leftarrow a + n$ realizza un accumulatore.

Tabella di traccia. Tabella che permette di tenere traccia dei valori assunti dalle singole variabili durante l'esecuzione di un algoritmo su specifici valori di input; essa prevede una colonna per ogni variabile utilizzata dall'algoritmo, mentre sulle righe, una per ogni istruzione eseguita, viene annotato il valore assunto dalle singole variabili. Questo è uno strumento molto utile per verificare passo per passo la funzionalità di un algoritmo.

Macchina di Turing (MDT). Macchina immaginaria ideata dal matematico inglese Alan Turing e capace di eseguire ogni tipo di calcolo su numeri e simboli; essa, di fatto, costituisce un modello formale per la rappresentazione e l'esecuzione di algoritmi.

Struttura di una MDT. Nella sua formulazione classica una MDT prevede:

- un insieme finito di simboli (alfabeto) comprendente un simbolo vuoto (*blank*);
- un nastro illimitato suddiviso in celle di cui solo una sua parte finita inizialmente contiene simboli diversi da *blank*;
- una testina che può spostarsi sopra il nastro per leggere o scrivere un simbolo da/in una cella;
- un dispositivo di controllo dotato di stato interno che determina il comportamento della macchina in base al suo stato e al contenuto della cella del nastro sopra cui si trova la testina.

Matrice funzionale di una MDT. Il comportamento di una MDT può essere programmato definendo un insieme di regole, o *quintuple* del tipo:

(*stato-corrente*, *simbolo-letto*, *nuovo-stato*, *simbolo-scritto*, *direzione*)

Il programma costituito dalle quintuple che possono essere applicate a una MDT viene detto *matrice funzionale* e, a seconda dello stato interno e del simbolo letto, determina: il nuovo stato da assumere; il simbolo da scrivere nella cella attualmente sotto la testina; lo spostamento da effettuare (a destra o a sinistra, ma di una sola cella).

Funzionamento di una MDT. Una MDT opera come segue:

- determina la regola da applicare in base allo stato interno e al simbolo letto dalla testina;
- se esiste una tale regola cambia lo stato, scrive il simbolo nella cella corrente e si sposta come indicato dalla regola;
- se non esiste una regola applicabile l'esecuzione termina.

Complessità degli algoritmi. L'efficienza con cui diversi algoritmi risolvono uno stesso problema diventa rilevante quando la quantità di dati da elaborare diventa grande. In generale la complessità computazionale o efficienza viene valutata rispetto all'uso di risorse quali lo *spazio*, ovvero la memoria necessaria per l'esecuzione dell'algoritmo, e il *tempo* richiesto per l'esecuzione dell'algoritmo stesso. La risorsa critica è il tempo; esso viene calcolato in funzione del numero di operazioni elementari eseguite durante l'applicazione di un algoritmo su specifici dati di input.

QUESITI

1 **Quali delle seguenti affermazioni relative a un algoritmo sono corrette?**

A Può essere composto da un numero illimitato di passi.
B Ogni suo passo non deve essere ambiguo.
C Ogni suo passo deve essere eseguibile dall'esecutore preposto.
D Uno o più passi possono essere ripetuti indefinitamente.

2 **Una variabile è …**

A … un contenitore di più valori che possono variare durante l'esecuzione di un algoritmo.
B … un contenitore che può contenere un solo valore che può variare durante l'esecuzione di un algoritmo.
C … un contenitore che può contenere un solo valore che deve variare durante l'esecuzione di un algoritmo.
D … un contenitore che può contenere un solo valore che non può variare durante l'esecuzione di un algoritmo.

3 **L'operatore di assegnamento è un operatore che permette di …**

A … modificare il valore dell'espressione che sta alla sua sinistra.
B … effettuare una operazione di input di una variabile.
C … modificare il contenuto della variabile che sta alla sua sinistra.
D … modificare il contenuto della variabile che sta alla sua destra.

4 **Che cosa è un DAB?**

A Una tecnica grafica per la rappresentazione di un algoritmo.
B Un diagramma che permette di simulare con diversi valori la funzionalità di un algoritmo.

C La descrizione di un algoritmo che può essere direttamente eseguita da un computer.

D Una tecnica per descrivere un algoritmo senza fare uso di variabili.

5 **Che cosa è la NLS?**

A La stessa cosa di un DAB.

B Una tecnica simile al linguaggio naturale per rappresentare algoritmi.

C Una tecnica che permette la descrizione di un algoritmo in modo tale da poter essere direttamente eseguita da un computer.

D Una tecnica per descrivere un algoritmo senza fare uso di variabili.

6 **Una sequenza è …**

A … uno schema fondamentale di composizione di un algoritmo in cui le singole istruzioni vengono eseguite nell'ordine stesso in cui sono state scritte.

B … uno schema fondamentale di composizione di un algoritmo in cui le singole istruzioni vengono eseguite in un ordine casuale.

C … uno schema fondamentale di composizione di un algoritmo che prevede la ripetizione di una o più istruzioni.

D … uno schema fondamentale di composizione di un algoritmo che permette la scelta di un gruppo di istruzioni da eseguire in funzione del verificarsi o meno di una certa condizione.

7 **La selezione è …**

A … uno schema fondamentale di composizione di un algoritmo in cui le singole istruzioni vengono eseguite nell'ordine stesso in cui sono state scritte.

B … uno schema fondamentale di composizione di un algoritmo in cui le singole istruzioni vengono eseguite in un ordine casuale.

C … uno schema fondamentale di composizione di un algoritmo che prevede la ripetizione di una o più istruzioni.

D … uno schema fondamentale di composizione di un algoritmo che permette la scelta di un gruppo di istruzioni da eseguire in funzione del verificarsi o meno di una certa condizione.

8 **La ripetizione è …**

A … uno schema fondamentale di composizione di un algoritmo in cui le singole istruzioni vengono eseguite nell'ordine stesso in cui sono state scritte.

B … uno schema fondamentale di composizione di un algoritmo in cui le singole istruzioni vengono eseguite in un ordine casuale.

C … uno schema fondamentale di composizione di un algoritmo che prevede la ripetizione di una o più istruzioni.

D … uno schema fondamentale di composizione di un algoritmo che permette la scelta di un gruppo di istruzioni da eseguire in funzione del verificarsi o meno di una certa condizione.

9 **Un ciclo si dice indeterminato quando …**

A … non termina mai.

B … non viene eseguito nemmeno una volta.

C … viene eseguito in funzione del verificarsi o meno di una condizione che ne controlla l'arresto.

D … viene eseguito almeno una volta.

10 **Quali delle seguenti affermazioni sono vere in relazione a un ciclo indeterminato con controllo in testa?**

A Può non terminare mai.

B Può non essere eseguito nemmeno una volta.

C Si conosce a priori il numero di volte che verrà eseguito.

D Viene eseguito almeno una volta.

11 **Quali delle seguenti affermazioni sono vere in relazione a un ciclo indeterminato con controllo in coda?**

A Può non terminare mai.

B Può non essere eseguito nemmeno una volta.

C Si conosce a priori il numero di volte che verrà eseguito.

D Viene eseguito almeno una volta.

12 **Un ciclo si dice determinato quando …**

A … non termina mai.

B … viene eseguito in ogni caso.

C … viene eseguito almeno una volta.

D … si conosce a priori il numero di volte che verrà eseguito.

13 **Quali delle seguenti affermazioni sono vere in relazione a un ciclo determinato?**

A Non termina mai.

B Viene eseguito in ogni caso.

C Utilizza in maniera più o meno esplicita una variabile contatore per tenere traccia del numero di volte che viene eseguito.

D Si conosce a priori il numero di volte che verrà eseguito.

14 **Quali delle seguenti affermazioni sono vere in relazione a un ciclo?**

A Prevede l'uso di una condizione per il controllo del suo arresto.

B Prevede comunque l'esecuzione della prima istruzione al uso interno.

C Le istruzioni al suo interno saranno eseguite sicuramente almeno una volta.

D Le istruzioni al suo interno possono essere eseguite neanche una volta.

15 **Una MDT è …**

A … un vecchio tipo di computer.

B … un modello formale per la rappresentazione e l'esecuzione di algoritmi.

C … un esecutore intelligente.

D … una macchina astratta con memoria illimitata.

16 **Che cosa rappresenta la matrice funzionale di una MDT?**

A I dati di input su cui la MDT lavora.

B I dati di output prodotti dalla MDT.

C L'insieme delle istruzioni che la MDT può eseguire.

D L'insieme degli stati riconosciuti dalla MDT.

17 **In ogni riga della matrice funzionale di una MDT viene descritta …**

A ... la coppia <stato-corrente, simbolo-letto> in funzione della tripla <nuovo-stato, simbolo-scritto, spostamento>.

B ... la tripla <nuovo-stato, simbolo-scritto, spostamento> in funzione della coppia <stato-corrente, simbolo-letto>.

C ... la configurazione corrente del nastro della MDT.

D ... la quintupla <stato-corrente, simbolo-letto, nuovo-stato, simbolo-scritto, spostamento> in funzione della configurazione iniziale del nastro della MDT.

18 **Che cosa rappresenta il nastro di una MDT?**

A Il supporto su cui sono memorizzati i dati di input su cui la MDT lavora.

B Il supporto su cui sono memorizzati i dati di output prodotti dalla MDT.

C Il supporto su cui sono memorizzati sia i dati di input sia quelli di output della MDT.

D Il supporto su cui è memorizzato l'insieme delle istruzioni che la MDT esegue.

19 **La notazione a grafo di una MDT è ...**

A ... l'equivalente della matrice funzionale per quella MDT.

B ... l'equivalente del nastro della MDT.

C ... l'equivalente della testina della MDT.

D ... l'insieme degli stati in cui può trovarsi la MDT.

ESERCIZI

Progettazione di algoritmi

Tutti gli esercizi di questa sezione possono essere risolti utilizzando DAB o NLS; suggeriamo di risolverli utilizzando entrambe le notazioni e di verificare il risultato compilando almeno una tabella di traccia.

Esercizi risolvibili senza uso della struttura iterativa

1 **Progettare un algoritmo che, letto il valore *r* del raggio, calcoli e scriva l'area del cerchio relativo.**

2 Lo spazio s espresso in metri di frenata di un'automobile – supposto essere di 1 secondo il tempo di reazione del guidatore – è stimato mediante la seguente formula:

$$s = \frac{v^2}{250 \cdot f}$$

dove v è la velocità in km/h ed f un coefficiente relativo alle condizioni stradali:

Condizioni stradali	f
Asfalto ruvido	0,6
Asfalto liscio	0,5
Asfalto bagnato	0,4
Asfalto ghiacciato	0,1

Progettare un algoritmo che calcola lo spazio di frenata a partire dalla velocità v e dal coefficiente f.

3 È esperienza comune che la presenza di vento fa percepire una temperatura inferiore a quella reale. La seguente formula, messa a punto da scienziati americani durante la seconda guerra mondiale, consente di calcolare la temperatura t percepita a partire dalla temperatura T reale e dalla velocità v del vento:

$$t = 33 + (0{,}45 + 0{,}29 \cdot \sqrt{v} - 0{,}02 \cdot v) \cdot (T - 33)$$

Le temperature sono espresse in °C e la velocità del vento in miglia per ora (mph); la formula è valida solo per velocità del vento superiori a 5 mph, al di sotto della quale t è uguale a T.
Progettare un algoritmo che calcoli la temperatura percepita a partire dalla temperatura reale e dalla velocità del vento.

4 Un professore assegna i livelli di valutazione (A, B, C, D o E) ai propri studenti in base alla seguente tabella dei punteggi:

Punteggio	Valutazione
0-40	E
41-60	D
61-70	C
71-85	B
86-100	A

Progettare un algoritmo che consenta di determinare il livello di valutazione a partire dal punteggio.

5 Gli abbonamenti alla metropolitana di Roma possono essere settimanali (S), mensili (M) o annuali (A) e valgono per la sola zona centrale (tipo 1), per la sola zona periferica (tipo 2) o per entrambe le zone (tipo 3). I costi sono quelli indicati nella seguente tabella:

Tipo	Settimanale	Mensile	Annuale
1	10 €	30 €	250 €
2	5 €	20 €	150 €
3	15 €	40 €	300 €

Gli abbonamenti che non consentono di viaggiare nelle ore di punta (bassa priorità) hanno una riduzione del 20%.
Progettare un algoritmo che calcola il costo dell'abbonamento a partire dalla durata, dal tipo e della priorità.

Esercizi non risolvibili senza uso della struttura iterativa

6 Progettare un algoritmo che legga una sequenza di valori numerici fino alla lettura di un valore 0 e scriva quanti valori sono stati letti e la loro somma.

7 Progettare un algoritmo che, dato un valore numerico k, legga n valori e conti quanti di essi sono maggiori di k scrivendo il risultato.

8 Modificare l'algoritmo dell'esercizio precedente scrivendo tre risultati: quanti sono i valori maggiori di k, quanti i valori uguali a k e quanti quelli minori.

9 Progettare un algoritmo che, dato un valore numerico k, legga n numeri e conti quanti di questi sono multipli di k scrivendo il risultato.

10 Progettare un algoritmo che legga una sequenza di valori numerici fino a che la loro somma è minore di 100 e scriva la somma ottenuta e quanti sono i valori letti.

11 Progettare un algoritmo che, dati due valori numerici *h* e *k*, legga *n* valori e conti quanti di essi sono compresi tra *h* e *k* scrivendo il risultato.

12 Progettare un algoritmo che, leggendo *n* valori numerici, verifichi se essi sono forniti in ordine crescente o meno.

13 Progettare un algoritmo che, dato un valore numerico *k*, legga *n* coppie di valori e conti quante di queste coppie hanno come prodotto il valore *k*.

14 Dati *n* valori numerici in ordine crescente, progettare un algoritmo che scriva se i numeri forniti a partire dal secondo differiscono ognuno dal precedente di un valore costante. In caso affermativo l'algoritmo deve scrivere il valore della differenza, in caso negativo l'algoritmo deve scrivere il valore massimo delle differenze.

15 Dato il DAB di FIGURA **15**, supponendo che la variabile *x* sia un intero non negativo, indicare che cosa calcola l'algoritmo che esso rappresenta e che cosa rappresentano le due variabili *w* e *x*.

16 Una leggenda orientale narra di un matematico che, in cambio di alcuni servigi resi al re, chiese la seguente ricompensa: «un chicco di riso per la prima casella di una scacchiera, due chicchi di riso per la seconda casella di una scacchiera, quattro chicchi di riso per la terza casella… e così via per tutte le 64 caselle della scacchiera». Progettare un algoritmo che, a partire dal numero *N* di caselle che si intendono riempire, calcoli il numero complessivo di chicchi di riso che spettano come ricompensa.

17 Un'onda marina anomala dimezza la propria altezza ogni chilometro percorso e scompare raggiungendo un'altezza pari a zero quando l'altezza scende al di sotto del metro.

a) Progettare un algoritmo che calcoli, a partire dai valori dell'altezza iniziale *h* e dal numero

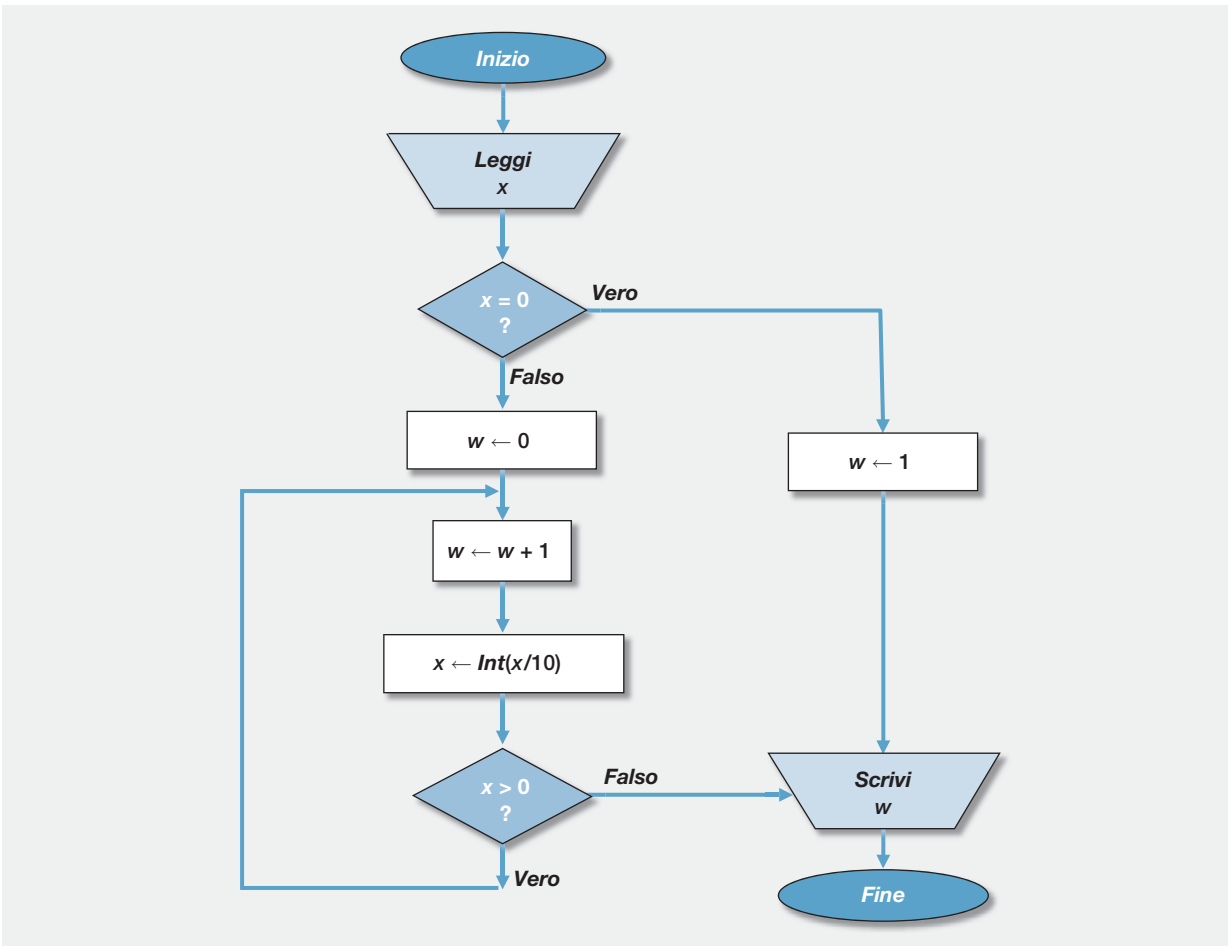


FIGURA **15**

di chilometri percorsi *k*, l'altezza raggiunta dall'onda.

b) Modificare l'algoritmo precedente in modo che, a partire dalla sola altezza iniziale dell'onda *h*, determini il numero di chilometri necessario prima che essa scompaia.

18 Nella disintegrazione atomica dei materiali radioattivi la massa perduta nel periodo di un anno è data dal prodotto della massa residua per una costante di decadimento caratteristica del tipo di materiale.

a) Progettare un algoritmo che calcoli, a partire dai valori della massa iniziale espressa in grammi, della costante di decadimento e del numero di anni trascorsi, la massa residua di materiale.

b) Modificare l'algoritmo precedente in modo che, a partire dalla massa iniziale espressa in grammi e dalla costante di decadimento, determini il numero di anni necessario prima che la massa residua di materiale sia inferiore a 1 g.

19 La popolazione di un particolare batterio raddoppia ogni ora. Progettare un algoritmo che, a partire dal numero di ore trascorse e dal valore espresso in «unità di carica batterica» della consistenza iniziale della popolazione batterica, ne calcoli la consistenza finale raggiunta.

20 La massa di un particolare materiale radioattivo dimezza ogni millennio. Progettare un algoritmo che, a partire dal numero di millenni trascorsi e dal valore espresso in grammi della massa iniziale del materiale radioattivo, calcoli la massa finale residua.

21 Dato un valore numerico costante *k* (non necessariamente intero), l'*N*-esimo numero di Bernoulli è dato dalla somma dei primi *N* numeri interi elevati alla potenza *k*; per esempio per $N = 5$:

$$1^k + 2^k + 3^k + 4^k + 5^k$$

Progettare un algoritmo che determini, a partire dai valori della costante *k* e del numero *N*, il numero di Bernoulli relativo.

22 La media geometrica di *N* numeri $x_1, x_2, \ldots, x_N$ è data dalla seguente formula:

$$(x_1 \cdot x_2 \cdot \ldots \cdot x_N)^{1/N}$$

Progettare un algoritmo che calcoli la media geometrica di *N* numeri positivi inseriti dall'utente.

23 Il filosofo Zenone di Elea motivava il fatto che il moto è solo un'illusione con la seguente argomentazione: «dovendo percorrere una certa distanza si dovrà coprire con un primo spostamento metà della distanza, con un secondo spostamento metà della distanza rimanente, con un terzo spostamento metà della distanza ancora rimanente e così via senza arrivare mai a destinazione». Progettare un algoritmo che, data la distanza da percorrere e il numero di spostamenti effettuati, calcoli la distanza effettivamente coperta.

24 In una acciaieria il semilavorato metallico grezzo viene prodotto con uno spessore di alcuni centimetri e viene successivamente lavorato passando per una serie di *N* laminatoi, ciascuno dei quali diminuisce lo spessore del 10%.

a) Progettare un algoritmo per determinare lo spessore del laminato a partire dallo spessore del semilavorato grezzo e dal numero di laminatoi presenti nel processo di lavorazione.

b) Modificare l'algoritmo precedente in modo che determini il numero di laminatoi necessari nel processo di lavorazione per ottenere un laminato di spessore definito a partire dallo spessore del semilavorato.

25 Un foglio di carta in formato A0 ha dimensioni 118,8 × 84 cm; a partire da questo un foglio in formato A1 ha il lato lungo uguale al lato corto del formato A0 (84 cm) e il lato corto uguale alla metà del lato lungo del formato A0 (118,8 cm : 2 = = 59,4 cm). Per calcolare le dimensioni dei formati A2, A3, A4, … si procede sempre nello stesso modo: il lato lungo è uguale al lato corto del foglio immediatamente più grande, mentre il lato corto è esattamente la metà del lato lungo del foglio immediatamente più grande e così via. Progettare l'algoritmo che calcola le dimensioni di un foglio in formato AN, dove N viene fornito come dato di ingresso.

26 Una popolazione di insetti ha un accrescimento mensile dato dalla seguente formula:

$$k \cdot P \cdot \left(1 - \frac{P}{M}\right)$$

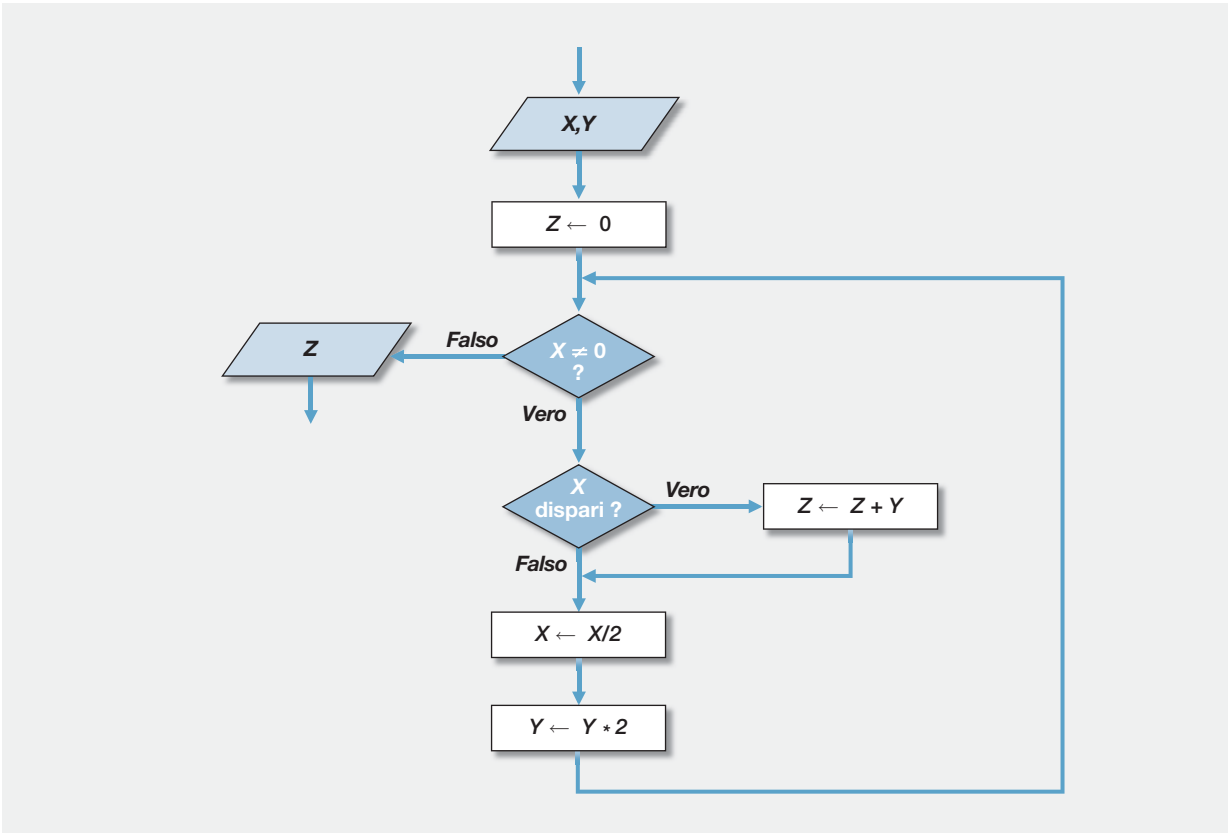


FIGURA **16**

dove:

- *P* è la popolazione di insetti;
- *k* è la costante di accrescimento;
- *M* è la massima popolazione sostenibile dall'ambiente locale.

Realizzare un algoritmo che abbia come dati

- il numero *N* dei mesi;
- la costante *k*;
- il massimo *M*;
- la popolazione iniziale *Pi*

e che calcoli come risultato la popolazione di insetti trascorsi *N* mesi dal momento iniziale.

27 L'accrescimento della popolazione umana è guidato da una semplice legge matematica: l'incremento della popolazione tra un anno e il successivo è dato dal prodotto di una costante (il tasso di accrescimento) per la dimensione della popolazione. Realizzare l'algoritmo che consente di simulare anno per anno i valori della dimensione della popolazione a partire da:

- l'anno iniziale della simulazione;
- il valore della dimensione iniziale della popolazione;
- il valore del tasso di accrescimento;
- l'anno finale della simulazione.

28 Una pianta cresce ogni mese della metà di quanto è cresciuta il mese precedente (il primo mese cresce della metà dell'altezza iniziale). Progettare l'algoritmo che calcola l'altezza finale della pianta a partire dall'altezza iniziale e dal numero di mesi.

29 Dato l'algoritmo rappresentato dal DAB di FIGURA **16**, dopo averlo completato delle parti mancanti, produrre una tabella di traccia per valori casuali delle variabili *X* e *Y* e valutare che cosa viene calcolato dalla variabile *Z* a partire dal valore di *X* e *Y*.

30 Nel 1593 il matematico dilettante francese François Viète approssimò il valore numerico di π con il seguente metodo, che fornisce risultati

progressivamente più precisi al crescere del numero n di iterazioni:

$$\begin{cases} c_0 = 0 & p_0 = 2 \\ c_n = \sqrt{\dfrac{1 + c_{n-1}}{2}} & p_n = \dfrac{p_{n-1}}{c_n} \end{cases}$$

Realizzare un algoritmo che, dato il numero n di iterazioni da calcolare, produca la stima p_n del valore di π nell'ipotesi di un esecutore che sia in grado di calcolare direttamente la radice quadrata di un numero.

31 La prima stima precisa del valore di π è stata fornita dal matematico cinese Tsu Chung-chi nel V secolo d.C. utilizzando il seguente metodo iterativo:

$$x_0 = \sqrt{2}$$
$$x_{n+1} = \sqrt{2 - \sqrt{4 - x_n^2}}$$

dove, a partire dal valore iniziale x_0, viene calcolato ogni volta il valore successivo x_{n+1} a partire dal valore precedente x_n; al termine del procedimento – dopo N iterazioni del calcolo – la stima del valore di π è data dalla seguente formula:

$$s = 2^N \cdot x_N$$

Progettare un algoritmo che, a partire dal numero di iterazioni N, calcoli una stima s del valore di π nell'ipotesi di un esecutore che sia in grado di calcolare direttamente la radice quadrata di un numero.

Programmazione della MDT

32 Programmare una MDT che, a partire da un nastro iniziale contenente la rappresentazione decimale di un numero intero positivo N, termini la sua esecuzione lasciando sul nastro la rappresentazione decimale $N \times 100$ (esempio: «505» diventa «50500»).

33 Programmare una MDT che, dato un nastro iniziale contenente una sequenza di simboli «A» e «B», termini la sua esecuzione lasciando sul nastro un solo simbolo «V» se la sequenza iniziale contiene almeno una B, un solo simbolo «F» altrimenti.

34 Programmare una MDT che, dato un nastro iniziale contenente una sequenza di cifre decimali, termina la sua esecuzione lasciando sul nastro una sequenza che si ottiene eliminando tutte le cifre «0» iniziali; se la sequenza iniziale è composta solo da cifre uguali a «0» la computazione deve terminare con un solo simbolo «0».

35 Programmare una MDT che, dato un nastro iniziale contenente una sequenza di A e B comprendente almeno una B, termina la sua esecuzione lasciando sul nastro la sequenza di sole B consecutive (senza spazi) ottenuta eliminando tutte le A (esempio: «AABAABBAB» diventa «BBBB»).

36 Programmare una MDT che divida un numero intero – codificato sul nastro in base 10 – per 10. Nel caso di numero non divisibile per 10 dovrà essere prodotto un risultato in formato decimale.

37 Programmare una MDT che riconosca se un numero intero – codificato sul nastro in base 10 – sia divisibile per 25; in caso affermativo scriverà «SI» a destra del numero, «NO» altrimenti.

38 Programmare una MDT che riconosca se un numero intero – codificato sul nastro in base 10 – sia divisibile per 20; in caso affermativo scriverà «SI» a destra del numero, «NO» altrimenti.

39 Programmare una MDT che inverta l'ordine di una successione di simboli «0» e «1» contigui inizialmente presenti sul nastro (al termine la successione invertita dovrà essere separata dalla successione originale da un carattere «*»).

40 Programmare una MDT in modo che termini la sua esecuzione scrivendo «SI» se la parola presente inizialmente sul nastro contiene almeno una lettera estranea all'alfabeto italiano («J», «K», «W», «X» o «Y»), «NO» altrimenti.

41 Programmare una MDT in modo che, a partire da una parola scritta sul nastro, verifichi che una eventuale lettera «Q» sia sempre immediatamente seguita da una lettera «U»; il programma terminerà scrivendo «NO» sul nastro se si trova una lettera «Q» non seguita da una «U», «SI» in caso contrario. Esempi:

Nastro iniziale	Nastro finale
RIQUADRO	SÌ
PIPPO	SÌ
QWAIT	NO

42 Programmare una MDT in modo che, a partire da una parola scritta sul nastro, conti quante volte è presente la sequenza di lettere «QU» senza prevedere un limite superiore per il numero di sequenze presenti nella parola stessa. Esempi:

Nastro iniziale	Nastro finale
QUIQUOQUA	3
PIPPO	0
ORDAUQ	0
QUQUQUQUQUQUQUQU	9

43 Nel corso delle guerre galliche Giulio Cesare comunicava con i suoi generali mediante un codice segreto che consisteva nel sostituire le lettere di una parola secondo lo schema di FIGURA **17**. Programmare una MDT che trasformi una parola applicando il cifrario di Cesare.

44 Programmare una MDT per determinare la prima posizione in cui il simbolo «X» si trova in una stringa composta esclusivamente da simboli «X» e «Y» scrivendola sul nastro come numero decimale (scrivendo 0 se il simbolo «X» non compare nella stringa). Esempi:

Nastro iniziale	Nastro finale
YYYYXXYXY	5
XYYYYYY	1
YYYY	0
YYYYYYYYYYYYXYYXX	13

45 Programmare una MDT che, a partire da una sequenza di simboli «X» e «Y», scriva sul nastro «SI» se la sequenza contiene un numero uguale di «X» e di «Y», «NO» altrimenti (suggerimento: prima di tutto riscrivere la sequenza in modo che tutti i simboli «X» precedano i simboli «Y», o viceversa). Esempi:

Nastro iniziale	Nastro finale
YXYX	SÌ
YYXYYXX	NO
YYYY	NO
XXYYYXXXYY	SÌ

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
Z	**Y**	**X**	**W**	**V**	**U**	**T**	**S**	**R**	**Q**	**P**	**O**	**N**	**M**	**L**	**K**	**J**	**I**	**H**	**G**	**F**	**E**	**D**	**C**	**B**	**A**

FIGURA **17**

greatest common divisor
massimo comun divisore (MCD)

remainder
resto

recipe
ricetta

merely
banalmente

features
caratteristiche

inputs
dati di ingresso

outputs
dati in uscita (risultati)

1.1 Algorithms

The notion of an *algorithm* is basic to all computer programming, so we should begin with a careful analysis of this concept.
The word "algorithm" itself is quite interesting; at first glance it may look as though someone intended to write "logarithm" but jumbled up the first four letters. The word did not appear in Webster's New World Dictionary as late as 1957: we find the older form "algorism" with its ancient meaning, i.e., the process of doing arithmetic using Arabic numerals. In the middle ages, abacists computed on the abacus and algorists computed by algorism. Following the middle ages, the origin of this word was in doubt, and early linguists attempted to guess at its derivation by making combinations like *algiros* [painful] + *arithmos* [number]; others said no, the word comes from "King Algor of Castile." Finally, historians of mathematics found the true origin of the word algorism: it comes from the name of a famous Arabic textbook author, Abu Ja'far, Mohammed ibn Mûsâ al-Khowârizmî (c. 825) – literally, "Father of Ja'far, Mohammed, son of Moses, native of Khowârizm." Khowârizm is today the small Uzbek city of Khiva. Al-Khowârizmî wrote the celebrated book *Kitab al jabr w'al-muqabala* ("Rules of restoration and reduction"); another word, "algebra," stems from the title of his book, although the book wasn't really very algebraic.
Gradually the form and meaning of "algorism" became corrupted; as explained by the Oxford English dictionary, the word was "erroneously refashioned by learned confusion" with the word *arithmetic*. The change from "algorism" to "algorithm" is not hard to understand in view of the fact that people have forgotten the original derivation of the word. An early German mathematical dictionary, *Vollständiges Mathematisches Lexicon* (Leipzig, 1747), gives the following definition of the word Algorithmus: "Under this designation are combined the notions of four types of arithmetic calculations, namely addition, multiplication, subtraction, and division." The latin phrase *algorithmus infinitesimalis* was at that time used to denote "ways of calculation with infinitely small quantities, as invented by Leibnitz."
By 1950, the word algorithm was most frequently associated with "Euclid's algorithm," a process for finding the greatest common divisor of two numbers, which appears in Euclid's Elements (book vii, proposition i and ii). It will be instructive to exhibit Euclid's algorithm here:

Algorithm E (*Euclid's algorithm*). Given two positive integers m and n, find their greatest common divisor, i.e., the largest positive integer which evenly divides both m and n.
E1. [Find remainder.] Divide m by n and let r be remainder (we will have $0 \le r < n$.)
E2. [Is it zero?] If $r = 0$, the algorithm terminates; n is the answer.
E3. [Interchange.] Set $m \leftarrow n$, $n \leftarrow r$, and go back step E1.
[…]

The modern meaning for algorithm is quite similar to that of *recipe*, *process*, *method*, *technique*, *procedure*, *routine*, except that the word "algorithm" connotes something just a little different. Besides merely being a finite set of rules which gives a sequence of operations for solving a specific type of problem, an algorithm has five important features:

1 Finiteness. An algorithm must always terminate after a finite number of steps. Algorithm E satisfied this condition, because after step E1 the values of r is less than n, so if $r \neq 0$, the value of *n decreases* the next time that the step E1 is encountered. A decreasing sequence of positive integers must eventually terminate, so step E1 is executed only a finite number of times for any given original value of n. Note, however, that the number of steps can become arbitrarily large: certain huge choices of m and n will cause step E1 to be executed over a million times.
(A procedure which has all of the characteristics of an algorithm except that it possibility lacks finiteness may be called a "computational method." Besides his algorithm for the

gratest common common divisor of two integers, Euclid also gave a geometrical construction that is essentially equivalent to Algorithm E, except it is a procedure for obtaining the "greatest common measure" of the lengths of two line segments; this is a computational method that does not terminate if the given lengths are "incommensurate.")

2 **Definiteness**. Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case […] In algorithm E, the criterion of definiteness as applied to step E1 means the reader is supposed to understand exactly what it means to divide *m* by *n* and what the remainder is. […]

3 **Input**. An algorithm has one or more inputs, i.e., quantities which are given it initially before the algorithm begins. These inputs are taken from a specified sets of objects. In algorithm E, for example, there are two inputs, namely *m* and *n*, which are both taken from the set of *positive integers*.

4 **Output**. An algorithm has one or many outputs, i.e., quantities which have a specified relation to the inputs. Algorithm E has one output, namely *n* in step E2, which is the greatest common divisor of the two inputs. […]

5 **Effectiveness**. An algorithm is also generally expected to be effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using pencil and paper. Algorithm E uses only the operations of dividing one positive integer by another, testing if an integer is zero, and setting the value of one variable equal to the value of another. These operations are effective, because integers can be represented on a paper in a finite manner and there is at least one method (the "division algorithm") for dividing one by another. But the same operation would not be effective if the values involved were arbitrary real number specified by an infinite decimal expansion nor if the values were the length of physical line segments, which cannot be specified exactly. Another example of a noneffective step is, "If 2 is the largest integer n for which there is a solution to the equation $x^n + y^n = z^n$ in positive integers x, y, and z, then goto step E4." Such a statement would not be an effective operation until someone succeeds in showing that there is an algorithm to determine whether 2 is, or is not, the largest integer with the stated property. […]
We should remark that the finiteness restriction is really not strong enough for practical use; a useful algorithm should require not only a finite number of steps, but a very finite number, a reasonable number. […]
In practice we not only want algorithms, we want good algorithms in some loosely-defined aesthetic sense. One criterion of goodness is the length of time taken to perform the algorithm; this can be expressed in terms of the number each step is executed. Other criteria are the adaptability of the algorithm to computers, its simplicity and elegance, etc.

[D. Knuth, *The art of computer programming*, vol. 1, Addison Wesley, 1969]

QUESTIONS

a Can an algorithm be infinite?

b Change Algotithm E (for the sake of efficiency) so that at step E3 we do not interchange values but immediately divide *n* by *r* and let *m* be the remainder. Add appropriate new steps to avoid all trivial replacement options. Write this new algorithm in the style of Algorithm E, and call it Algorithm F.

c What is the greatest common divisor of 2,166 and 6,099?

d What does the "definiteness" rule mean?

A3 Linguaggi di programmazione

Nel capitolo precedente è stato risolto il problema di determinare in quale giorno della settimana cade una certa data; l'algoritmo individuato per conseguire questo scopo è rappresentabile nella forma NLS come segue:

Inizio
  Leggi *anno*
  Leggi *mese*
  Leggi *giorno*
  $Y \leftarrow anno$
  $M \leftarrow mese - 2$
  Se $M <= 0$ ***allora***
    Inizio
      $Y \leftarrow Y - 1$
      $M \leftarrow M + 12$
    Fine
  $D \leftarrow ((giorno + Y + \textbf{\textit{INT}}(Y : 4) - \textbf{\textit{INT}}(Y : 100) + \textbf{\textit{INT}}(Y : 400) +$
      $+ \textbf{\textit{INT}}(31 \times M : 12))$ ***MOD*** $7)$
  Scrivi *D*
Fine

Se l'esecutore è un computer, questo tipo di formalismo, come pure quello relativo alla notazione DAB, non è direttamente utilizzabile: è necessario «tradurre» le istruzioni dell'algoritmo seguendo le regole di un linguaggio di programmazione per fare in modo che la macchina sia in grado di interpretarle ed eseguirle.

ESEMPIO

Il seguente programma, scritto in linguaggio C++, rappresenta una possibile implementazione dell'algoritmo:

```
void main(void)
{
  int anno, mese, giorno, Y, M, D;

  cin>>anno;
  cin>>mese;
  cin>>giorno;
  Y = anno;
  M = mese-2;
  if (M <= 0)
    {
      Y = Y-1;
      M = M+12;
    }
```

▶

```
    D = (giorno+Y+(Y/4) - (Y/100) + (Y/400) + (31*M/12)) % 7);
    cout<<D;
}
```

Senza voler prendere in esame dettagli che saranno discussi in seguito, risulta abbastanza immediato riconoscere nel codice del programma le parti fondamentali dell'algoritmo.

Un **linguaggio di programmazione** è una notazione formale per implementare algoritmi in modo tale che essi possano essere eseguiti da una macchina. Di conseguenza un **programma** è la descrizione, interpretabile da un computer, di un algoritmo secondo le regole di uno specifico linguaggio di programmazione.

1 Evoluzione dei linguaggi di programmazione

Ada, Algol, APL, Assembler, BASIC, C/C++, CoBoL, ForTran, Java, LISP, Logo, Pascal, Perl, PHP, PL/1, ProLog, Python, Simula, SmallTalk, ... Sono solo alcuni, in ordine alfabetico, dei numerosi linguaggi di programmazione che, a partire dagli anni '50 del secolo scorso, hanno caratterizzato l'evoluzione dello sviluppo software.

Ma perché tanti linguaggi diversi? L'obiettivo è sempre stato quello di fornire strumenti che permettessero allo sviluppatore software di focalizzare l'attenzione sul problema da risolvere, nascondendo i dettagli necessari a garantire il funzionamento dei programmi su uno specifico computer: i linguaggi di programmazione si sono di conseguenza evoluti verso il modo in cui la mente umana ragiona, astraendosi sempre di più dal livello della macchina esecutrice. Inoltre molti linguaggi di programmazione comprendono istruzioni progettate esplicitamente per essere utilizzate in specifiche aree applicative, come la matematica, le applicazioni gestionali, l'intelligenza artificiale ecc.; altri, invece, hanno caratteristiche più generali che ne permettono l'impiego in contesti diversi.

In ogni caso, come abbiamo già avuto modo di vedere, all'interno del computer «tutto è numero» e i programmi, indipendentemente dal linguaggio in cui sono codificati, sono memorizzati nella memoria come codici di macchina binari, ovvero istruzioni che il processore è in grado di interpretare ed eseguire. All'alba della storia dell'informatica gli sviluppatori di software erano costretti a scrivere direttamente i codici binari delle istruzioni che costituivano un programma: il livello di produttività era ovviamente molto basso e i programmi, anche se semplici, erano di difficile realizzazione e manutenzione. Si pensò quindi di interporre tra l'utente programmatore e la macchina hardware alcuni «strati» di software che facilitassero lo sviluppo di programmi più complessi, senza che il programmatore si dovesse preoccupare delle caratteristiche peculiari dei computer su cui questi sarebbero stati eseguiti.

1. Gli indirizzi e i codici delle istruzioni sono rappresentati in formato esadecimale, invece che binario.

ESEMPIO

Un semplice programma avente una dimensione di 34 byte e il solo scopo di visualizzare il messaggio «Hello, World!», viene memorizzato nella memoria del computer nel modo seguente[1]:

```
47B5:0000 B8B647
47B5:0003 8ED8
47B5:0005 8D160200
47B5:0009 B409
47B5:000B CD21
47B5:000D B8004C
47B5:0010 CD21
47B5:0012 48
47B5:0013 656C
47B5:0015 6C
47B5:0016 6F
47B5:0017 2C20
47B5:0019 57
47B5:001A 6F
47B5:001B 726C
47B5:001D 64210A
47B5:0020 0D2400
```

Il programma effettivo termina con l'istruzione contenuta nell'indirizzo 475B:0010, mentre i rimanenti byte altro non sono che la codifica numerica del caratteri che compongono il messaggio «Hello World!».

OSSERVAZIONE La difficoltà di gestire programmi come quello dell'esempio precedente, dove per i non esperti risulta già difficile individuare le singole istruzioni, portarono allo sviluppo di linguaggi ***assembly*** di tipo simbolico, dove le istruzioni sono ben identificate e caratterizzate da formati mnemonici più espliciti rispetto allo scopo della loro esecuzione. Il programma dell'esempio precedente assume in linguaggio *assembly* il seguente formato:

```
47B5:0000  MOV       AX,47B6
47B5:0003  MOV       DS,AX
47B5:0005  LEA       DX,Word Ptr [0002]
47B5:0009  MOV       AH,09
47B5:000B  INT       21
47B5:000D  MOV       AX,4C00
47B5:0010  INT       21
```

L'immagine precedente è stata ottenuta utilizzando uno strumento «disassemblatore» che ricostruisce i codici simbolici mnemonici a partire dai codici binari delle istruzioni che costituiscono il programma. Il disassemblatore non distingue i dati dalle istruzioni e opera anche sui dati, visualizzandoli come se fossero istruzioni:

```
47B5:0012  DEC       AX
47B5:0013  INSB
47B5:0015  INSB
47B5:0016  OUTSW
47B5:0017  SUB       AL,20
47B5:0019  PUSH      DI
47B5:001A  OUTSW
47B5:001B  JB        0089
47B5:001D  AND       Word Ptr FS:[BP+SI],CX
47B5:0020  OR        AX,0024
```

Uno speciale programma denominato **assemblatore** veniva impiegato per tradurre le istruzioni dal formato simbolico al formato numerico della macchina.

ESEMPIO

Il codice simbolico in linguaggio *assembly* che ha generato il programma dell'esempio precedente è il seguente:

```
 1                          .data
 2 0000 48 65 6C 6C 6F 2C   message db "Hello, World!", 10, 13,"$"
 3     20 57 6F 72 6C 64
 4     21 0A 0D 24
 5                          .code
 6 0000                     _inizio:
 7 0000 B8 ---- R           mov ax,@data
 8 0003 8E D8               mov ds,ax
 9 0005 8D 16 0000 R        lea dx,message
10 0009 B4 09               mov ah,9
11 000B CD 21               int 21h
12 000D B8 4C00             mov ax,4c00h
13 0010 CD 21               int 21h
14                          end _inizio
```

È immediato rendersi conto come sia migliore questo modo di scrivere i programmi rispetto all'uso diretto del codice macchina, ma sono ancora presenti molti dettagli legati all'hardware del computer (istruzioni del processore, nomi di registri, indirizzi di memoria ecc.) che rendono oneroso lo sviluppo del software.

L'idea alla base dell'evoluzione dei linguaggi di programmazione è stata quella di dotarli di istruzioni più «potenti» rispetto a quelle della macchina esecutrice – il computer – e allo stesso tempo più «semplici» da usare per il programmatore. A supporto di questa semplificazione sono stati sviluppati traduttori sempre più sofisticati che hanno notevolmente aumentato il livello di astrazione e di indipendenza della pratica della programmazione dai dettagli della piattaforma di esecuzione: i **compilatori**.

ESEMPIO

Un programma con lo stesso scopo dei programmi degli esempi precedenti codificato nel linguaggio C++ avrebbe il seguente formato:

```
void main(void)
{
  cout<<"Hello, World!";
}
```

In ogni caso l'effetto è sempre quello di visualizzare il solito messaggio.

OSSERVAZIONE Dal punto di vista teorico è stato dimostrato che – per quanto riguarda la capacità computazionale, cioè la capacità di implementare algoritmi – tutti i linguaggi non banali sono di fatto equivalenti: di conseguenza un programma codificato in un certo linguaggio di programmazione può essere tradotto in un qualsiasi altro linguaggio, a meno che esso non utilizzi caratteristiche peculiari che per alcuni linguaggi sono spesso strettamente dipendenti dall'hardware del computer (come, per esempio, la gestione dell'interfaccia grafica).

Linguaggi di programmazione

Per quanto ogni linguaggio di programmazione abbia le proprie caratteristiche specifiche, gli obiettivi più comuni che i progettisti si pongono sono i seguenti: semplicità di apprendimento, facilità di comprensione del codice, rapidità di realizzazione dei programmi, modificabilità e riusabilità del codice sviluppato, efficienza del programma in termini di velocità di esecuzione del codice, portabilità dei programmi tra computer di tipo diverso.

Anche se in passato vi sono stati diversi tentativi di creare un linguaggio di programmazione universale, lo scopo non è mai stato raggiunto e i linguaggi di programmazione effettivamente utilizzati sono moltissimi: diverse centinaia. Anche alcuni linguaggi «storici» come CoBol e ForTran sono ancora utilizzati per diverse ragioni, quali l'attaccamento dei suoi utilizzatori e la grande quantità di software già sviluppato e consolidato nel tempo (software *legacy*) che è alla base delle attività informatiche quotidiane di molte organizzazioni.

OSSERVAZIONE I linguaggi di programmazione differiscono dai linguaggi naturali per una loro **minore espressività** ma per una **maggiore precisione**: in generale sono semplici e poveri (poche regole e parole chiave) ma privi di ambiguità.

Non è facile classificare i linguaggi di programmazione, ma una prima distinzione, che indirettamente abbiamo già proposto negli esempi precedenti, è quella tra i **linguaggi di basso livello** e quelli **di alto livello**.
Ogni processore ha un proprio linguaggio articolato in un insieme di istruzioni macchina i cui codici numerici sono interpretati come operazioni elementari (per esempio il caricamento di un registro interno al processore con un numero, oppure la somma tra il valore contenuto in una locazione di memoria e il valore contenuto in un registro del processore). Questo tipo di linguaggio, molto distante dall'usuale modo di ragionare dell'uomo, comporta un lavoro complesso per l'implementazione degli algoritmi. Inizialmente questo era l'unico modo per programmare un computer: è per migliorare questo stato di cose che si realizzarono linguaggi intermedi basati su una codifica simbolica delle istruzioni macchina.

OSSERVAZIONE Un programma codificato mediante un linguaggio diverso dal linguaggio delle istruzioni macchina non è più direttamente eseguibile dal processore del computer: richiede una fase di traduzione del codice originale, il programma sorgente, in una sequenza di istruzioni del linguaggio macchina, il programma oggetto o eseguibile.

Il primo linguaggio di questo tipo è stato l'*assembly*, in cui per ogni singola istruzione macchina viene usato uno specifico codice mnemonico: anche se garantiva una relativa semplificazione del lavoro, il livello di riferimento era ancora quello del processore del computer. La necessità di astrarre l'implementazione degli algoritmi dalla logica di funzionamento dei processori ha portato ai cosiddetti **linguaggi di alto livello**, orientati non più al funzionamento della macchina, ma alla risoluzione di problemi.

Questi linguaggi, essendo virtualmente indipendenti da uno specifico processore con le sue particolari istruzioni, hanno consentito di scrivere software portatile, permettendo l'utilizzo di uno stesso codice sorgente su computer di tipo diverso.

Una ulteriore classificazione dei linguaggi di programmazione di alto livello è quella tra **linguaggi *problem-oriented***, ovvero linguaggi realizzati per la risoluzione di problemi in specifiche aree applicative, e i **linguaggi *general-purpose***, ovvero linguaggi adatti per la risoluzione di problemi di tipo diverso.

Il **linguaggio C++**, il cui studio affronteremo nel seguito, è uno dei linguaggi di programmazione più utilizzati al mondo: è ovviamente un linguaggio di alto livello, ma ha mantenuto alcune caratteristiche del linguaggio C, da

cui deriva, finalizzate all'interazione del software con l'hardware su cui viene eseguito, che lo rendono meno astratto di altri linguaggi simili. Il codice generato dai compilatori C++ è noto per la sua efficienza di esecuzione e il linguaggio è il riferimento per la realizzazione del software di sistema per molti computer attuali. La standardizzazione del linguaggio garantisce inoltre una buona portabilità dei programmi C++ tra diversi tipi di computer e sistemi operativi.

2 Paradigmi di programmazione

▶ Un **paradigma di programmazione** è l'insieme degli strumenti concettuali forniti da un determinato linguaggio per la codifica di un programma e definisce il modo con cui il programmatore concepisce il programma stesso.

La storia dell'informatica è stata fortemente caratterizzata dai diversi paradigmi di programmazione: spesso nuovi paradigmi sono nati come evoluzione dei precedenti, aggiungendo alle tecniche di programmazione nuovi concetti e mantenendo al tempo stesso quelli già affermati, in modo da mantenere le pratiche emerse come regole di buona programmazione.

ESEMPIO

La programmazione strutturata negli anni '80 del secolo scorso ha introdotto le strutture di controllo standard (sequenza, selezione e ripetizione) e ha messo al bando l'uso dell'istruzione di salto «***goto***». All'epoca i rischi di un programma legato all'uso indiscriminato dell'istruzione di salto (illeggibilità, difficoltà di manutenzione ecc.) erano già noti a molti programmatori ed erano diffuse regole di stile che suggerivano di restringerne l'uso con modalità tali da corrispondere alle future strutture di controllo della programmazione strutturata.

Dal momento che un linguaggio realizza un determinato paradigma di programmazione se consente di scrivere i programmi in accordo con esso, i paradigmi costituiscono un metodo di classificazione dei linguaggi. Ad oggi i paradigmi di programmazione più utilizzati sono i seguenti.

- **Programmazione imperativa**. Un programma è composto da istruzioni che realizzano trasformazioni di stato (lo **stato** è l'insieme di tutti i valori di un insieme di variabili in un dato momento dell'esecuzione). Linguaggi di questo tipo sono, per esempio, C e Pascal.
- **Programmazione funzionale**. Un programma è visto come una funzione che deve essere valutata per ottenere un risultato. Linguaggi di questo tipo sono, per esempio, LISP, ML e F#.
- **Programmazione logica**. Un programma è costituito da un insieme di fatti e regole logiche e la sua esecuzione equivale a una dimostrazione. Il linguaggio emblema della programmazione logica è ProLog.

Istruzione GOTO

L'istruzione `GOTO` o `GO TO` (letteralmente «vai a»), prevista da molti linguaggi di programmazione, permette di definire nel corpo di un programma un salto incondizionato tra un punto e un altro del codice.

Un suo uso improprio porta alla realizzazione di programmi di difficile interpretazione (relativamente al loro aspetto algoritmico) e manutenzione nel tempo.

Nei linguaggi a basso livello come l'*assembly* l'analoga istruzione *jump* è fondamentale per controllare il flusso dell'esecuzione nell'implementazione delle strutture di selezione e dei cicli.

Il `GOTO` era molto utilizzato anche in linguaggi come il BASIC, dove le singole righe di un programma erano numerate progressivamente. Esso veniva utilizzato con una sintassi del tipo:

`GOTO` <*numero riga*>.

Ormai l'uso di questa istruzione in linguaggi ad alto livello viene fortemente sconsigliato e generalmente considerato indice di cattiva programmazione.

Come dimostrato da Böhm e Jacopini, un'attenta scrittura del codice basato sulle regole della programmazione strutturata può evitare di ricorrere all'uso del `GOTO`.

ESEMPIO

Alcuni sistemi di intelligenza artificiale si basano sul paradigma della programmazione logica: a partire da una base di conoscenza e di regole di inferenza, il programma affronta un nuovo problema applicando le regole per determinare in maniera automatica la soluzione. Con la risoluzione di nuovi problemi la base di conoscenza viene ampliata e con essa la capacità del sistema di affrontare nuovi problemi.

- **Programmazione a oggetti**. Questo tipo di programmazione si basa sul concetto di **classe**, che è un modello da cui vengono derivati **oggetti** dotati di **proprietà** (dati) e **metodi** (procedure). Esempi di linguaggi di questo tipo sono C++, Eiffel, Java e C#.

3 Fasi di sviluppo di un programma

La realizzazione di un programma passa attraverso le seguenti fasi.

- **Codifica (*editing*)**. L'algoritmo viene implementato mediante le istruzioni del linguaggio di programmazione scelto. Per la codifica viene utilizzato un **editor**, un programma che consente di salvare in uno o più file il codice digitato. Il prodotto della codifica viene comunemente chiamato **codice sorgente**.
- **Compilazione.** Il codice sorgente viene tradotto dal compilatore del linguaggio in **codice oggetto**, costituito da istruzioni macchina e direttamente eseguibile dal processore. Il **compilatore** è un programma che svolge un ruolo fondamentale, agendo come traduttore che consente di trasformare un programma codificato in un linguaggio di programmazione *human-oriented* in un programma equivalente di tipo *computer-oriented*. In questo modo i programmatori possono ignorare i dettagli dipendenti dall'hardware dello specifico computer su cui il programma viene eseguito. Inoltre il compilatore rileva eventuali errori di sintassi commessi nella codifica del programma sorgente.

OSSERVAZIONE Normalmente a ogni singola istruzione di un linguaggio di alto livello corrispondono molte istruzioni del linguaggio macchina: quanto più il linguaggio di programmazione è astratto rispetto alla struttura del computer, tanto più il lavoro di traduzione è complesso.

- ***Linking***. Nel caso in cui la costruzione del programma finale richieda l'unione di uno o più moduli di codice oggetto compilati separatamente e/o necessiti di integrare il codice contenuto nelle librerie del linguaggio si utilizza il ***linker***, che provvede al collegamento producendo come risultato un unico **programma eseguibile**.

OSSERVAZIONE Ogni programma non banale è costituito da più moduli e praticamente tutti i programmi necessitano del codice della/e libreria/e, per cui la fase di *linking* è di fatto obbligatoria e tutti i compilatori invocano automaticamente il *linker* al termine della loro attività, se non sono stati riscontrati errori nel codice.

Ambienti integrati per lo sviluppo

I programmatori di ieri utilizzavano separatamente i vari strumenti software necessari per realizzare un nuovo programma: l'editor, il compilatore, il *linker* e il *debugger*.
Oggi utilizzano ambienti integrati di sviluppo (IDE, *Integrated Development Environment*) che uniscono in un unico software tutti gli strumenti necessari per la codifica, la compilazione e il collegamento e la ricerca di errori.

Bug

Il termine *debug* indica l'attività di ricerca e correzione degli errori che causano il malfunzionamento di un programma.
Letteralmente significa «togliere i bachi» (infatti *bug* significa «insetto»).
Questa terminologia risale agli inizi dell'era informatica, quando alcuni malfunzionamenti dei grandi computer a valvole erano causati dagli insetti che penetravano all'interno e che dovevano essere regolarmente rimossi.

- ***Debugging***. Il ***debugger*** consente di eseguire un programma passo per passo controllando le istruzioni effettivamente eseguite e ispezionando il contenuto delle variabili, al fine di scoprire ed eliminare gli errori che causano malfunzionamenti.

Oltre a queste fasi esistono altre attività attinenti al ciclo di sviluppo del software.

- **Progettazione**. Molti programmi presentano una complessità non banale, per cui risulta impossibile idearne la struttura e gli algoritmi contestualmente alla codifica. Questa fase deve essere proceduta da una fase di progettazione del codice da implementare, eventualmente con l'ausilio di strumenti grafici (come, per esempio, i DAB).
- **Testing**. L'esecuzione senza malfunzionamenti evidenti non garantisce la correttezza di un programma. La validazione prevede la verifica metodica di tutte le condizioni operative mediante la variazione sistematica degli input e il controllo dei relativi output prodotti. Questa importante attività viene in genere condotta da personale distinto da quello che ha sviluppato il programma oggetto di test.
- **Documentazione**. La documentazione di un programma prevede normalmente la redazione di due diverse tipologie di documenti: il *manuale utente* e il *manuale tecnico*. Il primo è destinato all'utente operatore del programma e ne descrive le funzionalità, mentre il secondo è destinato a un informatico e descrive la struttura del software realizzato.
- **Manutenzione**. La manutenzione del software è continua: i programmi sono modificati per correggere gli errori riscontrati (manutenzione correttiva) e per integrare nuove funzionalità (manutenzione evolutiva).

Programmazione in pratica

Ai programmatori piace scherzare in merito ai vari aspetti della loro professione.
Per esempio il «principio delle 11 P» recita:

«Prima
Pensa
Poi
Programma
Perché
Programmi
Poco
Pensati
Producono
Pericolosi
Pasticci».

Il principio intende sottolineare il ruolo fondamentale del programmatore nella realizzazione di programmi per computer privi di malfunzionamenti.
A questo proposito, la legge di Murphy applicata alla programmazione recita: «se un programma contiene un errore, esso prima o poi si manifesterà e lo farà nel momento in cui potrà causare il maggior danno possibile».

4 Traduzione del codice sorgente in codice eseguibile

Un programma scritto in un linguaggio ad alto livello deve necessariamente essere tradotto in linguaggio macchina per poter essere eseguito dal computer. Esistono fondamentalmente due diversi approcci per questa azione: l'approccio compilato e quello interpretato.

4.1 Approccio compilato

La tecnica della compilazione prevede l'uso di un programma – il **compilatore** – che provvede a tradurre in un'unica soluzione il codice sorgente in codice oggetto: prima viene tradotto l'intero programma che successivamente verrà eseguito. Il compilatore verifica la correttezza sintattica del codice sorgente in accordo con le regole stabilite dallo specifico linguaggio di programmazione utilizzato; nel caso siano presenti errori il compilatore li elenca, in caso contrario genera il codice oggetto.

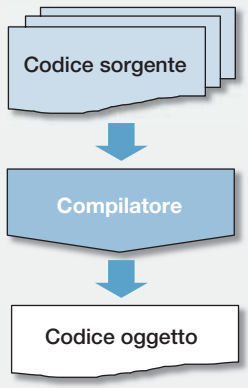


FIGURA 1

Schematicamente il processo di compilazione può essere rappresentato come in FIGURA 1.

OSSERVAZIONE Se il compilatore rileva degli errori, il programmatore dovrà effettuare una nuova fase di *editing* per la loro rimozione.

Il codice oggetto generato dal compilatore non è direttamente eseguibile dal computer; esso deve subire una ulteriore trasformazione: si deve unire al codice oggetto il codice delle funzionalità presenti nelle librerie del linguaggio di programmazione per ottenere un programma eseguibile. Questa operazione di unione viene effettuata dal programma *linker* secondo lo schema di FIGURA 2.

OSSERVAZIONE Il codice oggetto del programma eseguibile è contenuto in un file: solo quando sarà eseguito il sistema operativo provvederà a caricarlo nella memoria del computer.

OSSERVAZIONE La fase di *linking* del codice oggetto generato dal compilatore con il codice delle librerie può avvenire con due diverse modalità: **statica** o **dinamica**. Nel primo caso il collegamento avviene una sola

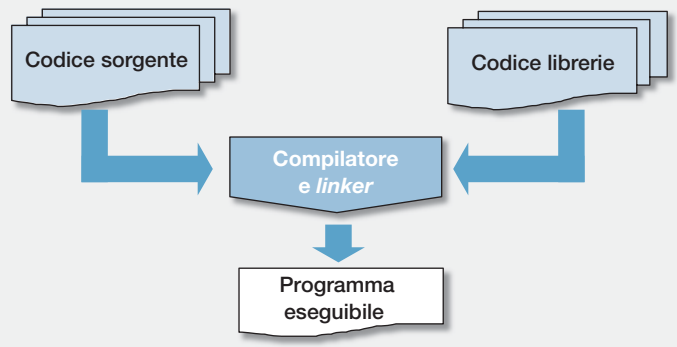


FIGURA 2

volta nella fase in cui viene costruito il programma eseguibile (*compile-time*). Nel secondo caso, invece, avviene a ogni esecuzione del programma (*run-time*): il codice oggetto viene direttamente eseguito e un *linker* dinamico gestito dal sistema operativo provvederà a unire il codice delle librerie quando necessario.

4.2 Approccio interpretato

Una diversa tecnica di traduzione del codice sorgente in codice oggetto consiste nel ricorso a un **interprete**. L'interprete è un programma che traduce ed esegue istruzione per istruzione il codice sorgente. Lo schema dell'approccio interpretato è quello di FIGURA **3**.

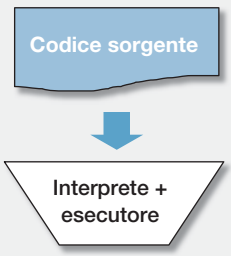


FIGURA **3**

Nell'approccio interpretato il collegamento con il codice delle librerie del linguaggio avviene in modalità dinamica.

OSSERVAZIONE Il fatto che l'interprete traduca le istruzioni singolarmente comporta che istruzioni ripetute (per esempio le istruzioni comprese in un ciclo) siano tradotte più volte; per questo motivo un programma interpretato è più lento in fase di esecuzione rispetto a un equivalente programma compilato, dove il processo di traduzione viene effettuato una sola volta in fase di compilazione.
Inoltre, dal momento che nel flusso di esecuzione di un programma non tutte le istruzioni sono sistematicamente eseguite (in funzione del verificarsi o meno delle condizioni dei costrutti di selezione) e che gli errori sintattici sono rilevati solo nel momento della traduzione, questo potrebbe avere come conseguenza che un programma interpretato manifesti errori di sintassi anche dopo molte esecuzioni corrette. Per mitigare questo problema molti linguaggi interpretati moderni hanno comunque un «compilatore» capace di verificare la correttezza sintattica dell'intero programma senza generare il codice oggetto.

OSSERVAZIONE Volendo commercializzare il software, è bene capire come la cessione di un programma interpretato preveda, da parte dell'utente finale, anche l'acquisto dell'interprete del linguaggio. Inoltre

– anche se esistono meccanismi di sicurezza previsti da specifici linguaggi interpretati – un programma interpretato viene fornito al cliente nella forma di codice sorgente con poche garanzie, in relazione alla proprietà intellettuale degli algoritmi e delle soluzioni tecniche adottate.

4.3 Un nuovo tipo di approccio

Un ulteriore e più recente approccio alla traduzione ed esecuzione dei programmi è quello seguito dai progettisti del linguaggio **Java**, il cui motto è «*write once, run anywhere*» («scrivi il codice una sola volta per eseguirlo dappertutto»). La portabilità dei programmi è sempre stato uno degli obiettivi perseguiti da chi progetta i linguaggi di programmazione: i programmi in linguaggio C/C++ (che pure rappresenta uno dei primi e migliori esempi di portabilità) devono essere ricompilati per essere eseguiti su computer o sistemi operativi di tipo diverso. Il compilatore del linguaggio Java, a partire dal codice sorgente, non genera codice oggetto per uno specifico processore, ma un codice per una macchina «virtuale» denominato ***byte-code***. Per ogni tipo di computer attualmente in commercio, praticamente, esiste poi un programma (denominato JVM, *Java Virtual Machine*) che simula la macchina virtuale in grado di eseguire il *byte-code*.
Il processo di compilazione ed esecuzione di un programma in linguaggio Java può essere schematizzato come in FIGURA **4**.

OSSERVAZIONE Con questa soluzione gli sviluppatori di software in linguaggio Java sono svincolati dalle preoccupazioni relative all'ambiente di esecuzione dei propri programmi: sarà infatti la specifica *Java Virtual Machine* del computer di esecuzione a garantire la corretta interpretazione del *byte-code*.

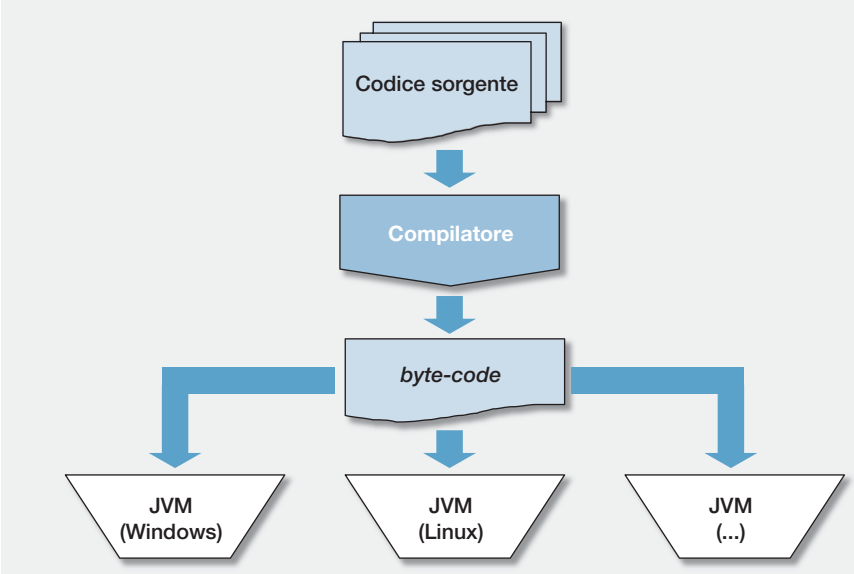


FIGURA **4**

Sintesi

Linguaggio di programmazione. Notazione formale per implementare algoritmi in modo tale da poter essere eseguiti su un computer.

Programma. Insieme di istruzioni che costituiscono la descrizione, eseguibile da un computer, di un algoritmo scritto secondo le regole di uno specifico linguaggio di programmazione.

Editor. Modulo software che permette al programmatore la scrittura e la modifica delle istruzioni che compongono un programma.

Linguaggio di basso livello. Linguaggio di programmazione le cui istruzioni sono molto vicine alle istruzioni macchina riconosciute dal processore.

Linguaggio di alto livello. Linguaggio di programmazione orientato alla soluzione di problemi secondo un modo di ragionare «umano», indipendente dalle caratteristiche dell'hardware del computer utilizzato. Questa indipendenza virtuale dal processore permette di rendere i programmi «portabili» su diverse piattaforme di esecuzione.

Programma sorgente. È l'insieme delle istruzioni scritte dal programmatore in un determinato linguaggio di programmazione per implementare un algoritmo.

Assemblatore. Strumento software che permette la traduzione di un programma scritto in *assembly* nelle corrispondenti istruzioni del linguaggio macchina.

Compilatore. Strumento software che permette la traduzione in linguaggio macchina di un programma scritto in un linguaggio di alto livello. Il processo di traduzione del programma avviene in un'unica soluzione.

Codice oggetto. Codice in linguaggio macchina ottenuto dalla compilazione del codice sorgente.

Libreria. Insieme di moduli software che implementano specifiche funzioni. Questi sono in genere forniti direttamente dal linguaggio di programmazione per aumentare la produttività del programmatore nella stesura dei programmi.

Linker. Strumento software che permette di unire in un unico programma più moduli oggetto ed eventualmente sezioni delle librerie del linguaggio che questi usano internamente.

Programma eseguibile. È il programma direttamente eseguibile dal computer. Esso è normalmente il risultato del processo compilazione/*linking*.

Run-time. Fase di esecuzione di un programma.

***Linking* statico/dinamico**. Nel *linking* statico l'unione dei vari moduli oggetto/librerie avviene prima della fase di *run-time* con la produzione di un modulo eseguibile monolitico. Nel *linking* dinamico l'unione dei vari moduli oggetto/librerie avviene a *run-time*.

Interprete. Modulo software che provvede a tradurre ed eseguire il codice sorgente. A differenza del compilatore l'interprete traduce (ed esegue) una istruzione alla volta.

Debugger. Strumento software che consente di eseguire passo per passo un programma, controllando le istruzioni effettivamente eseguite e ispezionando il contenuto delle variabili, al fine di scoprire ed eliminare gli errori che causano malfunzionamenti del programma.

Paradigma di programmazione. Insieme degli strumenti concettuali forniti da un determinato linguaggio per la codifica di un programma; definisce il modo con cui il programmatore concepisce il programma stesso.

QUESITI

1 Un linguaggio di programmazione è ...

A ... l'insieme delle istruzioni macchina riconosciute da uno specifico processore.

B ... un insieme di istruzioni scritte secondo le regole di un certo linguaggio di programmazione per implementare un algoritmo.

C ... un insieme di regole che definiscono una notazione formale per implementare algoritmi in modo tale da poter essere eseguiti da una macchina.

D Nessuna delle precedenti affermazioni.

2 Un programma è ...

A ... la stessa cosa di un algoritmo.

B ... un insieme di regole che definiscono una notazione formale per implementare algoritmi in modo tale da poter essere eseguiti da una macchina.

C ... un insieme di istruzioni scritte secondo le regole di un certo linguaggio di programmazione per implementare un algoritmo.

D Nessuna delle precedenti affermazioni.

3 Un assemblatore è ...

A ... un linguaggio simbolico di basso livello.

B ... un traduttore da programmi scritti in *assembly* simbolico a programmi in linguaggio macchina.

C ... un traduttore da programmi scritti in linguaggio macchina a programmi in *assembly* simbolico.

D Nessuna delle precedenti affermazioni.

4 Un compilatore è ...

A ... un linguaggio ad alto livello.

B ... un traduttore da programmi scritti in un linguaggio ad alto livello a programmi in linguaggio macchina.

C ... un traduttore da programmi scritti in linguaggio ad alto livello a programmi in *assembly* simbolico.

D Nessuna delle precedenti affermazioni.

5 Un interprete è ...

A ... un traduttore/esecutore da programmi scritti in un linguaggio ad alto livello a programmi in linguaggio macchina.

B ... un traduttore da programmi scritti in un linguaggio ad alto livello a programmi in linguaggio macchina.

C ... un esecutore di programmi in *assembly* simbolico.

D Nessuna delle precedenti affermazioni.

6 Quali delle seguenti affermazioni sono corrette?

A A *run-time*, un linguaggio interpretato è più veloce di uno compilato.

B A *run-time*, un linguaggio compilato è più veloce di uno interpretato.

C A *run-time*, un linguaggio interpretato è veloce quanto uno compilato.

D La velocità di un linguaggio dipende esclusivamente dall'hardware utilizzato.

7 Un *linker* è ...

A ... un traduttore/esecutore da programmi scritti in un linguaggio ad alto livello a programmi in linguaggio macchina.

B ... un traduttore da programmi scritti in un linguaggio ad alto livello a programmi in linguaggio macchina.

C ... un esecutore di programmi scritti in un linguaggio di medio livello.

D Nessuna delle precedenti affermazioni.

8 Un programma sorgente è ...

A ... un insieme di istruzioni scritte dal programmatore in un certo linguaggio di programmazione.

B ... un insieme di istruzioni in linguaggio macchina generate da un compilatore o un assemblatore.

C ... un insieme di istruzioni in linguaggio macchina generate da un *linker*.

D Nessuna delle precedenti affermazioni.

9 Un programma oggetto è ...

A ... un insieme di istruzioni in linguaggio macchina generate da un *linker*.

B ... un insieme di istruzioni scritte dal programmatore in un certo linguaggio di programmazione.

C ... un insieme di istruzioni in linguaggio macchina generate da un compilatore o un assemblatore.

D Nessuna delle precedenti affermazioni.

10 **Un editor è ...**

A ... un termine tecnico per indicare il programmatore stesso, ovvero colui che edita un programma.

B ... uno strumento software per la scrittura/modifica di programmi sorgente.

C ... uno strumento software per la scrittura/modifica di programmi oggetto.

D Nessuna delle precedenti affermazioni.

11 **Una libreria di un linguaggio è ...**

A ... un insieme di funzioni già predisposte che il programmatore può richiamare dai suoi programmi.

B ... un insieme di regole che definiscono una notazione formale per implementare algoritmi in modo tale da poter essere eseguiti da una macchina.

C ... un modulo software che serve al compilatore per poter effettuare la traduzione dei programmi sorgente.

D Nessuna delle precedenti affermazioni.

12 **La funzione di un *linker* è quella di ...**

A ... unire tra loro più programmi oggetto e le librerie del linguaggio.

B ... unire tra loro un programma sorgente con le librerie del linguaggio.

C ... tradurre ed eseguire un programma scritto in un linguaggio ad alto livello.

D Nessuna delle precedenti affermazioni.

13 **Quali delle seguenti affermazioni sono corrette?**

A Un compilatore traduce in un'unica soluzione tutto un programma.

B Un interprete traduce ed esegue un programma una istruzione alla volta.

C Un compilatore traduce ed esegue un programma una istruzione alla volta.

D Un interprete traduce in un'unica soluzione tutto un programma.

E Un *linker* unisce sempre programmi oggetto e librerie producendo un programma eseguibile.

F Un *linker* può unire programmi oggetto e librerie anche a *run-time*.

14 **Un *debugger* è ...**

A ... uno strumento software che permette al programmatore di eseguire passo-passo le istruzioni di un programma al fine di individuare eventuali *bug*.

B ... uno strumento software che permette al programmatore di tradurre passo-passo le istruzioni di un programma al fine di individuare eventuali errori sintattici.

C ... uno strumento software che permette al programmatore di modificare le istruzioni di un programma al fine di rimuovere eventuali *bug* individuati a *run-time*.

D Nessuna delle precedenti affermazioni.

15 **Indicare l'ordine corretto delle fasi di sviluppo di un programma compilato.**

A *Editing*, compilazione, *linking*, esecuzione, *debugging*.

B *Editing*, *linking*, compilazione, esecuzione, *debugging*.

C *Editing*, compilazione, *debugging*, *linking*, esecuzione.

D Nessuna delle precedenti affermazioni.

16 **Un paradigma di programmazione definisce ...**

A ... se un linguaggio adotta un approccio compilato o interpretato.

B ... le modalità attraverso le quali il programmatore concepisce i programmi mediante l'uso degli strumenti concettuali forniti da un determinato linguaggio per lo sviluppo delle applicazioni.

C ... le modalità operative offerte da un linguaggio attraverso le quali il programmatore opera per individuare gli eventuali errori semantici dei suoi programmi.

D Nessuna delle precedenti affermazioni.

bug
un errore in un programma

syntax
strutture di un programma

semantics
significato
di un programma

syntax error
errore sintattico
in un programma
che impedisce
la compilazione

run-time error
errore rilevato
in fase di esecuzione
di un programma

logical error
errore in un programma
che lo fa comportare
in modo diverso
da come il programmatore
si aspetterebbe

debugging
processo di ricerca
e rimozione degli errori

1.3 What is debugging?

Programming is a complex process, and since it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called *bugs* and the process of tracking them down and correcting them is called *debugging*. There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

1.3.1 Compile-time errors

The compiler can only translate a program if the program is syntactically correct; otherwise, the compilation fails and you will not be able to run your program. Syntax refers to the structure of your program and the rules about that structure.
For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a syntax error. So does this one.
For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of e e cummings without spewing error messages.
Compilers are not so forgiving. If there is a single syntax error anywhere in your program, the compiler will print an error message and quit, and you will not be able to run your program.
To make matters worse, there are more syntax rules in C++ than there are in English, and the error messages you get from the compiler are often not very helpful. During the first few weeks of your programming career, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

1.3.2 Run-time errors

The second type of error is a run-time error, so-called because the error does not appear until you run the program. For the simple sorts of programs we will be writing for the next few weeks, run-time errors are rare, so it might be a little while before you encounter one.

1.3.3 Logic errors and semantics

The third type of error is the logical or semantic error. If there is a logical error in your program, it will compile and run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. Specifically, it will do what you told it to do. The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying logical errors can be tricky, since it requires you to work

backwards by looking at the output of the program and trying to figure out what it is doing.

1.3.4 Experimental debugging

One of the most important skills you should acquire [...] is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming. In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that lead to the results you see. Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. [...]

[Allen B. Downey, *How to think like a computer scientist*, 1999]

QUESTIONS

a What is debugging?

b Which are the different kinds of errors that can occur in a program?

c Which are the differences between a syntax error and a semantics error?

d Why the name "loop"?

statement
istruzione

squiggly-braces
parentesi graffe

stickler
pedante: eccessivamente meticoloso ed osservante di tutte le regole

1.5 The first program

Traditionally the first program people write in a new language is called "Hello, World." because all it does is print the words "Hello, World." In C++, this program looks like this:

```
#include <iostream.h>
// main: generate some simple output
void main ()
{
  cout << "Hello, world." << endl;
  return 0
}
```

Some people judge the quality of a programming language by the simplicity of the "Hello, World." program. By this standard, C++ does reasonably well. Even so, this simple program contains several features that are hard to explain to novice programmers. For now, we will ignore some of them, like the first line. The second line begins with //, which indicates that it is a **comment**. A comment is a bit of English text that you can put in the middle of a program, usually to explain what the program does. When the compiler sees a //, it ignores everything from there until the end of the line. In the third line, you can ignore the word **`void`** for now, but notice the word `main`. `main` is a special name that indicates the place in the program where execution begins. When the program runs, it starts by executing the first statement in `main` and it continues, in order, until it gets to the last statement, and then it quits. There is no limit to the number of statements that can be in `main`, but the example contains only one. It is a basic **output** statement, meaning that it outputs or displays a message on the screen. `cout` is a special object provided by the system to allow you to send output to the screen. The symbol **<<** is an **operator** that you apply to cout and a string, and that causes the string to be displayed. `endl` is a special symbol that represents the end of a line. When you send an `endl` to `cout`, it causes the cursor to move to the next line of the display.

The next time you output something, the new text appears on the next line. Like all statements, the output statement ends with a semi-colon (;). There are a few other things you should notice about the syntax of this program. First, C++ uses squiggly-braces ({ and }) to group things together.

In this case, the output statement is enclosed in squiggly-braces, indicating that it is inside the definition of main. Also, notice that the statement is indented, which helps to show visually which lines are inside the definition.

At this point it would be a good idea to sit down in front of a computer and compile and run this program. […] the C++ compiler is a real stickler for syntax. If you make any errors when you type in the program, chances are

that it will not compile successfully. For example, if you misspell `iostream`, you might get an error message like the following:

```
hello.cpp:1: oistream.h: No such file or directory
```

There is a lot of information on this line, but it is presented in a dense format that is not easy to interpret. A more friendly compiler might say something like:
"On line 1 of the source code file named hello.cpp, you tried to include a header file named oistream.h. I didn't find anything with that name, but I did find something named iostream.h. Is that what you meant, by any chance?"
Unfortunately, few compilers are so accomodating. The compiler is not really very smart, and in most cases the error message you get will be only a hint about what is wrong. [...]
Nevertheless, the compiler can be a useful tool for learning the syntax rules of a language. Starting with a working program (like hello.cpp), modify it in various ways and see what happens. If you get an error message, try to remember what the message says and what caused it, so if you see it again in the future you will know what it means.

[Allen B. Downey, *How to think like a computer scientist*, 1999]

QUESTIONS

a What is main in a C++ program?

b What is a line that begins with //?

c How would you define the behaviour of a compiler to translate a program? [This is unclear, does translate mean compile, or interpret, or correct mistakes?]

d Why does C++ uses squiggly-braces?

A4 Il linguaggio di programmazione C++

Prendiamo nuovamente in considerazione l'algoritmo per il calcolo del giorno della settimana corrispondente a una data. Sia nel caso di rappresentazione in termini di NLS sia di DAB, affinché esso possa essere eseguito da un computer, è necessario operare una sua traduzione (implementazione) in un linguaggio di programmazione.

Il linguaggio di programmazione C++

Il linguaggio di programmazione C++ deriva dal linguaggio C. Nasce infatti nei laboratori «Bell» di AT&T nel 1983 per opera di Bjarne Stroustrup, che lo concepì come estensione del linguaggio C, includendovi il nuovo paradigma della programmazione OOP (*Object Oriented Programming*), che si andava affermando in quegli anni.

Il linguaggio C++ è molto diffuso e apprezzato sia per la sua ricchezza semantica sia per la grande varietà di librerie disponibili, che lo rendono un linguaggio molto espressivo e potente, ma anche complesso: al programmatore è richiesto molto tempo per poterlo padroneggiare completamente.

In alcuni casi la variabilità del comportamento dei vari tipi di compilatori nel gestirne le funzionalità avanzate sacrifica la portabilità dei programmi tra varie piattaforme, limitandone l'uso a specifiche architetture hardware e/o software.

ESEMPIO

Il seguente programma è una possibile implementazione dell'algoritmo nel linguaggio di programmazione C++:

```
#include <iostream>

using namespace std;

void main(void)
{
  int anno, mese, giorno, Y, M, D;

  cin>>anno;
  cin>>mese;
  cin>>giorno;
  Y = anno;
  M = mese-2;
  if (M <= 0)
    {
      Y = Y-1;
      M = M+12;
    }
  D = (giorno+Y+(Y/4) - (Y/100) + (Y/400) + (31*M/12)) % 7);
  cout<<D;
}
```

OSSERVAZIONE Per quanto simili strutturalmente, vi sono numerose differenze tra il programma C++ e la NLS riportata in apertura del capitolo precedente:

- il programma inizia con `void main(void)`;
- le variabili *anno*, *mese*, *giorno*, *Y*, *M* e *D* sono introdotte con la «dichiarazione» `int anno, mese, giorno, Y, M, D;` come variabili adatte a contenere valori numerici interi;

- ogni istruzione termina col simbolo «;»;
- le parole chiave ***Inizio*** e ***Fine*** sono sostituite rispettivamente dai simboli «{» e «}»;
- l'operatore di assegnamento «←» è sostituito dall'omologo operatore «=»;
- l'istruzione di selezione ***Se*** … ***allora*** è sostituita dall'istruzione `if (…);`
- l'istruzione di input ***Leggi*** è sostituita dall'istruzione `cin>>...;`
- nelle operazioni in cui sono previste delle divisioni intere è stata eliminata la funzione ***Int*** (essendo le variabili interessate limitate a contenere valori numerici interi, anche il risultato calcolato è un valore intero);
- l'istruzione di output ***Scrivi*** viene sostituita da `cout<<...`.

In questo capitolo saranno presentate le regole da osservare per implementare un algoritmo mediante un programma in linguaggio C++.

1 Struttura fondamentale di un programma

Ogni programma codificato in linguaggio C++ deve comprendere una «funzione» principale denominata ***main***.
Nel seguito si analizza il programma dell'esempio precedente per individuare quali sono le parti fondamentali della sua articolazione:

`/* Programma per determinare il giorno della settimana a cui fa riferimento una data: il valore restituito varia da 0 a 6 con il seguente significato: 0=Domenica, 1=Lunedi,..., 6=Sabato. */`	Commento iniziale che riassume lo scopo e la codifica dei valori di input/output del programma.
`#include <iostream>`	Istruzioni di precompilazione: sono le direttive per il preprocessore che, in questo caso, specificano l'inclusione del file di intestazione *iostream* contenente la definizione delle funzioni di libreria utilizzate dal programma; file come questo includono solitamente la definizione di elementi utili al programmatore.
`using namespace std;`	Dichiarazione di uso dello spazio dei nomi standard; gli spazi dei nomi permettono di evitare conflitti, per esempio dichiarando due variabili distinte aventi lo stesso nome.
`void main(void)`	Dichiarazione della funzione principale. ▶

Preprocessing

Il preprocessore è un programma che effettua sostituzioni testuali nel codice sorgente di un programma, procedimento noto come *preprocessing*.
I più comuni tipi di sostituzioni consistono nell'espansione di «macro», nell'inclusione di file e nella compilazione condizionale.
Tipicamente il preprocessore viene invocato automaticamente dal compilatore prima del processo di compilazione vero e proprio: il file risultante costituirà l'input del compilatore stesso.
Il C++ è dotato di un preprocessore, analogo a quello del linguaggio C, che viene utilizzato per includere i file di dichiarazione delle funzioni di libreria e per definire costanti simboliche.

`{`	Inizio del blocco di codice principale.
`// dichiarazione variabili` `int anno, mese, giorno, Y, M, D;`	Dichiarazione delle variabili utilizzate nel programma con indicazione del loro «tipo».
`// input elementi della data` `cin>>anno;` `cin>>mese;` `cin>>giorno;` `// inizializzazione variabili` `Y = anno;` `M = mese-2;` `if (mese <= 0)` `{` `Y = Y-1;` `M = M+12;` `}` `// calcolo giorno seriale` `D = giorno + Y + Y/4 - Y/100 +` `Y/400 + 31*M/12;` `// calcolo giorno in modulo 7` `D = D-D/7*7;` `// output risultato` `cout<<D;`	Sezione procedurale: è relativa all'implementazione dell'algoritmo.
`}`	Fine del blocco di codice principale e fine del codice del programma.

1.1 Blocchi di istruzioni

Dopo l'intestazione `void main(void)` si trova il «corpo» della funzione principale: i simboli «{» e «}» delimitano un **blocco di istruzioni** monolitico che può essere relativo al corpo di una funzione, come in questo caso, oppure comprendere istruzioni di codice successive a una istruzione di selezione o iterazione che dipendono dalla stessa istruzione di selezione o di iterazione.

ESEMPIO

Nel codice del programma di esempio l'esecuzione delle istruzioni `Y = Y-1;` e `M = M+12;` è condizionata dal fatto che la variabile *mese* assuma un valore minore o uguale a 2.

OSSERVAZIONE In generale le istruzioni che appartengono a un blocco costituiscono una sequenza da considerarsi come un'unica istruzione; inoltre un blocco può contenere a sua volta blocchi di istruzioni.

Per facilitare la leggibilità e la comprensione del codice, la maggior parte dei programmatori ricorre all'**indentazione delle istruzioni**: la convenzione fondamentale è quella di far rientrare di uno o più spazi ogni nuovo blocco dipendente da un'istruzione condizionale o di iterazione.

ESEMPIO

Nel codice del programma di esempio tutte le istruzioni sono rientrate rispetto alle parentesi che delimitano il corpo della funzione principale, mentre il blocco di istruzioni condizionate è rientrato rispetto all'istruzione di selezione.

1.2 Commenti di un programma

L'inserimento di commenti nel codice di un programma costituisce un elemento fondamentale per una sua più immediata comprensione futura o da parte di altri programmatori. Esistono due modalità per inserire commenti:

- **commento su linea singola**: inizia con il simbolo «`//`» e termina con la fine della riga stessa;
- **commento multilinea**: inizia con il simbolo «`/*`» e termina con il simbolo «`*/`».

Il compilatore ignora i commenti, che pertanto hanno una valenza puramente documentativa per chi sviluppa il codice.

2 Variabili e costanti

Nel capitolo dedicato agli algoritmi è stato introdotto il concetto di variabile.

In un programma C++ tutte le variabili devono essere dichiarate prima di poter essere utilizzate.

ESEMPIO

Nel codice del programma di esempio con la dichiarazione

```
int anno, mese, giorno, Y, M, D;
```

sono definite le variabili *anno*, *mese*, *giorno*, *Y*, *M* e *D* come contenitori per valori numerici esclusivamente interi.

2.1 Uso di variabili e tipi fondamentali

Quando un programma viene eseguito, le variabili che utilizza sono registrate nella memoria del computer; è importante che il programmatore abbia consapevolezza del tipo di dato che intende conservare in una variabile, perché memorizzare un carattere piuttosto che un valore numerico intero, o un valore numerico non intero, richiede spazi di memoria diversi, dal momento che i vari tipi di dato non hanno la stessa dimensione.
La memoria del computer è organizzata in byte: un byte rappresenta la quantità di memoria minima che è possibile gestire in un programma

C++, ma un solo byte può servire a memorizzare un singolo carattere, o al massimo un piccolo valore numerico intero. Per gestire dati più complessi, come grandi valori numerici interi, o valori numerici non interi, si devono utilizzare variabili che occupano nella memoria del computer sequenze di più byte consecutivi.

Il **tipo** di una variabile (o di una costante) definisce quindi l'insieme di valori che questa può assumere, determinando al tempo stesso l'insieme di operazioni che a essa possono essere applicate.

Nella TABELLA **1** sono riportati i principali tipi di dato previsti dal linguaggio C++.

TABELLA **1**

Tipo	Descrizione	Dimensione (byte)	Intervallo numerico
`char`	Singolo carattere numero intero *small*	1	`signed`: da –128 a 127 `unsigned`: da 0 a 255
`short int`	Numero intero *short*	2	`signed`: da –32 768 a 32 767 `unsigned`: da 0 a 65 535
`int`	Numero intero		Dipendente dal compilatore
`long int`	Numero intero *long*	4	`signed`: da –2 147 483 648 a 2 147 483 647 `unsigned`: da 0 a 4 294 967 295
`bool`	Valore booleano	1	`true`/`false`
`float`	Numero *floating-point*	4	da -3.4×10^{-38} a 3.4×10^{38}
`double`	Numero *floating-point* a doppia precisione	8	da -1.7×10^{-38} a 1.7×10^{38}

OSSERVAZIONE La dimensione del tipo `int` dipende dallo specifico compilatore che lo può implementare utilizzando 2, 4 o 8 byte. Normalmente essa coincide con la dimensione dei registri interni del processore, in modo che il ricorso a variabili di tipo `int` garantisca le massime prestazioni sulla specifica piattaforma di esecuzione.

Come risulta evidente dalla tabella precedente, l'intervallo dei valori che un determinato tipo di variabile numerica può contenere è condizionato dal fatto che la variabile stessa sia dichiarata o meno in modalità `unsigned` (`signed` è la modalità predefinita e non è necessario che sia esplicitata).

ESEMPIO

La dichiarazione

```
unsigned int alfa;
```

definisce una variabile denominata `alfa` che può contenere valori numerici interi «senza segno» compreso tra 0 e 4 294 967 295.

OSSERVAZIONE I valori numerici non interi sono rappresentati con la notazione anglosassone, ovvero utilizzando il punto come separatore decimale (per esempio 3.1416). Inoltre i valori costanti con cifre decimali sono normalmente interpretati come valori di tipo `double`, a meno di non specificarne esplicitamente il tipo utilizzando il simbolo finale «F» (per esempio 3.1415F, se si vuole esplicitamente indicare che il valore deve essere interpretato come un tipo `float`).

OSSERVAZIONE I valori costanti di tipo numerico non intero possono essere definiti in notazione esponenziale utilizzando il simbolo «E» (per esempio 6.022E23, che assume il valore di 6.022×10^{23}).

Nel caso in cui, nel corso della valutazione di un'espressione, si assegni a una variabile un valore esterno all'intervallo che il tipo con cui è stata dichiarata consente di rappresentare, i programmi in linguaggio C++ non segnalano errori e proseguono – con valori errati – l'esecuzione.

ESEMPIO

L'output fornito dal seguente frammento di codice C++:

```
…
short int i;
i = 40000;
cout<<i;
…
```

non è 40 000, ma –13 108! Infatti il valore numerico 40 000 non ricade nell'intervallo di rappresentazione del tipo `short int`, che comprende i valori interi tra 32 767 e –32 768, e causa un errore apparentemente inspiegabile.

Il tipo **char** è utilizzato per rappresentare singoli caratteri alfanumerici i cui valori costanti sono sempre compresi tra due simboli «`'`».

ESEMPIO

Il seguente frammento di codice visualizza la sequenza di caratteri «Ciao!»:

```
…
char c;
c='C';
cout<<c;
c='i';
cout<<c;
c='a';
cout<<c;
c='o';
cout<<c;
c='!';
cout<<c;
…
```

È possibile memorizzare in una variabile di tipo `char` piccoli valori numerici interi ed eseguire su di essi operazioni aritmetiche.

OSSERVAZIONE Utilizzando il tipo `char` accade di effettuare operazioni aritmetiche con caratteri, ma in realtà si opera sui codici numerici che internamente codificano i caratteri stessi[1].

1. Il linguaggio C++ codifica numericamente i caratteri alfanumerici secondo lo standard ASCII.

ESEMPIO

Il seguente frammento di codice

```
…
char c;
c='A';
c=c+3;
cout<<c;
…
```

visualizza come output il carattere «D».

Per i nomi delle variabili è possibile usare sia lettere maiuscole sia minuscole e cifre numeriche, ma in ogni caso essi devono iniziare con una lettera e non sono ammessi spazi bianchi, o caratteri «speciali», a eccezione del simbolo «_».

OSSERVAZIONE Il linguaggio C++ è *case sensitive*: due variabili denominate *x* e *X* sono per il compilatore a tutti gli effetti due variabili distinte. Allo stesso modo le parole chiave del linguaggio devono essere codificate usando esclusivamente caratteri minuscoli, altrimenti non saranno interpretate come tali da parte del compilatore.

OSSERVAZIONE Il linguaggio C++ non impone regole di stile per la denominazione delle variabili, ma è sempre bene che il programmatore le definisca utilizzando nomi significativi per garantire la leggibilità del codice: l'uso del nome *areaQuadrato* è senz'altro meno oscuro di *Aq*. A questo scopo il simbolo «_» viene spesso impiegato dai programmatori per costruire nomi di variabili apparentemente composti da più parole, come *data_nascita*.

OSSERVAZIONE È buona regola inizializzare una variabile con un valore prima di utilizzarla perché, in caso contrario, non è noto al programmatore il valore contenuto nell'area di memoria utilizzata per la memorizzazione della variabile stessa e, impiegandola in un'espressione, si otterranno risultati imprevisti e non corretti.

L'**inizializzazione**, o – più in generale – la modifica del valore di una variabile, può essere effettuata in C++ mediante l'operatore di assegnamento «=»: a sinistra dell'operatore di assegnamento vi è sempre una variabile, mentre a destra può esserci un valore costante o un'espressione comunque complessa che il linguaggio consente di valutare per ottenere un risultato il cui valore viene assegnato alla variabile.

ESEMPIO

I seguenti frammenti di codice illustrano modi corretti di assegnare un valore a una variabile:

```
…
int alfa, beta, gamma;
…
alfa = 5;
…
beta = alfa;
…
gamma = (alfa+3)*(beta-2);
…
```

OSSERVAZIONE L'inizializzazione di una variabile può avvenire anche contestualmente alla sua dichiarazione. Per esempio la dichiarazione

```
int delta=3, teta=0;
```

definisce le variabili intere *delta* e *teta* inizializzate rispettivamente ai valori 3 e 0.

È inoltre possibile assegnare un valore a una variabile mediante un'operazione di input.

ESEMPIO

L'istruzione

```
cin>>anno;
```

acquisisce un valore digitato dall'utente del programma e lo assegna alla variabile *anno*.

2.2 Uso di costanti

Il linguaggio C++ permette – mediante la parola chiave `const` – la definizione di costanti. Nel corso dell'esecuzione di un programma il valore di una costante non può essere modificato.

ESEMPIO

La dichiarazione

```
const float pi_greco = 3.14;
```

definisce la costante *floating-point* in singola precisione *pi_greco* il cui valore, non ulteriormente modificabile, viene inizializzato a 3.14.

OSSERVAZIONE Per comprendere la valenza dell'uso di una costante si immagini di avere un programma dove in diversi punti del codice sia richiesto l'uso del valore π.

- L'uso del nome simbolico *pi_greco* al posto della costante numerica `3.14` rende maggiormente leggibili le espressioni dove si utilizza π.

- Volendo aumentare la precisione dei calcoli, è sufficiente modificare la sola dichiarazione della costante, aggiungendo ulteriori cifre decimali senza modificare tale valore in tutte le singole espressioni che lo utilizzano.

2.3 Visibilità e classe di memorizzazione di variabili e costanti

Relativamente alle variabili e alle costanti si parla di *ambito di visibilità* (*scope*) e *tempo di vita* (*lifetime*) riferendoci rispettivamente all'ambito di azione e alla loro classe di memorizzazione.

▶ L'**ambito di visibilità** di una variabile è il blocco delle istruzioni in cui la variabile stessa è dichiarata.

L'ambito di visibilità di una variabile è, in pratica, l'area del programma in cui tale variabile è valida: si tratta delle istruzioni del blocco in cui essa è dichiarata, comprese le istruzioni dei blocchi interni al blocco stesso.

OSSERVAZIONE I blocchi possono essere **nidificati** uno dentro l'altro; naturalmente nella nidificazione dei blocchi il numero delle parentesi chiuse deve bilanciare quello delle parentesi aperte, inoltre ogni parentesi chiusa termina il blocco iniziato dalla parentesi aperta più interna.

È possibile classificare le variabili e le costanti in base alla loro visibilità, come indicato nel seguito.

- **Variabile globale**: è valida dal punto in cui è dichiarata fino al termine del codice contenuto in un singolo file. Una variabile è globale – cioè visibile in tutto il codice – solo se è definita esternamente a qualsiasi blocco di istruzioni.

OSSERVAZIONE

Le dichiarazioni (e le eventuali inizializzazioni) sono le uniche istruzioni del linguaggio C++ che possono essere collocate esternamente ai blocchi.

- **Variabile locale**: è visibile limitatamente al blocco in cui essa è dichiarata e non è possibile accedervi (per modificarne o utilizzarne il valore) al di fuori di esso.

È possibile dichiarare una variabile locale con lo stesso nome di una variabile globale; in questo caso la variabile locale, all'interno del blocco di definizione, nasconde la variabile globale.

Variabili condivise dal codice di file distinti

Nel linguaggio C++ la visibilità di una variabile globale è limitata al codice del file in cui è dichiarata.

I programmi complessi sono però costituiti da numerose funzioni organizzate in file distinti: per riferire una variabile globale dichiarata in un file diverso è possibile ripeterne la dichiarazione (senza inizializzazione) preceduta dalla parola chiave `extern`, altrimenti la coincidenza del nome genererà comunque variabili distinte.

Il programmatore che intenda impedire il riferimento esterno alle variabili globali che definisce deve premetterne la dichiarazione con la parola chiave `static`.

Con una ambiguità da molti criticata, questa stessa parola chiave consente di rendere permanente una variabile locale.

È anche possibile annidare dichiarazioni locali (in blocchi di istruzioni nidificati) per nascondere le variabili dichiarate nei blocchi più esterni, ma queste tecniche di codifica portano a realizzare programmi il cui codice è di difficile comprensione.

Relativamente alla classe di memorizzazione, una variabile può essere *permanente* o *temporanea*.

- **Variabile permanente**: *una variabile globale è sempre permanente*; essa è creata e inizializzata prima dell'avvio del programma e rimane attiva fino al termine dell'esecuzione.
- **Variabile temporanea**: *viene creata nella memoria del computer all'inizio dell'esecuzione delle istruzioni del blocco che ne comprende la dichiarazione*. Lo spazio di memoria occupato dalle variabili temporanee viene liberato al termine dell'esecuzione delle istruzioni del blocco e le variabili che esso comprende sono distrutte insieme con i valori contenuti. Inoltre, ogni volta che si eseguono le istruzione di un blocco, le variabili dichiarate all'interno del blocco stesso vengono nuovamente inizializzate.

ESEMPIO

Nel seguente programma:

```
void main(void)
{
  int x=1, y, z;

  cin>>y;
  cin>>z;
  cout<<x;

  if (y>=z)
    {
      int x;
      x=y+z;
      cout<<x;
    }
  cout<<x;
}
```

supponendo che l'utente assegni alla variabile `y` il valore 2 e alla variabile `z` il valore 3, l'output sarà:

```
1
1
```

Assegnando invece alla variabile `y` il valore 3 e alla variabile `z` il valore 2, l'output sarà:

```
1
5
1
```

Infatti nel primo caso le istruzioni del blocco condizionato dall'istruzione `if` non sono eseguite e l'output è determinato esclusivamente dalla prima e dall'ultima istruzione `cout`. Nel secondo caso sono eseguite tutte le istruzioni `cout` del programma; inoltre è dichiarata nel blocco condizionato dall'istruzione `if` una variabile locale `x` avente lo stesso nome di una variabile dichiarata in un blocco più esterno: quando sono eseguite le istruzioni del blocco interno, la variabile esterna viene nascosta dall'omonima variabile locale che assume il valore dell'espressione `y + z` e immediatamente visualizzato. Non appena l'esecuzione del blocco interno termina, la variabile `x` dichiarata nel blocco esterno diviene nuovamente visibile, mantenendo il valore assegnato prima dell'esecuzione del blocco interno.

3 Espressioni e condizioni

3.1 Operatori algebrici ed espressioni

Gli operatori algebrici più comuni utilizzati nelle espressioni sono riepilogati nella TABELLA **2**.

TABELLA **2**

Operatore	Descrizione
+	Somma o segno
–	Sottrazione o segno
*	Moltiplicazione
/	Divisione
%	Modulo (resto della divisione)

OSSERVAZIONE

Nella valutazione delle espressioni algebriche la precedenza degli operatori è quella classica: prima sono valutati gli operatori di segno delle singole variabili o costanti numeriche, successivamente sono valutate le operazioni di moltiplicazione, divisione e modulo e solo per ultime le operazioni di addizione e sottrazione. Questi criteri di precedenza possono essere modificati con il ricorso ai simboli delle parentesi «(» e «)».

ESEMPIO

Data la dichiarazione

```
int x=5, y=3, z=2, alfa;
```

con l'istruzione

```
alfa = x+y*z;
```

alla variabile *alfa* viene assegnato il valore 11, mentre con l'istruzione

```
alfa = (x+y)*z;
```

alla variabile *alfa* viene assegnato il valore 16.
Dopo l'esecuzione dell'istruzione

```
alfa = (x+(y+z)*3)*2;
```

il valore della variabile *alfa* sarà 40.

Oltre agli operatori appena visti, il linguaggio C++ rende disponibili gli operatori di **autoincremento** ++ e di **autodecremento** – – delle variabili, che possono essere utilizzati per incrementare/decrementare di una unità le variabili numeriche.

Per quanto ridondanti, sono molto utilizzati dai programmatori, perché hanno il pregio di rendere il codice molto compatto, anche se il loro abuso lo rende di difficile interpretazione.

ESEMPIO

L'istruzione

```
alfa++;
```

equivale all'assegnamento

```
alfa=alfa+1;
```

mentre l'istruzione

```
alfa--;
```

equivale a

```
alfa=alfa-1;
```

Questi operatori hanno due diverse notazioni: la *notazione prefissa*, raramente utilizzata, stabilisce che l'autoincremento/autodecremento avvenga prima della valutazione dell'espressione, e la più comune *notazione suffissa*, in cui l'autoincremento/autodecremento avviene dopo la valutazione dell'espressione.

ESEMPIO

L'istruzione (incremento postfisso)

```
cout<<alfa++;
```

è equivalente alla sequenza di istruzioni

```
cout<<alfa;
alfa = alfa+1;
```

mentre l'istruzione (incremento prefisso)

```
cout<<++beta;
```

equivale alle seguente sequenza di istruzioni:

```
beta = beta+1;
cout<<beta;
```

L'istruzione complessa

```
beta = ++delta + alfa--;
```

equivale alla sequenza di istruzioni

```
delta = delta+1;
beta = delta + alfa;
alfa = alfa-1;
```

OSSERVAZIONE Il linguaggio C++ mette a disposizione anche i seguenti operatori «compatti»: `+=`, `-=`, `*=` e `/=`. Il loro uso è esemplificato dall'istruzione

```
alfa += beta;
```

equivalente alla seguente:

```
alfa = alfa + beta;
```

3.2 La formulazione delle condizioni logiche

Gli operatori di relazione del linguaggio C++ utilizzabili per costruire le condizioni logiche sono elencati nella TABELLA **3**.

TABELLA **3**

Operatore	Descrizione
==	Uguale
<	Minore
<=	Minore o uguale
>	Maggiore
>=	Maggiore o uguale
!=	Diverso

OSSERVAZIONE Si noti che l'operatore di uguaglianza è il simbolo «==» e non «=». Dato che in caso di errore il compilatore genera un semplice avviso, ma non un messaggio di errore, è importante porre molta attenzione nella digitazione delle condizioni logiche.

Inoltre le condizioni logiche possono essere combinate in un'espressione logica – che assume il solo valore `true` o `false` – mediante i seguenti operatori booleani

Operatore	Descrizione
!	NOT (negazione)
&&	AND (congiunzione logica)
||	OR (disgiunzione logica)

la cui funzione è illustrata nelle seguenti **tabelle di verità**, riferite alle ipotetiche variabili booleane *x* e *y*:

x	!x
true	false
false	true

x	y	x || y
false	false	false
false	true	true
true	false	true
true	true	true

x	y	x && y
false	false	false
false	true	false
true	false	false
true	true	true

ESEMPIO

L'espressione logica

```
(alfa>=5)&&(alfa<=10)
```

risulterà `true` per valori della variabile numerica *alfa* compresi tra 5 e 10 inclusi e `false` per tutti gli altri valori.

OSSERVAZIONE Il linguaggio C++ ha ereditato dal linguaggio C il fatto che il valore di una variabile numerica intera viene interpretato come un valore booleano: il valore 0 corrisponde a `false`, mentre un qualsiasi altro valore equivale a `true`.

3.3 *Casting*

▶ La conversione di un'espressione di un certo tipo in un tipo diverso è denominata ***casting* del tipo**; il *casting* può essere implicito o esplicito.

Le **conversioni implicite** non richiedono alcun operatore per essere eseguite: esse sono eseguite automaticamente quando un valore viene utilizzato come un tipo compatibile.

ESEMPIO

Nel seguente frammento di programma

```
int alfa=2000;
float beta=alfa;
```

il valore contenuto nella variabile *alfa* viene automaticamente promosso dal tipo `int` al tipo `float` senza specificare alcuna operazione di *casting*.

Le conversioni implicite hanno effetto sui tipi fondamentali di dati e consentono trasformazioni come quelle tra tipi numerici. In generale vale la **regola del più forte**: se gli operandi di un'espressione sono di tipo diverso tra loro, il risultato sarà del tipo più forte, cioè quello in grado di memorizzare un intervallo più esteso.

OSSERVAZIONE Alcune conversioni – per esempio la conversione da un valore di tipo `float` a un valore di tipo `int` – possono implicare una perdita di precisione numerica che il compilatore segnala con un avviso, ma non con un errore. Questa situazione può spesso essere evitata applicando opportunamente un *casting* esplicito.

Il linguaggio di programmazione C++ è un linguaggio «fortemente tipizzato»: ogni variabile ha un tipo che non può essere modificato nel corso

dell'esecuzione del programma. Di conseguenza è spesso necessario operare **conversioni esplicite**; esistono due diverse notazioni per effettuare il *casting* esplicito, la notazione funzionale propria del linguaggio C++ e quella tradizionale ereditata dal linguaggio C illustrate nel seguente frammento di codice:

```
float alfa = 2.0;
int beta, gamma;
beta = (int)a; // notazione tradizionale
gamma = int(a); // notazione funzionale
```

ESEMPIO

Il seguente programma

```
void main(void)
{
  int uno=1, due=2;
  float tre;

  tre = uno/due;
  cout<<tre;
}
```

nonostante le aspettative indotte dal fatto che la variabile *tre* sia dichiarata di tipo `float`, produrrà come risultato il valore zero. Infatti la divisione viene effettuata su due valori di tipo `int` (il contenuto delle variabili *uno* e *due*) e il risultato viene calcolato temporaneamente in una variabile di tipo `int` e solo successivamente assegnato alla variabile *tre* di tipo `float`.
Affinché la divisione produca il risultato atteso è necessario operare una conversione esplicita degli operandi come nel seguente programma:

```
void main(void)
{
  int uno=1, due=2;
  float tre;

  tre = (float)uno/(float)due;
  cout<<tre;
}
```

In questo caso il programma fornisce il risultato 0.5 perché il quoziente viene calcolato tra due variabili di tipo `float`. In realtà sarebbe stato sufficiente convertire uno solo dei due operandi e contare sull'applicazione della «regola del più forte» da parte del compilatore:

```
void main(void)
{
  int uno=1, due=2;
  float tre;

  tre = float(uno)/due;
  cout<<tre;
}
```

4 Operazioni standard di input e output

I programmi che tutti noi siamo abituati a utilizzare prevedono un'interazione tra l'utente e il computer basata principalmente sull'inserimento di dati tramite l'unità standard di input (normalmente la tastiera) e la visualizzazione di risultati tramite l'unità standard di output (normalmente lo schermo). Il linguaggio di programmazione C++ consente di effettuare tali attività basandosi sulle funzioni che le librerie standard di I/O (Input/Output) mettono a disposizione. Le funzionalità di queste librerie sono definite in un file a cui il programma deve inizialmente fare riferimento con la seguente direttiva di inclusione: `#include <iostream>`.

4.1 Operazioni di output

Per poter visualizzare un messaggio sul video è possibile utilizzare l'operatore «<<» applicato all'oggetto predefinito `cout`. Come abbiamo visto negli esempi precedenti, una istruzione come la seguente

```
cout<<area;
```

ha come effetto la visualizzazione sullo schermo del valore contenuto nella variabile *area*, mentre l'istruzione

```
cout<<"Valore della superficie del triangolo ";
```

visualizza il messaggio – cioè la sequenza dei caratteri – compreso tra i simboli «"».
È possibile effettuare più visualizzazione con una sola istruzione; per esempio, combinando le due istruzioni precedenti, avremo la seguente istruzione:

```
cout<<"Valore della superficie del triangolo "<<area;
```

il cui effetto, nel caso in cui la variabile contenga il valore 5.2, sarà quello di ottenere la visualizzazione sullo schermo del seguente output:

```
Valore della superficie del triangolo 5.2
```

Per fare in modo che il cursore si posizioni all'inizio della linea di visualizzazione successiva, in modo da evitare che il successivo output sia visualizzato adiacente al precedente, l'istruzione di visualizzazione deve terminare con l'output del simbolo «`\n`»:

```
cout<<"Valore della superficie del triangolo "<<area<<'\n';
```

o, con un effetto equivalente, con la costante «`endl`» (*end-line*):

```
cout<<"Valore della superficie del triangolo "<<area<<endl;
```

Elementi come il simbolo «`\n`» sono definiti **sequenze di *escape***. Per quanto rappresentato da due caratteri, il simbolo «`\n`» viene in realtà interpretato dal compilatore C++ come un solo carattere (è infatti delimitato dai simboli «`'`», utilizzati per i singoli caratteri, e non dai simboli «"», utilizzati per le sequenze di più caratteri). Il linguaggio C++ definisce un insieme di sequenze di *escape*, ognuna delle quali ha una sua funzione specifica. Le sequenze di *escape* fondamentali sono riassunte nella TABELLA **4**.

TABELLA **4**

Simbolo	Descrizione
`/r`	Singolarmente o in coppia definiscono il ritorno a capo e/o l'invio
`/n`	
`/t`	Tabulazione
`//`	Carattere *backslash*
`/"`	Carattere «"» all'interno di una sequenza di caratteri delimitata da simboli «"»

Le sequenze di *escape*

Il carattere «\» (*backslash*) non viene visualizzato dalle istruzioni di output, ma, utilizzato in combinazione con altri caratteri, ha alcune funzionalità speciali. Queste sequenze di caratteri, interpretate come un solo simbolo, sono definite *escape sequence*.
Esse sono definite dal linguaggio C++ per specificare caratteri speciali non visualizzabili – come il ritorno a capo e le tabulazioni – oppure caratteri che singolarmente hanno una funzione particolare nella sintassi del linguaggio, come le virgolette e lo stesso *backslash*.

ESEMPIO

La seguente istruzione di output:

```
cout<<alfa<<"\t"<<beta<<"\t"<<gamma<<"\r\n";
```

visualizza in un'unica linea di testo i valori delle variabili *alfa*, *beta* e *gamma* distanziati in modo uniforme dalla presenza del carattere di tabulazione.

4.2 Operazioni di input

Per assegnare come input (normalmente inserito dall'utente mediante la tastiera) il valore di una variabile è necessario utilizzare l'operatore «>>» applicato all'oggetto predefinito `cin`.

ESEMPIO

L'istruzione

```
cin>>beta;
```

assegna alla variabile *beta* il valore inserito dall'utente.

Un'istruzione come quella dell'esempio precedente sospende l'esecuzione del programma in attesa dell'inserimento di una sequenza di caratteri seguiti dal carattere di invio; la variabile di destinazione assume il valore digitato dall'utente.

OSSERVAZIONE Al fine di migliorare l'interazione uomo-macchina, è consigliabile utilizzare istruzioni di output che precedono le istruzioni di input per indirizzare l'utente sul significato dei dati richiesti per l'inserimento. Per cui, invece di limitarsi alla seguente istruzione

```
cin>>raggio;
```

il cui effetto è quello di visualizzare il cursore in attesa della digitazione del dato, è sicuramente più corretto farla precedere dall'istruzione

```
cout<<"Inserire il raggio del cerchio ";
```

che qualifica l'input che l'utente deve fornire:

```
Inserire il raggio del cerchio _
```

Naturalmente questa istruzione è completamente inutile dal punto di vista dell'algoritmo che il programma implementa, ma è invece determinante per l'interazione dell'utente col programma stesso.

Standard input e output in C++

Il compilatore C++ fornisce classi e oggetti predefiniti per la gestione dell'input/output il cui uso deve essere preceduto dalla direttiva

`#include <iostream>`.

`cin` è un oggetto della classe `istream` che interfaccia lo standard input, mentre `cout` è un oggetto della classe `ostream` che interfaccia lo standard output.

Tipicamente molti sistemi operativi associano lo standard input di un programma alla tastiera e lo standard output a una finestra testuale sullo schermo.

Il programma riceve da `cin` dati in forma di caratteri (input) usando l'operatore di estrazione «>>».

Su `cout` è invece possibile inviare dati in forma di caratteri (output) usando l'operatore di inserzione «<<».

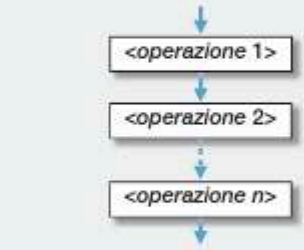
5 Controllo del flusso di esecuzione

Nel linguaggio di programmazione C++ la sezione algoritmica del codice si basa sui tre costrutti fondamentali: **sequenza**, **selezione** e **ripetizione**.

5.1 Sequenza

In C++ il concetto di sequenza coincide con il concetto di blocco di istruzioni delimitato dai simboli «{» e «}».
Nello schema di TABELLA 5 è possibile confrontare l'implementazione C++ di una sequenza di istruzioni con il formalismo adottato nelle due modalità di rappresentazione di un algoritmo.

TABELLA 5

Sequenza		
C++	**NLS**	**DAB**
`{` `  <istruzione 1>;` `  <istruzione 2>;` `  ...` `  <istruzione n>;` `}`	*<operazione 1>* *<operazione 2>* *...* *<operazione n>*	

5.2 Selezione

La selezione prevede i due tipi classici a una e a due vie mostrati nelle TABELLE 6 e 7.

TABELLA 6

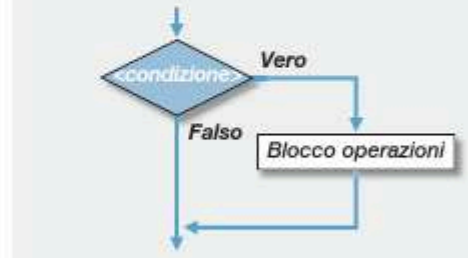
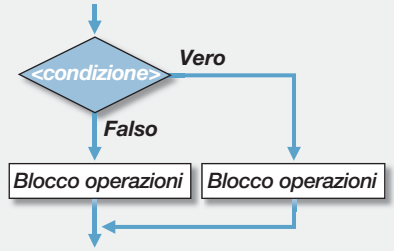
Selezione a una via		
C++	**NLS**	**DAB**
`if (<condizione>)` `  {` `  <istruzione 1>;` `  <istruzione 2>;` `  ...` `  <istruzione n>;` `  }`	*Se <condizione>* *allora ...*	

TABELLA 7

Selezione a due vie

C++	NLS	DAB
`if (<condizione>)` `  {` `  <istruzione 1>;` `  <istruzione 2>;` `  ...` `  <istruzione n>;` `  }` `else` `  {` `  <istruzione 1>;` `  <istruzione 2>;` `  ...` `  <istruzione n>;` `  }`	***Se*** *<condizione>* ***allora ...*** ***altrimenti ...***	

OSSERVAZIONE

Nel caso in cui l'istruzione `if` non sia seguita da un blocco delimitato dai simboli «{» e «}», la condizione si intende applicata **solo alla prima istruzione** che segue l'istruzione condizionale stessa.

Oltre ai due schemi di base per la selezione, il linguaggio C++ prevede un'istruzione specifica per la selezione multipla; esiste inoltre la possibilità di inserire in un'espressione una valutazione condizionale.
Nella costruzione di un programma può essere necessario condizionare l'esecuzione in seguito a più condizioni, per esempio in base ai diversi valori che può assumere una variabile. Questa situazione può essere gestita in linguaggio C++ utilizzando una serie di `if` (...) ... `else` ... nidificati, o con la sintassi `if` (...) ... `else if` (...) ... che permette una semplificazione del codice.

ESEMPIO

Il seguente programma determina la classe di merito (dalla migliore «A» alla peggiore «E») di uno studente in funzione del punteggio centesimale riportato in un test:

```
#include <iostream>

using namespace std;

void main(void)
{
  int punteggio;

  cout<<"Inserire il punteggio del test: ";
  cin>>punteggio;
  if (punteggio >= 90)
    cout<<"Classe di merito A"<<endl;
  else if (punteggio >= 80)
    cout<<"Classe di merito B"<<endl;
```

▶

```
    else if (punteggio >= 70)
      cout<<"Classe di merito C"<<endl;
    else if (punteggio >= 60)
      cout<<"Classe di merito D"<<endl;
    else
      cout<<"Classe di merito E"<<endl;
  }
```

Per effettuare selezioni multiple il linguaggio C++ rende disponibile la comoda istruzione **switch**, che presenta la seguente struttura sintattica:

```
switch (<variabile di controllo>)
  {
    case <valore1>:
                    {...}
                    break;
    case <valore2>:
                    {...}
                    break;
    ...
    ...
    ...
    case <valoreN>:
                    {...}
                    break;
    default:
                    {...}
  }
```

Il codice associato a ogni clausola **case** viene eseguito solo se la variabile di controllo assume il corrispondente valore; il codice associato alla **clausola opzionale default** viene eseguito se il valore della variabile di controllo non corrisponde ad alcuno dei valori indicati nelle clausole **case**. La parola chiave **break** indica l'uscita dall'istruzione e la prosecuzione dell'esecuzione dalla prima istruzione successiva.

ESEMPIO

La seguente tabella illustra la corrispondenza del punteggio decimale di un test con la relativa classe di merito (dalla migliore «A» alla peggiore «E») per la valutazione di uno studente:

Punteggi	Classe di merito
9 e 10	A
8	B
7	C
6	D
Inferiore a 6	E

Il seguente programma visualizza la classe di merito associata al punteggio fornito come input:

```
#include <iostream>

using namespace std;

void main(void)
{
  int punteggio;
  char classe;

  cout<<"Inserire il punteggio del test: ";
  cin>>punteggio;
  switch (punteggio)
    {
      case 9:
      case 10: classe = 'A';
               break;
      case 8:  classe = 'B';
               break;
      case 7:  classe = 'C';
               break;
      case 6:  classe = 'D';
               break;
      default: classe = 'E';
    }

  cout<<"Classe di merito "<<classe<<endl;
}
```

OSSERVAZIONE Nell'esempio precedente si noti come sia possibile associare lo stesso codice da eseguire a valori distinti della variabile di controllo dell'istruzione `switch` semplicemente scrivendo più clausole `case` una di seguito all'altra. È anche possibile proseguire l'esecuzione del codice di una clausola con il codice della clausola immediatamente successiva omettendo la parola chiave `break`, ma è una pratica sconsigliata per la sua scarsa leggibilità.

Esiste nel linguaggio C++ un operatore di confronto «ternario» – denominato ***if* aritmetico** – che ha come parametri:

- la condizione da valutare;
- l'espressione da restituire se la condizione risulta vera;
- l'espressione da restituire nel caso in cui la condizione sia falsa.

La sua forma generale è:

```
(<condizione>) ? <valore se vera> : <valore se falsa>;
```

Questo operatore permette di codificare in modo compatto e nel contesto di un'espressione semplici selezioni, come quelle per la determinazione del valore massimo o minimo tra due variabili, che risulterebbero inutilmente ingombranti se implementate con una istruzione `if`.

ESEMPIO

Per assegnare alla variabile *max* il valore massimo tra il contenuto di due variabili numeriche *x* e *y*, è sufficiente la seguente istruzione:

```
max = (x>y) ? x : y;
```

Le parentesi prima del simbolo «?» non sono strettamente necessarie, ma è sempre bene utilizzarle per evitare di incorrere nell'errore di usare nella condizione un operatore con priorità inferiore all'*if* aritmetico, alterando il significato dell'espressione.

OSSERVAZIONE L'*if* aritmetico non è un'istruzione, ma un operatore il cui uso è sempre limitato al contesto di un'espressione.

ESEMPIO

L'operatore *if* aritmetico permette anche la valutazione di condizioni non banali. La seguente istruzione seleziona il valore massimo tra il contenuto delle tre variabili *x*, *y* e *z*:

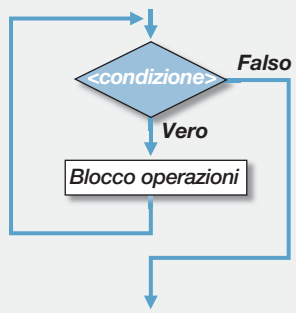
```
max = (x>y)?((x>z)?x:z):(y>z)?y:z);
```

5.3 Ripetizione

In relazione alla struttura di ripetizione il linguaggio C++ consente l'implementazione dei tre tipi base dello schema iterativo: **ciclo indeterminato con controllo in testa** (TABELLA **8**), **ciclo indeterminato con controllo in coda** (TABELLA **9**) e **ciclo determinato** (TABELLA **10**).

TABELLA **8**

Ciclo indeterminato con controllo in testa

C++	NLS	DAB
`while (<condizione>)` `{` `  <istruzione 1>;` `  <istruzione 2>;` `  ...` `  <istruzione n>;` `}`	***Finché*** *<condizione>* ***Esegui*** ***Inizio*** ... ***Fine***	

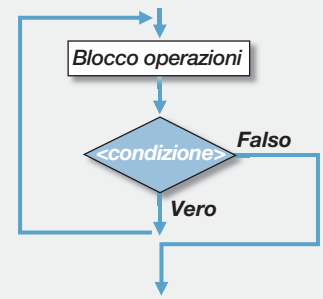
ESEMPIO

Il programma seguente visualizza la sequenza dei numeri pari compresi tra i valori interi *n* ed *m* forniti come input.

```
#include <iostream>

using namespace std;

void main(void)
{
  int n, m;

  cout<<("Inserire estremo inferiore intervallo: ");
  cin>>n;
  cout<<("Inserire estremo superiore intervallo: ");
  cin>>m;

  if (n%2 != 0) // se l'estremo inferiore è dispari
    n = n+1;    // si parte dal numero pari successivo

  while (n<=m)  //finche' n minore o uguale all'estremo superiore
    {
      cout<<n<<endl;  // si visualizza il numero pari corrente
      n = n + 2;      // e si determina il numero pari successivo
    }
}
```

TABELLA **9**

Ciclo indeterminato con controllo in coda

C++	NLS	DAB
`do` `{` `  <istruzione 1>;` `  <istruzione 2>;` `  ...` `  <istruzione n>;` `}` `while (<condizione>)`	*Esegui* *Inizio* … *Fine* *Finché* <*condizione*>	

ESEMPIO

Il programma che segue effettua la trasformazione di un numero *N* intero dalla base 10 alla base 2 producendone in output le singole cifre binarie. L'algoritmo che implementa prevede di eseguire divisioni successive per 2 a partire dal numero *N*: per ogni divisione viene calcolato il resto, che costituisce una cifra binaria (limitata ai valori 1 o 0); a ogni iterazione del procedimento il numero da dividere è il quoziente calcolato al passo precedente e la conversione ha termine quando si ottiene il valore 0 per il quoziente.

```
#include <iostream>

using namespace std;

void main(void)
{
  int N, quoziente, resto;
```

```
    cout<<"Inserire un numero intero da convertire: ");
    cin>>N;
    do
      {
        quoziente = N/2;      // calcolo del quoziente
        resto = N%2;          // calcolo del resto
        cout<<resto<<endl;    // visualizzazione cifra binaria
        N=quoziente;          // il quoziente attuale e' il prossimo numero
      } while (quoziente!=0);
  }
```

OSSERVAZIONE L'algoritmo del programma dell'esempio precedente fornisce le cifre binarie dalla meno significativa alla più significativa, per cui le cifre saranno visualizzate in senso inverso rispetto a quello della corretta interpretazione numerica!

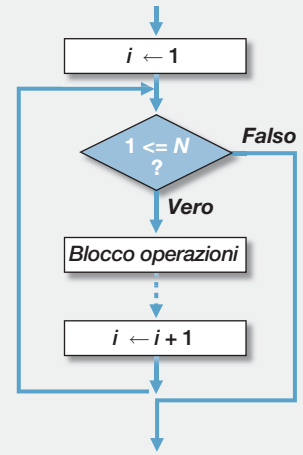
Per implementare cicli determinati il linguaggio C++ prevede l'istruzione *for*.

OSSERVAZIONE L'istruzione `for` del linguaggio C++ è atipica, in quanto il test di terminazione del ciclo può essere espresso tramite la formulazione di una condizione generica che può non coincidere con il controllo del valore della variabile il cui incremento ripetuto permette di contare il numero di iterazioni.

Nel caso più semplice la situazione è quella rappresentata nella TABELLA **10**, dove è stata utilizzata la variabile contatore *i*, che assume i valori compresi tra 1 ed *N* (in questo caso l'incremento della variabile è unitario ed è espresso in forma compatta utilizzando l'operatore di postincremento `i++` del C++).

TABELLA **10**

Ciclo determinato

C++	NLS	DAB
`for (i=1; i<=n; i++)` `{` `  <istruzione 1>;` `  <istruzione 2>;` `  ...` `  <istruzione n>;` `}`	*Per N volte esegui* *Inizio* ... *Fine*	

ESEMPIO

Il seguente programma produce come output la sequenza dei quadrati dei numeri interi compresi tra 1 ed *n*, con *n* fornito come input.

```
#include <iostream>

using namespace std;

void main(void)
{
  int i, n;

  cout<<"Inserire estremo superiore intervallo: ");
    cin>>n;
    for (i=1; i<=n; i++)   // per tutti i valori di i tra 1 ed n
      {
        cout<<i*i<<endl;   // si visualizza il quadrato di i
      }
}
```

Si noti come in questo caso la variabile di controllo *i* abbia un duplice scopo: quello di controllare il numero di iterazioni fatte e quello di rappresentare i valori da visualizzare.

Il linguaggio di programmazione C++ ha due specifiche istruzioni per controllare l'esecuzione dei cicli: **`break`** e **`continue`**.
L'istruzione ***break***, eseguita nel corpo di un ciclo, forza l'uscita immediata da esso interrompendo il processo iterativo e proseguendo l'esecuzione dall'istruzione successiva al ciclo stesso.
L'istruzione ***continue*** causa la prosecuzione dell'esecuzione della prima istruzione del ciclo, ignorando le istruzioni del corpo del ciclo che eventualmente sono specificate dopo di essa.

OSSERVAZIONE Le istruzioni **`break`** e **`continue`** sono sempre inserite nel corpo di un ciclo, in modo che la loro esecuzione sia condizionata dal verificarsi di specifiche condizioni. In caso contrario esse altererebbero il significato del blocco di istruzioni la cui esecuzione deve essere ripetuta.

ESEMPIO

Il programma che segue fornisce un semplice esempio di uso delle istruzioni **`break`** e **`continue`**; il programma acquisisce in input valori interi applicando a ciascuno le seguente regole:

- se il valore è negativo, si visualizza un messaggio di errore e si termina il processo di acquisizione;
- se un valore è maggiore di 100, esso viene ignorato e si prosegue con l'acquisizione del valore successivo;
- ogni altro valore acquisito viene visualizzato.

Il programma termina l'esecuzione quando il valore acquisito è uguale a 0 o, nel caso in cui si verifichi un errore, fornendo in output la somma e il numero dei valori validi acquisiti. ▶

```
#include <iostream>

using namespace std;

void main(void)
{
  int x, validi=0, somma=0;

  do
    {
      cout<<"Inserire un numero intero:");
      cin>>x;

      if (x<0)
        {
          cout<<"Valore non ammesso: "<<x<<endl;
          break; // interruzione del ciclo
        }
      if (x>100)
        {
          cout<<"Valore ignorato: "<<x<<endl;
          continue; // si prosegue con la prossima iterazione
        }
      cout<<"Numero acquisito: "<<x<< endl;
      somma = somma+x;
      validi++:
    }
    while (x!=0);
    cout<<"Valori validi acquisiti: "<<validi<<endl;
    cout<<"Somma dei valori validi: "<<somma<<endl;
}
```

OSSERVAZIONE

Come nel caso dell'istruzione `if`, anche per i cicli `while` e `for`, se le istruzioni non sono seguite da un blocco delimitato dai simboli «{» e «}», il corpo del ciclo è costituito da **una singola istruzione**: **la prima successiva alla parola chiave**.

6 Esempi di implementazione di algoritmi in linguaggio C++

6.1 Esempio 1: determinazione della data della Pasqua

Il seguente algoritmo in NLS determina il giorno e il mese in cui cade la festa di Pasqua a partire dall'anno:

Inizio
  Leggi X
    $B \leftarrow$ ***Int***$(X : 100)$
    $C \leftarrow X -$ ***Int***$(X : 100) * 100$
    $A \leftarrow (5 * B + C) -$ ***Int***$((5 * B + C) : 19) * 19$
    $R \leftarrow$ ***Int***$((3 * B + 75) : 4)$
    $S \leftarrow (3 * B + 75) -$ ***Int***$((3 * B + 75) : 4) * 4$
    $T \leftarrow$ ***Int***$((8 * B + 88) : 25)$
    $H \leftarrow (19 * A + R - T) -$ ***Int***$((19 * A + R - T) : 30) * 30$
    $G \leftarrow$ ***Int***$((A + 11 * H) : 319)$
    $J \leftarrow$ ***Int***$((300 - 60 * S + C) : 4)$
    $K \leftarrow (300 - 60 * S + C) -$ ***Int***$((300 - 60 * S + C) : 4) * 4$
    $M \leftarrow (2 * J - K - H + G) -$ ***Int***$((2 * J - K - H + G) : 7) * 7$
    $N \leftarrow$ ***Int***$((H - G + M + 110) : 30)$
    $Q \leftarrow (H - G + M + 110) -$ ***Int***$((H - G + M + 110) : 30) * 30$
    $P \leftarrow (Q + 5 - N) -$ ***Int***$((Q + 5 - N) : 32) * 32$
  Scrivi P, N
Fine

Il programma C++ che segue implementa l'algoritmo precedente:

```
#include <iostream>

using namespace std;

void main(void)
{
  int X,B,C,A,R,S,T,H,G,J,K,M,N,Q,P;

  cout<<"Anno? ";
  cin>>X; // input del valore dell'anno
  B = X/100;
  C = X-(X/100)*100;
  A = (5*B+C)-((5*B+C)/19)*19;
  R = ((3*B+75)/4);
  S = (3*B+75)-((3*B+75)/4)*4;
  T = ((8*B+88)/25);
  H = (19*A+R-T)-((19*A+R-T)/30)*30;
  G = ((A+11*H)/319);
  J = ((300-60*S+C)/4);
  K = (300-60*S+C)-((300-60*S+C)/4)*4;
  M = (2*J-K-H+G)-((2*J-K-H+G)/7)*7;
  N = ((H-G+M+110)/30);
  Q = (H-G+M+110)-((H-G+M+110)/30)*30;
  P = (Q+5-N)-((Q+5-N)/32)*32;
  cout<<"Giorno= "<<P<<endl; // output del valore del giorno
  cout<<"Mese= "<<N<<endl;   // output del valore del mese
}
```

OSSERVAZIONE Il fatto di avere dichiarato tutte le variabili di tipo `int` impone che le operazioni di divisione siano divisioni intere, come richiesto dall'algoritmo NLS.

6.2 Esempio 2: calcolo del costo di una spedizione

La TABELLA **11** riassume le condizioni economiche di un'agenzia di spedizioni.

TABELLA **11**

Classe	0-100 km	100-500 km	Oltre 500 km
NORMALE	1 €/kg	1,5 €/kg	2 €/kg
URGENTE	1,5 €/kg	2 €/kg	3 €/kg

L'algoritmo, rappresentato nel diagramma a blocchi di FIGURA **1** computa il costo di una spedizione a partire dalla sua classe, dalla distanza e dal peso.

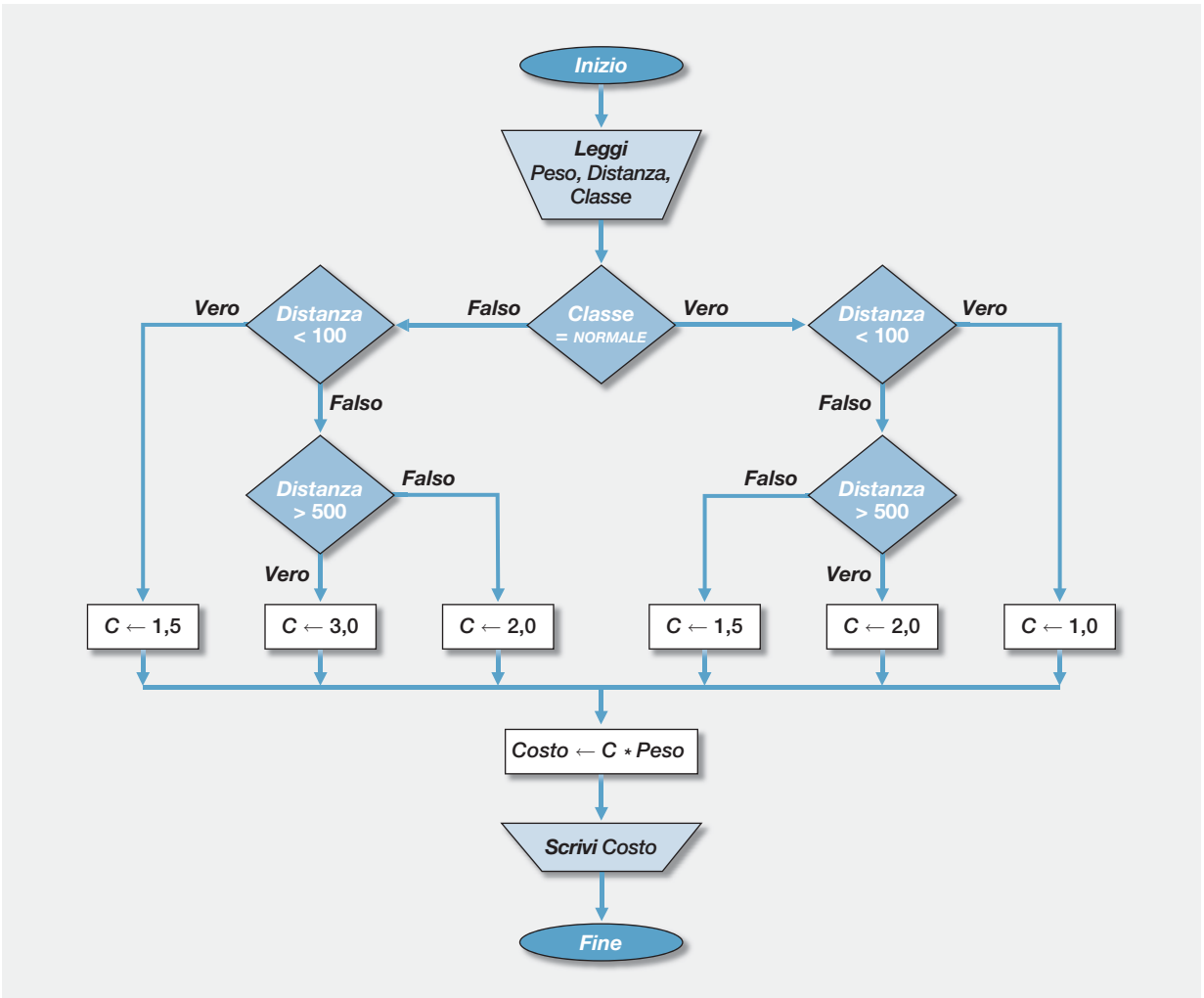


FIGURA **1**

Il programma C++ che segue implementa l'algoritmo precedente:

```
#include <iostream>

using namespace std;

void main(void)
{
  char classe;
  float peso, C, costo;
  int distanza;

  cout<<"Inserire la classe (N=Normale/U=Urgente) ";
  cin>>classe;
  cout<<"Inserire il peso (Kg) ";
  cin>>peso;
  cout<<"Inserire la distanza (Km) ";
  cin>>distanza;

  if (classe == 'U' || classe == 'u')
    {
      // classe Urgente
      if (distanza<100)
        C = 1.5;
      else if (distanza>500)
        C = 3.0;
      else C = 2.0;
    }
  else
    {
      // classe Normale
      if (distanza<100)
        C = 1.0;
      else if (distanza>500)
        C = 2.0;
      else C = 1.5;
    }
  costo = C*peso;
  cout<<"Il costo della spedizione e' "<<costo<<" Euro"<<endl;
}
```

OSSERVAZIONE La classe di spedizione è stata rappresentata mediante un singolo carattere, che viene verificato sia nella forma maiuscola sia in quella minuscola.

6.3 Esempio 3: rimbalzi di una pallina di gomma

L'algoritmo rappresentato con la notazione lineare strutturata che segue calcola il numero di rimbalzi effettuati da una pallina di gomma che ogni

volta rimbalza a un'altezza pari all'80% della precedente, arrestandosi quando l'altezza è pari o inferiore a 1 cm:

Inizio
  Leggi *altezza*
    $N = 0$
  Esegui
    Inizio
      $altezza \leftarrow altezza * 0{,}8$
      $N \leftarrow N + 1$
    Fine
  Finché $(altezza > 0{,}01)$
  Scrivi N
Fine

Il programma C++ che segue implementa l'algoritmo precedente:

```
#include <iostream>

using namespace std;

void main (void)
{
  int N;
  float altezza;

  cout<<"Inserire l'altezza iniziale (cm) ";
  cin>>altezza;
  N = 0; // inizializzazione del contatore di rimbalzi
  do
    {
      altezza = altezza * 0.8; // aggiornamento dell'altezza
      N++; // incremento del contatore di rimbalzi
    } while (altezza > 0.01);
  cout<<"Il numero dei rimbalzi e' "<<N<<endl;
}
```

6.4 Esempio 4: sequenza dei numeri di Fibonacci

L'algoritmo rappresentato con la notazione lineare strutturata che segue visualizza la sequenza dei numeri di Fibonacci, in cui ciascun numero è pari alla somma dei due precedenti:

Inizio
  Leggi N
    $F1 \leftarrow 1$
  Scrivi $F1$
    $F2 \leftarrow 1$ ▶

Scrivi $F2$
$N \leftarrow N - 2$
Esegui
Inizio
$F \leftarrow F1 + F2$
Scrivi F
$N \leftarrow N - 1$
$F2 \leftarrow F1$
$F1 \leftarrow F$
Fine
Finché $(N > 0)$
Fine

Il programma C++ che segue implementa l'algoritmo precedente:

```
#include <iostream>

using namespace std;

void main(void)
{
  int N, F1, F2, F;

  cout<<"Inserire la lunghezza della sequenza ";
  cin>>N;
  F1 = 1; // primo valore della sequenza
  cout<<F1<<endl;
  F2 = 1; // secondo valore della sequenza
  cout<<F2<<endl;
  N = N-2; // inizializzatore del contatore di iterazioni
  do
    {
      F = F1+F2; // calcolo del successivo valore della sequenza
      cout<<F<<endl; // visualizzazione del valore calcolato
      N--; // decremento del contatore di iterazioni
      F2 = F1;
      F1 = F;
    } while (N > 0);
}
```

7 Funzioni della libreria matematica

Il linguaggio C++ rende disponibili le funzioni matematiche più comuni nella propria libreria matematica, ereditata dall'analoga libreria del linguaggio C. La direttiva

```
#include <cmath>
```

consente di utilizzare nel codice del programma le funzioni della libreria, di cui si riportano le principali nella TABELLA **12**.

TABELLA **12**

Funzione	Descrizione
`abs`	Valore assoluto (valori interi)
`acos`	Arcocoseno (angolo in radianti)
`asin`	Arcoseno (angolo in radianti)
`atan`	Arcotangente (angolo in radianti)
`ceil`	Valore intero superiore (valori *floating-point*)
`cos`	Coseno (angolo in radianti)
`exp`	Esponenziale (base *e*)
`fabs`	Valore assoluto (valori *floating-point*)
`floor`	Valore intero inferiore (valori *floating-point*)
`log`	Logartitmo naturale (base *e*)
`log10`	Logaritmo decimale (base 10)
`pow`	Potenza
`sin`	Seno (angolo in radianti)
`sqrt`	Radice quadrata
`tan`	Tangente (angolo in radianti)

ESEMPIO

Il seguente programma C++ calcola la distanza tra due punti nel piano di cui sono fornite in input le coordinate cartesiane utilizzando la seguente formula:

$$d = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$$

```
#include <iostream>
#include <cmath>

using namespace std;

void main(void)
{
  double xA, xB, yA, yB, d, D;

  cout<<"Inserire l'ascissa del punto A ";
  cin>>xA;
  cout<<"Inserire l'ordinata del punto A ";
  cin>>yA;
  cout<<"Inserire l'ascissa del punto B ";
  cin>>xB;
  cout<<"Inserire l'ordinata del punto B ";
  cin>>yB;

  D = pow(xA-xB,2) + pow(yA-yB,2);
  d = sqrt(D);

  cout<<"La distanza e' "<<d;
}
```

Sintesi

Programma base in linguaggio C++. Insieme di dichiarazioni e di istruzioni comprese tra i simboli «{» e «}» che delimitano il corpo del programma introdotto dalla classica intestazione `void main(void)`. Tutte le istruzioni sono terminate dal simbolo «;».

Variabile. In C++ una variabile può essere vista come una tripla (nome, tipo, valore) dove il **nome** è l'identificatore della variabile, il **tipo** definisce l'insieme di valori che essa può assumere e le operazioni che su questi posso essere applicate, il **valore** è il contenuto della variabile. La dichiarazione delle variabili è obbligatoria e avviene elencando uno o più nomi di variabile – in quest'ultimo caso separati dal simbolo «,» – dopo la parola che ne definisce il tipo. È possibile inizializzare le variabili contestualmente alla loro dichiarazione. Le variabili non inizializzate assumono inizialmente un valore indefinito.

Tipi. I tipi fondamentali del linguaggio C++ sono elencati nella tabella riportata in fondo alla pagina.

Costante. In C++ una costante è un valore che non può essere modificato e a cui ci si può riferire nel corso di un programma tramite un nome simbolico. La dichiarazione delle costanti avviene nello stesso modo delle variabili, premettendo alla specifica del tipo e del nome simbolico la parola chiave `const`.

Blocco. In C++ è un insieme di istruzioni comprese tra i simboli «{» e «}» e che vengono viste come un insieme monolitico. All'interno di un blocco è possibile dichiarare variabili «locali» indipendenti da eventuali variabili omonime dichiarate in blocchi più esterni.

Commento. Esistono due modalità per inserire un commento nel codice:

- **commento su linea singola**: inizia con il simbolo «//» e termina con la fine della riga stessa;
- **commento multilinea**: inizia con il simbolo «/*» e termina con il simbolo «*/».

Assegnamento. La modifica del valore di una variabile può essere effettuata in C++ mediante l'operatore di assegnamento «=»: a sinistra dell'operatore di assegnamento vi è sempre una variabile, mentre a destra può esserci un valore costante o un'espressione comunque complessa che il linguaggio consente di valutare per ottenere un risultato il cui valore viene assegnato alla variabile.

Scope* e *lifetime. Con i termini *scope* e *lifetime* ci si riferisce rispettivamente alla visibilità (ambito di azione) e al tempo di vita (classe di memorizzazione) di variabili e costanti. Per quanto riguarda l'ambito di azione, variabili e costanti vengono classificate in **globali** (valide dal punto in cui sono state dichiarate fino alla fine del file) e **locali** (valide limitatamente al blocco in cui sono state dichiarate). Relativamente alla classe di me-

Tipo	Descrizione	Dimensione (byte)	Intervallo numerico
`char`	Singolo carattere numero intero *small*	1	`signed`: da –128 a 127 `unsigned`: da 0 a 255
`short int`	Numero intero *short*	2	`signed`: da –32 768 a 32 767 `unsigned`: da 0 a 65 535
`int`	Numero intero		Dipendente dal compilatore
`long int`	Numero intero *long*	4	`signed`: da –2 147 483 648 a 2 147 483 647 `unsigned`: da 0 a 4 294 967 295
`bool`	Valore booleano	1	`true`/`false`
`float`	Numero *floating-point*	4	da -3.4×10^{-38} a 3.4×10^{38}
`double`	Numero *floating-point* a doppia precisione	8	da -1.7×10^{-38} a 1.7×10^{38}

morizzazione variabili e costanti vengono classificate in **permanenti** (create e inizializzate prima dell'avvio del programma, rimangono attive fino al termine dell'esecuzione), o **temporanee** (create nella memoria del computer all'inizio dell'esecuzione del blocco di istruzioni che ne comprende la dichiarazione e distrutte al termine del blocco stesso).

Operatori algebrici. Oltre ai classici operatori aritmetici «+», «–», «*» e «/», il linguaggio C++ fornisce l'operatore per il calcolo del resto della divisione intera (modulo) «%», e gli operatori di autoincremento «++» e autodecremento «––» utilizzabili nella notazione prefissa e postfissa. Nella valutazione delle espressioni algebriche la precedenza degli operatori è quella classica: prima sono valutati gli operatori di segno delle singole variabili o costanti numeriche, successivamente sono valutate le operazioni di moltiplicazione, divisione e modulo e solo per ultime le operazioni di addizione e sottrazione. Questi criteri di precedenza possono essere modificati con il ricorso ai simboli delle parentesi «(» e «)».

Operatori di relazione. Per la formulazione delle condizioni logiche il linguaggio C++ mette a disposizione gli operatori «==» (uguale), «>», «>=», «<», «<=» e «!=» (diverso).

Operatori booleani. Per esprimere condizioni logiche composte combinando tra loro più condizioni logiche semplici il linguaggio C++ dispone degli operatori logici «!» (NOT), «&&» (AND) e «||» (OR).

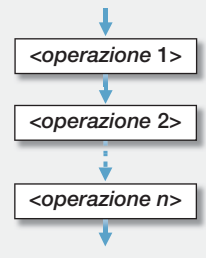
Casting. Operazione che consente di convertire un'espressione di un dato tipo in un altro tipo. I principali tipi di casting sono: **implicito** (eseguito automaticamente quando un valore viene utilizzato come un tipo compatibile) ed **esplicito** (invocato dal programmatore per richiedere una differente interpretazione del tipo dei valori di variabili o espressioni). Le conversioni implicite hanno effetto sui tipi fondamentali di dati e consentono trasformazioni come quelle tra tipi numerici. In generale vale la **regola del più forte**: se gli operandi di un'espressione sono di tipo diverso tra loro, il risultato sarà del tipo più forte, cioè quello in grado di memorizzare un intervallo più esteso.

Operazioni di input/output. Il compilatore C++ fornisce classi predefinite per la gestione dell'input/output: `cin` è un oggetto predefinito della classe *istream* che rappresenta lo standard input, mentre `cout` è un oggetto predefinito della classe *ostream* per lo standard output. Mediante `cin` è possibile ottenere dati in forma di caratteri usando l'operatore di estrazione «>>»; con `cout` è possibile visualizzare caratteri usando l'operatore di inserzione «<<».

Struttura del controllo del linguaggio C++. È basata sulle strutture fondamentali di **sequenza**, **selezione** e **ripetizione** e presenta costrutti specifici per la loro gestione.

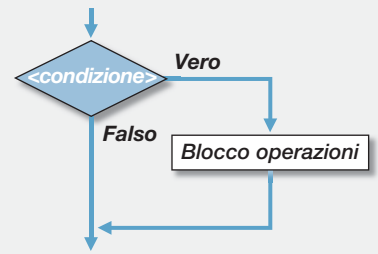
Sequenza. In pratica, il concetto di sequenza in C++ coincide con il concetto di blocco di istruzioni delimitato dai simboli «{» e «}»:

Sequenza

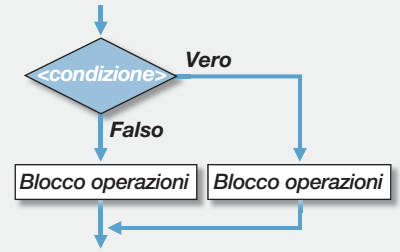
C++	NLS	DAB
`{` `  <istruzione 1>;` `  <istruzione 2>;` `  ...` `  <istruzione n>;` `}`	*<operazione 1>* *<operazione 2>* *...* *<operazione n>*	

Selezione. Per la selezione sono previsti i due tipi a una via e a due vie:

Selezione a una via

C++	NLS	DAB
`if (<condizione>)` `  {` `  <istruzione 1>;` `  <istruzione 2>;` `  ...` `  <istruzione n>;` `  }`	*Se* <*condizione*> *allora ...*	

Selezione a due vie

C++	NLS	DAB
`if (<condizione>)` `  {` `  <istruzione 1>;` `  <istruzione 2>;` `  ...` `  <istruzione n>;` `  }` `else` `  {` `  <istruzione 1>;` `  <istruzione 2>;` `  ...` `  <istruzione n>;` `  }`	*Se* <*condizione*> *allora ...* *altrimenti ...*	

Selezione multipla. Il linguaggio C++ dispone del costrutto di selezione multipla `switch-case`, che sulla base del valore di una variabile permette di articolare una selezione su più casi distinti:

```
switch (<variabile di controllo>)
  {
    case <valore1>:
                    {...}
                    break;
    case <valore2>:
                    {...}
                    break;
    ...
    ...
    ...
    case <valoreN>:
                    {...}
                    break;
    default:
                    {...}
  }
```

Ripetizione. Anche per la ripetizione il linguaggio C++ permette l'implementazione dei tre classici tipi di ciclo: ciclo indeterminato con controllo in testa, ciclo indeterminato con controllo in coda e ciclo determinato.

Ciclo indeterminato con controllo in testa

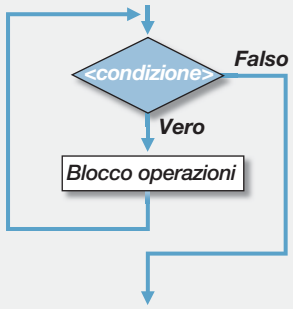
C++

```
while (<condizione>)
{
   <istruzione 1>;
   <istruzione 2>;
   ...
   <istruzione n>;
}
```

NLS

Finché <condizione>
Esegui
  Inizio
  …
  Fine

DAB



Ciclo indeterminato con controllo in coda

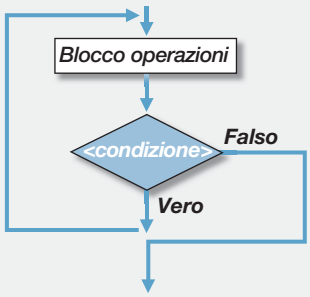
C++

```
do
{
   <istruzione 1>;
   <istruzione 2>;
   ...
   <istruzione n>;
}
while (<condizione>)
```

NLS

Esegui
  Inizio
  …
  Fine
Finché <condizione>

DAB



Ciclo determinato

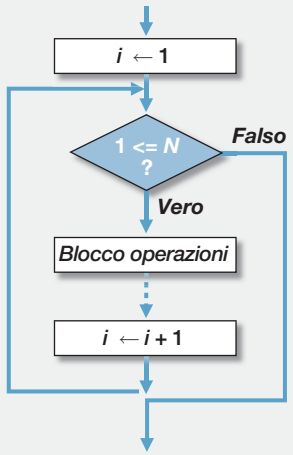
C++

```
for (i=1; i<=n; i++)
{
   <istruzione 1>;
   <istruzione 2>;
   ...
   <istruzione n>;
}
```

NLS

Per N volte esegui
  Inizio
  …
  Fine

DAB



Controllo dei cicli. Il linguaggio di programmazione C++ ha due specifiche istruzioni per controllare l'esecuzione dei cicli: `break` e `continue`. L'istruzione ***break***, eseguita nel corpo di un ciclo, forza l'uscita immediata da esso interrompendo il processo iterativo e proseguendo l'esecuzione dall'istruzione successiva al ciclo stesso. L'istruzione ***continue*** causa la prosecuzione dell'esecuzione della prima istruzione del ciclo, ignorando le istruzioni del corpo del ciclo che eventualmente sono specificate dopo di essa.

Direttive di precompilazione. Sono comandi per il preprocessore che specificano in particolare l'inclusione dei file di intestazione contenenti la definizione delle funzioni di libreria utilizzate dal programma. Questi file includono solitamente la definizione di funzioni di utilità come `sqrt` (radice quadrata) e `pow` (elevamento a potenza), oppure di oggetti come `cin` e `cout`, molto utilizzati dai programmatori. Queste funzionalità aggiuntive sono comprese nelle librerie standard del linguaggio C++ e le definizioni necessarie al loro uso corretto sono presenti nei file di intestazione che si includono prima dell'inizio del programma con la direttiva `#include`.

QUESITI

1 **Indicare quali delle seguenti affermazioni relative a un programma C++ sono corrette.**

A È l'implementazione di un algoritmo.

B È la stessa cosa della rappresentazione di un algoritmo in NLS.

C Deve contenere una funzione principale denominata *main*.

D Deve contenere almeno una costante.

2 **Un blocco è…**

A … una speciale istruzione che provoca il blocco dell'esecuzione di un programma.

B … un punto del programma dove il programma si blocca per permettere l'analisi del contenuto delle variabili.

C … un insieme di istruzioni comprese tra i simboli «{» e «}».

D … Nessuna delle risposte precedenti.

3 **Indicare quali delle seguenti affermazioni relative a una variabile C++ sono corrette.**

A Il suo valore può essere modificato tramite l'uso dell'operatore «=».

B All'atto delle sua dichiarazione è possibile assegnarle un valore.

C Non è necessario dichiararla esplicitamente prima del suo uso.

D Nella sua dichiarazione deve essere specificato oltre al nome anche il tipo.

4 **Indicare quali delle seguenti affermazioni relative a una costante C++ sono corrette.**

A Il suo valore può essere modificato nel corso del programma tramite l'uso dell'operatore «=».

B All'atto delle sua dichiarazione non è necessario assegnarle un valore.

C È necessario dichiararla esplicitamente prima del suo uso.

D Nella sua dichiarazione deve essere specificato oltre al nome anche il tipo.

5 **Indicare per quali valori di *a*, *b* e *c* si ottiene come output «il primo coefficiente è negativo» eseguendo il seguente frammento di codice:**

```
if (a>0)
if(b>0)
if (c>0) cout<<"sono tre coefficienti
                positivi";
else cout<<"il primo coefficiente è
          negativo";
```

A $a < 0, b > 0, c <= 0$.

B $a > 0, b > 0, c <= 0$.

C $a > 0, b > 0, c > 0$.

D Nessuna delle risposte precedenti.

6 **Al termine dell'esecuzione di un blocco di istruzioni...**

A ... le variabili locali vengono perse.

B ... i valori delle variabili locali sono copiati in quelle globali con lo stesso nome.

C ... i valori delle variabili locali sono mantenuti a meno di omonimia con le variabili globali.

D Nessuna delle risposte precedenti.

7 **Se una variabile locale di un blocco e una variabile globale hanno lo stesso nome...**

A ... la variabile globale smette di esistere.

B ... la variabile locale smette di esistere.

C ... la variabile locale nasconde la variabile globale durante l'esecuzione del blocco.

D Nessuna delle risposte precedenti.

8 **Se la variabile *A* vale 5 e la variabile *B* vale 1, indicare quale valore assumono le variabili *A*, *B* e *C* alla fine di ognuna delle seguenti sequenze di istruzioni:**

```
A += B;          A = A*B;       A = B++;
B = (B--)*A;     C = ++A*B;     B = 2*A--;
C = B-A;         B = B-C;       C = 2*++B;
```

9 **Dati i seguenti cicli in linguaggio C++, indicare per ciascuno di essi quante volte viene ripetuto.**

A

```
c = 1;
do
{
    c--;
    cout<<c;
}
while (c>1);
```

B

```
c = 0;
while (c>=0)
{
    c++;
    cout<<c;
}
```

C

```
c = 2;
while (c>2)
{
    c++;
    cout<<c;
}
```

D

```
c = 0;
do
{
    c=c+2;
    cout<<c;
}
while (c>=2);
```

10 **Indicare quali valori vengono visualizzati eseguendo il seguente frammento di codice:**

```
a=0
for (i=1; i<10; i++)
   {
   a+=2;
   if (a>7)
     break;
}
```

A $i = 4, a = 8$

B $i = 3, a = 6$

C $i = 10, a = 20$

D Nessuna delle risposte precedenti.

11 **Indicare quando la seguente espressione logica risulta vera:**

```
(a>5) && (b<=7)
```

A Mai.

B Solo se $5 < a <= 7$.

C Solo se $5 < b <= 7$ e $5 < a <= 7$.

D Nessuna delle risposte precedenti.

12 **Indicare quando sono eseguite le istruzioni del blocco:**

```
if (a>5)
   if (b<=7)
      {...}
```

A Mai.

B Solo se $5 < a <= 7$.

C Solo se $5 < b <= 7$ e $5 < a <= 7$.

D Nessuna delle risposte precedenti.

13 Indicare quale dei seguenti insiemi di istruzioni calcola nella variabile *j* la media di *n* valori interi positivi, posto che *j* sia inizializzata a zero.

A
```
for (i=0; i<n; i++) {cin>>k; j=k;}
```
B
```
for (i=0; i<=n; i++) {cin>>k; j+=k; j/=n;}
```
C
```
for (i=0; i<n; i++) {cin>>k; j+= k; j=j/n;}
```
D Nessuna delle risposte precedenti.

14 Dati i due operatori «==» e «=», stabilire quali delle seguenti affermazioni sono corrette.

A Il primo è un operatore di assegnamento, mentre il secondo è un operatore di confronto logico.

B Il secondo è un operatore di tipo distruttivo, mentre il primo no.

C Sono due operatori equivalenti.

D Nessuna delle risposte precedenti.

15 Sia dato un valore *k*; dopo l'esecuzione del seguente frammento di codice:

```
for (i=0; i<10; i++)
  {
    cin>>n;
    if(k<n)
      k=n;
  }
```

A … *k* conterrà il più piccolo valore *n* tra 10 input.

B … *k* conterrà l'ultimo valore *n* letto.

C … *k* conterrà il più grande valore *n* tra 9 input.

D Nessuna delle risposte precedenti.

16 È dato il seguente frammento di codice:

```
while (i < MAX)
  {
    cin>>k;
    if (k>=0)
      i++;
  }
```

Perché l'esecuzione del ciclo avvenga almeno una volta e lo stesso termini è necessario che…

A … siano inseriti esclusivamente valori positivi.

B … il valore MAX sia al massimo 3525.

C … il valore della variabile *i* sia minore di MAX e i valori di *k* non siano negativi.

D Nessuna delle risposte precedenti.

17 In una istruzione `if`, la clausola `else` viene eseguita solo se…

A … la condizione è vera.

B … la condizione è falsa, ma non contiene l'operatore «==».

C … la condizione è falsa.

D Nessuna delle risposte precedenti.

18 È data la seguente istruzione:

```
x=((a>b)?((a>c)?a:c):((b>c)? b:c));
```

A quali dei seguenti frammenti di codice è equivalente?

A
```
if (a>b && a>c)
  x=a;
else
  x=b;
```
B
```
if (a>b)
  if (a>c)
    x=a;
  else
    x=c;
  else
    if (b>c)
      x=b;
    else
      x=c;
```
C
```
if (a>b && a>c)
  x=a;
else
  if (b>c)
    x=b;
  else
    x=c;
```
D Nessuna delle risposte precedenti.

19 **Una operazione di *casting* rappresenta…**

A … la conversione di un'espressione di un certo tipo in un tipo diverso.

B … lo scambio dei valori di due variabili dello stesso tipo.

C … lo scambio dei valori di due variabili di tipo diverso.

D Nessuna delle risposte precedenti.

20 **Con l'istruzione `switch` è possibile implementare:**

A … un ciclo determinato.

B … un ciclo indeterminato con controllo in testa.

C … un ciclo indeterminato con controllo in coda.

D Nessuna delle risposte precedenti.

ESERCIZI

Gli esercizi di questo capitolo prevedono l'implementazione e la verifica in forma di programmi C++ degli algoritmi progettati eseguendo gli esercizi che chiudono il precedente capitolo «Algoritmi» (esclusi gli esercizi relativi alla MDT).

loop
istruzione che cicla ripetutamente finché una condizione è vera o qualche condizione è soddisfatta

infinite loop
un loop la cui condizione è sempre vera

body
corpo di un loop: insieme di istruzioni all'interno di un loop

iteration
una singola esecuzione del corpo di un loop

squiggly-braces
parentesi graffe

blastoff!
letteralmente: "salta fuori!"

lather
schiuma

to rinse
risciacquare

6.2 Iteration

One of the things computers are often used for is the automation of repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.
We have seen programs that use recursion to perform repetition, such as `nLines` and `countdown`. This type of repetition is called **iteration**, and C++ provides several language features that make it easier to write iterative programs.
The two features we are going to look at are the *while* statement and the *for* statement.

6.3 The while statement

Using a `while` statement, we can rewrite `countdown`:

```
void countdown (int n) {
  while (n > 0) {
    cout << n << endl;
    n = n-1;
  }
  cout << "Blastoff!" << endl;
}
```

You can almost read a `while` statement as if it were English. What this means is, "While n is greater than zero, continue displaying the value of n and then reducing the value of n by 1. When you get to zero, output the word 'Blastoff!'"
More formally, the flow of execution for a **while** statement is as follows:

1 Evaluate the condition in parentheses, yielding **true** or **false**.

2 If the condition is false, exit the **while** statement and continue execution at the next statement.

3 If the condition is true, execute each of the statements between the squiggly-braces, and then go back to step 1.

This type of flow is called a loop because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the statements inside the loop are never executed. The statements inside the loop are called the **body** of the loop.

The body of the loop should change the value of one or more variables so that, eventually, the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop.

In the case of **countdown**, we can prove that the loop will terminate because we know that the value of **n** is finite, and we can see that the value of **n** gets smaller each time through the loop (each **iteration**), so eventually we have to get to zero.
In other cases it is not so easy to tell:

```
void sequence (int n) {
  while (n != 1) {
    cout << n << endl;
    if (n%2 == 0) { // n is even
      n = n / 2;
    } else { // n is odd
      n = n*3 + 1;
    }
  }
}
```

The condition for this loop is n! = 1, so the loop will continue until n is 1, which will make the condition false.
At each iteration, the program outputs the value of n and then checks whether it is even or odd. If it is even, the value of n is divided by two. If it is odd, the value is replaced by 3n+1. For example, if the starting value (the argument passed to sequence) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.
Since n sometimes increases and sometimes decreases, there is no obvious proof that n will ever reach 1, or that the program will terminate. For some particular values of n, we can prove termination. For example, if the starting value is a power of two, then the value of n will be even every time through the loop, until we get to 1. The previous example ends with such a sequence, starting with 16.
Particular values aside, the interesting question is whether we can prove that this program terminates for all values of n. So far, no one has been able to prove it *or* disprove it! […]

10.3 for loops

The loops we have written so far have a number of elements in common. All of them start by initializing a variable; they have a test, or condition, that depends on that variable; and inside the loop they do something to that variable, like increment it.
This type of loop is so common that there is an alternate loop statement, called *for*, that expresses it more concisely. The general syntax looks like this:

```
for (INITIALIZER; CONDITION; INCREMENTOR) {
   BODY
}
```

This statement is exactly equivalent to

```
INITIALIZER;
while (CONDITION) {
   BODY
   INCREMENTOR
}
```

except that it is more concise and, since it puts all the loop-related statements in one place, it is easier to read. For example:

```
for (int i = 0; i < 4; i++) {
   cout << count[i] << endl;
}
```

is equivalent to

```
int i = 0;
while (i < 4) {
   cout << count[i] << endl;
   i++;
}
```

[Allen B. Downey, *How to think like a computer scientist*, 1999]

QUESTIONS

a What is iteration?

b What is an infinite loop?

c What has to happen in the body of the loop so that the loop itself terminates?

d Why the name "loop"? [This is a repeat of question d) from debugging]

A5 Le funzioni in C++

Quanti modi ci sono di scegliere k elementi da un insieme di n elementi? La teoria del calcolo combinatorio definisce **combinazioni** queste modalità e il loro numero è dato dalla seguente formula[1]:

$$C_{n,k} = \frac{n!}{k! \cdot (n-k)!}$$

ESEMPIO

Quante combinazioni di gioco prevede la lotteria *Win-for-life*? Si tratta di scegliere 10 numeri da un gruppo di 20, quindi:

$$C_{20,10} = \frac{20!}{10! \cdot (20-10)!} = \frac{20!}{10! \cdot 10!} = \frac{2\,432\,902\,008\,176\,640\,000}{3\,628\,800 \cdot 3\,628\,800} = 184\,756$$

La probabilità di fare un 10 è quindi una su 184 756.

Volendo scrivere un programma C++ che calcola le combinazioni di n elementi presi a gruppi di k si può effettuare il calcolo del fattoriale, per esempio del numero n, mediante un semplice ciclo:

```
long fattoriale=1;
for (i=1; i<=n; i++)
  fattoriale = fattoriale * i;
```

Il programma completo può quindi essere il seguente:

```
void main (void)
{
  int i, n, k;
  long Nfattoriale=1, Kfattoriale=1, NKfattoriale=1;

  cin>>n;
  cin>>k;
  for (i=1; i<=n; i++)
    Nfattoriale=Nfattoriale*i;
  for (i=1; i<=k; i++)
    Kfattoriale=Kfattoriale*i;
  for (i=1; i<=(n-k); i++)
    NKfattoriale=NKfattoriale*i;
  cout<<Nfattoriale/(Kfattoriale*NKfattoriale)<<endl;
}
```

1. La simbologia $N!$ denota il *fattoriale*, cioè il prodotto di tutti i numeri naturali da 1 a N inclusi:
$N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot 2 \cdot 1$
Per convenzione $0! = 1$.

OSSERVAZIONE I tre cicli *for* nel codice implementano lo stesso calcolo applicato a tre diversi valori (n, k e $n - k$). Scrivere una sola volta questo codice per applicarlo ai tre valori distinti renderebbe il programma più leggibile e modulare. Inoltre una eventuale modifica alla tecnica di calcolo del fattoriale (per esempio per correggere un errore) dovrebbe essere effettuata una sola volta.

1 Definizione e invocazione di una funzione

Il linguaggio C++ dispone di una soluzione specifica per evitare la replicazione del codice: le **funzioni**. Il programma per il calcolo delle combinazioni può essere così strutturato:

```
long fattoriale(int x)
{
  int i;
  long f=1;

  for (i=1; i<=x; i++)
    f=f*i;

  return f;
}

void main(void)
{
  int n, k;

  cin>>n;
  cin>>k;
  cout<<fattoriale(n)/(fattoriale(k)*fattoriale(n-k));
}
```

Il nuovo programma prevede l'enucleazione del calcolo del fattoriale dal corpo del programma con la definizione della funzione *fattoriale* invocata dal programma principale (*main*) tre volte con argomenti diversi: n, k e $n - k$.

In generale un programma C++ è formato da una o più funzioni di cui la principale assume il nome ***main***: si tratta della funzione che viene eseguita per prima all'avvio del programma.

OSSERVAZIONE Analizzando la funzione per il calcolo del fattoriale si vede come essa costituisce un «piccolo» programma autonomo.

- La prima riga della funzione ne definisce la **firma** (*signature*): essa comprende il nome, i parametri (con i relativi tipi) e il tipo del valore restituito.

Nel caso specifico la funzione sarà invocata con il nome *fattoriale*, riceverà in input un parametro denominato *x* di tipo `int` e restituirà in output come risultato **un solo valore** di tipo `long`.

- Le altre righe rappresentano il **corpo** della funzione che, nel linguaggio di programmazione C++, costituiscono un blocco delimitato dai simboli «{» e «}».

In generale, la definizione di una funzione ha la seguente struttura di massima:

```
tipo-risultato nome-funzione(tipo parametro-1,          } FIRMA
                             tipo parametro-2, …)       }
{
  tipo variabile-1;                                     }
  tipo variabile-2;                                     }
  …                                                     }
                                                        } CORPO
  istruzione;                                           }
  istruzione;                                           }
  …                                                     }
                                                        }
  return risultato;                                     }
}
```

OSSERVAZIONE L'istruzione `return` conclude l'esecuzione del codice del corpo di una funzione e costituisce il meccanismo per la restituzione del risultato della computazione eseguita dalla funzione stessa: il valore della variabile, o dell'espressione, che segue la parola chiave `return` è il valore della valutazione della funzione al momento dell'invocazione.

L'**invocazione di una funzione** precedentemente definita avviene specificandone il nome e i parametri su cui deve essere eseguito il calcolo che essa implementa.

ESEMPIO

L'istruzione

```
long alfa;
...
alfa = fattoriale(5);
```

assegna alla variabile *alfa* il risultato del calcolo del fattoriale del valore 5 (essendo il tipo del risultato della funzione *fattoriale* `long`, la variabile *alfa* deve essere dichiarata dello stesso tipo o di un tipo compatibile).

Allo stesso modo, l'istruzione

```
int z;
long alfa;
...
alfa = fattoriale(z);
```

esegue il calcolo del fattoriale del valore numerico che la variabile *z* assume al momento dell'invocazione della funzione stessa.

OSSERVAZIONE Oltre che nel contesto di un assegnamento del risultato a una variabile, una funzione può essere invocata anche tramite un suo utilizzo diretto all'interno di un'altra istruzione; per esempio:

```
cout<<fattoriale(n);
```

oppure

```
combinazioni = fattoriale(n)/(fattoriale(k)*fattoriale(n-k));
```

Nell'uso delle funzioni si distinguono i **parametri formali** dai **parametri attuali**: i primi sono quelli definiti nell'intestazione della funzione, mentre i secondi sono quelli specificati al momento della chiamata della funzione e sono anche chiamati argomenti della funzione.

ESEMPIO

Nella definizione della funzione *fattoriale* la variabile *x* rappresenta un parametro formale, mentre il valore costante 5 o la variabile *z* dell'esempio precedente sono esempi di parametri attuali.

Al momento dell'invocazione della funzione il valore dei parametri attuali viene «copiato» nelle variabili che costituiscono i parametri formali della funzione stessa ed è in base a questo valore che viene eseguita la computazione. Nel caso in cui una funzione preveda più di un parametro formale, nella sua invocazione devono essere specificati altrettanti parametri attuali di tipo conforme a quelli formali. L'associazione tra parametri formali e attuali avviene per posizione: il primo con il primo, il secondo con il secondo e così via.

ESEMPIO

La funzione di cui segue la definizione calcola e restituisce la potenza tra due valori interi (*b*, base, ed *e*, esponente) forniti come argomento:

```
int potenza(int b, int e)
{
  int p = 1;

  while (e > 0)
    {
      p = p*b;
      e--;
    }
  return p;
}
```

La seguente invocazione calcola il valore di 2^3 assegnando il risultato alla variabile *y*:

```
int x=2, y;
…
y = potenza(x,3);
…
```

Al momento dell'invocazione i parametri attuali (argomenti) sono copiati nelle variabili che costituiscono i parametri formali della funzione associandoli per posizione: il valore della variabile *x* (primo argomento) viene assegnato al primo parametro *b* e il valore costante 3 (secondo argomento) viene assegnato al parametro *e*.

OSSERVAZIONE Una funzione può non avere parametri: in questo caso la sua firma conterrà al posto dell'elenco dei parametri la parola chiave `void` e la sua invocazione avverrà specificandone il nome della funzione seguito dai simboli «(«e»)».

ESEMPIO

Una funzione che genera e restituisce un numero casuale potrebbe avere una firma di questo tipo:

```
float caso(void)
```

ed essere invocata come nel seguente frammento di codice:

```
float n;
...
n = caso();
```

La restituzione del risultato da parte di una funzione mediante l'istruzione `return` deve soddisfare le seguenti regole.

- Una funzione restituisce al più un unico valore in accordo con il tipo di risultato dichiarato nella sua firma.
- Una funzione può non restituire valori; in questo caso il tipo di risultato da dichiarare nella definizione della funzione è il tipo `void`.

OSSERVAZIONE Una funzione di tipo `void`, non restituendo alcun risultato, viene invocata direttamente, senza dover dipendere da istruzioni di assegnamento o di altro tipo.

- Una funzione può comprendere più istruzioni `return`: la prima istruzione di questo tipo raggiunta nel corso dell'esecuzione del codice della funzione ne causa la terminazione, con conseguente ripresa dell'esecuzione nel punto in cui era stata sospesa al momento dell'invocazione della funzione stessa.
- Una funzione di tipo `void` può non comprendere alcuna istruzione `return`; in questo caso l'esecuzione del codice della funzione termina con l'ultima istruzione della funzione stessa.
- Una funzione il cui risultato è un tipo qualsiasi diverso da `void` deve comprendere almeno un'istruzione `return` che specifichi un valore o una variabile di tipo coerente con il tipo definito nella dichiarazione della funzione stessa.

OSSERVAZIONE L'esecuzione del codice di una funzione *f* invocata dal codice di una funzione *F* può essere rappresentato come in FIGURA 1.

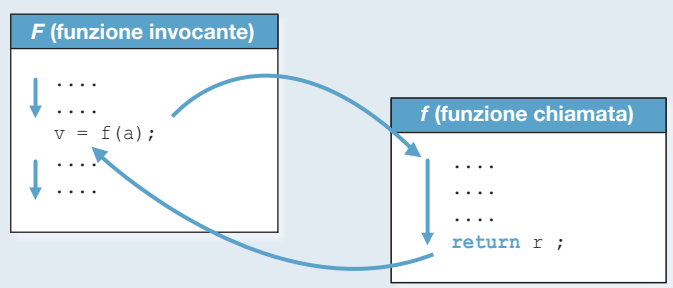


FIGURA 1

L'invocazione di una funzione prevede i seguenti passi:

- invocazione della funzione e memorizzazione del punto corrente di esecuzione nella funzione invocante;
- trasferimento dell'esecuzione alla prima istruzione della funzione invocata con relativa «copia» del valore dei parametri attuali nelle variabili che costituiscono i parametri formali;
- esecuzione del codice della funzione invocata;
- ripresa dell'esecuzione dal punto di sospensione nel codice della funzione invocante memorizzato al momento dell'invocazione con eventuale assegnazione del valore calcolato restituito dalla funzione invocata.

OSSERVAZIONE

L'ambiente interno di una funzione è locale alla funzione stessa: le variabili definite nel corpo della funzione – così come le variabili che costituiscono i parametri formali – hanno una visibilità e un tempo di vita limitato al tempo di esecuzione della funzione stessa.
Solo le variabili definite a livello globale (esternamente al corpo di una qualsiasi funzione, compresa la funzione *main*) sono visibili da parte di tutte in tutte le funzioni che compongono un programma. Valgono in proposito le stesse regole che si applicano a un blocco: se in una funzione viene definita una variabile che ha lo stesso nome di una variabile definita a livello globale, la variabile locale maschererà quella globale nel corso dell'esecuzione della funzione.

In ogni caso condividere mediante variabili definite a livello globale dati e risultati tra funzioni è fortemente scoraggiato perché rende di difficile interpretazione e manutenzione il codice.

Effetti collaterali nell'invocazione delle funzioni

Il ricorso a variabili globali è scoraggiato come pratica di programmazione perché altera il naturale meccanismo di input/output che le funzioni C++ mutuano dal concetto matematico di funzione. Il programmatore che analizza l'istruzione

```
y = f(x);
```

è portato, anche senza conoscerne il codice, a vedere la funzione *f* come un meccanismo per calcolare il valore da assegnare a *y* a partire dal valore di *x* e non a considerare che eventualmente anche il valore di una variabile globale *z* possa essere modificato. Gli effetti che una funzione può avere al di là della variazione del valore dei parametri passati per riferimento e della restituzione del risultato sono definiti **collaterali** (*side effect*).

ESEMPIO

La funzione *max* restituisce il massimo tra due valori numerici:

```
int max(int a, int b)
{
  if (a>b)
    return a;
  else
    return b;
}

void main(void)
{
  int m, x, y;

  cin>>x;
  cin>>y;
  m = max(x, y);
  cout<<m;
}
```

La funzione *max* ha due istruzioni `return`: il punto di uscita dal codice della funzione dipenderà dal valore degli argomenti passati alla funzione al momento della sua invocazione. In alternativa è possibile organizzare il codice della funzione in modo da avere un'unica istruzione `return`:

```
int max(int a, int b)
{
  int m;

  if (a>b)
    m=a;
  else
    m=b;
  return m;
}
```

ESEMPIO

La seguente funzione visualizza un messaggio che indica se il valore passato come argomento è positivo o negativo:

```
void visualizzaSegno(int a)
{
  if (a > 0)
    cout<<"positivo";
  else
    cout<<"negativo";
}

void main(void)
{
  int n;

  cin>>n;
  cout<<"il numero"<<n<<"e'";
  visualizzaSegno(n);
}
```

Nella funzione *visualizzaSegno* non compare l'istruzione **return**; infatti, essendo stata definita di tipo **void**, non è previsto un valore di ritorno e l'esecuzione del codice della funzione termina con l'ultima istruzione. L'introduzione esplicita di una istruzione **return** come ultima istruzione della funzione non ne avrebbe modificato la semantica.
In questo esempio la funzione produce direttamente un output: a meno che l'inserimento di dati, o la visualizzazione di risultati, non sia il compito specifico per cui una funzione viene realizzata, è bene seguire un approccio più generico in cui una funzione restituisce un risultato e il programma principale effettua le operazioni di input/output.

OSSERVAZIONE Così come i nomi delle variabili, anche i nomi delle funzioni dovrebbero essere significativi: a questo scopo è spesso necessario che siano composti da più parole che possono essere congiunte da un simbolo «_», oppure giustapposte con la tecnica della capitalizzazione dell'iniziale come nel caso dell'esempio precedente.

2 Passaggio dei parametri per valore e per riferimento

Nel linguaggio di programmazione C++ sono previste due distinte modalità per il passaggio dei parametri alle funzioni.

- **Passaggio dei parametri per valore**: al momento dell'invocazione della funzione viene effettuata una copia del valore dei parametri attuali nelle variabili dei corrispondenti parametri formali.

OSSERVAZIONE Nel passaggio dei parametri per valore qualsiasi modifica del valore dei parametri effettuata nel codice della funzione invocata non si rifletterà in alcun modo nel valore degli argomenti della funzione invocante.

- **Passaggio dei parametri per riferimento**: al momento dell'invocazione della funzione vengono associate le variabili che costituiscono i parametri attuali alle variabili che costituiscono i parametri formali, ovvero i parametri formali fanno riferimento alle stesse variabili che

costituiscono i parametri attuali. Il passaggio di parametri per riferimento si effettua anteponendo nell'intestazione della funzione al nome dei parametri formali il simbolo «&».

OSSERVAZIONE In pratica, nel passaggio dei parametri per riferimento, la funzione invocata e la funzione invocante utilizzano le stesse variabili e le eventuali modifiche apportate al valore dei parametri formali nel codice della funzione invocata si riflettono sul valore delle variabili corrispondenti passate come parametri attuali nel codice della funzione invocante.

ESEMPIO

La funzione *scambia* dovrebbe effettuare lo scambio del contenuto di due variabili:

```
void scambia(int a, int b)
{
  int t;

  t=a;
  a=b;
  b=t;
}

void main(void)
{
  int n, m;

  cin>>n;
  cin>>m;
  scambia(n, m) ;
  cout<<n;
  cout<<m;
}
```

In questo caso i parametri sono passati per valore: lo scambio dei valori avviene nel codice della funzione, ma non si riflette all'esterno nel contenuto delle variabili fornite alla funzione stessa come argomento. Inserendo i valori numerici 5 e 3 rispettivamente come parametri attuali per *n* ed *m*, dopo l'esecuzione della funzione *scambia* il loro contenuto sarà invariato; schematicamente la situazione è quella di FIGURA **2**.

Se invece i parametri sono passati per riferimento i valori della variabili *m* ed *n*, dopo l'invocazione della funzione *scambia*, risulteranno scambiati anche nel codice del programma principale:

```
void scambia(int &a, int &b)
{
  int t;

  t=a;
  a=b;
  b=t;
}

void main(void)
{
  int n, m;

  cin>>n;
  cin>>m;
  scambia(n, m);
  cout<<n;
  cout<<m;
}
```

Schematicamente la situazione è quella rappresentata in FIGURA **3**.

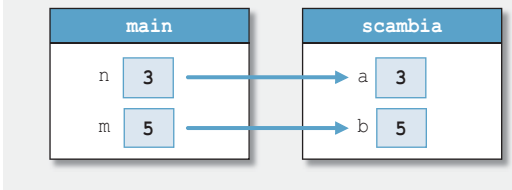


FIGURA **2**

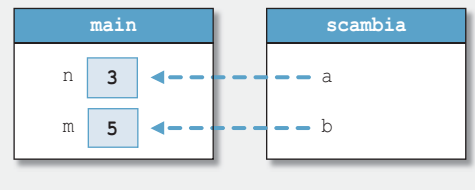


FIGURA **3**

OSSERVAZIONE La possibilità di passare a una funzione parametri per riferimento e non solo per valore consente di scrivere funzioni che producono più di un risultato – propagato al codice della funzione invocante mediante i parametri e non tramite l'istruzione `return` – senza utilizzare variabili definite a livello globale.

ESEMPIO

Molti programmatori riservano il valore restituito tramite l'istruzione `return` per fornire al codice che invoca la funzione un'informazione relativa all'eventuale presenza di errori negli argomenti o nella computazione eseguita. In questo caso, anche se il risultato è dato da un solo valore, è necessario utilizzare un parametro passato per riferimento, come nella seguente funzione per il calcolo della radice quadrata di un numero che invoca a sua volta la funzione di libreria *sqrt*:

```
#include <math.h>

// calcola la radice quadrata
// del parametro N nel parametro R
// restituisce il valore -1 in caso
// di errore, 0 altrimenti
int radice(float n, float &r)
{
  if (n<0)
    return -1;

  r = sqrt(n);
  return 0;
}

void main(void)
{
  int esito;
  float valore, risultato;

  cin>>valore;
  esito = radice(valore, risultato);
  if (esito < 0)
    cout<<"Errore";
  else
    cout<<risultato;
}
```

3 Prototipazione delle funzioni

Una funzione può invocare altre funzioni, che a loro volta ne possono invocare altre; quindi, nella stesura del codice sorgente, emerge la necessità di seguire un ordine nella dichiarazione delle funzioni. Il compilatore del linguaggio richiede infatti che al momento dell'invocazione una funzione sia stata precedentemente definita; in caso contrario – cioè se il compilatore incontra l'invocazione a una funzione prima della sua definizione – non conoscendone il numero e il tipo dei parametri sarebbe costretto a effettuare assunzioni arbitrarie e potenzialmente non corrette.

ESEMPIO

Se la funzione *f*3 invoca la funzione *f*2, che a sua volta invoca la funzione *f*1, allora le tre funzioni devono essere definite nell'ordine: *f*1, *f*2, *f*3.

Dato che, in particolare, se il programma è articolato in numerose funzioni, non è sempre agevole rispettare questa regola, è opportuno avvalersi della tecnica di prototipazione delle funzioni.

▶ Il **prototipo** di una funzione è costituito dalla sola firma della funzione (tipicamente un'unica riga di codice composta dal tipo e dal nome della funzione, seguiti dalla lista dei tipi dei singoli parametri formali) seguita dal simbolo «;».

ESEMPIO

Il prototipo della funzione *fattoriale* è il seguente:

```
long fattoriale(int);
```

OSSERVAZIONE Nella lista dei parametri del prototipo di una funzione è possibile – anche se non obbligatorio – inserire il nome dei parametri; in questo caso il nome può differire da quello dei parametri formali effettivi presenti nell'intestazione della definizione della funzione e utilizzati nel codice.
Quindi il prototipo della funzione *fattoriale* potrebbe essere scritto anche in questo modo:

```
long fattoriale(int N);
```

La definizione preventiva dei prototipi delle varie funzioni utilizzate in un programma permette al compilatore di conoscere le caratteristiche formali delle singole funzioni prima della loro invocazione, liberando il programmatore dalla necessità di seguire un ordine gerarchico nella loro definizione.

ESEMPIO

Volendo rendere disponibile una funzione per il calcolo del numero delle combinazioni di *n* elementi presi a gruppi di *k* si deve disporre della funzione *fattoriale*; il codice può essere organizzato come di seguito:

```
long fattoriale(int);

long combinazioni(int n, int k)
{
  long c = fattoriale(n)/(fattoriale(k)*fattoriale(n-k));
  return c;
}
```

La funzione fattoriale può essere definita, grazie alla presenza del prototipo che precede la definizione della funzione *combinazioni*, addirittura dopo la funzione *combinazioni* stessa.

Scomporre un programma in molteplici funzioni è una consolidata tecnica impiegata da molti programmatori: per dare ordine al codice, in particolare nel caso di «librerie» composte da numerose funzioni, si preferisce separarne le «interfacce» (i prototipi) dalle «implementazioni» (il codice).

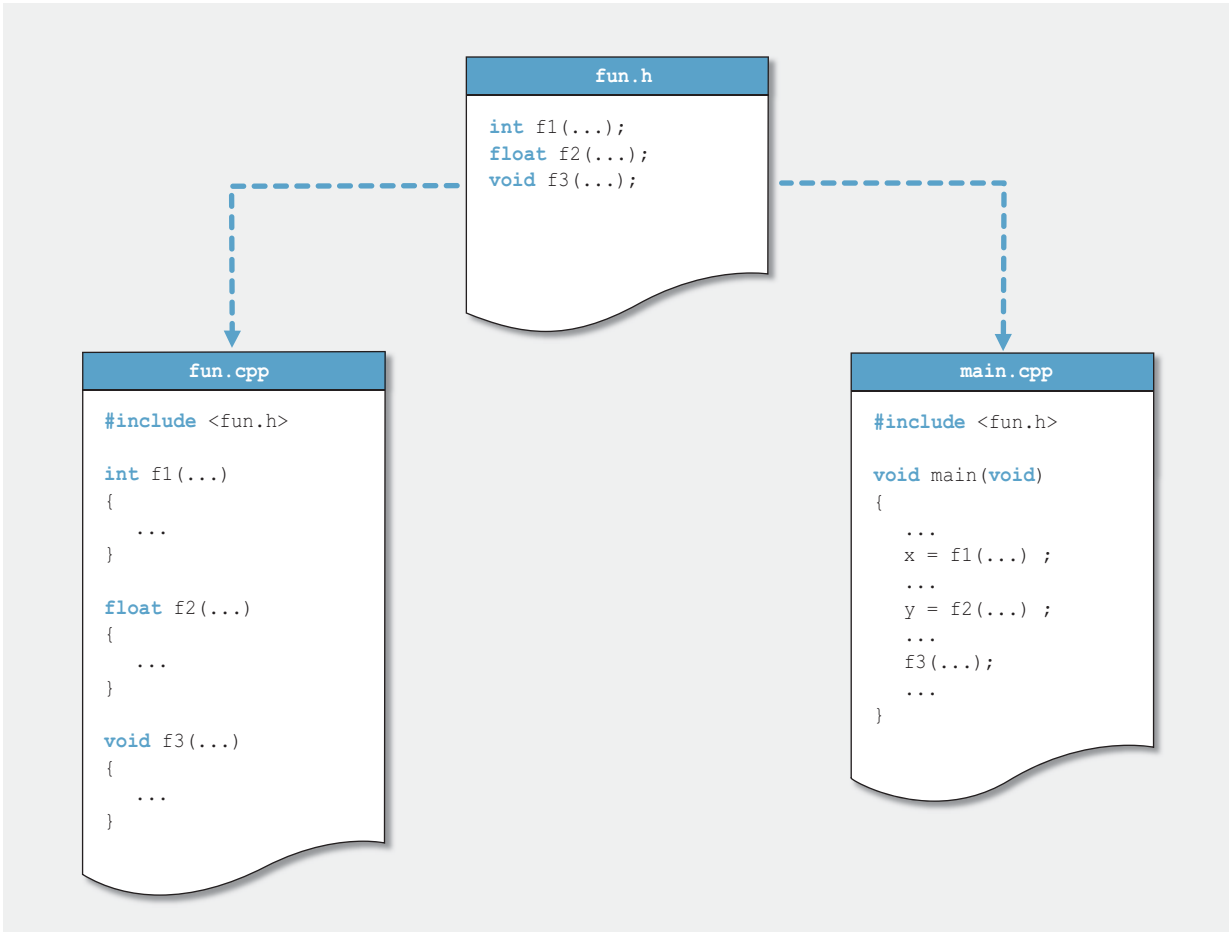


FIGURA **4**

A questo scopo i programmatori C++ sono soliti creare due file distinti: un file di intestazione (normalmente con estensione del nome «.h» o «.H») comprendente i soli prototipi e un file di codice (tipicamente con estensione del nome «.cpp» o «.CPP») per le funzioni complete del proprio corpo. Il file di intestazione deve essere incluso (con la direttiva `#include "nome-file"`) sia nel file che contiene il codice delle funzioni sia nel file che contiene il codice che invoca le funzioni stesse, come evidenziato nello schema di FIGURA **4**.

ESEMPIO

Il codice che segue implementa una libreria delle funzioni utili nel calcolo combinatorio di cui si riportano le formule[2]:

$$Combinazioni(n, k) = \frac{n!}{k! \cdot (n-k)!}$$

$$Permutazioni(n) = n!$$

$$Disposizioni(n, k) = n^k$$

Il file di intestazione *formule.h* comprende i soli prototipi delle funzioni utilizzabili e un breve commento sul loro scopo e il significato dei parametri:

2. Le *combinazioni* di n oggetti distinti presi a gruppi di k si contano trascurando l'ordinamento degli oggetti stessi, mentre le *disposizioni* di n oggetti distinti presi a gruppi di k si contano distinguendo casi con gli stessi oggetti in ordine diverso. Le *permutazioni* di n oggetti sono il numero di combinazioni in cui i gruppi sono composti da tutti gli n oggetti.

```
// calcola le permutazioni di N oggetti
long permutazioni(int N);

// calcola le combinazioni di N oggetti a gruppi di K
long combinazioni(int N, int K);

// calcola le disposizioni di N oggetti a gruppi di K
long disposizioni(int N, int K);
```

Il file di implementazione *formule.cpp* è invece costituito dalla definizione completa delle funzioni:

```
#include "formule.h"

static long fattoriale(int x)
{
  int i;
  long f=1;

  for (i=1; i<=x; i++)
    f=f*i;
  return f;
}

static int potenza(int b, int e)
{
  int p = 1;

  while (e > 0)
    {
      p = p*b;
      e--;
    }
  return p;
}

long combinazioni(int n, int k)
{
  long c = fattoriale(n)/(fattoriale(k)*fattoriale(n-k));
  return c;
}

long permutazioni(int n)
{
  return fattoriale(n);
}

long disposizioni(int n, int k)
{
  long d = potenza(n,k);
  return d;
}
```

Nel file in cui si intende invocare le funzioni della libreria è sufficiente includere il file di intestazione mediante la direttiva `#include "formule.h"`.

OSSERVAZIONE Le funzioni *fattoriale* e *potenza* dell'esempio precedente non sono rese disponibili per l'invocazione esternamente al codice delle sole funzioni contenute nel file *formule.cpp*. Infatti i relativi prototipi non sono presenti nel file di intestazione e la parola chiave `static` ne stabilisce la non visibilità dall'esterno, consentendo di avere in altri file dello stesso programma funzioni con lo stesso nome.

4 *Overloading* dei nomi delle funzioni

Nell'ambito dello stesso programma il linguaggio C++ permette di fornire più definizioni di funzioni aventi lo stesso nome, se queste prevedono parametri formali e/o tipi restituiti diversi.

Il compilatore distingue tra loro tali funzioni in base alla loro firma. Questa tecnica di definizione multipla di una funzione prende il nome di ***overloading*** (letteralmente «sovracaricamento») ed è una forma di **polimorfismo** (termine che deriva dal greco e ha il significato di «assumere più forme»).

Il polimorfismo nei linguaggi di programmazione

Il polimorfismo è implementato nei linguaggi di programmazione in modi diversi: nei moderni linguaggi a oggetti si impiegano sia il **polimorfismo per inclusione** sia il **polimorfismo parametrico**, le due forme che gli informatici teorici Luca Cardelli e Peter Wegner nel 1985 classificavano come «universali».
L'***overloading*** dei nomi delle funzioni viene invece considerata una forma di polimorfismo *ad hoc*, così come la coercizione del tipo. Quest'ultima forma di polimorfismo è molto utilizzata, anche se spesso in modo implicito, nei programmi C++. Per esempio, ogni volta che si esegue una moltiplicazione tra un valore di tipo `int` e un valore di tipo `float`, il compilatore trasforma il valore intero in un valore in virgola mobile, perché le operazioni aritmetiche possono essere effettuate solo tra tipi numerici omogenei. Si noti infine che gli operatori aritmetici del C++ sono di fatto polimorfici: per esempio lo stesso simbolo «*» viene utilizzato per moltiplicare tra loro due variabili di tipo `int` così come due variabili di tipo `float`.

ESEMPIO

La funzione illustrata in un esempio precedente consentiva di scambiare tra loro i valori di due variabili di tipo `int`:

```
void scambia(int &a, int &b)
{
  int t;

  t=a;
  a=b;
  b=t;
}
```

Se proviamo a utilizzarla per scambiare tra loro il contenuto di due variabili di tipo `float` il compilatore segnalerà un errore di tipo; d'altra parte sarebbe oltremodo noioso realizzare una libreria di funzioni *scambiaInt*, *scambiaLong*, *scambiaFloat*, *scambiaDouble*,… dove ciascuna funzione deve essere invocata in coerenza con il tipo dei parametri. Grazie all'*overloading* è invece possibile definire funzioni aventi lo stesso nome, ma con parametri di tipo diverso:

```
void scambia(float &a, float &b)
{
  float t;

  t=a;
  a=b;
  b=t;
}
```

Il compilatore selezionerà la funzione corretta in base al tipo di parametri; nel compilare il seguente frammento di codice

```
int x, y;
…
scambia(x,y);
```

sarà selezionata la prima variante della funzione, mentre nel compilare il seguente frammento di codice

```
float x, y;
…
scambia(x,y);
```

sarà individuata la seconda variante.

OSSERVAZIONE Con l'*overloading* dei nomi delle funzioni, così come con le altre forme di polimorfismo, si ottiene un equilibrio fra due esigenze apparentemente in contrasto:

- controllo stretto dei tipi allo scopo di rilevare in fase di compilazione il maggior numero possibile di errori;
- adeguata flessibilità per il programmatore.

Alcuni linguaggi di programmazione – come, per esempio, i linguaggi di *scripting* Java-script e PHP – non prevedono un controllo stretto dei tipi delle variabili e i controlli di coerenza sono effettuati in fase di esecuzione: la flessibilità per il programmatore è massima, ma alcuni errori di programmazione possono rimanere nascosti nel codice per periodi di tempo molto lunghi prima di manifestarsi.

Sintesi

Funzione. Le funzioni costituiscono il mezzo per realizzare la scomposizione del codice di un programma in «sottoprogrammi» riutilizzabili, evitando così inutili duplicazioni. In genere una funzione realizza una specifica computazione, restituendo se necessario un risultato.

Ambiente di una funzione. L'ambiente interno di una funzione è locale, nel senso che le variabili definite in essa hanno una visibilità e un tempo di vita relativo al tempo di invocazione ed esecuzione della funzione stessa. Solo le variabili «globali» dichiarate esternamente al corpo di qualsiasi funzione sono visibili nel codice di tutte le funzioni.

Firma di una funzione. La firma (o *signature*) di una funzione è rappresentata dall'intestazione della funzione stessa in cui sono dichiarati: il nome, i parametri e i tipi relativi e il tipo del valore restituito (nel caso in cui non sia previsto alcun valore di ritorno la funzione sarà definita di tipo `void`).

Parametri formali. I parametri formali di una funzione costituiscono i dati di input; il nome e il tipo dei singoli parametri formali sono dichiarati in forma di elenco nell'intestazione della funzione stessa (nel caso in cui non sia previsto alcun parametro l'elenco conterrà la sola parola chiave `void`).

Parametri attuali (o argomenti). Sono i parametri (variabili o valori costanti) forniti a una funzione all'atto della sua invocazione. L'associazione tra parametri attuali e parametri formali avviene per posizione; il numero e il tipo dei parametri attuali deve coincidere con il numero e il tipo dei parametri formali della funzione stessa.

Passaggio di parametri per valore. Viene effettuata una copia dei parametri attuali nei corrispondenti parametri formali che sono utilizzati nel contesto dell'ambiente locale del corpo della funzione. Il passaggio di parametri per valore fa sì che una loro qualsiasi modifica effettuata all'interno della funzione invocata non si rifletterà in variazioni nelle variabili della funzione invocante.

Passaggio di parametri per riferimento. Viene passato un «riferimento» alle variabili che costituiscono gli argomenti della funzione, ovvero i parametri formali fanno riferimento alle stesse variabili che contengono i valori dei parametri attuali. La funzione invocata e il codice invocante utilizzano le stesse variabili ed eventuali modifiche che i parametri formali subiscono all'interno della funzione invocata si riflettono nelle variabili relative ai corrispondenti parametri attuali anche nel codice invocante. Il passaggio di parametri per riferimento si effettua ponendo davanti al nome dei parametri formali il simbolo «&».

`return`. È l'istruzione di uscita dalla funzione che restituisce il controllo dell'esecuzione alla funzione invocante. Se la funzione invocata prevede un valore di ritorno, questo deve essere specificato contestualmente all'istruzione ed esso deve essere congruente al tipo dichiarato per la funzione. Una funzione può avere più di una istruzione `return`: la prima di queste a essere eseguita causerà la terminazione della funzione. Una funzione dichiarata di tipo `void` può anche non avere una istruzione `return`; in questo caso essa è da considerarsi come implicitamente inserita dopo l'ultima istruzione della funzione.

Prototipo. Il prototipo di una funzione è costituito dalla sola intestazione che ne comprende il tipo e il nome seguito dalla lista dei tipi dei singoli parametri formali. La prototipazione delle funzioni serve per comunicare al compilatore le caratteristiche formali delle stesse ed evitare al programmatore di rispettare, nella codifica di un programma, una gerarchia delle dichiarazioni delle funzioni basata sull'ordine delle invocazioni (una funzione deve infatti essere definita prima delle funzioni che la invocano).

***Overloading* del nome di una funzione**. Nell'ambito di uno stesso programma il linguaggio C++ permette di fornire più definizioni di una stessa funzione, se queste prevedono parametri formali diversi. Il compilatore distingue tra loro tali funzioni in base alla loro firma. La pratica di definire più volte la stessa funzione con parametri diversi nasce dalla necessità di consentirne un utilizzo più flessibile.

QUESITI

1 **Al termine dell'esecuzione di una funzione ...**

A ... i valori delle variabili locali sono persi.

B ... i valori delle variabili locali sono copiati in quelle globali aventi lo stesso nome.

C ... i valori delle variabili locali sono restituiti alla funzione invocante come valori di ritorno.

D ... i valori delle variabili locali sono mantenuti fino alla prossima invocazione della funzione.

2 **Se una variabile locale di una funzione e una variabile globale hanno lo stesso nome ...**

A ... è come se la variabile globale non fosse stata dichiarata.

B ... è come se la variabile locale non fosse stata dichiarata.

C ... la variabile locale nasconde la variabile globale durante l'esecuzione della funzione.

D ... la variabile globale nasconde la variabile locale durante l'esecuzione della funzione.

3 **Indicare quali delle seguenti affermazioni sono corrette.**

A Il passaggio di parametri per riferimento assicura che le variabili interessate non subiscano variazioni durante l'esecuzione della funzione.

B Nel codice di una funzione l'istruzione `return` non sempre è obbligatoria.

C Il passaggio di parametri per valore consente di fornire alla funzione invocata il valore contenuto in una o più variabili.

D Nel codice che invoca una funzione sono visibili le variabili locali delle funzioni invocate.

4 **È dato il seguente programma C++ che invoca la funzione *f*:**

```
void f(int x)
{
  int k, j;

  j=x;
  for (k=0; k<=3; k++)
    x-=k;
  cout<<x;
}
void main(void)
{
  int a=10;

  f(a);
  cout<<a;
}
```

Indicare quali valori visualizza quando eseguito.

A 10, 9, 8, 7.

B 10, 9, 7, 4.

C 4.

D Nessuna delle precedenti risposte è corretta.

5 **Sono date le seguenti funzioni C++:**

```
void f1(int a, int b)
{
  int t=a;

  a=b
  b=t;
  return;
}

void f2(int &a, int &b)
{
  int t=b;

  b=a;
  a=t;
}
```

Scrivere il valore delle variabili *x* e *y* al termine dell'esecuzione dei seguenti frammenti di codice:

```
int x=5, y=7;
…
f1(x,y);

int x=5, y=7;
…
f2(x,y);
```

6 **È dato il seguente frammento di codice:**

```
int t = -1;
for (int x=1; x<=n; x++)
  if (f(x))
    t = x;
if (t>=0)
  cout<<t;
```

Stabilire quali delle seguenti affermazioni a proposito di esso sono corrette.

A Visualizza il più piccolo intero fra 1 e n tale che $f(x) \neq 0$; se tale intero non esiste il codice entra in un ciclo infinito.

B Visualizza il più grande intero fra 1 e n tale che $f(x) \neq 0$; se tale intero non esiste il codice entra in un ciclo infinito.

C Visualizza il più piccolo intero fra 1 e n tale che $f(x) \neq 0$; se tale intero non esiste il codice non visualizza nulla.

D Visualizza il più grande intero fra 1 e n tale che $f(x) \neq 0$; se tale intero non esiste il codice non visualizza nulla.

7 **Sono date le seguenti definizioni di funzioni:**

```
int f(int x)
  {
    return x*x;
  }

float f(float x)
  {
    return x+x;
  }

int f(int x, int y)
  {
    return x/y;
  }

float f(float x, float y)
  {
    return x-y;
  }
```

Scrivere il valore delle variabili a, b, c e d dopo l'esecuzione del seguente frammento di codice:

```
int a=1, b=2;
float c = 0.0, d=-1.0;
…
a = f(a);
b = f(a, b);
c = f(c);
d = f(c, d);
```

8 **Che cosa verifica la seguente funzione C++?**

```
bool mistero(int n)
{
  int x;

  x = sqrt(n);
  if (x*x == n)
    return true;
  else
    return false;
}
```

9 **Che cosa visualizza il seguente programma C++?**

```
void main(void)
{
  int N;

  cout<<"Inserire un numero intero
        positivo: ";
  cin>>N;

  while (N > 0)
    {
      if (mistery(N))
        cout<<N<<endl;
      N--;
    }
}
```

ESERCIZI

1 **Il tachimetro per biciclette determina la velocità espressa in metri/minuto a partire dal numero g di giri della ruota conteggiati in un periodo di 10 s in base alla seguente formula:**

$$v = 3{,}1416 \cdot d \cdot g \cdot 6$$

dove d è il diametro della ruota espresso in metri. Scrivere una funzione C++ che, a partire dal diametro della ruota e dal numero di giri conteggiati in 10 s, determini la velocità in km/h della bicicletta.

2 **Un pendolo oscilla con un periodo P espresso in secondi che è funzione della lunghezza l in metri del filo di sospensione secondo la seguente formula:**

$$P = 6{,}2832 \cdot \sqrt{\frac{l}{g}}$$

dove la costante g è l'accelerazione di gravità, che sulla Terra vale 9,8 m/s^2. Scrivere una funzione C++ che determini la lunghezza di un pendolo, dato come parametro il periodo con cui deve oscillare.

3 **Scrivere una funzione C++ che, a partire dalle coordinate cartesiane di due punti, ne calcoli la distanza. Scrivere un programma C++ che, dopo avere richiesto all'utente le coordinate di tre pun-**

ti nel piano cartesiano, calcoli il perimetro del triangolo avente i tre punti come vertici.

4 Le macchine fotografiche digitali rappresentano i colori come combinazione del rosso, del verde e del blu: il colore di ogni singolo punto dell'immagine è rappresentato da un numero compreso tra 0 e 255; le stampanti digitali rappresentano i colori come combinazione del celeste, del magenta e del giallo: anche in questo caso il colore di ogni singolo punto della stampa è rappresentato da un numero compreso tra 0 e 255. Le seguenti formule consentono di trasformare i colori dal sistema rosso/verde/blu nei colori del sistema celeste/magenta/giallo:

```
Celeste = 255 - Rosso
Magenta = 255 - Verde
 Giallo = 255 - Blu
```

Scrivere una funzione C++ che determini le singole componenti di un colore di stampa (celeste, magenta e giallo) a partire dalle singole componenti di un colore fotografico (rosso, verde e blu).

5 Scrivere una funzione C++ che determini il MCD (massimo comune divisore) tra due numeri interi *m* ed *n* (con $m > n$) utilizzando il seguente algoritmo attribuito ad Euclide:

1) determinare il resto *r* della divisione tra *m* ed *n*;
2) se *r* è uguale a 0 terminare: il massimo comune divisore tra *m* ed *n* è *n*;
3) porre *m* uguale al valore di *n*;
4) porre *n* uguale al valore di *r* e riprendere dal passo 1.

Scrivere un programma C++ che calcoli il MCD tra due numeri forniti in input utilizzando la funzione precedente.

6 Scrivere una funzione C++ che, a partire dal valore di due parametri interi che rappresentano rispettivamente il numeratore e il denominatore di una frazione, li modifichi in modo da rendere la frazione risultante ridotta ai minimi termini[3].

7 Scrivere una funzione C++ che aggiorni il valore di una variabile numerica non intera fornita come argomento al valore dell'intero più prossimo.

8 Scrivere una funzione C++ che determini il prodotto di due frazioni i cui numeratori e denominatori siano forniti come argomenti.

3. La riduzione di una frazione ai minimi termini si ottiene dividendo sia il numeratore sia il denominatore per il loro MCD.

9 Scrivere una funzione C++ che restituisca `true` se il numero intero fornito come argomento è primo, `false` altrimenti (per verificare se un numero intero *n* è primo è sufficiente verificare che non è divisibile per nessuno dei numeri interi compresi tra 2 e *n*/2 inclusi). Scrivere un programma C++ che, utilizzando la funzione precedente, visualizzi quali numeri primi sono presenti tra 1 e un valore *N* fornito in input.

10 Un famoso «gioco» matematico consiste nel costruire una sequenza di numeri interi in modo che ogni numero sia la metà del precedente se esso è pari, altrimenti il successore del suo triplo: partendo da un numero qualsiasi si prosegue fino a raggiungere 1. Scrivere una funzione C++ che abbia come argomento il numero iniziale e restituisca come risultato il numero di iterazioni necessario per concludere la sequenza. Scrivere un programma C++ che, utilizzando la funzione precedente, visualizzi la lunghezza delle sequenze per numeri iniziali compresi tra due valori forniti in input.

11 Scrivere una funzione C++ che, accettato come parametro un intero rappresentante il valore di un anno, restituisca 1 se l'anno è bisestile, 0 altrimenti.

12 Scrivere una funzione C++ che, accettati come parametri sei interi, rappresentanti rispettivamente giorno, mese e anno di due date, restituisca come risultato il numero di giorni che intercorrono tra la prima e la seconda data. Realizzare la funzione per l'anno commerciale, ovvero l'anno di 360 giorni in cui tutti i mesi sono di trenta giorni. Provare a realizzare la funzione anche per l'anno solare, ovvero tenendo conto dei giorni reali dei singoli mesi e degli eventuali anni bisestili.

13 Scrivere una funzione C++ che, accettati come parametri quattro interi, rappresentanti, i primi tre, giorno, mese e anno di una data, calcoli nei primi tre parametri giorno, mese e anno della data ottenuta sommando il numero di giorni rappresentato dal quarto alla data di partenza (si tenga presente che il quarto parametro può essere sia positivo sia negativo). Realizzare la funzione per l'anno commerciale, ovvero l'anno di 360 giorni in cui tutti i mesi sono di trenta giorni. Provare a realizzare la funzione anche per l'anno solare, ovvero tenendo conto dei giorni reali dei singoli mesi e degli eventuali anni bisestili.

A6 Gli *array* in C++

Supponiamo di disporre dei valori, espressi in millimetri di pioggia, delle precipitazioni nei vari mesi dell'anno per una certa regione geografica e di volerne calcolare prima il valore medio, per poi individuare quali sono i mesi in cui le precipitazioni sono risultate al di sopra della media.
Le operazioni da eseguire sono nell'ordine:

- acquisire i dati relativi alle precipitazioni nei 12 mesi;
- calcolarne il valore medio;
- individuare i mesi i cui valori di precipitazione sono superiori alla media annua.

Risulta immediatamente chiaro che un approccio risolutivo basato sull'uso di variabili semplici (ne occorrerebbero 12 solo per memorizzare i millimetri di pioggia caduti da gennaio a dicembre) conduca alla realizzazione di un programma poco compatto e stilisticamente scadente dal punto di vista del codice da implementare.
Un ulteriore esempio in tal senso è rappresentato da quei problemi matematici che per la loro risoluzione richiedono l'impostazione di sistemi di equazioni: i coefficienti delle incognite devono essere visti come un insieme di valori in forma matriciale.
In molte situazioni l'uso di variabili semplici non permette una gestione efficiente dei dati che caratterizzano lo scenario del problema da risolvere: a questo proposito i linguaggi di programmazione prevedono tipi di dati strutturati, denominati *array*, che permettono un'organizzazione e una gestione razionale di insiemi di dati tra di loro correlati.

1 *Array* mono e bidimensionali

1.1 *Array* e indici

In prima istanza possiamo immaginare un *array* come una collezione di variabili a cui ci si può riferire come a un unico insieme.

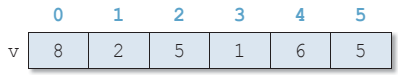
▶ Un ***array*** è un insieme omogeneo di dati identificato da un nome e dove ogni singolo elemento è identificato dal nome dell'*array* con associata una lista di indici.

Esaminiamo i dettagli di tale definizione:

- **Insieme omogeneo**: tutti gli elementi sono dello stesso tipo (tutti valori interi, oppure tutti caratteri, oppure…).
- **Insieme identificato da un nome**: all'insieme nel suo complesso viene associato un nome, così come avviene nel caso di una variabile semplice.

ESEMPIO

Di seguito è rappresentato un *array* monodimensionale («vettore») di nome *v* composto da 6 valori interi:



- **Ogni singolo elemento è identificato dal nome dell'*array* con associata una lista di indici**: con la notazione *v*[3] si fa riferimento all'elemento di indice 3 dell'insieme *v*; in proposito c'è da notare come nei linguaggi di programmazione – in generale e in particolare per il linguaggio C++ – il valore dell'indice assuma valori nell'intervallo 0, $N-1$, dove N è il numero di elementi dell'*array*; in questo senso un indice di valore i farà riferimento alla $(i+1)$-esima posizione dell'*array*.

ESEMPIO

Nel caso del vettore *v* di cui sopra abbiamo:

```
v[0] = 8    v[1] = 2    v[2] = 5
v[3] = 1    v[4] = 6    v[5] = 5
```

OSSERVAZIONE Si ponga attenzione alla differenza che c'è tra la posizione (identificata dall'indice) di un elemento dell'*array* e il suo contenuto. Per esempio, nell'*array* dell'esempio precedente il terzo elemento ha indice 2 e valore 5.
Con le espressioni *v*[*i*] e *v*[*j*] si indicano rispettivamente l'elemento di indice *i* e quello di indice *j* del vettore *v* (che coincidono solo nel caso che il valore di *i* sia uguale a quello di *j*).
Le espressioni *v*[*i*] e *w*[*i*] indicano due elementi di identica posizione in due vettori diversi, rispettivamente nel vettore *v* e nel vettore *w*.

Da quanto appena detto è chiaro che una variabile indice non è mai legata esclusivamente a uno specifico *array*.

- Una ulteriore puntualizzazione è necessaria in relazione al fatto che si faccia riferimento a una **lista di indici**. Nell'esempio che segue è rappresentato un *array* bidimensionale (**matrice**) *m* composto da un totale di 15 elementi organizzati su due dimensioni: 3 insiemi orizzontali detti **righe** e 5 insiemi verticali detti **colonne**. In questo contesto, con la

notazione $m[i][j]$ (essendo l'*array* bidimensionale occorre specificare una lista di due indici di cui il primo è relativo alla riga e il secondo alla colonna) viene identificato l'elemento che si trova alle coordinate (i, j) ovvero all'incrocio tra la i-esima riga e la j-esima colonna.

ESEMPIO

m	0	1	2	3	4
0	4	2	5	1	6
1	3	8	9	7	10
2	11	12	13	**14**	15

Con la notazione *m*[2][3] viene individuato l'elemento che si trova alle coordinate (2, 3), ovvero all'incrocio tra la terza riga e la quarta colonna (si ricordi che la numerazione sia delle righe sia delle colonne parte dall'indice 0) il cui contenuto è il valore 14.

OSSERVAZIONE In generale è possibile definire ***array* a più dimensioni**: per individuare uno specifico elemento sarà necessario specificare il nome dell'*array* associato a tanti indici quante sono le dimensioni dell'*array*. Nella nostra trattazione ci limiteremo a esaminare le caratteristiche degli *array* monodimensionali e bidimensionali (a una o due dimensioni), denominati rispettivamente **vettori** e **matrici**.

1.2 Vettori

In C/C++ la dichiarazione di un vettore viene effettuata tramite una istruzione del tipo:

```
float alfa[10];
```

dove il valore indicato tra parentesi rappresenta il numero degli elementi in singola precisione del vettore *alfa*.

La **dimensione di un *array*** viene fissata all'atto della sua dichiarazione e rimane inalterata durante l'esecuzione del programma in cui esso è utilizzato.

OSSERVAZIONE Dal momento che un *array* ha una dimensione fissa, all'atto della dichiarazione il numero dei suoi elementi deve essere espresso da una costante. Sono quindi accettate dichiarazioni del tipo:

```
int alfa[10];
```

oppure

```
const int dim=10;
int alfa[dim];
```

Non è invece ammessa un dichiarazione del tipo:

```
int d=10;
int alfa[d];
```

essendo *d* una variabile.

L'inizializzazione di un vettore può avvenire contestualmente alla sua dichiarazione con istruzioni del tipo:

```
int beta[5] = {3,2,5,4,3};
```

oppure con

```
int gamma[10] = {0};
```

Nel primo caso agli elementi del vettore *beta* sono assegnati i valori espressi tra parentesi graffe, rispettivamente a partire dalla posizione di indice 0 fino a quella di indice 4; nel secondo caso i dieci elementi di *gamma* sono inizializzati tutti a 0.

Come avviene per le variabili, gli elementi di un vettore non inizializzati contengono un valore indefinito.

Ovviamente è possibile modificare singolarmente gli elementi di un vettore assegnando loro un nuovo valore; per esempio con l'istruzione:

```
beta[2] = 9;
```

si intende assegnare all'elemento di indice 2 (posizione 3) del vettore *beta* il valore 9.

Nella gestione degli *array* rivestono una particolare importanza i **costrutti iterativi del linguaggio**: è proprio tramite l'uso di questi che si riesce a esprimere la potenza dell'uso degli *array*.

Con il codice seguente:

```
for (int i=0; i<10; i++)
  cout<<v[i]<<" ";
```

è possibile visualizzare il contenuto dei primi dieci elementi del vettore *v* (si noti che per visualizzare il contenuto di dieci diverse variabili sarebbero occorse altrettante istruzioni), poiché l'istruzione interna al ciclo viene eseguita dieci volte, cambiando per ognuna di esse il valore dell'indice *i*, che assumerà, uno alla volta, tutti i valori compresi tra 0 e 9.

Array associativi

In alcuni linguaggi di programmazione, come PHP e Javascript, è possibile utilizzare *array* i cui elementi non sono tra loro omogenei e sono accessibili, oltre che mediante il classico indice numerico, mediante nomi cui il contenuto dell'elemento corrispondente viene associato.
Nell'esempio che segue, nel vettore PHP $prezzo sono inseriti i prezzi relativi alla frutta:

```
$prezzo["mele"]=1.00;
$prezzo["pere"]=1.50;
```

Usando un indice di questo tipo non è necessario conoscere la posizione di un certo elemento, ma solo il nome che lo identifica, come nell'esempio che segue:

```
$spesa =
$prezzo["mele"]*1.5 +
$prezzo["pere"]*2;
```

L'interposizione di un carattere *blank* tra ogni valore visualizzato e il successivo si rende necessaria per far sì che questi risultino chiaramente distinguibili (utilizzando `"\t"` invece del *blank* è possibile separare i valori del vettore mediante il valore predefinito della tabulazione orizzontale).
Allo stesso modo con il ciclo

```
for (int i=0; i<10; i++)
  cin>>v[i];
```

è possibile inserire da tastiera, uno alla volta, i primi dieci elementi del vettore *v*.

OSSERVAZIONE È bene sottolineare come, a differenza di quanto avviene in altri linguaggi di programmazione, il C++ non effettua alcun controllo né segnalazione circa il fatto che nell'utilizzo di un indice si possa fuoriuscire dalla dimensione prefissata dell'*array*. Dal momento che questa responsabilità viene affidata completamente al programmatore, la questione diventa di primaria importanza se si considera che nell'allocazione dello spazio di memoria riservato a un programma può accadere che contiguamente a un vettore siano state memorizzate altre variabili: fuoriuscendo inavvertitamente dallo spazio riservato a un *array* si può andare a operare erroneamente su dati che non appartengono all'*array* stesso, con conseguenze imprevedibili.

OSSERVAZIONE Gli indici degli *array* devono essere in generale numeri interi, per cui possono essere espressi come costanti (0, 1, 2, …), come valori di una variabile (*i*, *j*, …) o come risultato di una qualsiasi espressione la cui valutazione fornisca un numero intero ($j + 1$, $i + k \cdot 2$, …).

ESEMPIO

Prendiamo in considerazione le due seguenti espressioni:

```
v[i]+1;
v[i+1];
```

dove *v* è il seguente vettore:

	0	1	2	3	4	5
v	8	2	4	1	6	5

Il significato delle due espressioni è profondamente diverso: nel primo caso si somma 1 al valore dell'elemento di indice *i* del vettore *v*, nel secondo si fa riferimento al valore dell'elemento di indice $i + 1$. Supponendo $i = 2$, la valutazione della prima espressione fornisce come risultato 5, mentre la valutazione della seconda espressione vale 1.

Per valutare correttamente espressioni che fanno riferimento a elementi di un *array* conviene operare con una tecnica che potremmo definire **per sostituzione**. Per esempio, facendo riferimento al vettore *v* definito nell'esempio precedente, nel seguente insieme di istruzioni:

```
...
i = v[2];
j = 1;
v[i-j] = v[i]+j;
...
```

si ottengono i seguenti passaggi successivi:

1	2	3	4
i=4; j=1; v[i-j]=v[i]+j;	i=4; j=1; v[4-1]=v[4]+2;	i=4; j=1; v[3]=6+2;	i=4; j=1; v[3]=8;

al termine dei quali il contenuto di *v* sarà:

	0	1	2	3	4	5
v	8	2	4	8	6	5

Si noti che nel processo di valutazione le espressioni del tipo *v*[…] vengono completamente sostituite con un valore solo, se si trovano a destra dell'operatore di assegnamento, mentre se si trovano a sinistra, viene sostituita completamente solo la parte che rappresenta l'indice (nell'ultimo passaggio si ottiene *v*[3] = 8). Questo perché a sinistra di un operatore di assegnamento deve comunque esserci una variabile che assuma il risultato dell'espressione che sta a destra.

ESEMPI

■ Il seguente frammento di codice moltiplica per 3 e somma 1 agli elementi dispari di un vettore *v* di 100 interi e dimezza gli elementi pari:

```
...
int v[100];
...
for (int i=0; i<100; i++)
  if (v[i] % 2 == 0) // pari
    v[i] = v[i]/2;
  else // dispari
    v[i] = v[i]*3 + 1;
...
```

■ Il seguente frammento di codice somma nella variabile *s* il valore di tutti gli elementi di un vettore *v* di interi dopo avere moltiplicato per 3 gli elementi di posizione dispari:

```
...
int v[12], s=0;
...
for (int i=0; i<12; i++)
  if (i % 2 != 0) // dispari
    v[i] = v[i]*3;
for (int i=0; i<12; i++)
  s = s + v[i];
...
```

Siamo ora in grado di risolvere il problema del paragrafo iniziale, scrivendo un programma completo:

```
void main(void)
{
  int pioggia[12];
  int media, somma = 0;

  // inserimento dei valori mensili delle precipitazioni
  for (int i=0; i<12; i++)
    {
      cout<<"Inserire i mm di pioggia del mese "<<(i+1);
      cin>>pioggia[i];
    }

  // calcolo della somma e della media delle precipitazioni
  for (int i=0; i<12; i++)
    somma = somma + pioggia[i];
  media = somma/12;

  // ricerca delle precipitazioni superiori alla media
  for (int i=0; i<12; i++)
    if (pioggia[i] > media)
      cout<<"Il mese "<<(i+1)<<" le precipitazioni hanno
            superato la media";
}
```

In molte situazioni si ha la necessità di scambiare tra di loro due elementi di un vettore: effettuare lo scambio significa scambiare il **contenuto** di due elementi. Lo scambio avviene nella stessa modalità usata per lo scambio del valore di due variabili semplici: usando una variabile intermedia. La seguente funzione scambia il contenuto dell'elemento di indice *i* del vettore *v* con l'elemento di indice *j* del vettore di valori numerici in singola precisione *v* dichiarato globalmente rispetto alla funzione (in generale la variabile intermedia dovrà avere lo stesso tipo di *v*):

```
int v[1000];
...
void swap(int i, int j)
{
  int t;

  t = v[i];
  v[i] = v[j];
  v[j] = t;
}
```

1.3 Matrici

Nel caso di *array* bidimensionali la scansione degli elementi, per esempio per la visualizzazione del loro contenuto, necessita dell'uso di due cicli nidificati.

ESEMPIO

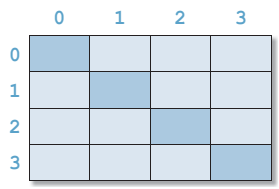
Il codice che segue visualizza una matrice *m* di 4 righe per 7 colonne:

```
...
float m[4][7];
...
for (int i=0; i<4; i++)
  {
    for (int j=0; j<7; j++)
      cout<<m[i][j]<<"\t";
    cout<<endl;
  }
...
```

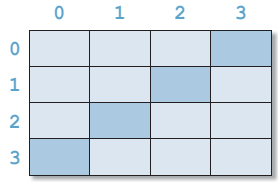
La matrice viene visualizzata «per righe», nel senso che per ogni singola riga vengono visualizzati i valori delle varie colonne. Invertendo l'ordine dei due cicli `for` si ottiene una visualizzazione «per colonne». Si noti come sia stato utilizzato `cout<<endl;` per andare a capo nella visualizzazione ogni volta che termina il ciclo interno (completamento della visualizzazione degli elementi di una riga).

Nel caso di una matrice si usa dire che questa è $N \times M$, indicando con N il numero delle righe e con M il numero delle colonne; se i due valori M ed N sono uguali, la matrice viene detta **quadrata**.
Per una matrice m quadrata $N \times N$ si individuano almeno quattro sottoinsiemi di valori notevoli:

- **diagonale principale**: sono gli elementi della matrice in cui gli indici di riga e di colonna sono coincidenti ($m[0][0]$, $m[1][1]$, ..., $m[i][i]$, ..., $m[N][N]$)

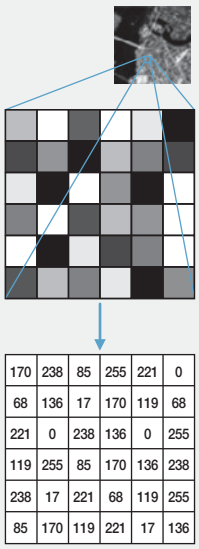


- **diagonale secondaria**: sono gli elementi della matrice i cui indici sono del tipo (i, $N-i-1$)



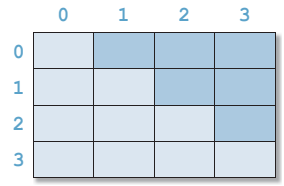
Immagini digitali

Le immagini digitali sono matrici di elementi, denominati **pixel**, il cui valore numerico ne rappresenta l'intensità:

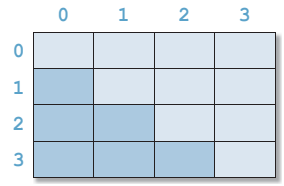


Nel caso di immagini a colori ogni singolo pixel è codificato mediante l'intensità dei 3 colori fondamentali: rosso, verde e blu.

- **sottomatrice triangolare superiore**: è l'insieme degli elementi per cui l'indice di riga è minore dell'indice di colonna (è la sottomatrice al di sopra della diagonale principale)



- **sottomatrice triangolare inferiore**: è l'insieme degli elementi per cui l'indice di riga è maggiore dell'indice di colonna (è la sottomatrice al di sotto della diagonale principale)



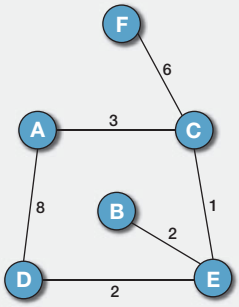
ESEMPIO

Il seguente frammento di codice calcola nelle variabili *somma_diag*, *somma_sup* e *somma_inf* rispettivamente la somma degli elementi della diagonale principale, della sottomatrice triangolare superiore e della sottomatrice triangolare inferiore della matrice di valori numerici *m*:

```
...
float m[10][10];
float somma_diag = 0.0;
float somma_sup = 0.0;
float somma_inf = 0.0;
...
for (int r=0; r<10; r++)
  {
    for (int c=0; c<10; c++)
      {
        if (r == c) // diagonale principale
          somma_diag = somma_diag + m[r][c];
        else
          {
            if (r<c) // triangolare superiore
              somma_sup = somma_sup + m[r][c];
            else // triangolare inferiore
              somma_inf = somma_inf + m[r][c];
          }
      }
  }
...
```

Matrici e grafi

Le matrici sono utilizzate per rappresentare numericamente **grafi** di reti di comunicazione (stradali, ferroviarie ecc.) e di servizi (distribuzione acqua, gas, elettricità ecc.). Per esempio nella figura è rappresentato un grafo dove i nodi sono etichettati con le lettere *A*, *B*, *C*, *D*, *E* ed *F* e i valori sui singoli archi rappresentano il «costo» del percorso tra i vari nodi:



I valori degli elementi della matrice che segue rappresentano le connessioni dirette tra i nodi con indicazione del relativo «costo»; i valori mancanti indicano la mancanza di connessione diretta:

	A	B	C	D	E	F
A	0		3	8		
B		0			2	
C	3		0		1	6
D	8			0	2	
E		2	1	2	0	
F			6			0

Nella memoria del computer gli elementi di un *array* sono disposti tutti sequenzialmente contigui, indipendentemente dal fatto che si tratti di un *array* monodimensionale o bidimensionale: è la modalità con cui accediamo a essi che mostra la loro organizzazione in termini di una matrice piuttosto che di un vettore.
Per esempio, è possibile usare un vettore come una matrice effettuando semplici calcoli sugli indici: supponendo di disporre di un vettore *v* di 20 valori, questo potrebbe essere «visto» come se fosse una matrice 2 righe per 10 colonne, oppure 4 righe per 5 colonne (FIGURA 1).

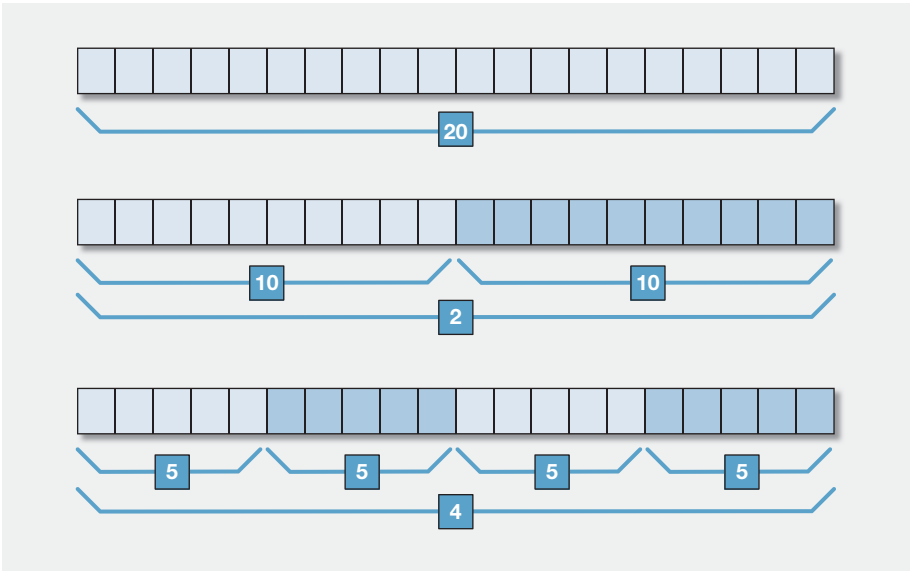


FIGURA 1

Nel caso di una matrice di 4 righe e 5 colonne avremmo che l'indice di riga *r* può assumere valori nell'intervallo 0-3, mentre quello di colonna *c* nell'intervallo 0-4. Per individuare all'interno di *v* il generico elemento di coordinate (*r*, *c*) sarà sufficiente utilizzare la notazione `v[r*4+c]`; infatti all'elemento di coordinate (0, 0) corrisponderà il primo elemento del vettore `v[0]`, mentre all'elemento di coordinate (4, 3) corrisponderà l'ultimo elemento del vettore `v[19]`.

2 *Array* come parametri di funzioni

Gli *array*, come le variabili semplici, possono essere utilizzati come parametri delle funzioni, ma il passaggio avviene sempre per indirizzo.

Nel linguaggio C/C++, usando un *array* come argomento di una funzione, in realtà viene passato l'indirizzo dell'elemento di indice 0 dell'*array* stesso; per evidenziare che il parametro formale è in realtà un vettore, di solito si usa la notazione *nome_parametro*[]. Inoltre, come già evidenziato, non esistendo alcun controllo sulla lunghezza dell'*array*, la dimensione di questo deve essere passata alla funzione come parametro, in modo da permettere

al codice della funzione stessa di rispettare il limite e non dover rischiare di fuoriuscire per andare a scrivere o leggere elementi in altre locazioni di memoria con effetti imprevedibili.

OSSERVAZIONE Volendo operare su di un *array* passato come parametro senza che le modifiche si riflettano all'esterno della funzione, l'unica possibilità è quella di creare un *array* locale delle stesse dimensioni di quello passato come parametro e copiarvi i valori di quest'ultimo.

Nella definizione di una funzione che ha come parametro un *array* monodimensionale, questo deve essere indicato con il nome seguito dalle parentesi quadre (aperta e chiusa) senza indicazione della dimensione; per esempio:

```
float media(int vettore[], int n);
```

L'invocazione della funzione, invece, prevede il passaggio dell'*array* utilizzando esclusivamente il nome:

```
m = media(v);
```

ESEMPIO

La funzione *cercaMinimo* restituisce il valore più piccolo tra i primi *n* elementi di un vettore di interi *v*:

```
int cercaMinimo(int v[], int n)
{
  int i, min;

  min = v[0];
  for (i=1; i<n; i++)
    if (v[i] < min)
      min = v[i];
  return min;
}
```

Una possibile chiamata di questa funzione potrebbe essere la seguente:

```
minimo = cercaMinimo(vettore,
      numero_elementi);
```

dove i due parametri attuali sono rispettivamente un vettore di interi di cui si vuole cercare il minimo e il numero degli elementi del vettore (la ricerca avverrà a partire dall'indice 0 fino all'indice immediatamente inferiore a questo numero: se la dimensione con cui è stato dichiarato il vettore è inferiore, si fuoriuscirà dai limiti dell'*array*).

OSSERVAZIONE Nella definizione di una funzione è possibile indicare la dimensione di un vettore passato come parametro tra le parentesi quadre che lo caratterizzano come *array*, ma questo valore non è reso disponibile al codice della funzione.

Nel caso in cui si voglia passare come parametro una matrice, è obbligatorio specificare almeno il numero delle colonne (cioè la dimensione della singola riga).

OSSERVAZIONE Il motivo per cui è necessario fornire la dimensione della riga (corrispondente al numero di colonne della matrice), negli *array* bidimensionali passati come argomento a una funzione, è per consentire al codice di determinare la corretta posizione di un elemento a partire dalle coordinate. Infatti una matrice è internamente memorizzata in modo sequenziale per righe – di fatto come un vettore – e il codice deve essere in grado di trasformare le coordinate bidimensionali (r, c) in un indice monodimensionale i. Nell'esempio di FIGURA **2** si ha una matrice di 4 righe ciascuna di 5 elementi (cioè di 4 righe per 5 colonne).

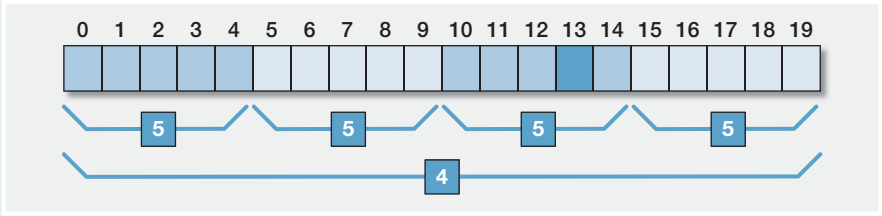


FIGURA **2**

Dovendo determinare l'indice dell'elemento di coordinate (2, 3), che si trova nella terza riga, è sufficiente calcolare il numero di elementi che costituiscono le prime 2 righe ($2 \times 5 = 10$) e sommarvi l'indice di colonna ($10 + 3 = 13$). Per svolgere questo calcolo, oltre agli indici di riga e di colonna, è necessario conoscere quanti elementi ha ogni riga, cioè quante colonne ha la matrice.

ESEMPIO

La funzione *sommaRighe* restituisce in un vettore passato come parametro la somma degli elementi di ciascuna riga di una matrice 10 × 10 (ma potrebbe avere un qualsiasi numero di righe, fermo restando che il numero delle colonne deve essere 10) fornita come argomento insieme alle sue dimensioni:

```
void sommaRighe(float m[][10], int R, float v[])
{
  float s=0;
  int r,c;
  for (r=0; r<R; r++)
    {
      s = 0.0;
      for (c=0; c<10; c++)
        s = s + m[r][c];
        v[r] = s;
      }
  return;
}
```

Una possibile chiamata di questa funzione potrebbe essere la seguente:

```
sommaRighe(valori, 10, somme);
```

dove *valori* è una matrice 10 × 10 e *somme* un vettore di 10 elementi dove la funzione calcolerà le somme delle singole righe della matrice. Il fatto che in C/C++ gli *array* sono tacitamente passati per indirizzo consente di usare un parametro-*array* come risultato senza notazioni sintattiche particolari.

Rivisitiamo ora la soluzione al problema iniziale del capitolo utilizzando una funzione per calcolare la media aritmetica dei valori contenuti in un vettore:

```
int mediaVettore(int v[], int n)
{
  int i, s=0;

  for (i=0; i<n; i++)
    s=s+v[i];
  return s/n;
}

void main(void)
{
  int mese, media, precipitazioni[12];

  // acquisizione valori precipitazioni mensili
  for (mese=0; mese<12; mese++)
    {
      cout<<"Inserire mm di pioggia del mese "<<(mese+1);
      cin>>precipitazioni[mese];
    }

  // calcolo media precipitazioni
  media = mediaVettore(precipitazioni, 12);

  // visualizzazione mesi piovosi
  for (mese=0; mese<12; mese++)
    if (precipitazioni[mese] > media)
      cout<<"Mese: "<<(mese+1)<<" pioggia:"
          <<precipitazioni[mese]<<"mm"<<endl;
}
```

L'esempio successivo è un classico «gioco» ideato dal matematico inglese John Conway verso la fine degli anni Sessanta per simulare l'evoluzione di una popolazione di semplici esseri viventi che vivono su un certo territorio. Questo è schematizzato da una griglia di caselle quadrate ognuna delle quali può ospitare un solo essere vivente; come è evidente (FIGURA **3**), ogni cella ha un intorno costituito da 8 celle adiacenti (in effetti quella che andiamo a presentare è una versione semplificata, in quanto il territorio dovrebbe avere una struttura «toroidale», dove la prima riga e la prima colonna confinano rispettivamente con l'ultima riga e l'ultima colonna).
L'individuo che occupa una cella può trovarsi in due stati, *vivo* o *morto*, e lo stato dell'intera popolazione evolve per generazioni successive calcolate a partire dagli stati degli individui che occupano le singole celle nella generazione precedente (tutte le celle del territorio vengono quindi aggiornate simultaneamente nel passaggio da una generazione alla seguente). I cam-

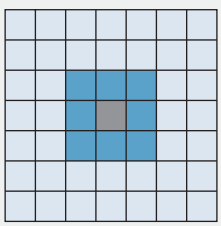


FIGURA **3**

biamenti di stato delle celle dipendono solo dal numero di celle adiacenti contenenti esseri vivi secondo le seguenti semplici regole:

- una cella vuota che ha esattamente 3 vicini vivi viene occupata da un nuovo essere vivente che nasce;
- una cella occupata da un essere vivente con esattamente 2 o 3 vicini vivi sopravvive;
- una cella occupata da un essere vivente con meno di 2 o più di 3 vicini vivi resta vuota perché l'essere muore per isolamento o sovraffollamento.

ESEMPIO

Le FIGURE **4**-**9** che seguono visualizzano l'evoluzione nel tempo di alcune configurazioni spaziali di esseri viventi.

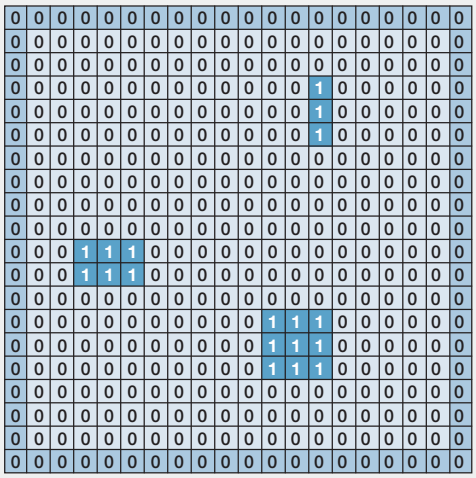


FIGURA **4**

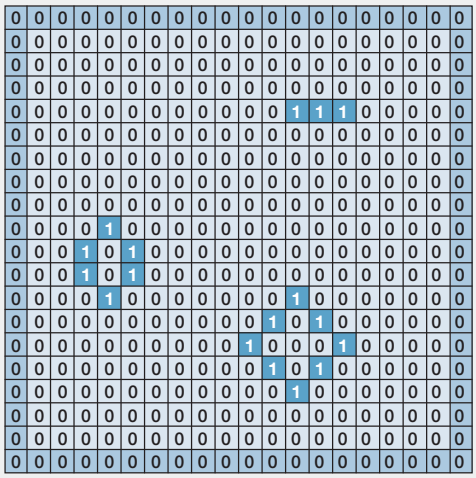


FIGURA **5**

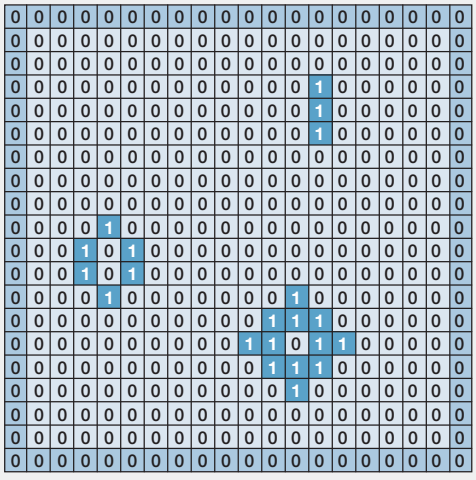


FIGURA **6**

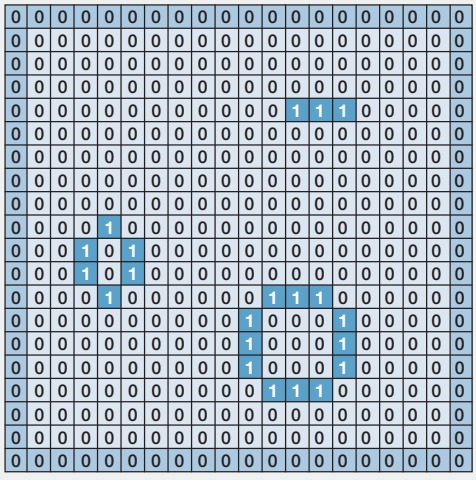


FIGURA **7**

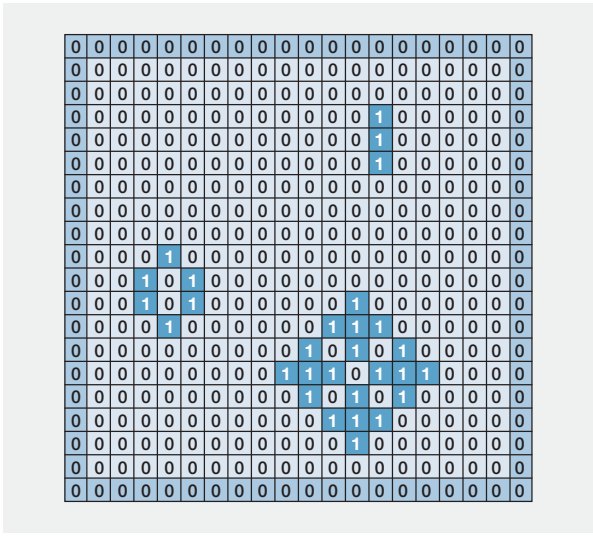


FIGURA **8**

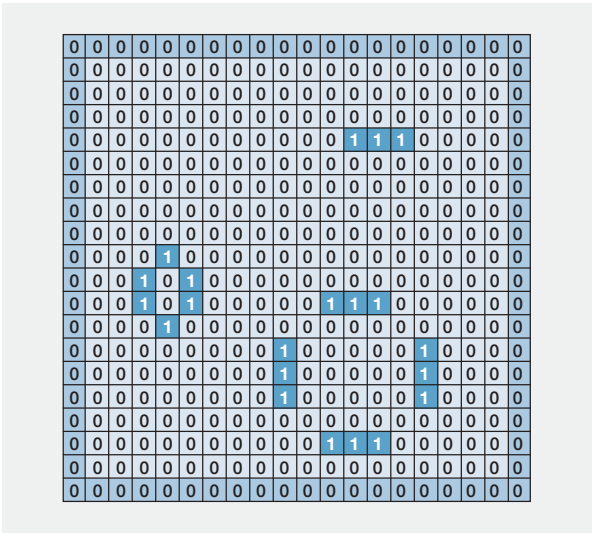


FIGURA **9**

Presentiamo di seguito il codice che simula il gioco Life operando su una griglia di *N* righe per *N* colonne, ma evitando le righe e le colonne dei bordi.
Per la soluzione del problema sono state utilizzate due matrici di caratteri *Gen_C* e *Gen_S*, entrambe di *N* righe per *N* colonne: *Gen_C* contiene la generazione corrente, mentre *Gen_S* quella successiva. In maniera iterativa le celle di *Gen_S* vengono aggiornate sulla base dello stato presente memorizzato in *Gen_C*; una volta terminato l'aggiornamento la nuova generazione diviene la generazione corrente semplicemente copiando il contenuto della matrice *Gen_S* in *Gen_C*.

```
const char N = 20;
const char VUOTA = ' ';
const char VIVA = '*';

char Gen_C[N][N], Gen_S[N][N];

// Inizializzazione del territorio del gioco
void inizializza(void)
  {
    int r,c;

    for (r=0; r<N; r++)
      for (c=0; c<N; c++)
        {
          Gen_C[r][c] = VUOTA;
          Gen_S[r][c] = VUOTA;
        }
    // configurazione iniziale predefinita
    Gen_C[13][3] = VIVA;
    Gen_C[13][4] = VIVA;
```

▶

```
    Gen_C[13][5] = VIVA;
    Gen_C[3][10] = VIVA;
    Gen_C[3][11] = VIVA;
    Gen_C[4][10] = VIVA;
    Gen_C[4][11] = VIVA;
    Gen_C[13][11] = VIVA;
    Gen_C[13][12] = VIVA;
    Gen_C[13][13] = VIVA;
    Gen_C[14][11] = VIVA;
    Gen_C[14][12] = VIVA;
    Gen_C[14][13] = VIVA;
    Gen_C[15][11] = VIVA;
    Gen_C[15][12] = VIVA;
    Gen_C[15][13] = VIVA;
  }

// Visualizzazione
void visualizza(char Gen[N][N])
  {
    int r,c;

    for (r=0; r<N; r++)
      {
        for (c=0; c<N; c++)
          cout<<Gen[r][c];
        cout<<endl;
      }
  }

// Copia
void copia(char Gen_1[N][N], char Gen_2[N][N])
  {
    int r,c;

    for (r=0; r<N; r++)
      for (c=0; c<N; c++)
        Gen_1[r][c] = Gen_2[r][c];
  }

// Calcola il nuovo stato di una cella
char cella(char Gen[N][N], int r, int c)
  {
    int i,j,c=0;

  /* l'intorno della cella (r,c) è costituito dalle celle
     con indici da (r-1,c-1) a (r+1,c+1) per un totale di 9
     elementi compreso quello di indici (r,c) */
  for (i=r-1; i<=r+1; i++)                                    ▶
```

```
    for (j=c-1; j<=c+1; j++)
      if (Gen[i][j] == VIVA)
        c++;

  // se la cella (r,c) e' viva si sottrae 1 dal conteggio
  if (Gen[r][c] == VIVA)
    c--;

  // applicazione delle regole di vita/morte
  if (c==3 && Gen[r][c] == VUOTA) // nascita
    return VIVA;

  if ((c==2 || c==3) && Gen[r][c] == VIVA) // sopravvivenza
    return VIVA;

  return VUOTA; // morte
}

// Evoluzione dalla generazione corrente alla successiva
void evolvi(char Gen_1[N][N], char Gen_2[N][N])
{
  int r,c;
  for (r=1; r<N-1; r++)
    for (c=1; c<N-1; c++)
      Gen_2[r][c] = cella(Gen_1, r, c);
}

void main(void)
{
  int i,j;
  char c;

  inizializza();

  // ciclo "infinito"
  while (1)
  {
    visualizza(Gen_C);
    // attesa input di un carattere (il carattere * termina)
    cin>>c;
    if (c == '*')
      break;
    // analisi generazione corrente per creare la successiva
    evolvi(Gen_C, Gen_S);
    // la nuova generazione diviene quella corrente
    copia(Gen_C, Gen_S);
  }
}
```

3 Stringhe di caratteri in stile C

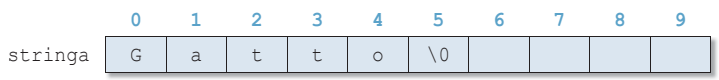
Il linguaggio C++ mantiene la gestione delle sequenze di caratteri – definite **stringhe** – nella tradizionale modalità del linguaggio C; in questo linguaggio, infatti, non esiste un vero e proprio tipo stringa (il tipo che in altri linguaggi di programmazione, compreso il C++ stesso, viene denominato *String*) e le stringhe vengono rappresentate mediante *array* unidimensionali di caratteri.

OSSERVAZIONE Utilizzando le stringhe in stile C, è necessario porre attenzione alle operazioni che si compiono e ricordare che si hanno a disposizione tutte le caratteristiche degli *array*, compresa quella di poter scorrere le posizioni dei singoli caratteri della stringa.

Ogni *array* di caratteri che contiene una stringa termina con un valore 0 (cioè la sequenza di *escape* «\0»): un vettore di N caratteri può dunque ospitare stringhe lunghe al più $N - 1$ caratteri, perché una cella è riservata al carattere terminatore «\0».
Per dichiarare una stringa lunga al massimo 9 caratteri (escluso il terminatore) è possibile utilizzare una dichiarazione del tipo:

```
char stringa[10];
```

Una occorrenza di stringa inserita in tale vettore potrebbe essere la seguente:



OSSERVAZIONE Si noti come le celle di indice superiore a 5 siano inutilizzate e possano contenere un valore casuale ininfluente al fine del contenuto associato alla stringa.

Una stringa può essere inizializzata come un qualsiasi *array* specificandone i singoli caratteri:

```
char stringa[10]={'g','a','t','t','o','\0'};
```

oppure mediante la forma specifica più compatta:

```
char stringa[10]="gatto";
```

In quest'ultimo caso il carattere di terminazione viene aggiunto automaticamente; in entrambi i casi è necessario prestare attenzione alla lunghezza della stringa inserita, perché se questa dovesse eccedere la dimensione dell'*array* il compilatore segnalerebbe un errore.

Il carattere terminatore caratterizza la modalità di elaborazione delle stringhe ed è particolarmente utile nella determinazione della lunghezza di una stringa contenuta in un *array*. È infatti semplice scrivere una semplice funzione che restituisce la lunghezza di una stringa fornita come parametro:

```
int lunghezzaStringa(char string[])
{
  int i=0;

  while (string[i] != '\0')
    i++;

  return i;
}
```

È ovviamente possibile utilizzare in alternativa il costrutto *for*:

```
int lunghezzaStringa(char string[])
{
  for (int i=0; string[i]!='\0'; i++);

  return i;
}
```

Una stringa può essere visualizzata come ogni altro *array* specificandone le singole componenti:

```
...
int i=0;
char s[101];
...
while (s[i] != '\0')
  {
    cout<<s[i];
    i++;
  }
```

oppure nella forma specifica più compatta:

```
...
int i=0;
char s[101];
...
cout<<s;
```

che può essere utilizzata anche per l'input dei suoi elementi da tastiera:

```
...
int i=0;
char s[101];
...
cin>>s;
```

OSSERVAZIONE Si noti come queste notazioni rappresentano eccezioni, perché in generale gli *array* non si possono caricare o visualizzare interamente, ma solo elemento per elemento.

In ogni caso non è possibile utilizzare questa impostazione per «copiare» una stringa, ma deve essere effettuato un ciclo come quello della seguente funzione:

```
void copiaStringa(char s1[], char s2[])
{
  for (int i=0; s1[i]!='\0'; i++)
    s2[i]= s1[i];
}
```

OSSERVAZIONE Si ponga attenzione al fatto che in fase di input la stringa inserita non contenga caratteri quali spazio bianco, tabulazione o ritorno a capo, perché questi vengono interpretati come fine stringa, causando la perdita degli eventuali caratteri che seguono.
L'acquisizione di una stringa che potenzialmente contenga caratteri di questo tipo deve essere effettuata utilizzando il metodo `cin.getline(string,n)`, dove *string* è un vettore di caratteri e *n* un valore intero che individua il numero massimo di caratteri di cui può essere composta la stringa (tipicamente la dimensione del vettore).

Per le stringhe vale il criterio di ordinamento lessicografico (regola di ordinamento corrispondente a quella utilizzata nei dizionari, anche se estesa a un qualunque insieme di simboli) definito per le variabili di tipo carattere: due stringhe devono essere sempre confrontate, carattere per carattere, a partire dal primo elemento e, in questo senso, la lunghezza della stringa non inflenza il criterio di precedenza nell'ordinamento.

ESEMPIO

	0	1	2	3	4	5	6	7	8	9
s1	g	a	t	t	o	\0				

	0	1	2	3	4	5	6	7	8	9
s2	g	a	t	t	i	n	o	\0		

s1 risulta essere «maggiore» di *s2*, in quanto i primi quattro caratteri sono identici, mentre il quinto carattere risulta essere successivo nell'ordinamento alfabetico per *s1*.

La seguente funzione di utilità restituisce *true*, se le due stringhe fornite come parametri sono identiche, *false* altrimenti:

```
bool confrontaStringhe(char s1[], char s2[])
{
  int i=0;

  // se le lunghezze sono diverse
  // allora le stringhe non sono uguali
  if (lunghezzaStringa(s1) != lunghezzaStringa(s2))
    return false;

  while (s1[i]!='\0' && s2[i]!='\0')
      {
        // se due caratteri corrispondenti sono diversi
        // allora le stringhe non sono uguali
        if (s1[i] != s1[i])
          return false;
      }
  return true;
}
```

Il fatto che in C++ gli *array* utilizzati come parametri delle funzioni siano passati per indirizzo permette di realizzare semplicemente funzioni che modificano le stringhe.

ESEMPIO

La seguente funzione trasforma in maiuscoli gli eventuali caratteri minuscoli di una stringa sfruttando il caratteristico ordinamento della tabella dei codici dei caratteri:

```
void stringaMaiuscola(char s[])
{
  for (int i=0; s[i]!='\0'; i++)
    if (s[i] >= 'a' && s[i] <= 'z')
      s[i] = s[i] - ('a' - 'A');
}
```

Sintesi

Array. Insieme omogeneo di valori identificato da un nome e dove ogni singolo elemento è identificato dal nome dell'*array* associato a una lista di indici (per esempio, $v[i]$, $m[i][j]$). Un *array* è come una collezione di variabili semplici tutte dello stesso tipo. Gli *array* monodimensionali sono denominati **vettori**, quelli bidimensionali **matrici**. Gli *array* sono strutture dati con dimensione fissa (una volta dichiarata non può essere modificata).

Indice. Riferito a un vettore, un indice è un valore numerico (espresso da una costante, una variabile o una espressione) che individua un elemento al suo interno. Riferito a una matrice, un indice è ancora un valore numerico che individua, in base alla posizione che occupa, una riga o una colonna (per esempio, in $m[i][j]$ i individua una generica riga, mentre j indica una generica colonna). Una variabile indice non è legata in alcun modo a uno specifico *array*: $v[i]$ e $w[i]$ identificano elementi di identica posizione in vettori distinti, mentre $v[i]$ e $v[j]$ identificano posizioni diverse dello stesso vettore (per $i \neq j$). In C/C++ il controllo di congruenza tra valore dell'indice e dimensione dell'*array* è completamente demandato al programmatore.

***Array* e costrutti iterativi**. I costrutti iterativi, e in particolare i cicli *for*, permettono di sfruttare al massimo la potenza espressiva dell'uso degli *array* in un programma; con poche istruzioni è possibile implementare un codice chiaro e sintetico per l'elaborazione di insiemi di dati difficilmente gestibili con l'utilizzo delle sole variabili semplici.

***Array* e funzioni**. In C/C++ il passaggio di un *array* come parametro di una funzione avviene per indirizzo. Un *array* viene definito come parametro formale in una funzione specificandone tipo e nome, seguito da parentesi quadre se si tratta di un vettore (per esempio, *void f*1(*int v*[], ...)), oppure specificando anche il numero delle colonne, se si tratta di matrice (per esempio, *void f*2(*int a*[][10], ...)). In entrambi i casi è buona norma passare come parametri separati anche le singole dimensioni dell'*array*. Per quanto riguarda le chiamate a funzioni con *array*, i parametri attuali relativi a questi ultimi vengono specificati indicandone solo il nome (per esempio, $n = f1(v, ...)$).

***Array* di caratteri**. Con gli *array* di caratteri si possono rappresentare stringhe alfanumeriche nello stile del linguaggio C. Ogni stringa è costituita da una sequenza di caratteri che termina con «\0», ovvero una stringa può essere lunga al massimo $N - 1$ caratteri, dove N è la dimensione dell'*array* che la contiene. In C++ è possibile utilizzare *cin* e *cout* per gestire l'input/output in un'unica soluzione di un *array* di caratteri.

QUESITI

1 **Quale delle seguenti affermazioni circa la validità del valore dell'indice di un vettore è corretta?**

A Viene controllata in fase di esecuzione.

B Viene controllata al momento della compilazione.

C Viene demandata al programmatore.

D Il valore dell'indice di un vettore è sempre valido.

2 **Quale delle seguenti affermazioni è corretta?**

A Si possono definire solo *array* di tipo *int*.

B Anche gli *array* bidimensionali sono memorizzati sequenzialmente.

C Ogni *array* è terminato da un carattere nullo («\0»).

D L'*n*-esimo elemento dell'*array* ha indice *n*.

3 **È dato il seguente frammento di programma**

```
int n = 0;
...
for (int i=0; i<MAX; i++)
  if (v[i]>=0)
    n++;
```

Affinché durante l'esecuzione siano esaminati tutti gli elementi del vettore è necessario che ...

A ... *v* contenga solo valori positivi.

B ... MAX sia il massimo valore contenuto in *v*.

C ... MAX sia inferiore o uguale al numero degli elementi di *v*.

D Nessuna delle risposte precedenti.

4 **Date le seguenti dichiarazioni:**

```
const int R = 3;
const int C = 4;
int M[R][C];
```

l'istruzione:

```
M[1][C-5] = 5;
```

è errata perché ...

A ... non si possono usare due indici.

B ... il secondo indice assume un valore inesistente.

C ... non è detto che M[1][C – 5] sia uguale a 5.

D Nessuna delle precedenti.

5 **È data la seguente matrice *m*:**

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Quali delle seguenti affermazioni sono vere?

A L'elemento *m*[2][1] ha valore 10.

B L'elemento *m*[*m*[0][0] + 1][*m*[0][1]] ha valore 11.

C L'insieme dei valori degli elementi della diagonale principale è {1, 6, 11, 16}.

D La somma dei valori degli elementi delle colonne di indice pari è maggiore della somma dei valori degli elementi delle righe di indice dispari.

6 **Data una stringa inserita in un vettore di caratteri con dimensione 10, indicare quali delle seguenti affermazioni sono false.**

A Si possono inserire al massimo stringhe lunghe 9 caratteri.

B Si possono inserire al massimo stringhe lunghe 10 caratteri.

C Inserendo una stringa più lunga di 9 caratteri si causa una fuoriuscita dal vettore.

D Inserendo una stringa più lunga di 9 caratteri si verifica un troncamento della stessa.

7 **Siano date due stringhe inserite rispettivamente nei vettori di caratteri *s1* e *s2*. Quali delle seguenti affermazioni sono vere circa il confronto tra *s1* ed *s2* per definirne l'ordinamento lessicografico?**

A Affinché si possa eseguire il confronto è necessario che i due vettori abbiano la stessa dimensione.

B La stringa inserita nel vettore di dimensione minore precederà quella nel vettore di maggiore dimensione.

C La dimensione dei due vettori è ininfluente, perché il confronto viene effettuato tra caratteri di uguale posizione nei due vettori fino

all'indice in cui l'elemento di un vettore è diverso dall'elemento corrispondente nell'altro vettore.

D La dimensione dei due vettori è determinante solo nel caso in cui le due stringhe abbiano uguali i primi $N - 1$, con N dimensione del vettore di dimensione minore.

ESERCIZI

Vettori

1 È dato il seguente vettore denominato *alfa*:

2	1	5	3	6	5	1	3	4	6	2

Indicare come esso si modifica in seguito a ognuno dei tre gruppi di istruzioni a partire dalla situazione iniziale:

a)
```
int i = alfa[3];
int j = 3;
alfa[i-j] = alfa[i]+j;
```

b)
```
int i = alfa[5];
int j = alfa[i-1]+1;
alfa[j] = alfa[i]+alfa[j];
```

c)
```
for (int i=1; i<=9; i++)
   {
     int s=0;
     for (j=0; j<i; j++)
       s += alfa[j];
     alfa[i] = s;
   }
```

2 Scrivere un programma C++ che acquisisca da tastiera un vettore di interi di dimensione *N* e calcoli minimo, massimo e media degli elementi.

3 Scrivere una funzione C/C++ che restituisca *true*, se un valore numerico intero positivo fornito come argomento è presente almeno una volta tra gli elementi di un vettore fornito anch'esso come argomento, *false* altrimenti.
Scrivere un programma C++ che, utilizzando la funzione precedente e costruendo il vettore a partire da valori forniti da tastiera, visualizzi, in risposta a singoli valori successivamente richiesti all'utente, «Vero» in caso di presenza nel vettore o «Falso» altrimenti (la richiesta terminerà in corrispondenza dell'inserimento del valore 0).

4 Scrivere una funzione C++ che inverta la posizione degli elementi di un vettore di interi di cui sia fornita la dimensione (risolvere il problema senza usare vettori ulteriori). Scrivere un programma C++ che, dopo avere acquisito da tastiera un vettore di valori interi, lo visualizzi trasformato dall'invocazione della precedente funzione.

5 Uno strumento di misura fornisce un dato ogni minuto nell'arco di un'ora. Per ovviare a possibili errori si vogliono elaborare i valori rilevati sostituendoli con una media a tre punti: ogni elemento viene sostituito dalla media di sé stesso, dell'elemento che lo precede e quello che lo segue; per i due elementi estremi viene considerato due volte il valore dell'elemento stesso e il successivo o il precedente nel caso si tratti rispettivamente del primo o dell'ultimo elemento. Realizzare un programma C++ che implementi l'elaborazione descritta acquisendo i dati da tastiera.

6 Dato un vettore di valori numerici, scrivere una funzione C++ che conti quanti elementi del vettore sono compresi tra un valore minimo e un valore massimo forniti come argomenti alla funzione stessa insieme alla dimensione del vettore. Scrivere un programma C++ che, utilizzando la funzione precedente e costruendo il vettore a partire da valori acquisiti da tastiera, visualizzi il risultato del conteggio dopo avere richiesto all'utente i valori minimo e massimo.

7 Scrivere una funzione C/C++ che, a partire da un vettore numerico di dimensione *N* fornito come parametro, copi in un secondo vettore (anch'esso fornito come parametro) i soli elementi del primo vettore compresi tra un valore minimo *a* e un valore massimo *b* (anch'essi argomenti della funzione); la funzione deve restituire la dimensione del secondo vettore. Scrivere un programma C++ che, utilizzando la funzione precedente, visualizzi gli elementi del secondo vettore dopo avere richiesto l'inserimento da tastiera dei valori del primo vettore.

8 Scrivere una funzione C++ avente il seguente prototipo:

```
void derive(float data[], int n,
   float difference[]);
```

che, a partire dal vettore *data* di dimensione *n*, costruisca il vettore *difference* contenente le differenze tra due elementi adiacenti del vettore *data*, cioè il primo elemento di *difference* dovrà essere la differenza tra il secondo e il primo elemento di *data*, il secondo elemento di *difference* dovrà essere la differenza tra il terzo e il secondo elemento di *data* e così via. Scrivere un programma C++ che, utilizzando la funzione precedente e costruendo il vettore a partire da valori acquisiti da tastiera, visualizzi il vettore risultato del calcolo.

9 Scrivere una funzione C++ avente il seguente prototipo:

```
float scalarProduct(float a[], float b[],
   int N);
```

dove *N* rappresenta la dimensione dei vettori *a* e *b*, che restituisca il prodotto scalare tra *a* e *b*, cioè la somma dei prodotti tra gli elementi corrispondenti (il primo con il primo, il secondo con il secondo e così via) dei due vettori.

10 Scrivere una funzione C++ che, a partire da due vettori numerici ordinati forniti come argomenti, ne costruisca un terzo avente dimensione pari alla somma delle due dimensioni e contenente tutti gli elementi dei due vettori in una sequenza ordinata (per esempio, da {1, 2, 4} e {1, 3, 9} si deve ottenere {1, 1, 2, 3, 4, 9}). Scrivere un programma C++ che, utilizzando la funzione precedente e costruendo i vettori iniziali a partire da valori acquisiti da tastiera, visualizzi il vettore risultato.

11 Scrivere una funzione C++ che restituisca la differenza tra il valore massimo e il valore minimo di un vettore di *N* elementi. Scrivere un programma C++ che calcoli e visualizzi il risultato dopo avere richiesto all'utente l'inserimento degli elementi.

12 I codici a barre dei prodotti sono composti da 13 cifre di cui l'ultima è la cifra di controllo che si determina a partire dalle prime 12 (il codice vero e proprio) con le seguenti regole:

- moltiplicare per 3 tutte le cifre in posizione «dispari» (la prima, la terza, … fino all'undicesima);
- moltiplicare per 1 tutte le cifre in posizione «pari» (la seconda, la quarta, … fino alla dodicesima) (*);
- sommare i 12 valori così ottenuti;
- prendere il resto della divisione per 10 della somma ottenuta.

Scrivere una funzione C/C++ che, a partire da un vettore di 12 numeri corrispondenti alle singole cifre di un codice a barre, restituisca la cifra di controllo calcolata con le regole illustrate. Scrivere un programma C++ che richieda all'utente l'inserimento delle singole cifre di un codice a barre e visualizzi la corrispondente cifra di controllo calcolata con la funzione realizzata in precedenza.

(*) Ovviamente questa operazione è inutile!

Matrici

13 Indicare lo scopo delle seguenti funzioni C/C++:

a)
```
const int N=…
...
void f(int t[N][N])
{
  int i,j;

  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      t[i][j]=(i+1)*(j+1);
  return;
}
```

b)
```
const int N=…
...
float mistery(float M[N][N])
{
  float s = 0.;
  int n = 0;

  for (int x=0; x<N; x++)
    for (int y=0; y<N; y++)
      if (M[x][y] == 0.)
        n++;
      else
        s += M[x][y];

  return (s/(N*N-n));
}
```

14 Scrivere una funzione C/C++ che restituisca il valore massimo di una matrice di dimensioni *N* × *N* fornita come argomento; la funzione deve fornire anche le coordinate (indice di riga e indice di colonna) di tale valore. Scrivere un programma C++

che, utilizzando la funzione precedente, visualizzi il valore massimo e le relative coordinate di una matrice i cui elementi sono inseriti da tastiera.

15 Scrivere una funzione C/C++ che determini, producendo come risultato *true* o *false*, se una matrice quadrata di $N \times N$ elementi è simmetrica, cioè se gli elementi in posizione simmetrica rispetto alla diagonale principale sono tutti uguali.

16 Scrivere una funzione C/C++ che costruisca, in una matrice quadrata di $N \times N$ elementi passata come argomento, la trasposta di una matrice quadrata di $N \times N$ elementi fornita anch'essa come argomento. La trasposta di una matrice è una matrice in cui le posizioni degli elementi sono modificate scambiando gli indici di riga e di colonna, per esempio la trasposta della matrice

$$\begin{bmatrix} 2 & 1 & 5 & 9 \\ 0 & 1 & 8 & 4 \\ 2 & 1 & 9 & 3 \\ 6 & 7 & 0 & 2 \end{bmatrix}$$

è la matrice

$$\begin{bmatrix} 2 & 0 & 2 & 6 \\ 1 & 1 & 1 & 7 \\ 5 & 8 & 9 & 0 \\ 9 & 4 & 3 & 2 \end{bmatrix}$$

Scrivere un programma C++ che, utilizzando la funzione precedente e costruendo la matrice iniziale a partire da valori forniti da tastiera, visualizzi la matrice trasposta disponendo gli elementi su più righe.

17 Un'area vulcanica viene monitorata dalla Protezione civile mediante alcuni sensori di temperatura numerati (0, 1, 2, …), che periodicamente forniscono una misura della temperatura del terreno in cui sono posizionati.
Si intende realizzare un programma C++ che consenta di inserire e memorizzare i valori di temperatura dei singoli sensori ogni volta che sono disponibili calcolandone i valori medio, minimo e massimo; lo stesso programma deve inoltre permettere di visualizzare la sequenza temporale delle misure di temperatura di un singolo sensore specificato dall'operatore e di calcolarne media, valore minimo e valore massimo.

18 Scrivere una funzione C/C++ che restituisca *true*, se una matrice quadrata di ordine N fornita come argomento è un quadrato magico (cioè se contiene tutti i numeri da 1 a N^2 e la somma di tutte le righe e di tutte le colonne è costante), *false* altrimenti.

19 Una matrice rappresenta le quote di un'area rettangolare di territorio. Scrivere tre funzioni C/C++ per la determinazione:

- della quota minima;
- della quota massima;
- della quota media.

Scrivere un programma C++ che – dopo avere richiesto all'utente l'inserimento delle quote nella matrice – calcoli e visualizzi la quota minima, la quota media e la quota massima del territorio.

20 Estendere il programma dell'esercizio precedente in modo che calcoli e visualizzi anche la percentuale della superficie di territorio che ha una quota superiore alla media.

21 Il prodotto tra due matrici A e B, quadrate e aventi la stessa dimensione, è una matrice quadrata C della stessa dimensione: l'elemento di coordinate (i, j) della matrice C è la somma dei prodotti degli elementi corrispondenti della i-esima riga di A e della j-esima colonna di B:

$$C(i, j) = A(i, 0) * B(0, j) + A(i, 1) * B(1, j) + \ldots$$

Scrivere una funzione C++ che determini il prodotto tra due matrici quadrate.

22 Il grafo della rete della metropolitana di Parigi, costituita da 298 stazioni, è rappresentato mediante una matrice quadrata di 298 × 298 elementi contenenti la distanza in kilometri tra le stazioni (per esempio l'elemento di coordinate 99 e 101 della matrice contiene la distanza tra la stazione numero 99 e la stazione numero 101, 0 se le stazioni non sono direttamente connesse).

a) Scrivere una funzione C/C++ che, a partire dalla matrice di rappresentazione del grafo, calcoli la lunghezza dell'intera rete della metropolitana. Scrivere un programma C++ che, dopo avere richiesto all'utente l'inserimento della matrice di rappresentazione del grafo (si consideri che solo pochi elementi della matrice sono diversi da 0), utilizzi la funzione del punto precedente per visualizzare la lunghezza complessiva della rete.

b) Scrivere un programma C++ che, disponendo della matrice di rappresentazione del grafo come variabile globale e a partire dal codice numerico di una stazione, elenchi le linee che raggiungono la stazione specificata. Scrivere un programma C++ che, disponendo della matrice di rappresentazione del grafo come variabile globale e a partire dal numero di una linea, elenchi i codici numerici delle stazioni che la costituiscono.

23 Scrivere una funzione C/C++ che, a partire da una matrice numerica di $R \times C$ elementi, costruisca due vettori di R elementi ciascuno contenenti rispettivamente i valori minimi e massimi di ogni riga della matrice. Scrivere un programma C++ che – dopo avere richiesto all'utente l'inserimento dei singoli elementi della matrice – visualizzi gli elementi dei due vettori ottenuti come risultato dell'applicazione della funzione precedente.

24 Una matrice si definisce «ordinata per righe» se gli elementi di ogni riga sono disposti in ordine crescente da sinistra verso destra e tutti gli elementi di ogni riga, esclusa la prima, sono maggiori di ciascuno degli elementi della riga precedente. Scrivere una funzione C/C++ che restituisca *true*, se una matrice fornita come argomento è ordinata per righe, *false* altrimenti.

25 Scrivere una funzione C/C++ che, a partire da due vettori di valori numerici aventi la stessa dimensione, costruisca una matrice quadrata in cui ogni singolo elemento di coordinate (r, c) sia ottenuto moltiplicando l'r-esimo elemento del primo vettore per il c-esimo elemento del secondo vettore. Per esempio, dati i vettori

$$[1 \quad 1] \qquad [0{,}5 \quad 2]$$

la matrice risultato deve essere

$$\begin{bmatrix} 1 \cdot 0{,}5 & 1 \cdot 2 \\ 1 \cdot 0{,}5 & 1 \cdot 2 \end{bmatrix} = \begin{bmatrix} 0{,}5 & 2 \\ 0{,}5 & 2 \end{bmatrix}$$

Scrivere un programma C++ che – dopo avere richiesto all'utente l'inserimento dei singoli elementi dei due vettori – visualizzi gli elementi della matrice ottenuta come risultato dell'applicazione della funzione precedente.

26 Scrivere una funzione C/C++ che, a partire da una matrice numerica di $R \times C$ elementi, costruisca due vettori rispettivamente di dimensione R e C: il primo vettore dovrà contenere le somme degli elementi di ciascuna riga della matrice, il secondo vettore le somme degli elementi di ciascuna colonna. Scrivere un programma C++ che – dopo avere richiesto all'utente l'inserimento dei singoli elementi della matrice – visualizzi gli elementi dei vettori ottenuti come risultato dell'applicazione della funzione precedente.

27 Una matrice si definisce «a predominanza diagonale» se per ogni riga il valore dell'elemento avente lo stesso indice di riga e di colonna è il maggiore della riga. Scrivere una funzione C++ che restituisca *true*, se una matrice fornita come argomento è a predominanza diagonale, *false* altrimenti.

28 Data una matrice m di $M \times N$ elementi interi, scrivere un programma che crei due vettori *media_colonne* di N elementi e *media_righe* di M elementi tali che:

- l'r-esimo elemento di *media_righe* contenga la media degli elementi della r-esima riga di m;
- il c-esimo elemento di *media_colonne* contenga la media degli elementi della c-esima colonna di m.

29 Si vuole implementare un programma C++ che simuli in parte il gioco della battaglia navale su un campo di 10 × 10 celle: le navi possono essere disposte in orizzontale o verticale; ognuna occupa un numero intero di celle e non devono essere in contatto tra loro. Sul campo sono schierate 4 navi da 1 cella, 3 navi da 2 celle, 2 navi da 3 celle e 1 nave da 4 celle. Sapendo che le celle occupate dalle navi contengono il carattere «*», mentre le altre contengono il carattere *blank*, scrivere una funzione C/C++ che, a partire dalla matrice di gioco dove sono state posizionate le navi descritte, individui la nave da 4 celle e ne visualizzi le coordinate degli estremi.

30 Data una matrice intera M di dimensioni qualsiasi, scrivere un programma C++ che calcoli e visualizzi la somma degli elementi che giacciono sulle cornici concentriche della matrice data (la cornice più esterna è formata dalla prima e ultima riga e dalla prima e ultima colonna, quella successiva dalla seconda e penultima riga e dalla seconda e penultima colonna, e così via).

Stringhe

31 Indicare lo scopo delle seguenti funzioni C/C++:

a)
```
int f(char s[])
{
  int i, n=0;

  for (i=0; s[i]!='\0'; i++)
    if ((s[i] >= '0') && (s[i] <= '9'))
      n++;
  return n;
}
```

b)
```
void mistery(char s[])
{
  int i=0;

  while (s[i] != '\0')
    {
      if (s[i] == ' ')
        s[i] = '_';
      i++;
    }
  return;
}
```

32 Scrivere un programma C++ che visualizzi il carattere che si ripete più volte in una stringa inserita dall'utente e il numero di ripetizioni.

33 Scrivere una funzione C/C++ che accettata come parametro una stringa *s*, ne calcoli il numero delle consonanti o delle vocali sulla base di un altro opportuno parametro che indichi il tipo di conteggio da effettuare (per esempio: «C» = consonanti, «V» = vocali).

34 Il CRC di una sequenza di caratteri è lo XOR (operatore C/C++ «^») dei codici di tutti i caratteri della sequenza: scrivere una funzione C/C++ che calcoli il CRC di una stringa di caratteri.

35 Data una stringa di caratteri, scrivere una funzione che verifichi se la stringa è o meno palindroma (cioè tale che, leggendo la stringa da sinistra verso destra o viceversa, si ottiene sempre la stessa sequenza di caratteri).

36 Una stringa di testo in formato CSV (*Comma Separated Values*) è formata da valori numerici in formato testuali separati dal carattere «;»; nella notazione anglosassone i valori numerici hanno le migliaia separate dal carattere «,» e i decimali che seguono il carattere «.», mentre nella notazione continentale i valori numerici hanno le migliaia separate dal carattere «.» e i decimali che seguono il carattere «,». Scrivere un programma C++ che trasformi una stringa CSV in formato anglosassone in una stringa CSV in formato continentale.

37 Date due stringhe *s*1 ed *s*2 di lunghezza qualsiasi, scrivere una funzione C/C++ che verifichi se *s*2 è una sottostringa di *s*1 (cioè se *s*2 è contenuta in *s*1) restituendo il valore intero –1 se non lo è, oppure un valore positivo che indichi la posizione di inizio della corrispondenza tra gli elementi di *s*2 e *s*1.

38 Scrivere una funzione C/C++ che, a partire da due stringhe fornite come argomenti, restituisca il conteggio di quanti caratteri della prima stringa sono presenti nella seconda (per esempio, il risultato restituito dalla funzione per «PIPPO» e «PIPPICALZELUNGHE» è 4: sono infatti presenti «P», «I», «P» e «P», ma non «O»).

39 Scrivere una funzione C/C++ avente il seguente prototipo

```
int stringNcopy(char src[], char dst[],
   int N);
```

che copi i primi *N* caratteri del vettore *src*[] nel vettore *dst*[]; il vettore di caratteri da copiare è terminato dal codice «\0» (stringa in stile C) e, se la lunghezza del suo contenuto è minore di *N*, devono essere copiati solo i caratteri che contiene; la funzione deve restituire il numero di caratteri effettivamente copiati.

40 Scrivere una funzione C++ avente il seguente prototipo

```
int stringConcat(char src1[], char src2,
   char dst[]);
```

che costruisca la stringa in stile C (terminata dal carattere «\0») *dst*[] concatenando le stringhe *src*1[] e *src*2[]; la funzione deve restituire il numero di caratteri contenuti nella stringa risultato *dst*[].

41 Nel *De bello gallico* Giulio Cesare racconta di avere utilizzato un codice segreto per comunicare in forma scritta con i propri generali: ogni singola lettera del messaggio veniva sostituita con la lettera che la seguiva nell'ordinamento alfabetico (A → B, B → C, …, Z → A), i simboli delle cifre numeriche e i segni di punteggiatura erano lasciati invariati. Scrivere due programmi C++: il primo deve codificare secondo la regola di Cesare una stringa di testo inserita dall'utente e visualizzare il risultato, il secondo deve accettare una stringa segreta e decodificarla visualizzando il testo originale.

42 Il ROT13 è una tecnica di codifica del testo basata sulla sostituzione di ogni singolo carattere alfabetico con il carattere che lo segue esattamente di 13 posizioni nella sequenza alfabetica: se si supera la posizione della «Z» si prosegue riprendendo dalla «A». La FIGURA **10** illustra la codifica della parola «HELLO», che diviene «URYYB».

Tenendo conto che lo stesso algoritmo si utilizza sia per la codifica sia per la decodifica, scrivere una funzione C/C++ che codifichi/decodifichi secondo la regola ROT13 una stringa di testo.

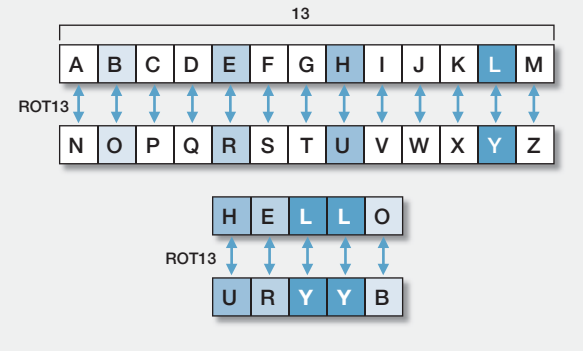


FIGURA **10**

43 Scrivere una funzione C/C++ che determini il numero di parole presenti in una stringa tenendo conto dei vari caratteri di separazione (spazi, punteggiatura ecc.).

44 Scrivere una funzione C/C++ avente il seguente prototipo:

```
bool subString(char s1[], int p1,
   int p2, char s2[]);
```

che copi i caratteri della stringa *s*1 compresi tra la posizione *p*1 e la posizione *p*2 nella stringa *s*2 (per esempio, se *s*1 è «Superpippicalzelunghe», *p*1 è 5 e *p*2 è 9, allora il risultato *s*2 deve essere «pippi»). La funzione deve restituire *true* se l'operazione viene effettuata, *false* se l'operazione non viene effettuata perché i valori *p*1 e *p*2 non sono corretti.

45 Scrivere una funzione C/C++ che abbia come parametri due stringhe e restituisca:

- –1 se la prima stringa precede in ordine alfabetico la seconda stringa;
- 0 se le due stringhe sono uguali;
- 1 se la seconda stringa precede in ordine alfabetico la prima stringa.

Scrivere un programma C++ che, dopo avere richiesto all'utente l'inserimento da tastiera di tre stringhe, le visualizzi sullo schermo in ordine alfabetico.

LABORATORIO

1 La mappa di un territorio viene schematizzata mediante una matrice di dimensioni $M \times N$ all'interno della quale ogni elemento contiene un numero intero che rappresenta in metri l'altezza del terreno in quella zona. In un elemento di cui sono note le coordinate di riga e di colonna è presente una sorgente d'acqua; sapendo che l'acqua può scendere in tutte le direzioni (verticale, orizzontale, diagonale) verso punti con quota minore o uguale ma, ovviamente, non può risalire verso punti di quota superiore, fornire la mappa delle zone che verranno allagate (per esempio rappresentando gli elementi allagati con un asterisco e quelli non allagati con un punto).

2 Realizzare una versione estesa del gioco Life dove il territorio abbia una struttura toroidale (la prima riga e la prima colonna confinano rispettivamente con l'ultima riga e l'ultima colonna) come mostrato in FIGURA **11**.

3 Un «quadrato magico» di ordine N è una tabella di $N \times N$ celle contenenti tutti i numeri naturali da 1 a N^2 in modo che la somma degli elementi di ciascuna riga e di ciascuna colonna produca sempre lo stesso numero.

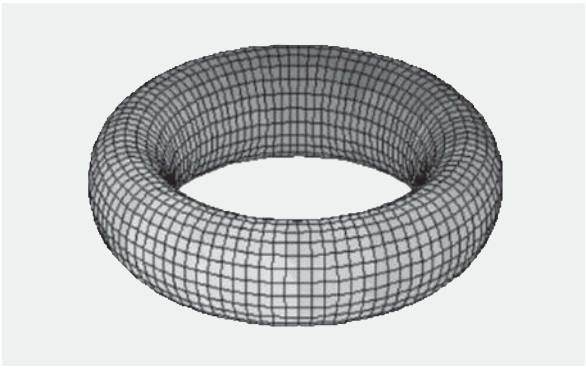


FIGURA **11**

Per esempio, in FIGURA **12** è rappresentato un quadrato magico di ordine 7 (quindi comprendente i numeri da 1 a 49) in cui la somma dei valori di ciascuna riga e di ciascuna colonna vale 175.
Per i quadrati magici di ordine dispari è nota dal XVII secolo la seguente tecnica costruttiva:

- posizionare il numero 1 al centro della prima riga;
- spostarsi di una posizione in alto e una posizione a sinistra per posizionare il numero successivo: se si fuoriesce dal quadrato continuare come se i due estremi di ciascuna riga e di ciascuna colonna fossero contigui;
- se si raggiunge una cella già occupata da un numero continuare dalla cella posizionata una riga più in basso.

28	19	10	1	48	39	30
29	27	18	9	7	47	38
37	35	26	17	8	6	46
45	36	34	25	16	14	5
4	44	42	33	24	15	13
12	3	43	41	32	23	21
20	11	2	49	40	31	22

FIGURA **12**

Scrivere un programma C++ che costruisca il quadrato magico di ordine (dispari) fornito dall'utente.

4 L'algoritmo SOUNDEX di Russell e Odell (brevettato nel 1918 e nel 1922) consente di determinare se due parole hanno un suono simile nella lingua inglese; si calcola la stringa SOUNDEX a partire dalla stringa contenente la parola applicando le seguenti regole:

- il primo carattere della stringa SOUNDEX è il primo carattere della parola;
- si devono eliminare le seguenti lettere dalla parola: *a*, *e*, *h*, *i*, *o*, *u*, *w*, *y*;
- dopo il primo carattere nella stringa SOUNDEX devono essere inseriti caratteri numerici corrispondenti alle lettere della parole secondo le seguenti sostituzioni:
- *b*, *f*, *p*, *v* → 1,
- *c*, *g*, *j*, *k*, *q*, *s*, *x*, *z* → 2,
- *d*, *t* → 3,
- *l* → 4,
- *m*, *n* → 5,
- *r* → 6;
- si devono rimuovere dalla stringa SOUNDEX i numeri successivi uguali, eccetto il primo;
- il risultato è dato dalla prima lettera e dalle prime 3 cifre; se sono meno di tre si aggiunge 0.

Scrivere una funzione C/C++ che determini la stringa SOUNDEX di una stringa fornita come argomento; scrivere un programma C++ che, utilizzando questa funzione, visualizzi la stringa SOUNDEX di una parola inserita dall'utente.

A7 Le strutture in C++

Esistono situazioni in cui le variabili semplici o gli *array* non consentono una efficiente gestione dei dati. Un esempio tipico è quello relativo alla gestione di una agenda di amici in cui, per ciascuno di essi, si intende memorizzare:

- il nominativo (cognome e/o nome);
- l'indirizzo;
- la città;
- il numero di telefono;
- l'indirizzo di posta elettronica;
- l'anno di nascita;
- il mese di nascita;
- il giorno di nascita.

Come si può vedere i dati da gestire *non* sono omogenei: alcuni di questi sono di tipo numerico e altri sono di tipo alfanumerico (stringhe di caratteri).
Si potrebbe pensare di risolvere la questione facendo uso di diversi vettori, tra i quali stabilire poi una correlazione in base al valore degli indici (il vettore dei nominativi, il vettore degli indirizzi, …): un'idea del genere è da scartare in quanto il risultato in termini di codice necessario per la gestione sarebbe inutilmente complicato. Ciò di cui abbiamo bisogno è una nuova struttura di dati detta *record*.

1 Le strutture come tipi di dato definiti dall'utente

▶ Un **record** è un insieme non necessariamente omogeneo di dati, identificato da un nome, dove ogni singola componente (**campo**) è identificata da un proprio nome giustapposto a quello dell'insieme stesso in una forma del tipo:

insieme.campo

Il linguaggio C++ permette all'utente programmatore di definire propri tipi di dato (UDT, *User-Defined data-Type*) a partire dai tipi di dati predefiniti forniti dal linguaggio. Tra questi le ***struct*** permettono di definire strutture di tipo record.

La seguente definizione implementa il nuovo tipo di dato *recRubrica* organizzato come un record: si vede chiaramente come i singoli campi abbiano un proprio nome e non siano omogenei tra di loro.

```
struct  recRubrica
{
  char  nominativo[20];  // cognome e/o nome
  char  indirizzo[30];   // indirizzo
  char  citta[20];       // citta'
  char  n_tel[15];       // numero di telefono
  char  e_mail[30];      // e-mail
  int   anno;            // anno di nascita
  int   mese;            // mese di nascita
  int   giorno;          // giorno di nascita
};
```

Una *struct* può essere definita in un programma sia a livello globale sia a livello locale. Dopo la definizione è possibile dichiarare variabili del nuovo tipo *recRubrica*. Con

```
recRubrica amico, contatto;
```

vengono dichiarate le variabili *amico* e *contatto* di tipo *recRubrica* e, per esempio, è possibile fare riferimento all'anno di nascita della variabile *amico* con la seguente notazione:

```
amico.anno
```

OSSERVAZIONE È possibile riferirsi al campo *amico.anno* in una qualsiasi istruzione, come un assegnamento

```
amico.anno = 1995;
```

o un'espressione

```
eta = anno_corrente - amico.anno;
```

Non avrebbero invece senso eventuali riferimenti a *recRubrica*, che è una definizione di struttura e **non** una variabile.

Un'altra modalità utilizzabile per la dichiarazione di variabili strutturate è la dichiarazione contestuale alla definizione della struttura:

```
struct  recRubrica
{
  char  nominativo[20];  // cognome e/o nome
  char  indirizzo[30];   // indirizzo
  char  citta[20];       // citta'
  char  n_tel[15];       // numero di telefono
  char  e_mail[30];      // e-mail
  int   anno;            // anno di nascita
  int   mese;            // mese di nascita
  int   giorno;          // giorno di nascita
} amico, contatto;
```

È infine possibile dichiarare una o più variabili strutturate senza assegnare un nome specifico al tipo di struttura, che pertanto rimane funzionale solo a tale dichiarazione:

```
struct
{
  char  nominativo[20];  // cognome e/o nome
  char  indirizzo[30];   // indirizzo
  char  citta[20];       // citta'
  char  n_tel[15];       // numero di telefono
  char  e_mail[30];      // e-mail
  int   anno;            // anno di nascita
  int   mese;            // mese di nascita
  int   giorno;          // giorno di nascita
} amico, contatto;
```

Dovendo assegnare il valore di una variabile di tipo *recRubrica* a un'altra variabile dello stesso tipo, è possibile copiare i valori di ogni singolo campo della variabile originale nel corrispondente campo della variabile destinazione (la funzione *copia Stringa()* è stata definita nel capitolo 6):

```
recRubrica nuovoAmico;
...
copiaStringa(nuovoAmico.nominativo,amico.nominativo);
copiaStringa(nuovoAmico.indirizzo,amico.indirizzo);
copiaStringa(nuovoAmico.citta,amico.citta);
copiaStringa(nuovoAmico.n_tel,amico.n_tel);
copiaStringa(nuovoAmico.e_mail,amico.e_mail);
nuovoAmico.anno = amico.anno;
nuovoAmico.mese = amico.mese;
nuovoAmico.giorno = amico.giorno;
```

oppure, e questo è uno dei maggiori vantaggi offerti dalle variabili di tipo *struct*, in un'unica soluzione con una istruzione come la seguente:

```
nuovoAmico = amico; // copia tutti i campi della struttura!
```

OSSERVAZIONE Si noti come l'operatore di assegnamento applicato alle strutture copi automaticamente anche il contenuto dei campi di tipo complesso come i vettori di caratteri, che non è altrimenti possibile assegnare singolarmente.

L'inizializzazione di una variabile strutturata può avvenire assegnando valori ai singoli campi, ma anche all'atto della sua dichiarazione:

```
recRubrica amico = {"Rossi Mario", "Via Roma, 30", "Livorno",
                    "0586123456", "mario95@posta.it",
                    1995, 12, 25};
```

OSSERVAZIONE I singoli campi di una *struct* possono essere a loro volta di tipo complesso (*array* o anche tipi definiti tramite *struct*).

ESEMPIO

La struttura della rubrica potrebbe essere ridefinita in modo da poter gestire la data di nascita come un unico campo oltre che mediante le singole componenti:

```
struct   recRubrica
{
  char   nominativo[20];
  char   indirizzo[30];
  char   citta[20];
  char   n_tel[15];
  char   e_mail[30];
  struct recDataNascita
  {
    int anno;    // anno di nascita
    int mese;    // mese di nascita
    int giorno;  // giorno di nascita
  } dataNascita;
};
```

Dovendo accedere ai singoli campi della data di nascita si utilizzerà una notazione come le seguenti:

```
recRubrica amico;
...
amico.dataNascita.anno = 1995;
amico.dataNascita.mese = 12;
amico.dataNascita.giorno = 25;
...
eta = anno_corrente - amico.dataNascita.anno;
```

In questo caso è superfluo definire un nome di tipo per la struttura *recDataNascita* in quanto, essendo una sottostruttura locale, la sua visibilità sarà limitata alla struttura *recRubrica* e quindi non utilizzabile al suo esterno. È invece fondamentale definire almeno un campo per questo tipo (*dataNascita*) per consentire l'accesso ai campi della struttura (*anno*, *mese*, *giorno*).

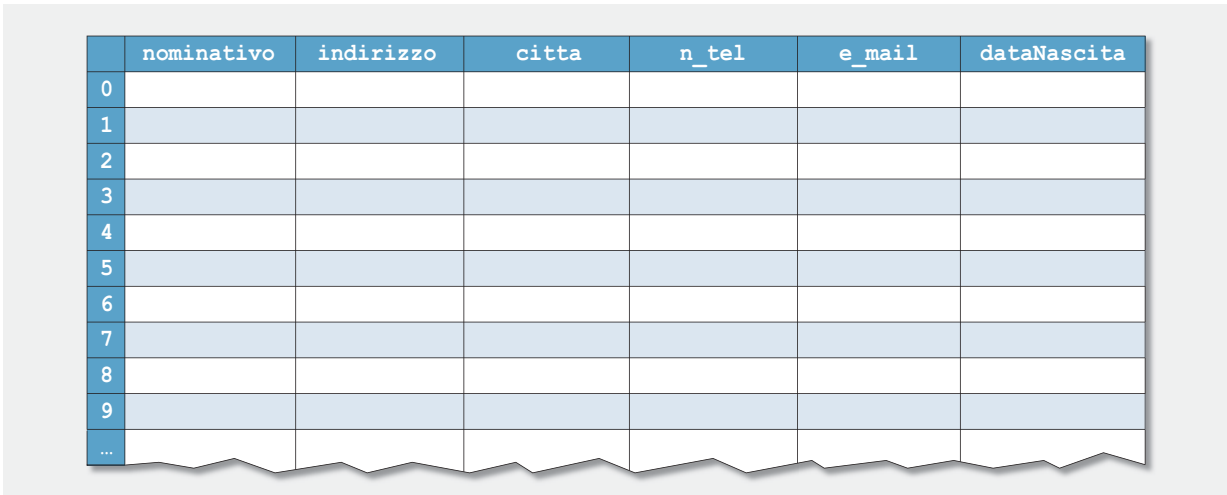
2 Tabelle come *array* di strutture

L'aver definito una struttura, e di conseguenza un nuovo tipo di dato con cui creare variabili strutturate, non risolve ancora completamente il problema dell'agenda presentato in apertura del capitolo. Supponendo di voler gestire i dati relativi a 100 diversi amici è necessario definire una tabella che contenga 100 voci di tipo *recRubrica*. Con una dichiarazione come la seguente

```
recRubrica agenda[100];
```

si crea un vettore di elementi di tipo *recRubrica*. Un *array* di elementi dichiarato con elementi di tipo *struct* viene denominato **tabella**.

OSSERVAZIONE Il nome di tabella è giustificato da una rappresentazione grafica della struttura complessiva che si viene a costituire (FIGURA **1**).



	nominativo	indirizzo	citta	n_tel	e_mail	dataNascita
0						
1						
2						
3						
4						
5						
6						
7						
8						
9						
...						

FIGURA **1**

OSSERVAZIONE Si noti come una tabella sia un *array* a tutti gli effetti perché tutti gli elementi, indipendentemente dal fatto che al loro interno possano avere campi disomogenei, sono tra di loro tutti dello stesso tipo, in accordo con la definizione della struttura che li caratterizza.

L'accesso ai singoli elementi di una tabella avviene con la tecnica propria degli *array*, ovvero specificando il nome dell'*array* e l'indice dell'elemento a cui si vuole accedere; l'accesso a uno specifico campo di un elemento avviene tramite il nome del campo. Per esempio, rispetto alla nostra agenda, con l'istruzione:

```
cout<<agenda[9].nominativo;
```

è possibile visualizzare il cognome/nome del contatto che occupa la decima posizione della tabella agenda, mentre con:

```
cin>>agenda[9].e_mail;
```

è possibile inserirne da tastiera l'indirizzo di posta elettronica.

ESEMPIO

I vertici di una poligonale sono punti nel piano cartesiano, ognuno dei quali ha coordinate *x* e *y*. La spezzata che si ottiene congiungendo tali vertici con dei segmenti due a due consecutivi costituisce la poligonale stessa. La poligonale può essere *chiusa* o *aperta*, a seconda del fatto che ogni segmento abbia due segmenti consecutivi oppure no. Una poligonale chiusa viene anche detta *poligono*.
La poligonale è *non intrecciata* se i segmenti, essendo consecutivi due a due, non hanno altri punti in comune; viceversa è *intrecciata* se i segmenti hanno anche altri punti in comune.
Di seguito viene presentato un programma che, definita la struttura per la memorizzazione delle coordinate di un punto e la tabella per la gestione di una poligonale non intrecciata di 4 vertici, ne calcola la lunghezza e stabilisce se tale poligonale è un poligono (nell'esempio, ci si è riferiti al fatto che il primo e l'ultimo punto abbiano stesse coordinate per determinare se siamo in presenza di un poligono). In quest'ultimo caso il programma fornisce la lunghezza del perimetro del poligono. ▶

ESEMPIO

```
struct PUNTO
{
  double x; // ascissa
  double y; // ordinata
};

// calcolo della distanza tra due punti
double distanza(PUNTO a, PUNTO b)
{
  return sqrt(pow((a.x - b.x),2) + pow((a.y - b.y),2));
}

// verifica poligonale chiusa: poligono
bool poligono(PUNTO poligonale[],int n_vertici)
{
  return(poligonale[0].x==poligonale[n-1].x)&&
        poligonale[0].y==poligonale[n-1].y);
}

void main(void)
{
  int v;
  double perimetro = 0.0;
  PUNTO poligonale[4];

  for (v=0; v<4; v++)
  {
    cout<<"Vertice "<<(v+1)<<endl;
    cout<<"ascissa: ";
    cin>> poligonale [v].x;
    cout<<"ordinata: ";
    cin>> poligonale [v].y;
  }

  for (v=1; v<4; v++)
    lunghezza=lunghezza+distanza(poligonale[v], poligonale[v-1]);
  if (poligono(poligonale,4)
    cout<<"Perimetro poligono= "<< lunghezza <<endl;
  else
    cout<<"Lunghezza poligonale= "<< lunghezza <<endl;
}
```

Sintesi

Record. Insieme non necessariamente omogeneo di dati, identificato da un nome, dove ogni singola componente (campo) viene identificata da un proprio nome giustapposto a quello dell'insieme.

Campo. Singola componente di un record che a sua volta può essere strutturata.

Struct. Costrutto dichiarativo del linguaggio C++ per la definizione di record.

Tabella. *Array* di elementi il cui tipo è definito in accordo con la definizione di una struttura; tutti gli elementi di una tabella sono omogenei, pur essendo al loro interno formati da campi di tipo eventualmente diverso.

QUESITI

1 **Indicare quali delle seguenti affermazioni sono vere e quali false.**

A Un campo di una struttura non può essere un *array*.

B Il campo *nome* di una *struct Studente* può essere riferito con *Studente.nome*.

C Un campo di una struttura può essere a sua volta una *struct*.

D Un *array* di elementi dichiarati di un tipo definito tramite una *struct* viene detto «tabella».

2 **Data la seguente definizione**

```
struct recStudente
{
  char nominativo[20];
  char sesso;
  int  voti[10];
};
```

indicare quali delle seguenti affermazioni sono vere e quali false.

A Per dichiarare una tabella di 10 studenti si deve scrivere

```
recStudente classe[10];
```

B Per accedere al terzo voto di uno studente si deve scrivere

```
recStudente.voti[2];
```

C Per poter accedere a un qualsiasi campo della struttura è necessario definire almeno una variabile del tipo *recStudente*.

D Una volta dichiarata una tabella *classe* di 10 elementi di tipo *recStudente* è possibile assegnare il voto 9 alla seconda prova sostenuta dal terzo studente scrivendo

```
classe[2].voti[1] = 9;
```

3 **Data la seguente definizione**

```
struct recStudente
{
  char nominativo[20];
  char sesso;
  int  voti[10];
  struct {
    int  anno_corso;
    char sezione;
    char indirizzo;
  } classe;
};
```

e la dichiarazione

```
recStudente Giovanni;
```

indicare quali delle seguenti affermazioni sono vere e quali false.

A Per visualizzare l'anno di corso dello studente Giovanni si deve scrivere

```
Giovanni.classe.anno_corso;
```

B La dichiarazione di classe costituisce un tipo di dato definito dall'utente.

C La dichiarazione di classe è errata perché una sottostruttura deve contenere tipi di dati omogenei.

D La dichiarazione di classe è errata perché il nome doveva essere posto vicino alla parola chiave *struct* (`struct classe{…}`)

4 **Data la seguente definizione**

```
struct
{
  long alfa, beta, gamma;
  char delta;
  int  teta[8];
} omega;
```

indicare quali delle seguenti affermazioni sono vere e quali false.

A *Omega* è il nome di una struttura.

B *Omega* è il nome di una variabile strutturata.

C È possibile dichiarare una variabile scrivendo.

```
omega variabile;
```

D Non è possibile scrivere `omega.teta[3]` perché *omega* è il nome di una *struct*.

ESERCIZI

1 

Definire la struttura dati Auto che descrive un'automobile in base a marca, cilindrata, anno di immatricolazione e acquirente (l'acquirente è caratterizzato dai soli nome e cognome). Scrivere un programma C++ che, dichiarata la tabella Auto-

salone di tipo Auto (per comodità se ne fissi la dimensione a 10 elementi), consenta di inserire da tastiera i dati delle auto vendute e di visualizzare il solo cognome degli acquirenti di auto di cilindrata superiore a 1500 cc, oltre al numero totale di auto che sono state immatricolate in un anno richiesto all'utente.

2 Definire la struttura dati Squadra che descrive una squadra di calcio ideale con il relativo allenatore. Una squadra è identificata da nome, colore della maglia, punteggio corrente e allenatore. L'allenatore è una persona caratterizzata da nome e cognome più il numero dei titoli vinti (un numero intero). Scrivere un programma C++ che, dichiarata una tabella Fantacalcio di tipo Squadra (per comodità se ne fissi la dimensione a 10 elementi), consenta di inserire da tastiera i dati delle squadre e dei relativi allenatori e di visualizzare il solo cognome degli allenatori di squadre che hanno più di 30 punti in classifica, oltre al numero totale di allenatori che hanno vinto almeno un titolo.

3 Un'immagine digitale a colori è una matrice di pixel, ognuno dei quali è caratterizzato dall'intensità percentuale dei tre colori primari (rosso, verde e blu). La seguente funzione C/C++

```
int colorToGray(int red, int green, int
                blue)
{
  float gray;

  gray = (float)(6*green + 3*red
         + 1*blu)/10;
  return (int)gray;
}
```

converte un colore nel corrispondente livello di grigio tipico delle immagini in bianco e nero. Dopo avere definito una struttura per la rappresentazione del colore, scrivere una funzione C++ che converta un'immagine a colori fornita come parametro in un'immagine in bianco e nero anch'essa fornita come parametro.

4 Dopo avere definito una struttura per la memorizzazione dei dati degli studenti di una classe (nominativo, sesso e anno di nascita) e dei voti relativi alle varie discipline (Italiano, Inglese, Matematica), si scriva una funzione C/C++ *piuFacile()* che, a partire da un vettore di studenti fornito come argomento, determini la disciplina che ha la media dei voti più alta.

5 Una biblioteca ha identificato tutti i libri della propria collezione mediante un codice numerico. Si deve realizzare un programma C++ che consenta di effettuare le seguenti operazioni memorizzando le informazioni relative a un libro (codice, titolo, autore, anno di pubblicazione, editore) in un vettore di strutture:

- aggiunta di un nuovo libro alla collezione;
- visualizzazione dell'elenco dei libri della collezione;
- visualizzazione delle informazioni relative a un libro a partire dal codice;
- visualizzazione delle informazioni relative a un libro a partire dal titolo.

LABORATORIO

1 Progettare e implementare una libreria di funzioni C/C++ di geometria analitica a partire dai seguenti tipi:

```
struct POINT
{
  double x; // ascissa
  double y; // ordinata
};

struct CIRCLE
{
  POINT centro;
  double raggio;
};

struct LINE
{
  double coefficienteAngolare;
  double intercetta; // intersezione
                     // con asse delle
                     // ordinate
};
```

La libreria dovrà comprendere le seguenti funzioni:

- punto medio tra due punti;
- distanza tra due punti;
- retta passante per due punti;
- circonferenza passante per tre punti;
- distanza tra una retta e un punto.

I tipi e i prototipi delle funzioni devono essere definiti in un file di intestazione; le funzioni devono essere implementate in un file separato. È richiesta la realizzazione di un programma C++ di test di tutte le funzioni realizzate con dati inseriti dall'utente.

A8 Ordinamento e ricerca

La necessità di ordinare un insieme di dati nasce da molte esigenze della vita quotidiana: fornire un elenco di nomi in ordine alfabetico, stilare la classifica di una gara in funzione dei punti totalizzati dai vari concorrenti, e così via.
Per esempio, volendo gestire il risultati di una gara campestre a cui partecipano sia ragazzi sia ragazze, la classifica potrebbe essere gestita tramite un *array* di elementi definiti secondo la seguente struttura:

```
struct recPartecipante
{
  int numero;           // numero di gara partecipante
  char nominativo[25];  // cognome e nome
  char sex;             // sesso (M/F)
  int eta;              // età dell'atleta
  int tempo;            // tempo impiegato espresso in secondi
};
```

e dichiarato come nel seguito:

```
Partecipante classifica[100];
```

Alcune operazioni a cui potremmo essere interessati sono le seguenti:

- fornire l'elenco in ordine alfabetico dei partecipanti;
- fornire la classifica finale assoluta;
- fornire le classifiche per sesso o per età;
- ricercare gli atleti che abbiano riportato specifici tempi (per esempio: il miglior tempo, il peggior tempo, inferiore a un certo tempo, superiore a quello di un diverso atleta, …).

Per i primi tre punti è necessario ordinare i dati in funzione dei valori del campo su cui si vuole operare e anche per quanto richiesto al quarto punto si può osservare che l'avere la classifica ordinata in funzione del tempo può facilitare l'operazione di ricerca: i due aspetti dell'ordinamento e della ricerca sono infatti strettamente connessi tra di loro.
Il nostro obiettivo, nel presente capitolo, è proprio quello di indagare le tecniche di ordinamento e di ricerca dei dati. Per praticità negli esempi utilizzeremo semplici *array* di interi, ma – come vedremo – sarà immediato adattali a situazioni più articolate come quella appena descritta.

1 Ordinamento

Ordinare un *array* significa confrontare tra di loro i suoi elementi e operare con opportuni scambi in modo tale da disporne il contenuto in ordine crescente o decrescente a seconda del tipo di ordinamento voluto. *Sort* è il termine tecnico inglese con cui si indica un algoritmo di ordinamento.

OSSERVAZIONE Si tenga presente che ordinare un insieme di dati tramite un algoritmo di ordinamento significa perdere la posizione iniziale che questi dati occupavano in seno all'insieme che, talvolta, potrebbe essere una informazione importante: si pensi a situazioni in cui i dati sono stati acquisiti con una cadenza temporale per cui il primo dato rappresenta la misura di un fenomeno al tempo t_0, il secondo al tempo t_1 e così via.

Esistono molti tipi di algoritmi di ordinamento più o meno complessi e più o meno veloci. In genere, al di là del risultato ultimo che è comunque quello di ottenere l'ordinamento dell'*array*, le diverse strategie adottate possono differire anche di molto tra di loro. In prima approssimazione si potrebbe affermare che la «complessità» dell'algoritmo è direttamente proporzionale alla velocità di ordinamento del medesimo. I moderni elaboratori sono così veloci che su piccoli *array* diventa praticamente ininfluente il tipo di algoritmo da scegliere per effettuare un ordinamento, per cui le prestazioni di un buon *sort* diventano significativamente apprezzabili solo se questo viene applicato a un vettore con un numero rilevante di elementi (nell'ordine dei milioni).
In ogni caso, visto che le due operazioni fondamentali su cui questi tipi di algoritmi si basano sono il confronto e lo scambio di elementi, risulta abbastanza chiaro che adottando una strategia che minimizzi il numero complessivo di queste si ottiene un ordinamento più veloce. La velocità di ordinamento dipende ovviamente anche dalla distribuzione iniziale dei valori nell'*array*, e questo non è tanto legato al numero di confronti quanto a quello degli scambi che saranno effettuati. Si tenga infatti presente che, in termini di velocità, un'operazione di scambio è più «costosa» di un'operazione di confronto.

Nel seguito analizzeremo e confronteremo tra loro due tra i più semplici algoritmi di ordinamento: l'***exchange-sort*** e il ***bubble-sort***. Nelle funzioni C++ che verranno presentate come implementazione degli algoritmi i parametri formali sono rispettivamente:

vet[]	*Array* su cui si esegue la ricerca
N	Numero degli elementi del vettore interessati all'ordinamento (sarà minore o uguale al numero degli elementi del vettore)

Come accennato in precedenza lo scambio tra gli elementi di un *array* è una delle operazioni su cui si basano gli algoritmi di ordinamento. La fun-

Complessità degli algoritmi di ordinamento

Come già visto, la valutazione della complessità di un algoritmo determina il costo del procedimento in termini di risorse di calcolo, quali il tempo di elaborazione e la quantità di memoria utilizzata. Nell'ipotesi semplificata di calcolarla in base al numero di operazioni svolte, ciascuna valutata a costo unitario in termini di tempo (nel caso del sort si tratta di confronti o di scambi), è ragionevole valutare il comportamento asintotico dell'algoritmo, ovvero calcolare il numero di operazioni eseguite per input di dimensione N potenzialmente infinita. Analizzando un ordinamento su vettori di grandi dimensioni ($N \to \infty$) conviene valutare due casi: vettore già ordinato (caso migliore) e ordinato in senso inverso (caso peggiore). È quindi possibile procedere al calcolo del numero dei confronti e degli scambi nelle due situazioni, determinarne il valore medio e successivamente l'ordine della complessità.

zione C++ che segue realizza tale operazione tra due variabili intere di cui vengano passati come parametri i rispettivi riferimenti e sarà utilizzata negli algoritmi di ordinamento che andiamo a introdurre:

```
void scambio (int &x, int &y)
{
  int z;

  z=x;
  x=y;
  y=z;
}
```

1.1 *Exchange-sort*

Questo tipo di algoritmo adotta la seguente strategia.

> A partire dal primo e fino al penultimo si fissa un elemento del vettore alla volta e lo si confronta con tutti i successivi, effettuando uno scambio se l'elemento fissato e l'elemento di confronto non sono ordinati secondo il criterio prescelto (crescente o decrescente).

La seguente funzione implementa quanto descritto sopra per un ordinamento crescente:

```
void exchangeSort (int vet[], int N)
{
  int i, j;

  for (i=0; i<N-1; i++) // ciclo esterno per elementi fissi
    for (j=i+1; j<N; j++) // ciclo interno per elementi successivi
      if (vet[i] > vet[j]) // confronto con elementi successivi
        scambio(vet[i], vet[j]); // eventuale scambio elementi
}
```

Si nota subito che l'algoritmo fa uso di due cicli determinati, il più esterno dei quali usa l'indice *i* per scandire il vettore e serve a fissare di volta in volta l'elemento da confrontare con tutti i successivi (si noti in particolare come questo ciclo arrivi solo fino al penultimo elemento del vettore).
Il ciclo più interno, che serve a confrontare l'elemento fissato con tutti i successivi individuati tramite l'indice *j*, parte appunto dall'elemento successivo a quello fissato nel ciclo più esterno e arriva fino all'ultimo elemento.

Complessità dell'*exchange-sort*

Considerando il vettore ordinato in senso inverso (caso peggiore) e il vettore già ordinato (caso migliore), per l'*exchange-sort* si hanno in ogni caso

$$\frac{N\cdot(N-1)}{2}$$

confronti e in media

$$\frac{N\cdot(N-1)}{4}$$

scambi che diventano

$$\frac{N\cdot(N-1)}{2}$$

nel caso peggiore.
La complessità per $N \to \infty$ è quindi nell'ordine di N^2.

ESEMPIO

Data la seguente occorrenza del nostro array *vet[]*:

3	2	5	1	6	5

l'algoritmo opera come in FIGURA **1**.

i	j	vet[i]	vet[j]	scambio
0	1	2	3	sì
0	2	2	5	no
0	3	2	1	sì
0	4	1	6	no
0	5	1	5	no

2	3	5	1	6	5

1	3	5	2	6	5

1	3	5	2	6	5

FIGURA **1**

A questo punto il ciclo interno termina (5 confronti in totale) e nella prima posizione del vettore vi è sicuramente l'elemento più piccolo. Si può quindi procedere all'individuazione del più piccolo tra i rimanenti applicando il confronto tra l'elemento di indice 1 e tutti i successivi (FIGURA **2**).

i	j	vet[i]	vet[j]	scambio
1	2	3	5	no
1	3	3	2	sì
1	4	2	6	no
1	5	2	5	no

1	2	5	3	6	5

1	2	5	3	6	5

FIGURA **2**

Il ciclo interno termina (4 confronti in totale) e le prime due posizioni del vettore sono già ordinate. Si procede a fissare l'elemento di indice 2 e a confrontarlo con tutti i successivi (FIGURA **3**).

i	j	vet[i]	vet[j]	scambio
2	3	5	3	sì
2	4	3	6	no
2	5	3	5	no

1	2	3	5	6	5

1	2	3	5	6	5

FIGURA **3**

Il ciclo interno termina (3 confronti in totale) e le prime tre posizioni del vettore sono già ordinate. Si procede a fissare l'elemento di indice 3 e a confrontarlo con tutti i successivi (FIGURA **4**).

i	j	vet[i]	vet[j]	scambio
3	4	5	6	no
3	5	5	5	no

1	2	3	5	6	5

1	2	3	5	6	5

FIGURA **4**

Il ciclo interno termina (2 confronti in totale) e le prime quattro posizioni del vettore sono già ordinate. Si procede a fissare l'elemento di indice 4 e a confrontarlo con il successivo e ultimo (FIGURA **5**).

i	j	vet[i]	vet[j]	scambio
4	5	5	5	sì

▶

1	2	3	5	5	6

FIGURA **5**

Il ciclo interno termina (1 confronto in totale) come pure quello esterno: il vettore è ora ordinato.

1	2	3	5	5	6

OSSERVAZIONE Si noti come utilizzando due cicli determinati questo algoritmo effettui sempre lo stesso numero di confronti; per un vettore di dimensione N il numero di confronti è dato da:

$$\frac{N \cdot (N-1)}{2}$$

Il numero dei confronti è fisso in ogni situazione anche nel caso limite in cui il vettore a cui si applica l'algoritmo sia già ordinato. Invece nulla si può dire, a priori, in merito al numero di scambi, che ovviamente dipende dalla distribuzione iniziale degli elementi nel vettore.

Riprendiamo ora l'esempio relativo alla gestione dei dati di una corsa campestre presentato in apertura del capitolo e applichiamo l'algoritmo di ordinamento appena visto per ottenere la classifica finale assoluta.
La funzione di ordinamento diverrà:

```
void exchangeSort (recPartecipante vet[], int N)
{
  int i,j;

  for (i=0; i<N-1; i++) // ciclo esterno per elementi fissi
    for (j=i+1; j<N; j++) // ciclo interno per elementi successivi
      //confronto con elementi successivi in funzione del tempo
      if (vet[i].tempo > vet[j].tempo)
        scambio(vet[i], vet[j]); // eventuale scambio elementi
}
```

mentre la sua invocazione avverrà come nel seguito:

```
…
exchangeSort(classifica, 100);
…
```

OSSERVAZIONE In riferimento all'esempio precedente si noti che anche gli elementi su cui lavora la funzione di scambio utilizzata all'interno di *exchange-sort* dovranno essere congruenti con il tipo di dati da scambiare:

```
void scambio(recPartecipante &x, recPartecipante &y)
{
  recPartecipante z;

  z=x;
  x=y;
  y=z;
}
```

In effetti, nell'esempio, mentre il confronto viene effettuato solo sul campo «tempo», lo scambio avviene coinvolgendo due righe intere della tabella prese in un'unica soluzione.

A questo punto risulta chiaro come sia possibile adattare la funzione *exchange-sort* a vari tipi di ordinamento intervenendo sulla sola istruzione condizionale.

ESEMPIO

L'istruzione

```
if (strcmp(vet[i].nominativo, vet[j].nominativo) > 0)
```

permette di ottenere un elenco alfabetico dei partecipanti (la funzione *strcmp* è una funzione della libreria C++ che consente di confrontare tra loro due stringhe fornendo come risultato un valore nullo se le due stringhe sono uguali, positivo se la prima stringa precede nell'ordinamento alfabetico la seconda e negativo se la seconda stringa precede la prima nell'ordinamento alfabetico).

ESEMPIO

È anche possibile ordinare la classifica per sesso e tempo utilizzando la seguente istruzione:

```
if ((vet[i].sex > vet[j].sex)||((vet[i].sex == vet[j].sex)
    && (vet[i].tempo > vet[j].tempo)))
  scambio(vet[i], vet[j]);
```

1.2 *Bubble-sort*

Questo tipo di algoritmo adotta la seguente strategia.

Si confrontano gli elementi a coppie, ognuno con il successivo: se la coppia non è ordinata secondo il criterio prescelto (crescente o decrescente) si effettua lo scambio e si annota l'avvenuto scambio; se al termine dei confronti si è effettuato almeno uno scambio si riparte con il confronto di tutti gli elementi come descritto in precedenza, altrimenti il vettore è ordinato.

La seguente funzione C++ implementa l'algoritmo descritto per un ordinamento crescente (*s* è la variabile di stato che tiene conto del fatto che siano avvenuti scambi o meno):

Complessità del *bubble-sort*

Per il *bubble-sort* nel caso peggiore si hanno

$$N \cdot (N-1)$$

confronti e

$$\frac{N \cdot (N-1)}{2}$$

scambi; nel caso migliore

$$N-1$$

confronti e nessuno scambio, mentre in media si hanno

$$\frac{(N+1) \cdot (N-1)}{2}$$

confronti e

$$\frac{N \cdot (N-1)}{4}$$

scambi.

La complessità per $N \to \infty$ è ancora nell'ordine di N^2, ma si noti che in media l'*exchange-sort* risulta essere più veloce del *bubble-sort*.

```
void bubbleSort (int vet[], int N)
{
  int i;
  bool s;

  do // ciclo da ripetere fino se è stato operato almeno uno scambio
    {
      s=false; // «azzeramento» variabile indicatore scambio
      for (i=0; i<N-1; i++) //ciclo di scansione degli elementi
        if (vet[i] > vet[i+1]) // confronto coppia elementi
          {
            scambio(vet[i],vet[i+1]);//eventuale scambio elementi
            s=true; // si «ricorda» lo scambio avvenuto
          }
    } while(s);
}
```

Anche in questo caso l'algoritmo fa uso di due cicli: il più interno è determinato e serve a confrontare, ed eventualmente scambiare tra loro, gli elementi a coppie; il più esterno, indeterminato, viene ripetuto fino a quando nel ciclo più interno è avvenuto almeno uno scambio. Il *bubble-sort* non è un algoritmo particolarmente efficiente, ma ha una sua valenza da un punto di vista didattico in virtù della sua semplicità.

ESEMPIO

Data la seguente occorrenza del nostro solito *array vet[]:*

3	2	5	1	6	5

l'algoritmo opera come in FIGURA **6**:

i	vet[i]	vet[i+1]	scambio	s
0	3	2	sì	V
1	3	5	no	V
2	5	1	sì	V
3	5	6	no	V
4	6	5	sì	V

2	3	5	1	6	5

2	3	1	5	6	5

2	3	1	5	5	6

2	3	1	5	5	6

FIGURA **6**

A questo punto il ciclo interno termina e nell'ultima posizione del vettore vi è sicuramente l'elemento più grande. È stato effettuato almeno uno scambio per cui si pone la variabile di stato *s* a «falso» e si riapplica il confronto a coppie degli elementi dall'inizio del vettore (FIGURA **7**).

i	vet[i]	vet[i+1]	scambio	s
0	2	3	no	**F**
1	3	1	sì	**V**
2	3	5	no	**V**
3	5	5	no	**V**
4	5	6	no	**V**

FIGURA **7**

Il ciclo interno termina e nelle ultime due posizione del vettore vi sono sicuramente gli elementi più grandi. È stato effettuato almeno uno scambio, per cui si pone la variabile di stato *s* a «falso» e si riapplica il confronto a coppie degli elementi dall'inizio del vettore (FIGURA **8**).

i	vet[i]	vet[i+1]	scambio	s
0	2	1	sì	**V**
1	2	3	no	**V**
2	3	5	no	**V**
3	5	5	no	**V**
4	5	6	no	**V**

FIGURA **8**

Il ciclo interno termina e nelle ultime tre posizione del vettore vi sono sicuramente gli elementi più grandi. Pur essendo il vettore ormai ordinato, è stato effettuato almeno uno scambio, per cui si pone la variabile di stato *s* a «falso» e si applica nuovamente il confronto a coppie degli elementi dall'inizio del vettore (FIGURA **9**).

i	vet[i]	vet[i+1]	scambio	s
0	1	2	no	**F**
1	2	3	no	**F**
2	3	5	no	**F**
3	5	5	no	**F**
4	5	6	no	**F**

FIGURA **9**

Il ciclo interno termina e la variabile di stato *s* è «falso»: il vettore è ora ordinato.

OSSERVAZIONE Si noti come, utilizzando questo algoritmo, per due cicli di cui uno indeterminato non si possa calcolare a priori il numero di confronti che esso effettuerà e nulla si può dire anche in merito al numero di scambi.
Entrambi i valori dipenderanno dalla distribuzione iniziale degli elementi nel vettore. Nel caso particolare in cui l'algoritmo venga applicato a un vettore già ordinato si può dire però che il numero dei confronti sarà

$$N - 1$$

e ovviamente non ci saranno scambi.

OSSERVAZIONE L'algoritmo *bubble-sort* tende a far «affondare» gli elementi più «pesanti» (dopo la prima scansione l'elemento più grande occupa sicuramente l'ultima posizione, dopo la seconda l'elemento più grande dei rimanenti occupa sicuramente la penultima posizione). Si può dunque pensare a una ottimizzazione della versione di tale algoritmo proposta in precedenza modificandolo in modo da diminuire di uno il numero degli elementi da confrontare a ogni scansione del vettore successiva alla prima. A una più attenta analisi, infatti, si può notare che dopo la prima scansione, e in tutte quelle successive, è sufficiente riconfrontare solo gli elementi del vettore di indice minore alla posizione in cui al passo precedente è avvenuto l'ultimo scambio: da quel punto in poi, infatti, gli elementi del vettore saranno sicuramente già ordinati. Queste considerazioni ci consentono di proporre la seguente versione ottimizzata dell'algoritmo:

```
void bubbleSortOtt (int vet[], int N)
{
  int i, s;

  do
    {
      s=0;
      for (i=0; i<N-1; i++)
          if (vet[i] > vet[i+1])
            {
              scambio(vet[i], vet[i+1]);
              s=i; // si «ricorda» la posizione dello scambio
            }
        if (s>0)
          N=s; // si determina il limite degli elementi
               // da controllare
      } while (s!=0);
}
```

ESEMPIO

Applicando il *bubble-sort* ottimizzato al solito esempio illustrato in precedenza si ha la situazione di FIGURA **10**.

i	vet[i]	vet[i+1]	scambio	s
0	3	2	sì	**0**
1	3	5	no	**0**
2	5	1	sì	**2**
3	5	6	no	**2**
4	6	5	sì	**4**

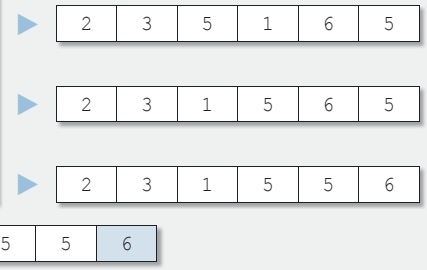


FIGURA **10**

A questo punto il ciclo interno termina e nell'ultima posizione del vettore vi è sicuramente l'elemento più grande. È stato effettuato almeno uno scambio, di cui l'ultimo in posizione 4, per cui si pone la variabile di stato *s* a 0 e si riapplica il confronto a coppie degli elementi dall'inizio del vettore fino all'indice 3 (FIGURA **11**).

i	vet[i]	vet[i+1]	scambio	s
0	2	3	no	**0**
1	3	1	sì	**1**
2	3	5	no	**1**
3	5	6	no	**1**

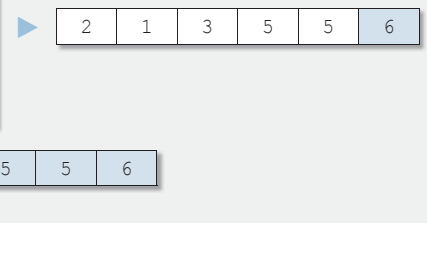


FIGURA **11**

Il ciclo interno termina e le ultime quattro posizioni del vettore sono sicuramente ordinate. È stato effettuato almeno uno scambio, per cui si pone la variabile di stato s a 0 e si riapplica il confronto a coppie degli elementi dall'inizio del vettore fino alla posizione 0 (FIGURA **12**).

i	vet[i]	vet[i+1]	scambio	s
0	2	1	sì	**0**

FIGURA **12**

Il ciclo interno termina, ed essendo avvenuto l'ultimo scambio in posizione di indice 0, l'algoritmo termina (*s* è uguale a 0) e il vettore è ordinato.

OSSERVAZIONE Rispetto all'algoritmo non ottimizzato si può notare come, pur essendo stato operato lo stesso numero di scambi (e non poteva essere altrimenti), sia stata effettuata solo la metà dei confronti (10 contro 20).

2 Ricerca

Ricercare un elemento in un insieme di dati è un problema comune in molti aspetti delle attività umane, si pensi alla ricerca di una parola in un vocabolario, o di un nominativo in un elenco telefonico.
Molto dipende dalla struttura dati utilizzata per contenere gli elementi su cui si vuole condurre la ricerca: le modalità di accesso consentite su questa potranno influenzare in maniera determinante il tipo e la velocità degli algoritmi di ricerca utilizzabili. Anche intuitivamente si può affermare che la ricerca su una struttura ad accesso diretto sarà in genere più performante rispetto a una ricerca su una struttura ad accesso sequenziale.
Nei prossimi paragrafi analizzeremo alcune tecniche di ricerca applicabili ad *array* monodimensionali (quindi strutture ad accesso diretto), e nello specifico a vettori di numeri interi, senza per questo perdere di generalità nell'esposizione degli algoritmi che saranno presentati.

2.1 Ricerca completa

Se il nostro vettore non è ordinato, l'unica possibilità di verificare se un valore sia presente o meno in esso è quella di applicare un algoritmo di ricerca completa:

> si scandiscono sequenzialmente uno dopo l'altro gli elementi del vettore fino a quando non si incontra un valore uguale a quello ricercato (successo), oppure non si raggiunge la fine del vettore (insuccesso).

La seguente funzione C++ rappresenta una semplice implementazione dell'algoritmo ritornando il valore i in caso di elemento trovato nella posizione *i*-esima del vettore, o il valore −1 altrimenti. I parametri formali della funzione sono i seguenti:

vet[]	*Array* su cui si esegue la ricerca
N	Numero degli elementi del vettore interessati alla ricerca (sarà minore o uguale agli elementi dei vettore)
K	Valore da ricercare

```
int ricercaCompleta (int vet[], int N, int K)
{
  int i=0;

  do
    {
      if (vet[i] == K)
```

```
        return i;
      i++;
    } while(i < N)
  return -1; // NON è stato trovato un valore uguale a K
}
```

OSSERVAZIONE Una diversa implementazione dello stesso algoritmo è realizzabile utilizzando il ciclo `for` invece del ciclo `do-while`:

```
int ricerca Completa (int vet[], int N, int K)
{
  int i;

  for (i=0; i<N; i++)
    if (vet[i] == K)
      return i;
  return -1; // NON è stato trovato un valore uguale a K
}
```

In entrambi i casi la non esistenza del valore ricercato può essere dedotta solo dopo aver esaminato tutti gli elementi del vettore. Se stiamo trattando un vettore di notevoli dimensioni questo aspetto può essere «costoso», in particolare se l'algoritmo deve essere applicato in maniera ripetitiva per verificare l'esistenza o meno di valori diversi.
In queste circostanze avere il vettore ordinato può aiutare molto le operazioni di ricerca. Infatti la ricerca completa su un vettore ordinato permette di utilizzare un criterio più intelligente nella verifica dell'esistenza o meno di un dato elemento:

si scandiscono uno dopo l'altro gli elementi del vettore fino a quando è verificata la doppia condizione per cui questi sono minori o uguali al valore ricercato (nel caso di ordinamento crescente, maggiori o uguali altrimenti) e non si raggiunge la fine del vettore. Il procedimento termina con successo se si incontra un valore esattamente uguale a quello ricercato.

Supponendo di avere il vettore ordinato in maniera crescente, l'algoritmo descritto è implementato dalla seguente funzione C++:

```
int ricercaCompleta(int vet[], int N, int K)
{
  int i=0;

  while ((i<N) && (vet[i]<=K))
        {
          if (vet[i] == K)
            return i;
          i++;
        }
  return -1; // NON è stato trovato un valore uguale a K
}
```

Complessità della ricerca completa

Per la ricerca completa in un vettore il costo dipende dalla posizione dell'elemento cercato.
Caso migliore: l'elemento è il primo del vettore (un solo confronto); caso peggiore: l'elemento è l'ultimo o non è presente, in quest'ultimo caso l'istruzione fondamentale (il confronto) è eseguita N volte (dove N è la dimensione del vettore).
In media il test viene quindi eseguito

$$\frac{N+1)}{2}$$

volte.
Il costo è quindi lineare e la complessità è nell'ordine di N.

OSSERVAZIONE Si noti come in questo caso si sia optato per il costrutto `while` con la condizione composta `((i<N) && (vet[i]<=K))` che fa avanzare il ciclo solo nel caso non si sia ancora raggiunta la fine del vettore ma che, al tempo stesso, l'elemento corrente del vettore sia minore o uguale del valore ricercato *K*. In altre parole, supponendo *K* uguale a 25 se `vet[i]` è uguale a 30, essendo il vettore ordinato da questo punto in poi, di sicuro non potrà esservi un elemento uguale a *K*.

2.2 Ricerca binaria (o dicotomica)

Se dovessimo cercare una parola su un dizionario, sapendo che questo è ordinato alfabeticamente, si potrebbe pensare di iniziare la ricerca non dal primo elemento, ma da quello centrale, cioè a metà del dizionario. A questo punto il valore ricercato viene confrontato con il valore dell'elemento preso in esame e se sono uguali la ricerca termina con successo, altrimenti si sceglie la metà del dizionario su cui continuare la ricerca (la prima metà se la parola cercata è minore di quella centrale, la seconda metà altrimenti) e si riapplica il procedimento descritto. Quando la parte del dizionario da scegliere per proseguire coincide con una singola parola, l'algoritmo termina perché l'elemento cercato non è presente.

Nel caso in cui si abbia il vettore ordinato in maniera crescente, è possibile utilizzare un algoritmo di ricerca simile a quello utilizzabile per il dizionario:

si calcola la posizione dell'elemento centrale *M* del vettore partizionando i rimanenti elementi in due sottoinsiemi ordinati *A* (con elementi tutti minori di *M*) e *B* (con elementi tutti maggiori di *M*). Se l'elemento cercato *K* è uguale a *M*, l'algoritmo termina con successo; in caso contrario, se $K < M$ si riapplica il procedimento al sottoinsieme *A*, altrimenti al sottoinsieme *B*. Se non è possibile determinare un elemento centrale l'algoritmo termina con insuccesso, senza che *K* sia stato trovato.

Il procedimento descritto non è difficile da implementare in C++, come mostrato di seguito:

```
int ricercaBinaria(int vet[], int N, int K)
{
int inizio, fine, centro;

inizio = 0;
fine = N-1;
while (inizio <= fine)
```

Complessità della ricerca binaria

Come per la ricerca completa, il costo dipende dalla posizione dell'elemento cercato.
Ma sarà più semplice trovarlo!
Nel caso migliore l'elemento cercato è il primo preso in esame nel vettore (quello mediano): in questo caso si fa un solo confronto. Nel caso peggiore l'elemento cercato non è nel vettore e il ciclo viene ripetuto finché l'intervallo di ricerca non è vuoto.
Poiché a ogni passo il vettore si dimezza, si hanno al più C passi (con C proporzionale a $\log_2 N$).
La complessità di questo algoritmo è dunque nell'ordine di $\log_2 N$.

```
    {
      centro = (inizio + fine )/2; // elemento centrale
      if (vet[centro] == K)
        return centro; // valore K trovato in posizione centro
        if (vet[centro] < K)
          inizio = centro + 1;
        else
          fine = centro - 1;
      }
  // se l'esecuzione arriva a questo punto significa che il valore K
  // NON è presente nel vettore:
return -1;
}
```

Nella funzione C++ illustrata le nuove variabili utilizzate hanno il seguente significato:

inizio	Indice del primo elemento dell'intervallo corrente di ricerca
fine	Indice dell'ultimo elemento dell'intervallo corrente di ricerca
centro	Indice dell'elemento centrale dell'intervallo di ricerca corrente

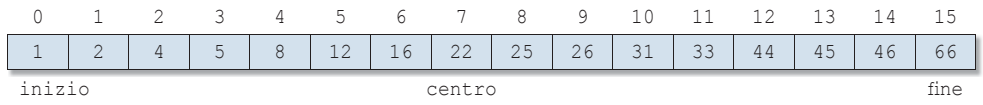
OSSERVAZIONE Si noti come, nel caso l'elemento cercato non sia presente nel vettore, procedendo nella scelta successiva dei vari intervalli di ricerca ci si trova a un certo punto di fronte a un intervallo di ricerca con `fine < inizio`, che indica la fine dell'algoritmo senza successo. La potenza di questo algoritmo è apprezzabile se la cardinalità dell'insieme su cui viene effettuata la ricerca è rilevante. Infatti, dividendo per due l'insieme di ricerca a ogni passo successivo dell'algoritmo, si riduce il numero di elementi da controllare in maniera logaritmica: la prima volta si dimezza il numero degli elementi, la seconda volta lo si riduce a un quarto, la terza a un ottavo e così via.
In pratica si può verificare che applicando questo procedimento a un insieme di 1024 elementi, in al più 10 passi si riesce a sapere se l'elemento cercato è presente o meno ($\log_2 1024 = 10$), e applicandolo a un insieme con 1 048 576 elementi sono sufficienti al più 20 passi ($\log_2 1\,048\,576 = 20$).

ESEMPIO

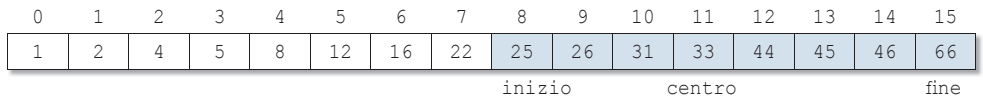
Supponendo di cercare il valore $K = 32$ nel seguente vettore di 16 elementi:

1	2	4	5	8	12	16	22	25	26	31	33	44	45	46	66

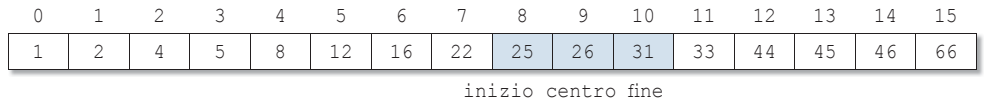
vediamo come lavora l'algoritmo di ricerca binaria:



vet[7] < *K* per cui

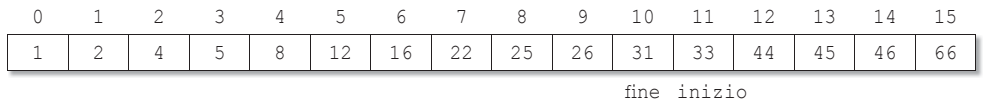


vet[11] > *K* per cui

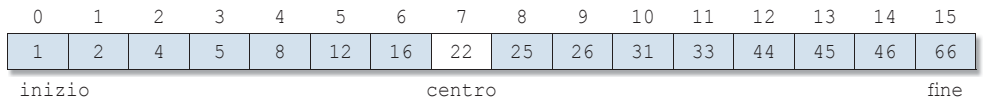


vet[9] < *K* per cui

vet[10] < *K* per cui



A questo punto si ha che *inizio*>*fine* per cui l'algoritmo termina senza aver trovato il valore *K* dopo il confronto di questo valore con soli 4 elementi del vettore (si noti che $\log_2 16 = 4$).
Nel caso in cui il valore da ricercare fosse stato il 22, anziché il 31, l'algoritmo si sarebbe arrestato dopo solo un'iterazione, in quanto tale valore è esattamente al centro del vettore originale.



Questa tanto fortunata quanto casuale situazione non altera il fatto per cui nel caso medio occorrono comunque $\log_2 N$ (con *N* dimensione del vettore) iterazioni per ricercare un elemento nel vettore.

Sintesi

Ordinamento. Algoritmi applicabili a insiemi di dati (in particolare *array*) per ottenere un ordinamento (crescente o decrescente) degli elementi contenuti in essi. Diversi tipi di ordinamento si basano su strategie che utilizzando operazioni di confronto e scambio degli elementi producono l'ordinamento voluto. Ordinare un vettore significa perdere la posizione iniziale degli elementi dell'insieme a cui viene applicato, a meno di non produrre come output un insieme diverso da quello fornito in input.

Exchange-sort. Prevede una strategia di ordinamento che, a partire dal primo fino al penultimo, fissa un elemento alla volta e lo confronta con tutti i successivi, effettuando uno scambio se l'elemento fissato e l'elemento di confronto non sono ordinati secondo il criterio prescelto (crescente o decrescente). Il numero dei confronti effettuati da questo algoritmo è fisso ed è funzione del numero degli elementi del vettore, piuttosto che della distribuzione iniziale degli elementi dell'insieme. In caso di ordinamento crescente, ogni volta che termina il confronto dell'elemento fisso con tutti i rimanenti, questo conterrà il valore minore tra tutti quelli esaminati: alla fine della prima scansione nel primo elemento si troverà il valore più piccolo, alla fine della seconda i primi due elementi saranno già ordinati, e così via.

Bubble-sort. La strategia di ordinamento prevede il confronto degli elementi dell'insieme da ordinare a coppie, ognuno con il successivo: se la coppia non è ordinata secondo il criterio prescelto (crescente o decrescente), si effettua lo scambio e viene annotato l'avvenuto scambio. Se al termine dei confronti si è effettuato almeno uno scambio, si riparte con il confronto di tutti gli elementi come descritto in precedenza, altrimenti il vettore è ordinato. Questo algoritmo è in grado di «accorgersi» in un'unica scansione se il vettore è già ordinato (se applicato a un insieme già ordinato di cardinalità N esegue esattamente $N - 1$ confronti). Nel caso di ordinamento crescente tende a far affondare gli elementi più pesanti: dopo la prima scansione l'elemento più grande occuperà sicuramente l'ultima posizione, dopo la seconda l'elemento più grande dei rimanenti occuperà sicuramente la penultima posizione, e così via.

Ricerca. Algoritmi applicabili a strutture dati per verificare l'esistenza o meno al loro interno di un elemento uguale a un valore dato. La ricerca può essere solo sequenziale e completa, se non è definito alcun ordinamento sugli elementi dell'insieme. In generale un insieme ordinato permette l'applicazione di algoritmi di ricerca che migliorano sensibilmente la velocità dell'operazione.

Ricerca binaria. Algoritmo applicabile solo a insiemi ad accesso diretto di dati ordinati. La strategia utilizzata calcola la posizione dell'elemento centrale M dell'insieme partizionando i rimanenti elementi in due sottoinsiemi ordinati A (con elementi tutti minori di M) e B (con elementi tutti maggiori di M). Se l'elemento ricercato K è uguale a M, l'algoritmo termina con successo; in caso contrario, se $K < M$, si riapplica il procedimento al sottoinsieme A, altrimenti al sottoinsieme B. Se non è più possibile determinare un elemento centrale (intervallo di ricerca vuoto) l'algoritmo termina con insuccesso, senza che K sia stato trovato. Applicato a un insieme di cardinalità N, completa la ricerca al massimo in $\log_2 N$ passi (per esempio, con un insieme di 1024 elementi, in al più 10 passi, con uno di 1 048 576 elementi sono sufficienti al più 20 passi).

QUESITI

1 Dato un vettore di 10 elementi interi, quali delle seguenti affermazioni sono corrette?

A Utilizzando *bubble-sort* si fanno esattamente 90 confronti.

B Utilizzando *bubble-sort* si fanno esattamente 90 scambi.

C Utilizzando *exchange-sort* si fanno esattamente 90 confronti.

D Utilizzando *exchange-sort* si fanno esattamente 100 scambi.

2 Dato un vettore di *N* elementi interi, quali delle seguenti affermazioni sono corrette?

A Alla fine del primo ciclo di un *bubble-sort* decrescente il primo elemento del vettore conterrà sicuramente il valore più grande

B Alla fine del primo ciclo di un *bubble-sort* decrescente l'ultimo elemento del vettore conterrà sicuramente il valore più piccolo.

C Alla fine del primo ciclo di un *exchange-sort* decrescente il primo elemento del vettore conterrà sicuramente il valore più grande.

D Alla fine del primo ciclo di un *exchange-sort* crescente l'ultimo elemento del vettore conterrà sicuramente il valore più grande.

3 Dato il seguente vettore a cui viene applicato un algoritmo *exchange-sort*:

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

dopo aver completato il passo di confronto dell'elemento in posizione 2 con tutti i successivi la situazione sarà:

A

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

B

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

C

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

D

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

4 È dato il seguente vettore ordinato a cui viene applicata una ricerca binaria per cercare in quale posizione si trovi il valore *K* = 33:

0	1	3	8	10	22	23	24	30	32	41	50

Indicare quale delle seguenti situazioni è verificata dopo aver terminato il secondo passo dell'algoritmo:

A inizio = 0, fine = 11, centro = 5.

B inizio = 0, fine = 11, centro = 7.

C inizio = 6, fine = 11, centro = 8.

D inizio = 9, fine = 11, centro = 10.

5 È dato il seguente vettore ordinato a cui viene applicata una ricerca binaria per cercare in quale posizione si trovi un valore *K* = 33:

0	1	3	8	10	22	23	24	30	32	41	50

Indicare quale delle seguenti situazioni è verificata al termine dell'algoritmo:

A inizio = 0, fine = 11.

B inizio = 8, fine = 8.

C inizio = 6, fine = 4.

D inizio = 10, fine = 9.

6 È dato il seguente vettore ordinato a cui viene applicata una ricerca binaria per cercare in quale posizione si trovi un valore *K* = 23:

0	1	3	8	10	22	23	24	30	32	41	50

Indicare quale delle seguenti situazioni è verificata al termine dell'algoritmo:

A inizio = 0, fine = 11.

B inizio = 6, fine = 7.

C inizio = 6, fine = 4.

D inizio = 10, fine = 9.

ESERCIZI

1 Scrivere un programma C++ che, dopo avere inserito tutti gli *N* elementi interi di un vettore (eventualmente anche in maniera casuale), richieda all'utente il tipo di ordinamento da effettuare

(crescente o decrescente) e visualizzi quindi gli elementi in accordo con quanto specificato.

2 Scrivere un programma C++ che, dopo avere inserito tutti gli *N* elementi interi di un vettore (eventualmente anche in maniera casuale), determini il valore massimo e quello minimo dopo aver ordinato il vettore. Si confronti e si commenti poi la soluzione rispetto al caso in cui il vettore non sia ordinato.

3 Scrivere un programma C++ che, dopo avere inserito tutti gli *N* elementi interi di un vettore (eventualmente anche in maniera casuale), determini e visualizzi il numero di scambi e il numero di confronti necessari per il suo ordinamento se a esso viene applicato il metodo *bubble-sort* o il metodo *exchange-sort*. (Nota: ci si assicuri di partire dalla stessa configurazione iniziale per entrambi i casi).

4 Dati due vettori *A* e *B* rispettivamente di *N* e *M* elementi, si vuole verificare quanti sono gli elementi di *A* presenti anche in *B* (si supponga che all'interno dei singoli vettori non vi siano duplicazioni). Scrivere un programma C++ che risolva il problema. La situazione migliora se *B* è ordinato? E se sia *A* sia *B* sono entrambi ordinati?

5 Si vogliono organizzare i dati relativi ai risultati di una corsa campestre sulla distanza dei 1000 metri. Gli iscritti sono meno di 100 e si intende memorizzarne il numero di gara, il nominativo, il sesso (M/F), l'anno di nascita e il tempo in secondi impiegato per portare a termine la prova. Realizzare un programma C++ che, una volta definita un'opportuna struttura per memorizzare i dati, provveda a visualizzare la classifica finale mostrando nell'ordine: numero di gara, nominativo, sesso, anno di nascita e tempo impiegato. (Nota: si faccia attenzione alla definizione della funzione di scambio degli elementi della tabella nell'eventuale algoritmo di ordinamento utilizzato).

6 Scrivere una funzione C++ che ricerchi i valori contenuti in un vettore all'interno di un secondo vettore non ordinato: la funzione dovrà semplicemente restituire un valore booleano «vero» se tutti i valori sono presenti, «falso» se anche solo uno dei valori del primo vettore non è presente nel secondo. Commentare la soluzione implementata dal punto di vista della complessità in termini di numero di confronti necessario. È migliorabile sotto questo aspetto?

7 Scrivere una funzione C++ che ordini i valori di una matrice quadrata numerica di ordine *N* posizionando il valore minore nell'elemento di coordinate (0, 0) e i successivi in sequenza crescente per riga e per righe successive, fino a posizionare il valore maggiore nell'elemento di coordinate (*N* – 1, *N* – 1).

8 Scrivere una funzione C++ che ordini i valori di una matrice quadrata numerica di ordine *N* posizionando il valore minore nell'elemento di coordinate (0, 0) e i successivi in sequenza crescente per colonna e per colonne successive, fino a posizionare il valore maggiore nell'elemento di coordinate (*N* – 1, *N* – 1).

9 È data la seguente struttura:

```
struct recUtente
{
  char nominativo [32];
  char telefono[16];
};
```

Scrivere un programma C++ che, definita una tabella di 100 utenti del tipo `recUtente`, ne esegua l'ordinamento in funzione del nominativo ed effettui quindi la ricerca binaria di un nominativo fornito da tastiera visualizzando, se trovato, il numero di telefono altrimenti il messaggio «Utente non trovato». In ogni caso il programma deve visualizzare il numero di elementi esaminati prima di pervenire alla visualizzazione del messaggio finale.

10 Relativamente all'esercizio precedente, con l'ordinamento definito sul nominativo, in quali modi è possibile effettuare la ricerca di un utente fornendo il suo numero di telefono da tastiera? Realizzare un programma C++ che permetta di effettuare la ricerca binaria sulla tabella in funzione del numero telefonico.

sorting algorithm
algoritmo di ordinamento

warehouse
magazzino

nevertheless
tuttavia

gain
guadagno

item
elemento

in situ
sul posto

key
chiave di ordinamento (per gli array il valore degli elementi stessi)

i.e.
idem est ovvero "vale a dire"

2 Sorting

2.1 Introduction

Sorting is generally understood to be the process of rearranging a given set of objects in a specific order. The purpose of sorting is to facilitate the later search for members of the sorted set. As such it is an almost universally performed, fundamental activity. Objects are sorted in telephone books, in income tax files, in tables of contents, in libraries, in dictionaries, in warehouses, and almost everywhere that stored objects have to be searched and retrieved. Even small children are taught to put their things "in order", and they are confronted with some sort of sorting long before they learn anything about arithmetic.

Hence, sorting is a relevant and essential activity, particularly in data processing. What else would be easier to sort than data! Nevertheless, our primary interest in sorting is devoted to the even more fundamental techniques used in the construction of algorithms. There are not many techniques that do not occur somewhere in connection with sorting algorithms. In particular, sorting is an ideal subject to demonstrate a great diversity of algorithms, all having the same purpose, many of them being optimal in some sense, and most of them having advantages over others. It is therefore an ideal subject to demonstrate the necessity of performance analysis of algorithms. The example of sorting is moreover well suited for showing how a very significant gain in performance may be obtained by the development of sophisticated algorithms when obvious methods are readily available.

The dependence of the choice of an algorithm on the structure of the data to be processed – an ubiquitous phenomenon – is so profound in the case of sorting that sorting methods are generally classified into two categories, namely, sorting of arrays and sorting of (sequential) files. The two classes are often called *internal* and *external sorting* because arrays are stored in the fast, high-speed, random-access "internal" store of computers and files are appropriate on the slower, but more spacious "external" stores based on mechanically moving devices (disks and tapes). […]

If we are given n items a_0, a_1, ..., a_{n-1} sorting consists of permuting these items into an array a_{k0}, a_{k1}, ... , $a_{k[n-1]}$ such that, given an ordering function f, $f(a_{k0}) \leq f(a_{k1}) \ldots \leq f(a_{k[n-1]})$

Ordinarily, the ordering function is not evaluated according to a specified rule of computation but is stored as an explicit component (field) of each item. Its value is called the *key* of the item. The other components represent relevant data about the items in the collection; the *key* merely assumes the purpose of identifying the items. As far as our sorting algorithms are concerned, however, the *key* is the only relevant component, and there is no need to define any particular remaining components. In the following discussions, we shall therefore discard any associated information and assume that the type *Item* be defined as INTEGER. This choice of the *key* type is somewhat arbitrary. Evidently, any type on which a total ordering relation is defined could be used just as well.

A sorting method is called *stable* if the relative order of items with equal *keys* remains unchanged by the sorting process. Stability of sorting is often desirable, if items are already ordered (sorted) according to some secondary *keys*, i.e., properties not reflected by the (primary) *key* itself. […]

2.2 Sorting Arrays

The predominant requirement that has to be made for sorting methods in arrays is an economical use of the available store. This implies that the permutation of items which brings the items into order has to be performed *in situ*, and that methods which transport items from an array *a* to a result array *b* are intrinsically of minor interest. […] A good measure of efficiency is obtained by counting the numbers *C* of needed *key* comparisons and *M* of moves (transpositions) of items. These numbers are functions of the number *n* of items to be sorted. Whereas good sorting algorithms require in the order of $n^*\log(n)$ comparisons, we first discuss several simple and obvious sorting techniques, called *straight methods*, all

of which require in the order of n^2 comparisons of *keys*. There are three good reasons for presenting straight methods before proceeding to the faster algorithms.

1 Straight methods are particularly well suited for elucidating the characteristics of the major sorting principles.
2 Their programs are easy to understand and are short. Remember that programs occupy storage as well!
3 Although sophisticated methods require fewer operations, these operations are usually more complex in their details; consequently, straight methods are faster for sufficiently small n, although they must not be used for large n.

Sorting methods that sort items *in situ* can be classified into three principal categories according to their underlying method:
Sorting by insertion
Sorting by selection
Sorting by exchange
[…] The procedures operate on a global variable *a* whose components are to be sorted *in situ*, i.e. without requiring additional, temporary storage. The components are the keys themselves. […] we will assume the presence of an array *a* and a constant *n*, the number of elements of *a* […]

2.2.1 Sorting by Straight Insertion

This method is widely used by card players. The items (cards) are conceptually divided into a destination sequence $a_1 \ldots a_{i-1}$ and a source sequence $a_i \ldots a_n$. In each step, starting with $i = 2$ and incrementing i by unity, the i th element of the source sequence is picked and transferred into the destination sequence by inserting it at the appropriate place.[…]

2.2.2 Sorting by Straight Selection

This method is based on the following principle:

1 Select the item with the least *key*.
2 Exchange it with the first item a_0.
3 Then repeat these operations with the remaining $n - 1$ items, then with $n - 2$ items, until only one item – the largest – is left.[…]

2.2.3 Sorting by Straight Exchange

The classification of a sorting method is seldom entirely clear-cut. Both previously discussed methods can also be viewed as exchange sorts. In this section, however, we present a method in which the exchange of two items is the dominant characteristic of the process. The subsequent algorithm of straight exchanging is based on the principle of comparing and exchanging pairs of adjacent items until all items are sorted. As in the previous methods of straight selection, we make repeated passes over the array, each time sifting the least item of the remaining set to the left end of the array. If, for a change, we view the array to be in a vertical instead of a horizontal position, and – with the help of some imagination – the items as bubbles in a water tank with weights according to their *keys*, then each pass over the array results in the ascension of a bubble to its appropriate level of weight […]. This method is widely known as the *Bubblesort*.

[N.Wirth, *Algorithms and Data Structures*, Prentice - Hall, 1985]

QUESTIONS

a What is a sorting algorithm?

b What is the difference between internal and external sorting?

c What is the predominant requirement of array sorting?

d Which parameters are used to evaluate the efficiency of an array sorting?

A9 La ricorsione

Come sappiamo il fattoriale di un numero n si esprime con la simbologia $n!$ ed è dato dal prodotto dei primi n numeri naturali. Si può anche osservare che

$$n! = n \cdot (n-1)!$$

ovvero che il fattoriale di un numero n è il prodotto tra n stesso e il fattoriale del numero precedente $n-1$. Ricordando che $0! = 1$ e $1! = 1$, applicando a ritroso il ragionamento appena illustrato possiamo impostare in maniera del tutto intuitiva una funzione per il calcolo di $n!$ (per $n \geq 0$):

```
long fattoriale(int n)
  {
    if (n<=1)
      return 1;
    else
      return n*fattoriale(n-1);
  }
```

Prima di entrare nel dettaglio della problematica osserviamo che, per come è formulata, la funzione *fattoriale* richiama sé stessa: questo meccanismo è alla base del concetto di *ricorsione*, oggetto del presente capitolo.

1 Induzione e ricorsione

Per calcolare la somma dei primi n numeri naturali si potrebbe utilizzare la seguente funzione che adotta un procedimento iterativo:

```
long sommaNum(int n)
{
  int i, s=0;

  for (i=1; i<=n; i++)
    s=s+i;
  return s;
}
```

oppure si potrebbe usare la formula:

$$s = \frac{n \cdot (n+1)}{2}$$

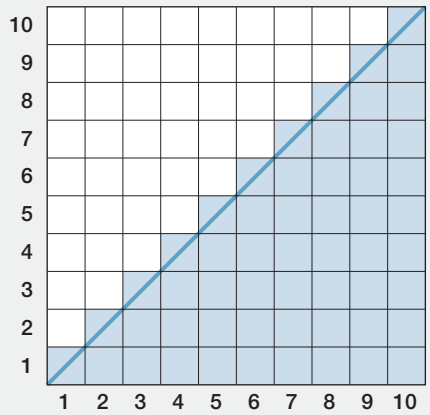


FIGURA 1

che il grande matematico Gauss, pare, scoprì da bambino, almeno per il caso $n = 10$, realizzando il disegno di FIGURA 1.
È interessante dimostrare rigorosamente come la formula precedente sia valida per ogni valore n; in matematica questo problema può essere risolto tramite una **dimostrazione per induzione**:

- ponendo $n = 1$ nella formula precedente si ha:

$$1 = \frac{1 \cdot (1 + 1)}{2}$$

  che banalmente è verificata (base dell'induzione);

- ora, supposta vera la formula per un valore n, proviamo che è vera anche per il valore $n + 1$, sostituendo a n proprio $n + 1$:

$$s = \frac{(n + 1) \cdot [(n + 1) + 1]}{2}$$

  da cui, svolgendo il prodotto:

$$s = \frac{n \cdot (n + 1) + n + (n + 1) + 1}{2}$$

  segue:

$$s = \frac{n \cdot (n + 1)}{2} + \frac{n + (n + 1) + 1}{2}$$

  Ma il primo termine non è altro che la formula della somma dei primi n numeri naturali, che avevamo supposto essere vera (ipotesi induttiva), a cui viene aggiunto il termine:

$$\frac{n + (n + 1) + 1}{2} = \frac{2n + 2}{2} = n + 1$$

  che altro non è che l'$(n + 1)$-esimo termine della somma.

Nel procedimento si è adottato un **ragionamento ricorsivo**: la somma dei primi $n + 1$ numeri naturali si ottiene aggiungendo $n + 1$ alla somma dei primi n numeri naturali.

Riassumendo: abbiamo verificato direttamente la formula per un valore di n, l'abbiamo supposta vera per n generico e quindi l'abbiamo verificata per $n + 1$ dimostrandone la validità generale.

Analizziamo ora quali sono le conseguenze di questo modo di ragionare nello sviluppo di una funzione. Come abbiamo già visto, una funzione può invocare altre funzioni; in generale, quindi, se si conviene che una funzione possa invocare una qualsiasi altra funzione il cui nome le sia accessibile, dobbiamo anche considerare la possibilità che questa possa auto-invocarsi.

In questo caso si parla di **programmazione ricorsiva** e l'azione di auto-invocazione è definita **ricorsione**.

Questo fatto può sembrare paradossale o di scarso rilievo pratico, ma in realtà non è così e, anzi, la ricorsione introduce spesso un livello di astrazione più vicino di quanto non si creda al nostro modo di pensare.

ESEMPIO

Nel caso della somma dei primi numeri naturali si potrebbe quindi utilizzare la seguente funzione ricorsiva, dove si è adottato il principio ricorsivo secondo cui «la somma dei primi n numeri è data da n più la somma dei primi $n - 1$ numeri naturali»:

```
long sommaNum(int n)
{
   if (n == 1)
     return n;
else
     return (n + sommaNum(n-1));
}
```

OSSERVAZIONE A ogni invocazione della funzione viene allocata un'area di memoria distinta per l'istanza corrente della funzione stessa, dove i valori delle variabili hanno una validità locale. Schematicamente, nell'esecuzione dell'esempio precedente, per $n = 4$ si ottiene la rappresentazione grafica di FIGURA **2**, dove:

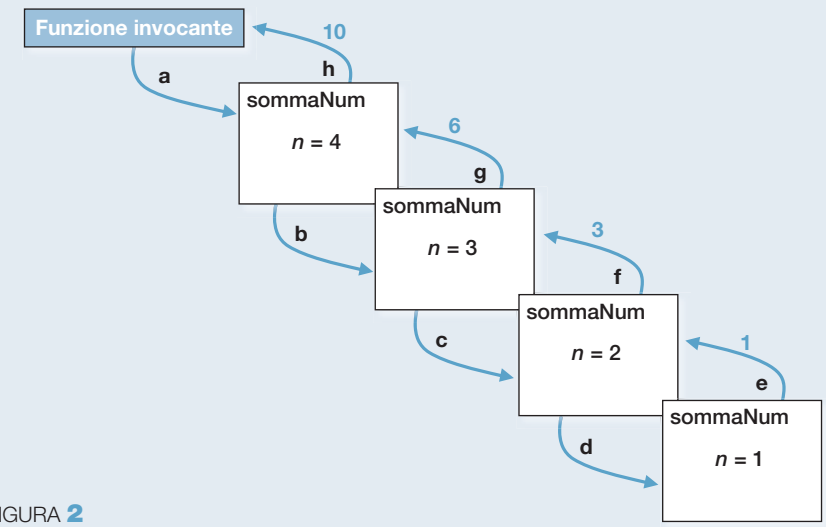


FIGURA **2**

- **a** rappresenta l'invocazione a *sommaNum* da parte di una funzione esterna;
- **b**, **c** e **d** rappresentano le invocazioni di *sommaNum* a sé stessa (ricorsione):
- **e**, **f**, **g** e **h** rappresentano il percorso inverso di de-allocazione degli spazi di memoria utilizzati durante la ricorsione (si noti come il risultato sia calcolato, passo per passo, durante il processo inverso di invocazione come somma tra il valore corrente di *n* e il valore restituito dall'invocazione effettuata nel passo precedente).

OSSERVAZIONE La funzione *sommaNum* verifica una condizione di terminazione delle invocazioni ricorsive, in questo caso il fatto che il numero *n* sia uguale a 1. Tutte le funzioni ricorsive devono prevedere la verifica di una condizione di terminazione, altrimenti si origina una sequenza potenzialmente infinita di invocazioni (in realtà la memoria disponibile è finita e quindi, anche se in modo catastrofico, la sequenza di invocazioni terminerà!).

2 Funzioni ricorsive

Prendiamo in esame l'algoritmo di Euclide per trovare il massimo comun divisore (MCD) tra due numeri qualsiasi *m* ed *n* (con $m > n$); a parole potrebbe essere espresso così:

per trovare MCD(*m*, *n*) si effettui la divisione fra *m* ed *n*; sia *r* il resto della divisione: se $r = 0$ allora *n* è il MCD, altrimenti il MCD(*m*, *n*) è uguale al MCD(*n*, *r*).

In effetti Euclide ha pensato e formulato questo algoritmo in versione «ricorsiva».

ESEMPIO

MCD(15, 9) = MCD(9, 6) = MCD(6, 3) = 3

Come si vede la ricorsione avviene su argomenti sempre più piccoli, fino a trovare il caso ($r = 0$) in cui non si ricorre più.

Possiamo facilmente programmare questa versione «ricorsiva» dell'algoritmo di Euclide:

```
int MCD(int m, int n)
{
  int r;
```

```
  r = m % n;
  if (r == 0)
    return n;
  else
    return MCD(n,r);
}
```

ESEMPIO

Il processo di esecuzione della funzione precedente per $m = 15$ ed $n = 9$ può essere rappresentato schematicamente come in FIGURA **3**, dove:

- **a** rappresenta l'invocazione di MCD da parte di una funzione esterna;
- **b** e **c** rappresentano le invocazioni di MCD a sé stessa (ricorsione);
- **d**, **e** ed **f** rappresentano il percorso inverso di de-allocazione degli spazi di memoria utilizzati durante la ricorsione (si noti come il risultato 3 venga propagato a ritroso da un'istanza all'altra della funzione MCD()).

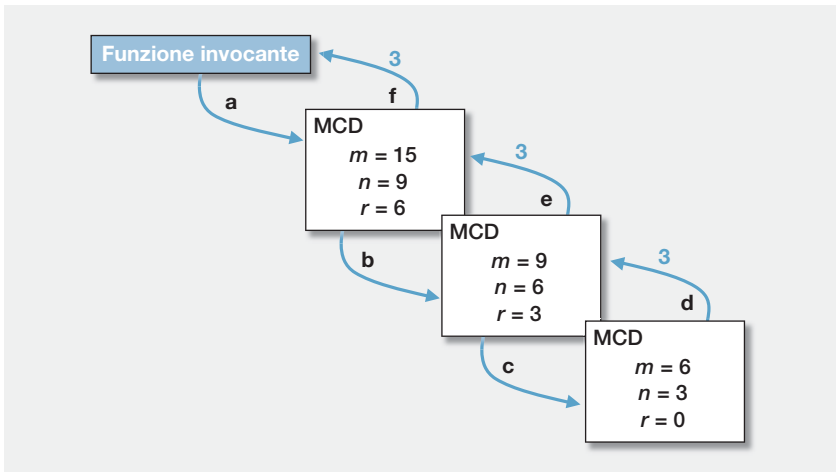


FIGURA **3**

OSSERVAZIONE Un aspetto caratterizzante degli algoritmi ricorsivi è l'assenza dei cicli, che ritroviamo invece nelle versioni iterative degli stessi algoritmi: una invocazione ricorsiva corrisponde in questo caso a una iterazione. Ed è proprio l'assenza di cicli espliciti che contribuisce a una maggiore leggibilità, eleganza e astrattezza dei programmi ricorsivi rispetto a quelli iterativi.
In teoria è possibile implementare qualsiasi algoritmo senza ricorrere a strutture iterative, ma solo in modo ricorsivo: esistono linguaggi di programmazione – in particolare i cosiddetti linguaggi «funzionali» – che privilegiano l'uso della ricorsione rispetto all'iterazione.

Anche se la ricorsione introduce un maggior grado di astrazione, non sempre è preferibile all'iterazione. A questo proposito si consideri la **successione numerica di Fibonacci**.

I numeri di Fibonacci sono gli elementi della successione così definita:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \quad (n \geq 2)$$

che genera la sequenza di numeri

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

in cui ogni elemento successivo al secondo è uguale alla somma dei due precedenti.
Si può senz'altro scrivere una funzione ricorsiva che restituisce l'n-esimo numero di Fibonacci F_n:

```
int F(int n)
{
  if (n == 0 || n == 1)
    return 1;
  else
    return (F(n-1) + F(n-2));
}
```

Questa funzione costituisce però un caso di abuso della ricorsione, perché genera un grande spreco di risorse. Infatti, per calcolare $F(n-1)$ occorre prima calcolare $F(n-2)$ che viene quindi calcolato due volte; analogamente si può verificare che $F(n-3)$ viene calcolato tre volte, $F(n-4)$ quattro volte e così via.
Questo inconveniente non si presenta nella soluzione iterativa, che appare comunque meno immediata alla lettura:

```
int F(int n)
{
  int i, ultimo, penultimo, t;

  if (n == 0 || n == 1)
    return 1;

  ultimo = 1;
  penultimo = 1;
  for (i=2; i<=n; i++)
    {
      t = ultimo;
      ultimo = ultimo + penultimo;
      penultimo = t;
    }
  return ultimo;
}
```

OSSERVAZIONE Anche escludendo i casi limite come la successione di Fibonacci, occorre dire che in generale un algoritmo ricorsivo richiede più memoria ed è più lento di una equivalente soluzione iterativa. Ciò è dovuto al meccanismo di allocazione (in fase di ricorsione) e de-allocazione della memoria (in fase di ritorno) che il procedimento stesso comporta. Questo rimane un dato di fatto, anche se le nuove generazioni di elaboratori rendono meno importanti questo tipo di considerazioni e, di conseguenza, non è sempre necessario sacrificare la chiarezza all'efficienza.

Come abbiamo visto in apertura, costruire un **algoritmo ricorsivo** per risolvere un dato problema corrisponde a dare una definizione per induzione della sua soluzione. Non a caso, due sono le condizioni necessarie, ma non sufficienti, affinché un algoritmo ricorsivo sia corretto (in pratica perché non cada in una sequenza infinita di invocazioni):

- deve esistere almeno una modalità di terminazione che evita l'invocazione ricorsiva (questa condizione corrisponde alla **base dell'induzione**);
- almeno una invocazione ricorsiva deve prevedere la soluzione di un sottoproblema ridotto rispetto all'originale (questa condizione corrisponde all'**ipotesi di induzione**).

Queste condizioni sono ovviamente verificate in tutti gli esempi proposti.

3 La ricorsione e gli *array*

Come abbiamo avuto modo di vedere nella trattazione degli *array*, l'elaborazione di questo tipo di dati strutturati implica un uso naturale dei cicli iterativi. Dato che mediante la ricorsione è possibile sostituire strutture iterative con invocazioni di una funzione a sé stessa, vediamo, per esempio, come è possibile risolvere ricorsivamente in due modi diversi il problema della visualizzazione del contenuto di un vettore di n elementi:

```
void visualizzaVettore(int v[], int n)
{
  if (n < 0)
    return;
  visualizzaVettore(v, n-1);
  cout<<v[n]<<endl;
}

void visualizzaVettore(int v[], int n)
{
  if (n < 0)
    return;
  cout<<v[n]<<endl;
  visualizzaVettore(v, n-1);
}
```

OSSERVAZIONE Dato che il punto di ritorno di una chiamata a funzione è costituito dall'istruzione immediatamente successiva alla chiamata, si avrà nel primo caso la visualizzazione degli elementi per valori crescenti dell'indice (prima viene sviluppata completamente la ricorsione quindi, nella fase del ritorno dalle invocazioni ricorsive, avviene la visualizzazione), mentre nel secondo caso la visualizzazione avviene durante il processo di ricorsione, per cui gli elementi sono visualizzati in senso inverso (dall'indice più grande al più piccolo).

ESEMPIO

Una tecnica usuale per l'implementazione di algoritmi ricorsivi che coinvolgono vettori o stringhe è quella di passare alla funzione ricorsiva due parametri interi come indici, uno superiore e l'altro inferiore, che delimitano gli elementi su cui operare. Per esempio la funzione:

```
bool palindroma(char s[], int a, int b)
{
  if (a > b)
    return true;
  if (s[a]!= s[b])
    return false;
  return palindroma(s,a+1,b-1);
}
```

determina se una stringa è palindroma o meno, confrontando a ogni invocazione i due caratteri estremi della stringa individuati dagli indici forniti come argomento; naturalmente nell'invocazione iniziale gli indici forniti come argomento dovranno specificare l'intera stringa:

```
palindroma("ANNA",0,3);
palindroma("MARIA",0,4);
```

ESEMPIO

Per quanto riguarda l'elaborazione di matrici, presentiamo la funzione che segue, il cui scopo è quello di determinare se una matrice quadrata $N \times N$ sia o meno simmetrica rispetto alla diagonale principale. Nel caso specifico è stato fatto riferimento a una matrice 4×4.

```
bool simmetrica(int matr[][4], int i, int j)
{
  //fine della ricorsione
  if (i==0) return true;

  // fine esame di una riga: si passa alla riga di indice inferiore
  if (j<0) return (simmetrica(matr, i-1, 0));

  // elemento sulla diagonale principale: si passa alla colonna di
  // indice inferiore
  if (i==j) return (simmetrica(matr, i, j-1));

  // elemento della triangolare inferiore da confrontare col
  // simmetrico della triangolare superiore
  return (matr[i][j]==matr[j][i]) && (simmetrica(matr, i, j-1));
}
```

L'elaborazione avviene confrontando gli elementi della triangolare inferiore (di indici `[i][j]`) con i corrispondenti della triangolare superiore (di indici `[j][i]`), a partire dalla riga con indice più grande a scalare fino a quella di indice 0.
La prima invocazione della funzione appena vista su una matrice Alfa 4×4 sarà effettuata specificando il valore massimo che *i* e *j* potranno assumere, ovvero con: `simmetrica(Alfa,3,3);`.

3.1 L'algoritmo di ordinamento *quick-sort*

Tra gli algoritmi di ordinamento più performanti dal punto di vista della velocità di esecuzione, presentiamo in questo paragrafo ***quick-sort***. Questo algoritmo, ideato da Tony Hoare nel 1961, fa uso della ricorsività adottando la seguente strategia basata sul paradigma *divide et impera*.

Complessità del *quick-sort*

Per *quick-sort* nel caso peggiore si ha una complessità per $N \to \infty$ pari a N^2, ma nel caso medio essa è pari a $N \cdot \log N$. Quindi *quick-sort* risulta essere in media molto più veloce degli algoritmi di ordinamento visti in precedenza.

Si sceglie un elemento ***pivot*** a caso (generalmente l'ultimo elemento dell'insieme) e si operano opportuni scambi in modo tale che tutti gli elementi più piccoli del *pivot* si trovino alla sua sinistra e quelli più grandi alla sua destra; applicando ricorsivamente il processo descritto (algoritmo di partizionamento) prima al sottoinsieme di sinistra e poi a quello di destra si perviene all'ordinamento del vettore.

Supponendo di operare su un vettore *v* formato da *n* elementi, la funzione *quickSort()* riportata nel seguito implementa l'algoritmo descritto per un ordinamento crescente. Le variabili *inf* e *sup* rappresentano rispettivamente il valore minimo e massimo degli indici relativi agli estremi della porzione di *array* in fase di ordinamento (alla prima invocazione si ha $inf = 0$ e $sup = n - 1$), la variabile *p* assume il valore dell'elemento scelto di volta in volta come pivot, *i* ed *s* sono indici che servono a scorrere gli elementi del vettore per effettuare la ripartizione dei valori del vettore (rispettivamente minori di *p* a sinistra e maggiori a destra):

```
void quickSort(int v[], int inf, int sup)
{
  int s, i, p, t;

  if (inf < sup)
    {
      i = inf;
      s = sup;
      p = v[sup];
      do
        {
        while (i<s && v[i]<=p)
          i++;
        while (s>i && v[s]>=p)
          s--;
        if (i < s)
          {
            t = v[i];
            v[i] = v[s];
            v[s] = t;
          }
        } while (i<s);
      t = v[i];
      v[i] = v[sup];
```

▶

```
        v[sup] = t;
        quickSort(v, inf, i-1); // ricorsione sull'insieme
                                // sinistro
        quickSort(v, i+1, sup); // ricorsione sull'insieme
                                // destro
    }
}
```

4 Un esempio di strategia risolutiva ricorsiva: il gioco della «Torre di Hanoi»

Narra la leggenda che all'inizio dei tempi Brahma portò nel grande tempio di Benares, sotto la cupola d'oro che si trova al centro del mondo, tre colonnine di diamante e sessantaquattro dischi d'oro, collocati su una di queste colonnine in ordine decrescente, dal più piccolo in alto, al più grande in basso. È la sacra Torre di Brahma che vede impegnati, giorno e notte, i sacerdoti del tempio nel trasferimento della torre di dischi dalla prima alla terza colonnina. Essi non devono contravvenire alle regole precise, imposte da Brahma stesso, che richiedono di spostare soltanto un disco alla volta e che non ci sia mai un disco sopra uno più piccolo. Quando i sacerdoti avranno completato il loro lavoro e tutti i dischi saranno riordinati sulla terza colonnina, la torre e il tempio crolleranno e sarà la fine del mondo. Se calcoliamo il numero dei movimenti necessari per spostare i dischi otteniamo 18 446 744 073 551 615 ($2^{64} - 1$). Nel caso in cui i sacerdoti impieghino un secondo per ogni movimento, ci vorranno più di cinque miliardi di secoli per spostare tutti i dischi da una colonnina all'altra. Possiamo quindi stare tranquilli, almeno da questo punto di vista, per il nostro futuro. Il gioco in realtà venne inventato nel 1883 da Edouard Lucas, studioso di teoria dei numeri e professore in un liceo di Parigi, che deve la sua fama all'analisi della successione di Fibonacci.

4.1 Regole e algoritmo

Il gioco inizia con tutti i dischi collocati in un solo piolo, ordinati in modo che in basso ci sia il disco più grande e in alto quello più piccolo. Si deve spostare tutta la pila di dischi in un diverso piolo muovendo un disco alla volta e senza mai collocare un disco più grande sopra uno più piccolo.
Nella FIGURA **4** i dischi appaiono inseriti sul piolo 1: si supponga che questi debbano essere spostati sul piolo 2. Supponiamo inoltre che tutti i dischi, meno l'ultimo, possano essere spostati in qualche modo corretto, dal piolo 1 al piolo 3, come nella situazione della FIGURA **5**.
Arrivati a questo punto si può spostare l'ultimo disco rimasto (l'*n*-esimo) dal piolo 1 al piolo 2; quindi, come in precedenza, si supponga di spostare in qualche modo il gruppo di dischi posizionati attualmente nel piolo

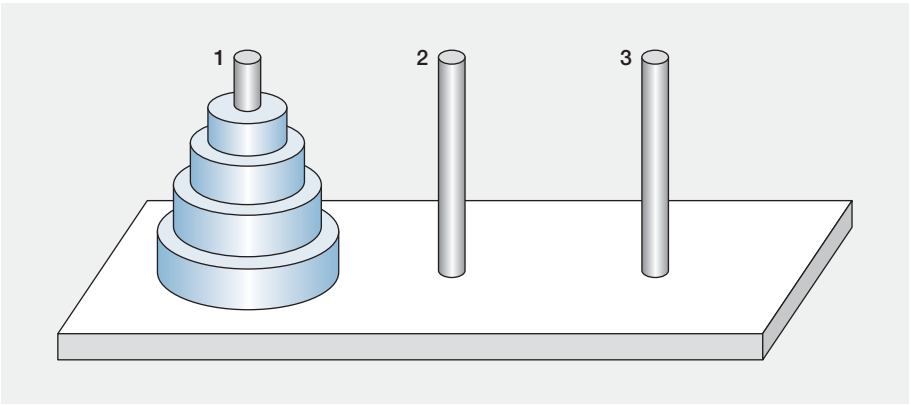


FIGURA **4**

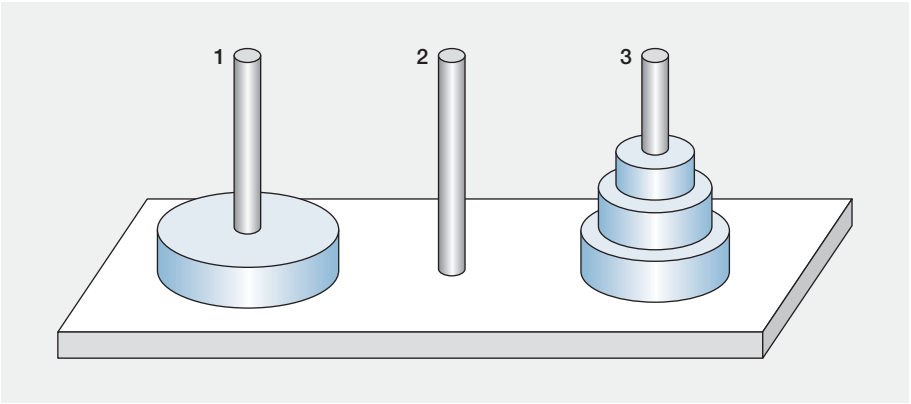


FIGURA **5**

3 fino al piolo 2, sopra il disco più grande. Operando in questo modo, l'algoritmo risolutivo del problema risulta essere ricorsivo e implementabile mediante un'unica funzione cha avrà la seguente struttura:

```
void hanoi(int n, int p1, int p2)
{
  if (n>0)
    {
      hanoi(n-1, p1, 6-p1-p2); // muove disco n da piolo p1
                               // a piolo p2
      hanoi(n-1, 6-p1-p2, p2);
    }
}
```

dove:

n	è la dimensione della torre espressa come numero di dischi numerati da 1 a *n*
*p*1	è il numero del piolo su cui si trova inizialmente la pila di *n* dischi
*p*2	è il numero del piolo su cui deve essere spostata la pila di dischi
6-*p*1-*p*2	è il numero del terzo piolo (questa formula determina il numero del terzo piolo se questi sono numerati da 1 a 3)

Se il numero n dei dischi da spostare è minore di 1, non si deve compiere alcuna azione. Se n è uguale a 1, le istruzioni interne alla struttura condizionale sono eseguite, ma nessuna delle invocazioni ricorsive ha luogo, dato che $n - 1$ è zero. In questo caso, supponendo $n = 1$, $p1 = 1$ e $p2 = 2$, il risultato è semplicemente quello di muovere il disco 1 dal piolo 1 al piolo 2. Il risultato è quindi corretto per una pila iniziale consistente di un solo disco.
Se n è uguale a 2, la prima invocazione ricorsiva sposta un disco ($n - 1 = 1$) dal piolo 1 al piolo 3 (ancora assumendo che i due dischi debbano essere spostati dal primo al terzo piolo) e si sa che questa è la mossa corretta. Quindi viene effettuato lo spostamento del secondo disco (l'n-esimo) dalla posizione 1 alla posizione 2; infine la seconda chiamata ricorsiva si occupa di spostare il disco collocato precedentemente nel terzo piolo sul secondo, sopra a quello che si trova già nella posizione finale corretta. In pratica, nel caso di due dischi che devono essere spostati dal primo al secondo piolo, vengono eseguite le tre operazioni seguenti:

- spostamento del disco 1 dal piolo 1 al piolo 3;
- spostamento del disco 2 dal piolo 1 al piolo 2;
- spostamento del disco 1 dal piolo 3 al piolo 2.

Allo stesso modo si potrebbe dimostrare il corretto funzionamento per un numero maggiore di dischi.
Il seguente programma C++ implementa l'algoritmo della torre di Hanoi simulando i pioli e i dischi tramite una matrice di N righe (corrispondenti a N dischi) e 3 colonne.

```
const int N = ...; // numero di dischi

// visualizza la matrice
void visualizza(int torre[][3])
{
  int i,j;
  char c;
  cout<<endl;
  for (i=0; i<N; i++)
    {
      for (j=0; j<3; j++)
        cout<<torre[i][j]<<'\t';
      cout<<endl;
    }
  cin>>c; // attesa input utente per avanzamento
}

// determina la posizione del disco da spostare
int posizioneDisco(int torre[][3], int piolo)
{
  int i;
```

▶

```
  for (i=0; i<N; i++)
    if (torre[i][piolo] != 0)
      return i;
  return i;
}

// determina la destinazione del disco da spostare
int destinazioneDisco(int torre[][3], int piolo)
{
  int i;

  for (i=N-1; i>=0; i--)
    if (torre[i][piolo] == 0)
      return i;
  return i;
}

// funzione ricorsiva
void hanoi(int torre[][3], int n, int p1, int p2)
{
  int disco, posizione, destinazione;

  if (n>0)
    {
      hanoi(torre, n-1, p1, 6-p1-p2); // invocazione ricorsiva
      visualizza(torre);
      cout<<"disco "<<n<<" da piolo "<<p1<<"
        a piolo "<<p2<<endl;
      posizione = posizioneDisco(torre, p1-1);
      disco = torre[posizione][p1-1];
      torre[posizione][p1-1] = 0; // rimozione disco
      destinazione = destinazioneDisco(torre, p2-1);
      torre[destinazione][p2-1] = disco; // posizionamento disco
      hanoi(torre, n-1, 6-p1-p2, p2); // invocazione ricorsiva
    }
}

void main(void)
{
   int torre[N][3] = {0};

   // posiziona N dischi sul piolo 1
   for (int i=0; i<N; i++)
     torre[i][0] = i+1;
   hanoi(torre, N, 1, 3); // sposta N dischi dal piolo 1
                          // al piolo 3
   visualizza(torre);
}
```

Sintesi

Principio di induzione. È un principio utilizzato per verificare la validità di una proposizione che afferma che qualcosa è vera per ogni numero naturale. Esso si articola nei seguenti tre passi:

- verifica che la proposizione sia vera per $n = 1$;
- supposizione che la proposizione sia vera per n;
- verifica che la proposizione sia vera per $n + 1$.

Se questi punti sono verificati allora la proposizione è valida per ogni numero naturale.

Ricorsione. Nella ricorsione una funzione richiama più volte sé stessa in un processo che per costruzione termina dopo un numero finito di volte. Fatto caratterizzante degli algoritmi ricorsivi è l'assenza dei cicli tipici delle versioni iterative degli stessi algoritmi (una chiamata ricorsiva corrisponde in questo caso a una iterazione). Ed è proprio l'assenza di cicli espliciti che contribuisce a una maggiore leggibilità, eleganza e astrattezza dei programmi ricorsivi rispetto a quelli iterativi. Costruire una soluzione ricorsiva per un dato problema equivale a dare una definizione per induzione della sua soluzione.

Applicabilità. In teoria una soluzione ricorsiva può sostituire una qualsiasi soluzione iterativa, tuttavia esistono tipologie di problemi che per loro natura non si prestano a questo tipo di approccio: per esempio, nel calcolo della successione di Fibonacci l'algoritmo ricorsivo dà luogo a un grande spreco di risorse, dal momento che a ogni passo per il calcolo di un nuovo termine è necessario effettuare una doppia chiamata ricorsiva per il calcolo dei due termini che lo precedono.

Condizioni. Due sono le condizioni necessarie, ma non sufficienti, affinché un algoritmo ricorsivo sia corretto (in pratica perché non cada in una sequenza infinita di invocazioni):

- deve esistere almeno una modalità di terminazione che evita l'invocazione ricorsiva (questa condizione corrisponde alla **base dell'induzione**);
- almeno una invocazione ricorsiva deve prevedere la soluzione di un sottoproblema ridotto rispetto all'originale (questa condizione corrisponde all'**ipotesi di induzione**).

Vantaggi e svantaggi. La ricorsione permette di scrivere poche linee di codice per risolvere problemi anche molto complessi. Eventuali svantaggi sono da ricercare nelle prestazioni: infatti una funzione ricorsiva, a seconda dei casi, occupa memoria per un numero di istanze della funzione pari alle chiamate necessarie per risolvere il problema. In questo senso funzioni che allocano una grande quantità di spazio in memoria, pur potendo essere implementate ricorsivamente, potrebbero creare problemi in fase di esecuzione. Pertanto, se le prestazioni sono importanti e/o non si dispone di memoria in quantità adeguata, è sconsigliato l'utilizzo della ricorsione.

QUESITI

1 **Quali delle seguenti affermazioni sono vere?**

A In generale l'esecuzione di una funzione ricorsiva è più veloce di una equivalente funzione iterativa.

B In generale l'esecuzione di una funzione ricorsiva richiede più memoria di una equivalente funzione iterativa.

C La ricorsività può essere implementata anche senza ricorrere all'uso di funzioni.

D Una funzione ricorsiva contiene al suo interno almeno un costrutto iterativo (*for*, *while*, …).

2 **Quali sono le condizioni necessarie per la terminazione di un procedimento ricorsivo?**

A Almeno un'invocazione ricorsiva della funzione deve prevedere la soluzione di un sottoproblema ridotto rispetto all'originale.

B La funzione ricorsiva deve contenere almeno una istruzione *return*.

C La funzione ricorsiva non deve mai invocare sé stessa.

D La funzione ricorsiva deve contenere almeno una modalità che eviti l'invocazione di sé stessa.

3 **Quali delle seguenti affermazioni sono vere rispetto all'algoritmo *quick-sort* applicato a un *array*?**

A La chiamata ricorsiva sull'insieme sinistro viene applicata in parallelo alla chiamata ricorsiva sull'insieme destro.

B Prima di applicare la chiamata ricorsiva sull'insieme destro vengono esaurite tutte le chiamate ricorsive sull'insieme sinistro.

C L'elemento *pivot* deve necessariamente essere l'elemento centrale dell'insieme.

D Nessuna delle risposte precedenti.

4 **Che cosa calcola la seguente funzione ricorsiva?**

```
long mistero(int n)
{
   if (n == 0)
     return 1;
   return (n*mistero(n-1));
}
```

A Il quadrato di un numero intero *n*.

B La somma dei primi *n* numeri naturali.

C Il fattoriale di *n*.

D Il prodotto $n \cdot (n-1)$.

5 **La seguente funzione**

```
void visualizzaVettore(int v[],int n)
{
   if (n > 0)
     visualizzaVettore(v,n-1);
   cout<<v[n]<<endl;
}
```

visualizza gli elementi di un vettore …

A … in ordine di indice.

B … in ordine inverso.

C … in ordine crescente.

D … in ordine casuale.

6 **Quale risultato calcola nella variabile *x* la seguente funzione C++ ricorsiva quando viene invocata come indicato?**

```
char stringa[] = «Ciao mondo!»;
...
int mistero(char s[], int p)
{
   if (s[p] == '\0')
     return 0;
   return (1 + mistero(s, (p+1));
}
...
x = mistero(stringa, 0);
```

A Il numero dei caratteri che compongono la stringa.

B Il numero delle vocali a partire dal carattere di indice 0 della stringa.

C Una nuova stringa senza i caratteri che occupano posizioni multiple di *p*.

D Nessuna delle risposte precedenti.

ESERCIZI

1 Scrivere una funzione ricorsiva C++ che calcoli il prodotto di due numeri interi forniti come parametri utilizzando solo l'operatore algebrico di addizione.

2 Scrivere una funzione ricorsiva C++ avente il seguente prototipo

```
float power(float a, unsigned int b)
```

che calcoli la potenza a^b.

3 La *sezione aurea* è il numero

$$\phi = \frac{\sqrt{5}-1}{2} = 0{,}618\ldots$$

Le potenze della sezione aurea godono della seguente proprietà:

$$\phi^{n+1} = \phi^{n-1} - \phi^n$$

Scrivere una funzione ricorsiva C++ che calcoli la potenza *n*-esima della sezione aurea (con *n* fornito come argomento).

4 Scrivere una funzione ricorsiva C++ che calcoli il quoziente della divisione tra interi. (Suggerimento: $x/y = (x - y + y)/y = 1 + (x - y)/y$)

5 Dato un vettore di *n* numeri interi, scrivere una funzione ricorsiva C++ che ne restituisca la somma.

6 Dato un vettore di *n* numeri interi, scrivere una funzione ricorsiva C++ che ne visualizzi gli elementi di indice pari (0, 2, 4, ...).

7 Dato un vettore di *n* numeri interi, scrivere una funzione ricorsiva C++ che ne scambi gli elementi fino a metterli in ordine inverso rispetto alla posizione iniziale (senza dichiarare vettori aggiuntivi).

8 Data una stringa *S* di caratteri, scrivere una funzione ricorsiva C++ che restituisca la stringa ottenuta da *S* eliminando tutti gli spazi bianchi.

9 Data una stringa *S* di caratteri e un carattere *c*, scrivere una funzione ricorsiva C++ che calcoli le occorrenze di *c* in *S*.

10 Data una stringa di caratteri, scrivere una funzione ricorsiva C++ che visualizzi tutti i possibili anagrammi realizzabili.

11 Scrivere una funzione ricorsiva C++ che, assegnati due interi *N* e *M*, restituisca la somma di tutti gli interi compresi tra *N* e *M*.

12 Dato un vettore di *n* numeri interi, scrivere una funzione ricorsiva C++ che calcoli il massimo valore dei suoi elementi.

13 Dato un vettore di *n* numeri in doppia precisione, scrivere una funzione ricorsiva C++ che restituisca la differenza minima tra ogni elemento e il successivo (ultimo escluso).

14 Dato un vettore di *n* numeri interi, scrivere una funzione ricorsiva C++ che restituisca il valore massimo della somma tra ogni elemento e il precedente (primo escluso).

15 Data una matrice quadrata $N \times N$ di interi, scrivere una funzione ricorsiva C++ che verifichi se questa è simmetrica o meno rispetto alla diagonale principale restituendo un opportuno valore booleano.

16 Data una matrice quadrata $N \times N$ di interi, scrivere una funzione ricorsiva C++ che verifichi se questa ha o meno due righe identiche restituendo un opportuno valore booleano.

A10 I file

Si devono gestire le prenotazioni per la prevendita dei posti della sala di un teatro: i posti sono disposti per file, ognuna delle quali ha un certo numero di posti che possono essere liberi, occupati oppure riservati. Gli spettatori possono effettuare le prenotazioni presso un centro telefonico modificando la situazione delle disponibilità.
Ragionevolmente si può scrivere un programma dove viene usata una matrice di valori interi per simulare la mappa della sala del teatro, dove le file dei posti corrispondono alle righe della matrice e l'insieme di uno specifico numero di posto nelle varie file corrisponde a una colonna della matrice. Inoltre si può associare a ogni posto libero il valore 0, ai posti occupati il valore 1 e a quelli riservati il valore 2. L'operatore che riceve le richieste dei clienti dovrà indicare ogni volta il numero di fila e il numero di posto in modo che, se questo è disponibile, la prenotazione possa essere effettuata.
La situazione è simile ad altre che abbiamo trattato studiando gli *array*: ma che cosa accade se le prenotazioni non vengono effettuate in un'unica giornata? Tutte le strutture dati di cui abbiamo trattato nei capitoli precedenti, così come quella che abbiamo ipotizzato di utilizzare in questo caso, hanno una caratteristica comune: sono realizzate nella memoria volatile del computer. Questo fatto rende la loro esistenza strettamente legata a quella dei programmi che le gestiscono, e quando un programma termina la sua esecuzione e il sistema operativo libera lo spazio di memoria a esso assegnato, i dati sono perduti per sempre.
In riferimento alla situazione descritta, significa che ogni volta che si spegne il computer o che si chiude il programma che gestisce le prenotazioni inevitabilmente i dati relativi alle prenotazioni stesse vengono perduti: questa cosa è ovviamente inaccettabile. D'altra parte non si può pensare di tenere il programma costantemente attivo, perché questo comporterebbe la necessità di mantenere il computer costantemente acceso per tutto il periodo di tempo in cui le prenotazioni sono accettate dal centro telefonico.

Dal momento che la maggior parte di esempi realistici di programmi per computer hanno a che fare, come in questo caso, con dati da gestire in modo persistente, la soluzione del problema prevede di utilizzare un nuovo tipo di struttura dati realizzata su memorie di tipo permanente (dischi magnetici o ottici, memorie a stato solido realizzate con tecnologia FLASH ecc.) da dove, anche in tempi successivi, possano essere recuperate per consultazione e/o aggiornamento. Strutture dati di questo tipo sono chiamate **file**.

La soluzione al problema della prevendita dei biglietti sarà basata sullo sviluppo di un programma che ogni volta terminerà la sua attività salvando su un file la situazione attuale della matrice dei posti, per poi acquisirla nuovamente nel momento in cui verrà nuovamente eseguito per effettuare una nuova sessione di prenotazioni.

1 Gestione dei file in C++

La tecnica della memorizzazione di dati su file è utilizzata nei casi più disparati: documenti di testo, fogli di calcolo, immagini, filmati, brani musicali ecc.: i dati sono memorizzati sulla memoria non volatile del computer in modo da essere conservati nel tempo.
Affronteremo i concetti di base dalla gestione dei file trattando alcune loro semplici organizzazioni e gli strumenti che il C++ mette a disposizione per la loro realizzazione.

▶ Un **file** è un insieme, non necessariamente omogeneo, di dati correlati, identificato da un nome, memorizzato permanentemente su un supporto di memoria non volatile e avente vita indipendente dal programma (o dai programmi) utilizzato/i per la sua creazione e/o modifica.

OSSERVAZIONE Il fatto che un file abbia vita indipendente dal programma utilizzato per la sua creazione e/o modifica significa che un file creato con un programma può essere successivamente elaborato sia mediante lo stesso programma, sia utilizzando programmi diversi.
In questo modo, per esempio, è possibile creare un file con un programma scritto in linguaggio C++ e riutilizzarlo successivamente mediante programmi realizzati con linguaggi di programmazione diversi, o utilizzando un'applicazione di utilità generale come un editor di testo.

L'accesso a un file si basa su operazioni piuttosto lente e complesse, dal punto di vista delle risorse del computer che entrano in gioco: la loro esecuzione richiede una conoscenza dettagliata dell'hardware del supporto di memorizzazione e può determinare situazioni di conflitto fra diversi programmi in esecuzione. È per queste ragioni che tali operazioni sono demandate al sistema operativo per conto dei programmi che ne fanno richiesta. Le operazioni che la gestione dei file comunemente prevede sono le seguenti.

- **Apertura**: **il programma comunica al sistema operativo la necessità di accedere a un specifico file.** Il sistema operativo controlla l'esistenza del file e che questo non sia già inutilizzato o bloccato, crea alcune strutture dati nascoste per gestire le operazioni successive e riserva un'area di

File system

Il *file system* è la componente del sistema operativo che gestisce i dati presenti nella memoria non volatile del computer (disco magnetico o ottico, memoria FLASH ecc.).
Esistono diversi tipi di *file system*, ma tutti organizzano i contenuti in cartelle (directory) e file.
In particolare ogni singolo oggetto memorizzato sulla memoria non volatile viene definito file: un file può essere un documento di testo, un programma, un foglio di calcolo, un disegno, una fotografia, un filmato, un brano musicale.
Anche le cartelle sono viste come file speciali in cui è memorizzato l'elenco delle cartelle e dei file che contengono.
Il sistema operativo gestisce la memoria non volatile del computer fornendone all'utente la visione astratta del sistema di cartelle e di file.
Quando viene creato un nuovo file è il sistema operativo che alloca e gestisce lo spazio di memoria necessario a contenerlo; e quando viene cancellato un file è sempre il sistema operativo che provvede a recuperare lo spazio da esso utilizzato per renderlo di nuovo disponibile.

memoria, comunemente denominata *buffer*, per memorizzare i dati in transito dal programma al file e viceversa dal file al programma; restituisce inoltre al programma un riferimento che esso utilizzerà nelle successive operazioni sullo stesso file. Altri programmi possono accedere allo stesso file solo in modo limitato (per esempio senza poterlo modificare) o non possono accedervi affatto.

- **Chiusura**: **il programma comunica al sistema operativo che non ha più necessità di accedere al file.** Tutte le strutture dati nascoste sono eliminate e lo spazio di memoria del *buffer* viene rilasciato; il sistema operativo elimina il file dalla lista dei file in uso consentendo ad altri programmi di accedervi liberamente.
- **Lettura**: **il programma richiede dei dati dal file come input.** Il sistema operativo acquisisce i dati, li memorizza temporaneamente nel *buffer* e comunica al programma l'avvenuta lettura, in modo che questo possa utilizzare i dati.
- **Scrittura**: **il programma intende scrivere dei dati in un file come output.** I dati sono temporaneamente memorizzati nel *buffer*, successivamente il programma ne comunica al sistema operativo la disponibilità e lascia che questo si occupi di effettuare la scrittura fisica sul supporto di memoria attendendo la comunicazione di avvenuta scrittura (FIGURA 1).

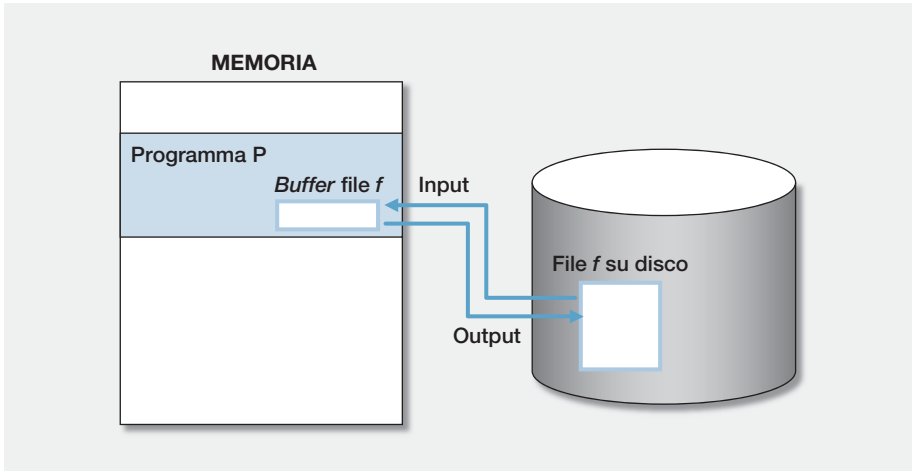


FIGURA 1

OSSERVAZIONE Un programma non può leggere o scrivere un file se prima non lo ha «aperto» e, una volta finito di usarlo, lo deve sempre «chiudere» per consentire al sistema operativo di liberare le risorse occupate.

Il linguaggio C++ permette al programmatore di definire vari tipi di organizzazione dei file che sono riferiti come *stream* (flussi). La trattazione che segue è limitata al tipo più semplice: i file sequenziali di tipo testuale.

Organizzazione e modalità di accesso

I dati in un file sono sempre memorizzati come una sequenza di byte. In relazione a un file si parla di **organizzazione** facendo riferimento alla tecnica con cui questi byte vengono gestiti. Tutti i sistemi operativi consentono almeno due diversi tipi di organizzazione dei file: l'organizzazione sequenziale e l'organizzazione *random*.

Senza entrare nel dettaglio, possiamo affermare che la **modalità di accesso** è il modo in cui i dati possono essere reperiti all'interno di un file: essa è strettamente legata al tipo di organizzazione scelta.

Un file organizzato sequenzialmente permette di accedere ai dati solo in maniera **sequenziale**, ovvero nello stesso ordine in cui questi sono stati scritti: per leggere un dato è necessario leggere prima tutti quelli che lo precedono, senza possibilità di tornare indietro.

L'organizzazione *random* permette l'accesso **diretto** a ogni singolo dato specificando la sua posizione all'interno del file, come se i dati fossero memorizzati all'interno di un vettore. Ovviamente anche con l'organizzazione *random* è possibile operare in modalità sequenziale, specificando in maniera ordinata la posizione dei singoli dati.

OSSERVAZIONE Si noti come lo standard input (in genere la tastiera) e lo standard output (normalmente la finestra di testo sul video) siano visti dal programma come file sequenziali di testo.

Il C++ prevede le seguenti classi di file di testo:

- ***ofstream***: file con accesso in sola scrittura (la «o» iniziale di *ofstream* sta per output);
- ***ifstream***: file con accesso in sola lettura (la «i» iniziale di *ifstream* sta per input);
- ***fstream***: file con accesso contemporaneo sia in lettura sia in scrittura (è però possibile scegliere solo una delle due opzioni in funzione della modalità di apertura, ovvero, se un file di classe *fstream* viene aperto in lettura per utilizzarlo in scrittura, è necessario prima chiuderlo e quindi riaprirlo).

Queste classi derivano dalle classi *istream* e *ostream* di cui abbiamo già visto gli oggetti ***cin*** e ***cout***: il primo è un oggetto di classe *istream*, mentre il secondo è di classe *ostream*. I file sequenziali possono essere utilizzati nello stesso modo con cui abbiamo utilizzato *cin* e *cout*, con la sola differenza che in questo caso i due oggetti sono associati a file fisici, con una propria posizione sul supporto di memorizzazione e un proprio nome.

OSSERVAZIONE Come è consuetudine della programmazione per oggetti che presenteremo con maggior dettaglio nel seguito, le classi per la gestione di file prevedono delle funzioni predefinite, comunemente note come **metodi**, che sono invocate in riferimento agli oggetti istanziati a partire da esse. L'invocazione di un metodo di un oggetto avviene mediante un'istruzione del tipo:

<nome oggetto>.<nome metodo>(<lista di parametri>)

dove <lista di parametri> può ovviamente anche essere vuota.

ESEMPIO

Il seguente programma C++:

```
#include <iostream>
#include <fstream>

using namespace std;

void main (void)
{
  ofstream miofile;

  miofile.open("testo.txt");
  miofile<<"Ciao mondo!"<<endl;
  miofile.close();

  return;
}
```

crea un file denominato «testo.txt» nella stessa directory del programma eseguibile e inserisce in esso la frase «Ciao mondo!» nello stesso modo cui siamo soliti usare lo standard output *cout*, con la sola differenza che, invece di usare come destinazione la finestra di testo sul video, abbiamo utilizzato il file *miofile*. L'oggetto *miofile* (istanziato a partire dalla classe *ofstream* con la dichiarazione iniziale) viene associato al file fisico «testo.txt» mediante l'invocazione del metodo *open()*. Le invocazioni dei metodi *open()* e *close()* realizzano rispettivamente l'**apertura** e la **chiusura** del file, mentre il noto operatore << realizza la scrittura del testo nel file.

La gestione di un file in un programma comporta l'identificazione del file stesso mediante un nome. In particolare, nell'ambito del programma vengono utilizzati due nomi distinti, uno fisico e un logico.

- **Nome fisico**: è il nome con cui il file viene salvato sul supporto di memorizzazione da parte del sistema operativo, per esempio «testo.txt».

OSSERVAZIONE In generale il nome fisico completo di un file viene detto ***pathname***, perché comprende l'identificazione del dispositivo di memoria di massa su cui esso è salvato e, a partire da questa, il percorso delle directory che ne definisce la posizione. Per esempio per il sistema operativo Windows, il *pathname* `"C:\alfa.txt"` indica che il file denominato `"alfa.txt"` è memorizzato direttamente nella radice del *file system* del disco `"C:"`, mentre il *pathname* `"C:\Documenti\esempi\beta.txt"` identifica il file `"beta.txt"` che si trova all'interno della cartella `"esempi"`, a sua volta memorizzata all'interno della cartella `"Documenti"` memorizzata nella radice del *file system* del disco `"C:"`.
Se il *pathname* specificato è **relativo**, cioè non specifica il dispositivo di memoria (spesso si include il solo nome del file), il file sarà ricercato a partire dalla stessa directory dove è memorizzato il file che contiene il codice eseguibile del programma. Per esempio, se il programma che apre il file è memorizzato in `"C:\Sviluppo"`, il file riferito mediante il *pathname* relativo `"\esercizi\gamma.txt"` sarà ricercato come se il *pathname* completo fosse `"C:\Sviluppo\esercizi\gamma.txt"`.
Si noti che il sistema operativo Linux non prevede la specificazione del dispositivo di memoria nei *pathname*.

- **Nome logico**: è il nome che nel programma viene associato al nome fisico all'atto dell'apertura del file tramite il metodo *open()*; nel corso dell'esecuzione del programma tutte le operazioni che saranno effettuate sul file faranno riferimento al solo nome logico del file. Nell'esempio precedente le operazioni effettuate sul file fisico `"testo.txt"`, dopo la sua apertura, sono riferite al nome dell'oggetto logico *miofile* che ne permette la gestione.

Nella FIGURA **2** è schematizzata una situazione in cui sono evidenziati i flussi di input/output di un programma che legge/scrive utilizzando sia i dispositivi standard di input/output (tastiera e finestra di testo su video), sia un dispositivo di memorizzazione non volatile; il codice del programma riportato legge da un file `"ingresso.txt"` q valori interi (q viene fornito dall'utente mediante la tastiera) e li scrive nel file `"uscita.txt"` (se `"ingresso.txt"` contiene meno di q valori interi tutto il suo contenuto viene scritto in `"uscita.txt"`).
Esaminiamo ora nel dettaglio questo esempio per introdurre le tecniche di gestione dei file testuali in C++:

- `#include <iostream>`: il file di intestazione «*iostream*» contiene la dichiarazione delle costanti enumerative ***ios::in*** e ***ios::out*** usate per l'apertura dei file utilizzati dal programma.

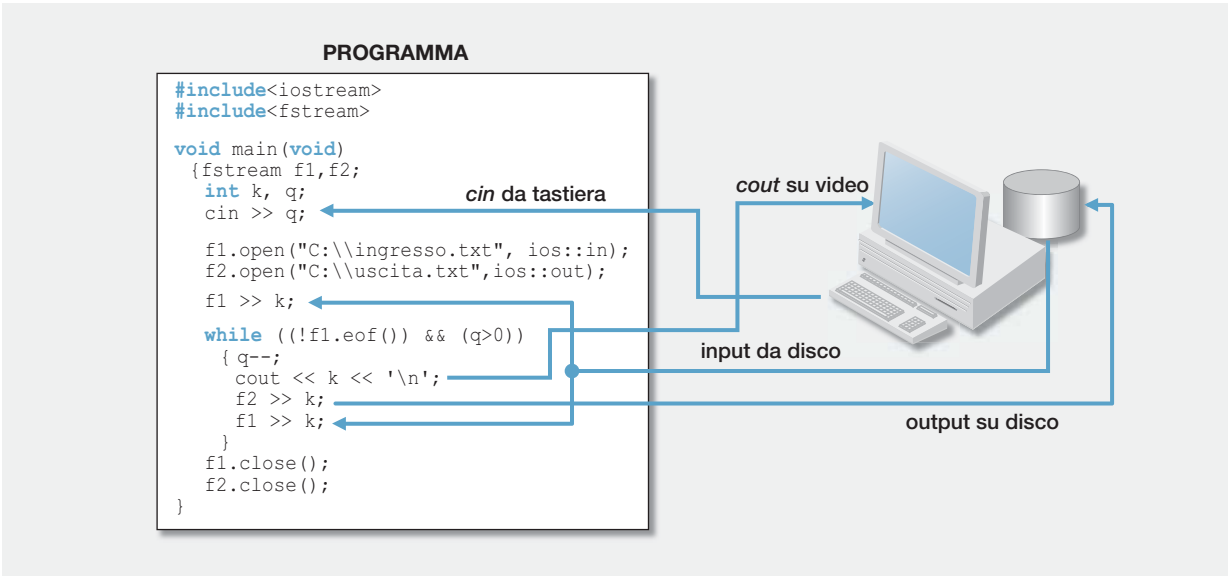


FIGURA **2**

- `#include <fstream>`: il file di intestazione *fstream* contiene la definzione della classe che permette l'uso di questo tipo di file.
- `fstream f1,f2`: dichiarazione dei due oggetti di tipo *fstream* *f*1 e *f*2 che saranno associati rispettivamente ai due file `"ingresso.txt"` e `"uscita.txt"`.
- `f1.open("C:\\ingresso.txt",ios::in)`: è il metodo che apre il file `"ingresso.txt"` (nome fisico) associandolo all'oggetto *f*1 (nome logico).

OSSERVAZIONE Il metodo *open()* della classe *fstream* prevede due parametri:

- il primo serve a specificare il *pathname* completo del file (la doppia «\» è resa necessaria dal fatto che il simbolo «\» all'interno di una stringa di testo viene interpretato dal compilatore C++ come un carattere speciale);
- il secondo indica la modalità di apertura del file. Le principali modalità di apertura di un *fstream* sono le seguenti:
  - ***ios::in***: il file viene aperto in lettura (input); ovviamente un file che viene aperto in lettura deve esistere, ovvero deve essere stato preventivamente creato da un qualsiasi programma (non necessariamente dallo stesso programma che ne richiede l'apertura). Questa modalità di apertura è automatica nel caso di file di tipo *istream*;
  - ***ios::out***: il file viene aperto in scrittura (output); il file viene creato e, nel caso in cui esistesse già un file con lo stesso nome fisico nella stessa posizione sul supporto di memorizzazione, questo viene sostituito dal nuovo, il che significa che il contenuto del file preesistente viene perduto. Questa modalità di apertura è automatica nel caso di file di tipo *ostream*;

- *ios::app*: il file viene aperto in output, ma in aggiunta (*append*); il file deve essere preesistente e il nuovo output andrà ad accodarsi al contenuto presente.

- `while ((!f1.eof()) && (q>0))`: è l'inizio del ciclo che permette la lettura di più valori da *f*1; la condizione di terminazione del ciclo è doppia e l'operatore logico AND («&&») impone che il ciclo sia eseguito fino a che entrambe le clausole sono verificate:
  - `(!f1.eof())`: serve a verificare che nel file vi siano ancora dati da leggere; il metodo *eof()* (*end of file*) restituisce il valore `true` se nel file sono terminati i valori da leggere, *False* altrimenti: l'operatore di negazione impone la continuazione del ciclo fino a che ci sono ancora dati nel file;
  - `(q>0)`: serve a soddisfare la condizione che impone di leggere al massimo *q* dati (*q* è un valore inserito dall'utente utilizzato come contatore a scalare per tenere traccia del numero dei valori letti).

OSSERVAZIONE La prima lettura da *f*1 viene eseguita fuori dal ciclo: questo è il modo corretto di procedere perché, nel caso in cui il file `"ingresso.txt"` non contenga valori, *eof()* restituirà immediatamente `true`, rendendo falsa la condizione del ciclo che non sarà eseguito neanche una volta.

- `f1 >> k` ed `f2 << k`: sono le due istruzioni che servono rispettivamente a effettuare una lettura dal file *f*1 e una scrittura nel file *f*2; gli operatori «<<» (operatore di scrittura) e «>>» (operatore di lettura), già visti in relazione agli oggetti predefiniti *cin* e *cout*, realizzano i due tipi di operazione.

OSSERVAZIONE Sia l'operatore di scrittura («<<») sia l'operatore di lettura («>>») sono **sequenziali** (da qui il nome dato a questo tipo di file); l'uso ripetuto (tipicamente all'interno di un ciclo) dello stesso operatore consentono di scrivere o leggere valori nel file in modo sequenziale: uno di seguito all'altro.

> In un file sequenziale, infatti, è consentito l'accesso ai dati nell'ordine in cui questi sono stati scritti: a ogni lettura un «indice» immaginario[1] si sposta da un dato al successivo, fino a quando non viene raggiunta la fine del file.

Supponendo di essere posizionati sul dato *i*-esimo, all'interno del file non vi è altra possibilità di posizionarsi sul dato precedente se non chiudendo il file, riaprirlo e rileggere tutti i dati fino a raggiungere quello in posizione $(i-1)$-esima.

1. In realtà questo indice è una delle informazioni che il sistema operativo mantiene nelle strutture dati nascoste utilizzate per la gestione dei file.

- `f1.close()` ed `f2.close()`: sono le istruzioni che chiudono i due file; l'operazione di chiusura è importante, in particolare per i file aperti in

output, perché in questo modo il programma richiede al sistema operativo il salvataggio definitivo sul supporto di memorizzazione di eventuali dati ancora presenti nel *buffer* di memoria centrale, evitando la loro possibile perdita.

L'operazione di apertura del file è fondamentale, sia che si intenda lavorare in lettura, come in scrittura. In entrambi i casi è necessario assicurarsi che il file sia stato aperto in maniera corretta (in lettura è ovviamente anche necessario verificare che il file specificato sia preesistente); questo è infatti il presupposto perché il nostro programma funzioni adeguatamente. Per verificare il successo o meno di un'operazione di apertura di un file, è possibile controllare se l'oggetto associato al file sia stato istanziato o meno; usualmente, dopo aver richiesto l'apertura del file, è possibile eseguire il seguente test:

```
...
miofile.open("dati.txt");
if (miofile)
  {
    // OK
    ...
    ...
    ...
  }
else
  {
    // ERRORE
    ...
  }
```

In alternativa è possibile usare i metodi *is_open()* o *fail()* che restituiscono entrambi un valore booleano:

- *is_open()* restituisce `true` se il file è aperto correttamente, `false` altrimenti;
- *fail()* ritorna `true` se il file non è aperto correttamente, `false` altrimenti.

ESEMPI

```
...
miofile.open("dati.txt");
if (miofile.is_open())
  {
    // OK
    ...
    ...
    ...
  }
else
  {
    // ERRORE
    ...
  }
```

```
...
miofile.open("dati.txt");
if (!miofile.fail())
  {
    // OK
    ...
    ...
    ...
  }
else
  {
    // ERRORE
    ...
  }
```

2 Tecniche per la gestione dei file testuali sequenziali

ESEMPIO

Vediamo un semplice esempio di codice che carica una matrice di valori numerici a partire dai dati memorizzati in un file:

```
#include <iostream>
#include <fstream>
using namespace std;
void main(void)
{
  ifstream matfile;
  int matrice[100][75];
  int i,j;

  matfile.open("matrice.txt");
  for (i=0; i<100; i++)
    for (j=0; j<75; j++)
      matfile >> matrice[i][j];
  matfile.close();
  ...
}
```

I valori numerici memorizzati nel file possono essere separati tra loro in vario modo: da spazi, da tabulazioni o da ritorni «a capo»[2].

2. Si ricordi che il ritorno «a capo» è dato dai caratteri ASCII CR (codice decimale 13) e LF (codice decimale 10).

OSSERVAZIONE Nel codice dell'esempio precedente si carica la matrice con la tecnica classica dei due cicli *for* nidificati. In questo modo, se il file non contiene almeno 750 valori interi, si corre il rischio di eseguire un'operazione di lettura quando gli elementi del file sono terminati e di incorrere in un errore di lettura. Per ovviare a questo problema è opportuno effettuare un test di fine file subito dopo l'istruzione di lettura ed eventualmente interrompere l'esecuzione del programma:

```
#include <iostream>
#include <fstream>

using namespace std;

void main(void)
{
  ifstream matfile;
  int matrice[100][75];
  int i,j;

  matfile.open("matrice.txt");
```

▶

File testuali e binari

In particolare i valori numerici possono essere memorizzati in un file in formato testuale (la sequenza dei codici ASCII delle cifre) o binario (la codifica binaria del valore numerico). La semplicità d'uso dei file testuali (normalmente «leggibili» dall'utente umano mediante un semplice editor di testo) è compensata dalla maggiore dimensione: un singolo valore numerico intero decimale di 8 cifre occupa 8 byte in formato testuale, ma sono sufficienti 4 byte per la sua codifica binaria.

```
  for (i=0; i<100; i++)
    {
      for (j=0; j<75; j++)
        {
          matfile >> matrice[i][j];
          if (matfile.eof())
            {
              cout<<"dati insufficienti!"<<endl;
              matfile.close();
              return;
            }
        }
    }
  matfile.close();
  ...
}
```

Si noti che il codice chiude correttamente il file anche se, in caso di dati insufficienti, il programma viene terminato prematuramente.

ESEMPIO

In alternativa si può operare in maniera opposta, caricando i dati nella matrice a partire dal contenuto del file, come nel codice che segue:

```
#include <iostream>
#include <fstream>

using namespace std;

void main(void)
{
  ifstream matfile;
  int matrice[100][75];
  int i=0,j=0,k;

  matfile.open("matrice.txt");
  matfile >> matrice[i][j];
  while (!matfile.eof())
    {
      j++;
      if (j > 74)
        {
          j = 0;
          i++;
          if (j > 99)
            break;
        }
      matfile >> matrice[i][j];
    }
  matfile.close();
  ...
}
```

OSSERVAZIONE Negli esempi illustrati i dati sono letti sequenzialmente, per cui possono essere memorizzati nel file nell'ordine corretto, ma separati tra loro utilizzando spazi, tabulazioni e «a capo» (caratteri «separatori») in un modo qualsiasi.
In questo senso è indifferente che siano in riga

3 4 6 2 ... 44 8 10 ...

in colonna

```
3
5
6
2
...
44
8
10
...
```

o in formato matriciale

```
3 5 6 2 ...
44 8 10 ...
...
```

oppure alternando in qualsiasi ordine l'uso dei caratteri separatori.

Vediamo ora un possibile approccio risolutivo al problema della prenotazione dei posti della sala di un teatro dove i posti sono disposti come in una matrice per file (righe) e numeri di posto (colonne) e dove ogni posto può essere libero (indicato con il valore «0»), occupato (indicato con il valore «1») oppure riservato (indicato con il valore «2»).
Il programma deve leggere da file la situazione corrente dei posti, ricevere in input dall'utente una serie di prenotazioni caratterizzate dalle coordinate (fila, posto), verificare per ognuna di esse se è accettabile o meno, effettuando in caso positivo l'eventuale prenotazione (la digitazione del valore «0» in corrispondenza del valore della fila di una prenotazione termina l'elaborazione e il programma che deve salvare permanentemente la nuova situazione di occupazione dei posti).

```
#include <iostream>
#include <fstream>

using namespace std;

#define FILE   35
#define POSTI  40

// funzione per il caricamento dal file delle prenotazioni
void caricaDati(int sala[][POSTI])
{
  fstream teatro;
  int i=0,j=0;

  teatro.open("teatro.txt",ios::in);
  teatro >> sala[i][j];
  while (!teatro.eof())
```

▶

```
    {
      j++;
      if (j >= POSTI)
        {
          j = 0;
          i++;
          if (i >= FILE)
            break;
        }
      teatro >> sala[i][j];
    }
  teatro.close();
}

// funzione per il salvataggio su file delle prenotazioni
void salvaDati(int sala[][POSTI])
{
  fstream teatro;
  int i=0,j=0;

  teatro.open("teatro.txt",ios::out);
  for (i=0; i<FILE; i++)
    {
      for (j=0; j<POSTI; j++)
        teatro << sala[i][j] << ' ';
      teatro << endl;
    }
  teatro.close();
}

void main(void)
{
  int sala[FILE][POSTI];
  int fila, posto;

  caricaDati(sala);

  while (true)
    {
      cout << "Inserire il numero di fila: ";
      cin >> fila;
      if (fila == 0)
        break;
      cout << "Inserire il numero di posto: ";
      cin >> posto;
      if ((fila <= 0)  || (fila > FILE) ||
          (posto <=0) || (posto > POSTI))
        cout << "Coordinate di posto non valide" << endl;
      else
        {
```

▶

```
            switch (sala[fila-1][posto-1])
              {
                case 0: cout << "Prenotazione effettuata" << endl;
                        sala[fila-1][posto-1]=1;
                        break;

                case 1: cout << "Posto già prenotato" << endl;
                        break;

                case 2: cout << "Posto riservato" << endl;
                        break;
              }
          }
      }

  salvaDati(sala);
}
```

OSSERVAZIONE Per predisporre i dati iniziali per il programma dell'esempio precedente, è possibile utilizzare un qualsiasi editor di testo. Per esempio, i dati iniziali dell'occupazione della sala possono essere memorizzati nel seguente formato:

```
0 0 0 0 0 0 0 0 2 0 2 2 0 0 0 0 0 0 2 0 2 2 0 0 0 0 2 2 2 0 0 2 0 0 0 0 0 0 2 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Allo stesso modo, dopo aver registrato alcune prenotazioni e averle salvate uscendo dal programma, è possibile utilizzare lo stesso editor per verificare lo stato di occupazione dei posti del teatro:

```
0 0 1 1 1 0 0 0 2 0 2 2 0 0 0 0 0 0 2 0 2 2 0 0 0 0 2 2 2 0 0 2 0 0 0 0 0 0 2 0
0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

I file sequenziali di testo rappresentano il modo più semplice per memorizzare dati sulla memoria non volatile; la loro struttura non è molto sofisticata e le loro prestazioni in termini di funzionalità offerte sono estremamente ridotte. Tuttavia file di questo tipo hanno ancora una loro ragione di esistere: spesso costituiscono la maniera più semplice – e talvolta l'unica – per scambiare dati tra applicazioni software che utilizzano formati di file dati non compatibili tra loro.

Un esempio tipico in questo senso è dato dai file in formato **CSV** (***Comma Separated Values***), che sono spesso utilizzati per l'importazione e l'esportazione di una tabella di dati da applicazioni come i gestori di fogli di calcolo o di database. Non esiste uno standard formale che definisce questa tipologia di file, ma solo alcune consuetudini consolidate: ogni riga della tabella è normalmente rappresentata da una linea di testo, a sua volta suddivisa in colonne (campi), ciascuna delle quali rappresenta un valore separato dagli altri da un apposito carattere separatore. Il formato CSV non specifica la codifica dei caratteri, né la convenzione per indicare il fine linea (anche se tipicamente sono utilizzati i caratteri ASCII CR e/o LF), né il carattere da usare come separatore tra campi (il punto e virgola e il carattere ASCII per la tabulazione sono molto diffusi) e nemmeno le convenzioni per rappresentare date o numeri (tutti i valori sono considerati semplici stringhe di testo) o se la prima riga sia da considerarsi di intestazione o meno.

ESEMPIO

Si consideri la tabella seguente:

	Cognome	Nome	Indirizzo	CAP	Città
Intestazione →					
Contatti →	Rossi	Marco	Via Roma, 12	57100	Livorno
	Neri	Andrea	Via del Mare, 15	56100	Pisa
	Bianchi	Giovanni	Via Ferruccio, 7	55100	Firenze

Esportata in un file in formato CSV potrebbe assumere il seguente formato:

```
Cognome;Nome;Indirizzo;CAP;Città<CR><LF>
Rossi;Marco;Via Roma, 12;57100;Livorno<CR><LF>
Neri;Andrea;Via del Mare, 15;56100;Pisa<CR><LF>
Bianchi;Giovanni;Via Ferruccio, 7;55100;Firenze<CR><LF>
```

dove chiaramente il simbolo «;» è il carattere separatore utilizzato per distinguere i singoli valori.

OSSERVAZIONE I caratteri terminatori di riga sono «invisibili» e solo per chiarezza, nell'esempio precedente, si è voluto rappresentare i simboli <CR> e <LF>.

OSSERVAZIONE Nell'economia di un programma C++ che debba costruire una tabella acquisendone i valori da un file in formato CSV, è necessario ricordare che, essendo gli spazi bianchi interpretati come separatori, un dato come «Via Roma, 12» dovrebbe essere acquisito con tre letture successive. Ovviamente, non avendo indicazioni su come possano essere strutturati i singoli dati, è necessario operare in una maniera

diversa: la strategia è quella di leggere ogni singola riga come un'unica stringa di caratteri e di individuarne successivamente le singole componenti (nell'esempio: Cognome, Nome, Indirizzo, ...) mediante una funzione che esamina la stringa alla ricerca del carattere di separazione, in modo da poterla suddividere nei vari elementi.
Il linguaggio C++ consente di leggere un'intera linea di testo da un file utilizzando il metodo *getline()*, i cui due parametri sono rispettivamente il vettore di caratteri dove inserire la stringa terminata dal carattere ASCII di codice numerico 0 e la lunghezza massima che questa può assumere (nel caso in cui una linea del file contenga un numero di caratteri superiore, questa risulterà troncata).

ESEMPIO

Il codice che segue è relativo alla lettura di un file in formato CSV contenente i dati della tabella dei contatti illustrata in precedenza, dove il simbolo «;» è il carattere di separazione tra i vari valori di ogni singola riga:

```
#include <iostream>
#include <fstream>

using namespace std;

#define MAX_LEN 256

// funzione per estrarre il prossimo elemento dalla stringa
void estraiElemento(char rigain[], char elemento[])
{
  int i,k,j;

  // ricerca fine stringa o carattere separatore
  for (i=0; (i<MAX_LEN)&&(rigain[i]!='\0')&&(rigain[i]!=';'); i++);
  // copia elemento
  for (k=0; k<i; k++)
    elemento[k]=rigain[k];
  elemento[k]='\0';
  // eliminazione dalla stringa dell'elemento trovato
  j=0;
  for (k=i+1; k<MAX_LEN; k++)
    {
      rigain[j]=rigain[k];
      j++;
    }
}

void main(void)
{
  ifstream tabfile;
  char riga[MAX_LEN], elemento[MAX_LEN];
  int i=1;
```

▶

```
  tabfile.open("tabella.txt");
  tabfile.getline(riga, MAX_LEN);
  while (!tabfile.eof())
    {
      cout << "Contatto n. " << i++ << endl;
      estraiElemento(riga, elemento);
      cout << "Cognome: " << elemento << endl;
      estraiElemento(riga,elemento);
      cout << "Nome: " << elemento << endl;
      estraiElemento(riga,elemento);
      cout << "Indirizzo: " << elemento << endl;
      estraiElemento(riga,elemento);
      cout << "CAP: " << elemento << endl;
      estraiElemento(riga,elemento);
      cout << "Citta': " << elemento << endl;
      tabfile.getline(riga,MAX_LEN);
    }
  tabfile.close();
}
```

Anche in questo caso è bene notare come siano state utilizzate due istruzioni

```
tabfile.getline(riga, MAX_LEN);
```

una immediatamente prima del ciclo e l'altra come ultima istruzione del ciclo stesso. La prima occorrenza serve a verificare che il file non sia vuoto. In tale evenienza, infatti, la condizione del **while**, `(!tabfile.eof())`, risulterà immediatamente falsa e il ciclo stesso non sarà eseguito neanche una volta.
Nel caso che il file contenga almeno una riga, questa verrà elaborata dalle istruzioni del corpo del ciclo e solo successivamente verrà tentata una nuova lettura dal file: se questo conterrà ancora dati, si darà luogo alla loro elaborazione; in caso contrario `tabfile.eof()` restituirà il valore **true** che, combinato con l'operatore di negazione `!`, verrà valutato come **false** e pertanto interromperà l'esecuzione del ciclo.

OSSERVAZIONE In situazioni simili a quella dell'esempio precedente è necessario porre particolare attenzione al fatto che le righe del file, in particolare l'ultima, non siano vuote (cioè contenenti solo <CR><LF>). Infatti una linea del tipo:

....
Bianchi;Giovanni;Via Ferruccio, 7;55100;Firenze<CR><LF>
<CR><LF>

impedisce il corretto funzionamento del programma, facendolo terminare in maniera anomala.

Sintesi

File. Insieme di dati correlati, identificato da un nome, memorizzato permanentemente su un supporto di memoria non volatile e avente vita indipendente dal programma (o dai programmi) utilizzato/i per la sua creazione e/o modifica.

Organizzazione sequenziale. Organizzazione di un file in cui i dati vengono scritti uno dopo l'altro e possono essere letti solo nell'ordine in cui sono stati scritti.

Operazioni su file. Le operazioni di base effettuabili su di un file ad accesso sequenziale sono: apertura, lettura o scrittura, chiusura.

Buffer. Area di memoria volatile riservata dal sistema operativo all'atto dell'apertura di un file per gestire lo scambio dei dati tra programma e memoria non volatile; l'operazione di chiusura di un file aperto in scrittura fa sì che eventuali dati contenuti nel *buffer* siano copiati nella memoria non volatile prima che questo venga rilasciato.

Nome fisico. È il nome con cui il file viene salvato sul supporto di memorizzazione, per esempio `"file.txt"`; di solito esso comprende il *pathname*, ovvero la stringa che indica il percorso delle cartelle in cui è contenuto il file a partire dall'identificatore del dispositivo di memoria.

Nome logico. È il nome che nell'ambito di un programma viene associato a un file fisico all'atto della sua apertura; nel programma si fa sempre riferimento al nome logico ogni volta che è necessario effettuare un'operazione sul file.

File sequenziali testuali in C++. In C++ sono previsti tre diversi tipi di file sequenziali testuali: *ifstream* (solo in lettura), *ofstream* (solo in scrittura), *fstream* (a ogni apertura è possibile scegliere una sola modalità tra: lettura, scrittura o accodamento). Si ricorda che, aprendo un file in scrittura, nel caso in cui questo sia già esistente, il suo eventuale contenuto preesistente viene irrimediabilmente perso.

Metodi base e operatori. I metodi base applicabili a oggetti di tipo *ifstream*, *ofstream* o *fstream* sono *open()* per l'apertura, *close()* per la chiusura, *eof()* per verificare il raggiungimento della fine di un file aperto in lettura. Gli operatori >> (operatore di lettura) e << (operatore di scrittura) servono rispettivamente a effettuare letture/scritture da/in un file sequenziale.

Elaborazione dei dati in input da un file sequenziale. Indipendentemente dal tipo di elaborazione che deve essere applicata a un file sequenziale che contiene dati, la struttura generale del codice che la implementa deve essere articolata su:

- apertura del file;
- una prima istruzione di lettura per verificare se il file contiene o meno dati;
- un ciclo indeterminato con controllo in testa che implementa l'elaborazione vera e propria dei dati, avente come criterio di terminazione la raggiunta fine del file (*eof()*) e come ultima istruzione una nuova lettura dal file, in modo tale da verificare se l'ultimo dato elaborato fosse o meno anche l'ultimo contenuto nel file.

Caratteri speciali. Nei file testuali sono utilizzati alcuni caratteri con significato speciale: i caratteri <CR> e <LF> (codici ASCII decimali 13 e 10) da soli o in sequenza sono utilizzati per indicare la fine di una riga, il carattere di tabulazione (codice ASCII decimale 9) e lo spazio (codice ASCII decimale 32) sono utilizzati come separatori dei dati nell'ambito di una singola riga.

File in formato CSV (*Comma Separated Values*). File testuali utilizzati per l'importazione e l'esportazione di una tabella di dati da applicazioni software per la gestione dei fogli di calcolo o dei database.

QUESITI

1 **Un file è un insieme di dati …**

A … memorizzati nella memoria volatile.
B … memorizzati in una memoria non volatile e avente vita dipendente dai programmi utilizzati per la sua gestione.
C … memorizzati in una memoria non volatile e avente vita indipendente dai programmi utilizzati per la sua gestione.
D … memorizzati in un'area della memoria denominata *buffer*.

2 **La gestione fisica di un file su un dispositivo di memorizzazione non volatile è di competenza …**

A … dei programmi usati per la sua gestione.
B … del sistema operativo.
C … della cartella che lo contiene.
D Nessuna delle risposte precedenti.

3 **Quale delle seguenti non è un'operazione applicabile a un file C++ di tipo *fstream* aperto in modalità *ios::app*?**

A Apertura.
B Chiusura.
C Lettura.
D Scrittura.

4 **La modalità *append* per un file C++ di tipo *fstream* prevede la possibilità di operare in …**

A … lettura e scrittura.
B … solo lettura.
C … solo scrittura.
D Nessuna delle risposte precedenti.

5 **Il *buffer* associato a un file è …**

A … un'area di memoria volatile tramite la quale avviene lo scambio di dati tra un programma e la memoria non volatile.
B … un'area di memoria volatile di ausilio alle sole operazioni di scrittura dei dati in un file.
C … un'area di memoria non volatile riservata allo specifico file.
D Nessuna delle risposte precedenti.

6 **Un file sequenziale di tipo testuale permette di accedere ai suoi dati con modalità …**

A … diretta sul singolo elemento.
B … strettamente sequenziale.
C … sia diretta sia sequenziale.
D Nessuna delle risposte precedenti.

7 **Supponendo di essere arrivati a leggere l'*n*-esimo dato di un file ad accesso sequenziale, per leggere il dato precedente è necessario …**

A … effettuare una banale lettura all'indietro.
B … effettuare due letture all'indietro e quindi una in avanti.
C … chiudere il file, riaprirlo e quindi effettuare $n - 1$ letture.
D Nessuna delle risposte precedenti.

8 **In generale i caratteri utilizzati per separare i singoli dati all'interno di un file sequenziale di tipo testuale sono:**

A tabulazione.
B «,» o «;».
C spazio.
D <CR> e/o <LF>.

9 **Volendo inserire un dato in posizione *n*-esima in un file di tipo *fstream* aperto in modalità *ios::out* è necessario …**

A … leggere i primi $n - 1$ dati e quindi effettuare la scrittura del nuovo dato tramite l'operatore di inserimento «<<».
B … l'operazione non è possibile; sarebbe infatti necessario chiudere e riaprire il file in input e un nuovo file in output, leggere i primi $n - 1$ dati dal primo file scrivendoli di volta in volta nel secondo, scrivere in quest'ultimo il nuovo dato, continuare a leggere i rimanenti dati dal primo file per scriverli nel secondo.
C … spostare i dati del file di una posizione in avanti a partire dalla $n + 1$-esima e quindi scrivere in posizione n-esima il nuovo dato.
D Nessuna delle risposte precedenti.

10 Volendo leggere uno o più dati da un file C++ di tipo *fstream* aperto in modalità *ios::out* è necessario …

A … chiudere il file e quindi riaprirlo in modalità *ios::in*, oppure in modalità *ios::app*.

B … utilizzare banalmente l'operatore «>>».

C … chiudere il file e quindi riaprirlo in modalità *ios::in* ed effettuare le letture volute tramite l'operatore «>>».

D Nessuna delle risposte precedenti.

11 Per leggere l'*n*-esimo dato di un file sequenziale testuale in C++ è necessario …

A … effettuare *n* letture.

B … effettuare solo una lettura del dato in posizione *n*.

C … effettuare *n* + 1 letture.

D Nessuna delle risposte precedenti.

12 Aprire e scrivere in output su un file sequenziale comporta …

A … la perdita dell'eventuale contenuto nel caso si tratti di un file preesistente.

B … in ogni caso l'accodamento dei nuovi dati anche se si tratta di un file preesistente.

C … l'accodamento dei nuovi dati se si tratta di un file preesistente aperto in modalità *ios::app*.

D Nessuna delle risposte precedenti.

13 Il metodo *eof()* applicato a un file sequenziale ritorna il valore `true` se …

A … è stata raggiunta la fine del file in fase di scrittura.

B … è stata raggiunta la fine del file in fase di lettura.

C … è stata raggiunta la fine del file sia che ci si trovi in fase di scrittura che di lettura.

D Nessuna delle risposte precedenti.

14 Il nome fisico di un file è …

A … quello che comprende il suo *pathname* completo.

B … quello che comprende il suo *pathname* completo o relativo.

C … quello utilizzato nel programma nelle varie operazioni di lettura e/o scrittura.

D Nessuna delle risposte precedenti.

15 In generale, nella lettura del contenuto di un file sequenziale, è buona norma prevedere …

A … un ciclo indeterminato che contenga come prima istruzione la lettura dal file.

B … un ciclo indeterminato che non contenga alcuna istruzione di lettura dal file.

C … una lettura esterna e un ciclo indeterminato che contenga come ultima istruzione la lettura dal file.

D Nessuna delle risposte precedenti.

16 In generale, nella lettura del contenuto di un file sequenziale, è buona norma prevedere …

A … un ciclo indeterminato con controllo in testa per verificare la raggiunta o meno fine del file.

B … un ciclo indeterminato con controllo in coda per verificare la raggiunta o meno fine del file.

C … un ciclo determinato perché in ogni caso è possibile conoscere a priori il numero delle letture da eseguire.

D Nessuna delle risposte precedenti.

ESERCIZI

Scrivere e verificare i programmi C++ che risolvano i seguenti problemi.

1 Data una sequenza di interi forniti dall'utente (il valore 0 interrompe la digitazione) memorizzare i soli multipli di 5 nel file di testo `"multipli5.txt"`.

2 Data una sequenza di interi forniti dall'utente (il valore 0 interrompe la digitazione), memorizzare i numeri positivi nel file `"positivi.txt"` e quelli negativi nel file `"negativi.txt"`.

3 Dato un file `"numeri.txt"` contenente valori interi e un valore intero *N* inserito dall'utente, me-

morizzare i valori contenuti nel file minori di *N* in un nuovo file `"minori.txt"` e quelli maggiori in un nuovo file `"maggiori.txt"`.

4 A partire da un file sequenziale «Alfa.txt», contenente un insieme di numeri interi ordinato in modo crescente, e da un valore *N* inserito dall'utente, creare un nuovo file `"Beta.txt"` contenente l'insieme dei dati di `"Alfa.txt"` più il valore *N*, in modo che anch'esso risulti ordinato in modo crescente.

5 Dato un file `"Testo.txt"` e una parola acquisita dallo standard input, ricercare la parola all'interno del file e visualizzare il numero delle occorrenze della parola nel testo contenuto nel file.

6 Dato un file di testo `"Testo.txt"` contenente stringhe di caratteri (ogni riga del file è lunga al massimo da 256 caratteri), creare un file di testo `"Maiuscolo.txt"` che contenga lo stesso testo contenuto nel primo file trasformato in lettere maiuscole (esempio: la parola «Aereo» del primo file sarà trasformata nel secondo file in «AEREO»).

7 Si intende simulare il gioco della battaglia navale. Un primo file `"Navi.txt"` contiene una mappa di dimensione 10 × 10 sulla quale sono disposte 3 navi (una da 3 elementi, una da 2 e una da 1 elemento). La mappa è costituita da 10 righe e 10 colonne e prevede quattro tipi di simboli: 0 = acqua, 1 = componente di una nave da 1 elemento, 2 = componente di una nave da 2 elementi, 3 = componente di una nave da 3 elementi. Un secondo file `"Colpi.txt"` contiene la serie di colpi effettuati da un singolo giocatore: i colpi sono specificati, uno per riga, nel formato numero-riga numero-colonna; il numero di colpi, e quindi quello delle righe del secondo file, non è noto a priori. Il programma deve calcolare il punteggio ottenuto con i colpi specificati nel secondo file applicandoli alla mappa acquisita dal primo file: il punteggio è semplicemente ottenuto come somma dei colpi andati a segno; inoltre il programma deve eventualmente visualizzare l'elenco delle navi affondate (nave da 1, nave da 2 o nave da 3).

8 Si dispone in un file `"Risultati.txt"` in formato CSV di un elenco di nomi di studenti completati dal numero di matricola, dall'età e dal voto di una verifica scritta (esempio: Rossi Mario 910017 16 9). Dopo avere definito una tabella di elementi aventi una struttura adeguata a contenere i dati descritti, scrivere nel file `"Risultati_ordinati.txt"` i dati degli studenti ordinati in senso decrescente rispetto al voto conseguito.

9 Sia dato un file di testo in formato CSV `"Persone.txt"` che contiene i dati di una serie di persone, una per riga; più precisamente ogni riga contiene nell'ordine i seguenti dati separati da uno spazio:

- il cognome (non più di 20 caratteri senza spazi intermedi);
- il nome (non più di 20 caratteri senza spazi intermedi);
- l'anno di nascita;
- un carattere che indica il sesso (`"M"` o `"F"`).

Si definisca una tabella *Elenco* di elementi del tipo definito dalla struttura *Persona* adeguata a contenere i dati sopra descritti: si deve caricare la tabella mediante lettura dei dati delle persone dal file e visualizzarla. Successivamente si definisca la funzione booleana *Compatibili()* che, a partire dagli indici di posizione di due persone presenti in *Elenco*, restituisca `true` se le due persone hanno sesso diverso e differenza di età non superiore ai 5 anni, *False* altrimenti. Il programma dovrà produrre l'elenco delle coppie di persone compatibili nel file di testo `"Coppie.txt"` scrivendo i dati delle due persone in un'unica riga.

10 Una biblioteca ha identificato tutti i libri della propria collezione mediante un codice numerico. Si deve realizzare un programma che consenta di effettuare le seguenti operazioni memorizzando le informazioni relative a un libro (codice, titolo, autore/i, anno di pubblicazione, editore) in un vettore di strutture:

- aggiunta di un nuovo libro alla collezione;
- ordinamento dell'elenco in base al codice del libro;
- visualizzazione dell'elenco dei libri della collezione;
- ricerca delle informazioni relative a un libro a partire dal codice;
- ricerca delle informazioni relative a un libro a partire dal titolo;
- salvataggio della collezione in un file di testo;
- caricamento della collezione da un file di testo.

A11 Introduzione alla programmazione orientata agli oggetti

Dovendo gestire quantità numeriche di tipo frazionario nella tradizionale notazione con numeratore e denominatore, è possibile rappresentare una frazione mediante una struttura:

```
struct FRAZIONE
{
  int num; // numeratore
  int den; // denominatore
};
```

e realizzare – disponendo di una funzione per calcolare il MCD di due numeri interi – una collezione di funzioni che implementano le operazioni fondamentali tra frazioni:

```
void semplifica(FRAZIONE &f)
{
  int div = MCD(f.num, f.den);
  f.num = f.num/div;
  f.den = f.den/div;
  return;
}

FRAZIONE somma(FRAZIONE f1, FRAZIONE f2)
{
  FRAZIONE f;
  f.den = f1.den * f2.den;
  f.num = f1.num * f2.den + f2.num * f1.den;
  semplifica(f);
  return f;
}

FRAZIONE differenza(FRAZIONE f1, FRAZIONE f2)
{
  FRAZIONE f;
  f.den = f1.den * f2.den;
  f.num = f1.num * f2.den - f2.num * f1.den;
  semplifica(f);
  return f;
}
```

▶

```
FRAZIONE prodotto(FRAZIONE f1, FRAZIONE f2)
{
  FRAZIONE f;
  f.num = f1.num * f2.num;
  f.den = f1.den * f2.den;
  semplifica(f);
  return f;
}

FRAZIONE quoziente(FRAZIONE f1, FRAZIONE f2)
{
  FRAZIONE f;
  f.num = f1.num * f2.den;
  f.den = f1.den * f2.num;
  semplifica(f);
  return f;
}
```

1 Tipi di dato astratto in C++

La soluzione precedente è funzionale, ma presenta il limite di separare i dati (la struttura `FRAZIONE`) dalle operazioni che a essi si applicano (le funzioni `semplifica`, `somma`, `differenza`, `prodotto` e `quoziente`): per esempio non è semplice gestire la condizione per cui il denominatore di una frazione deve essere sempre diverso da 0.
La **programmazione orientata agli oggetti** risolve questa difficoltà introducendo il concetto di *tipo di dato astratto* (**ADT**, ***Abstract Data-Type***).

▶ Un **tipo di dato astratto** è un insieme di *valori* e di *operazioni* definite su di essi in modo indipendente dalla loro implementazione.

OSSERVAZIONE L'indipendenza delle operazioni dalla loro implementazione comporta la necessità di prevedere un'**interfaccia** astratta, indipendente dalla soluzione che la implementa.

Il linguaggio di programmazione C++, così come altri linguaggi di programmazione orientati agli oggetti, consente di implementare gli ADT mediante il concetto di *classe*.

▶ Una **classe** in C++ è una estensione del concetto di struttura (è quindi un «tipo» utilizzabile per la definizione di variabili che in questo caso prendono il nome di **oggetti**) che comprende oltre ai campi, in questo caso denominati **attributi**, anche le funzioni, note come **metodi**, che consentono di operare su di essi.

Il tipo di dato astratto relativo alla frazione può essere implementato in C++ mediante una classe che presenta la seguente interfaccia:

Linguaggi per la programmazione orientata agli oggetti

Il primo linguaggio di programmazione orientato agli oggetti è stato *Simula* nel 1967. Esso, e i successivi *Smalltalk* (1972) ed *Eiffel* (1985), hanno avuto una diffusione limitata, ma hanno contribuito alla definizione di un paradigma di programmazione che, a partire dagli anni '90 del secolo scorso, è diventato imperante.
Il linguaggio C++ è nato nel corso degli anni '80, come evoluzione del linguaggio C, nei laboratori «Bell» della AT&T per opera di Bjarne Stroustrup, che intendeva estendere il C con funzionalità orientate agli oggetti. Oggi è, insieme a *Java*, uno dei linguaggi di programmazione più diffusi al mondo.

```
class Frazione
{
  private:

  /* Attributi */
  int num; // numeratore
  int den; // denominatore

  void semplifica();

  public:

  /* Costruttori */
  Frazione(void);  // inizializza una frazione
                   // nulla
  Frazione(int n, int d);  // inizializza
                           // una frazione

  /* Metodi */
  void setNum(int n);  // imposta il valore
                       // del numeratore
  void setDen(int d);  // imposta il valore
                       // del denominatore
  int getNum(void);    // recupera il valore
                       // del numeratore
  int getDen(void);    // recupera il valore
                       // del denominatore
  Frazione somma(Frazione f);
  Frazione differenza(Frazione f);
  Frazione prodotto(Frazione f);
  Frazione quoziente(Frazione f);
};
```

ATTRIBUTI

METODI

L'interfaccia della classe espone gli attributi e i metodi che essa definisce, nascondendo il codice che ne implementa le funzionalità.

OSSERVAZIONE La definizione della classe `Frazione` è suddivisa in due sezioni: la sezione privata (preceduta dalla parola chiave **`private`**) risulterà accessibile esclusivamente dal codice dei metodi della classe, mentre la sezione pubblica (preceduta dalla parola chiave **`public`**) sarà accessibile anche dal codice che dichiarerà gli oggetti di tipo `Frazione`.

Per questo motivo i valori degli attributi risulteranno non direttamente accessibili e sono stati definiti specifici metodi di accesso che – seguendo una diffusa tradizione tra i programmatori – assumono i nomi ***get*** e ***set*** seguiti dal nome dell'attributo il cui valore consentono di impostare o di recuperare.

Il codice dei metodi della classe può essere definito in un file separato[1]:

1. Una valida consuetudine prevede che la classe sia definita in un file di intestazione con estensione .H e implementata in un file di codice con intestazione .CPP e che entrambi i file abbiano come nome il nome della classe stessa.

```
/* Metodi privati */

void Frazione::semplifica()
{
  int div;

  if (num == 0)
    return;
  div = MCD(den,num);
  num = num/div;
  den = den/div;
}

/* Costruttori */

Frazione::Frazione(void)
{
  num = 0;
  den = 1;
}

Frazione::Frazione(int n, int d)
{
  if (d != 0)
    {
      num = n;
      den = d;
      semplifica();
    }
  else
    {
      num = 0;
      den = 1;
    }
}

/* Metodi di accesso */

void Frazione::setNum(int n)
{
  num = n;
  semplifica();
}

void Frazione::setDen(int d)
{
  if (d != 0)
    {
```

▶

```
      den = d;
      semplifica();
    }
}

int Frazione::getNum(void)
{
  return num;
}

int Frazione::getDen(void)
{
  return den;
}

/* Metodi di implementazione delle operazioni */

Frazione Frazione::somma(Frazione f)
{
  Frazione r;

  r.setDen(den*f.getDen());
  r.setNum(num*f.getDen() + f.getNum()*den);
  return r;
}

Frazione Frazione::differenza(Frazione f)
{
  Frazione r;

  r.setDen(den*f.getDen());
  r.setNum(num*f.getDen() - f.getNum()*den);
  return r;
}

Frazione Frazione::prodotto(Frazione f)
{
  Frazione r;

  r.setNum(num*f.getNum());
  r.setDen(den*f.getDen());
  return r;
}

Frazione Frazione::quoziente(Frazione f)
{
  Frazione r;

  r.setNum(num*f.getDen());
  r.setDen(den*f.getNum());
  return r;
}
```

OSSERVAZIONE Quando sono implementati esternamente alla definizione della classe, i metodi sono sempre preceduti dal nome della classe separato dal nome del metodo con il simbolo «::».

Una classe definisce un tipo e come tale è inutilizzabile direttamente nel codice, ma ha il solo scopo di dichiarare (il termine tecnico è **istanziare**) gli oggetti che sono successivamente utilizzati come variabili. Il **costruttore** di una classe è un metodo speciale – ha sempre il nome stesso della classe e non può restituire valori – che viene invocato automaticamente al momento della dichiarazione di un oggetto del tipo definito dalla classe stessa con lo scopo di inizializzarne gli attributi.

ESEMPIO

Il seguente frammento di codice C++ definisce tre oggetti di tipo `Frazione` inizializzando i primi due rispettivamente ai valori 2/3 e 3/2; il terzo oggetto viene utilizzato per il risultato del prodotto dei primi due:

```
Frazione x(2,3), y(3,2), z;
z = x.prodotto(y);
```

OSSERVAZIONE In realtà nell'esempio precedente anche l'oggetto *z* viene inizializzato; infatti la classe `Frazione` definisce due distinti costruttori:

```
Frazione(int n, int d);
Frazione(void);
```

Il primo permette di impostare i valori del numeratore *n* e del denominatore *d*, mentre il secondo consente di non specificare alcun valore del numeratore e del denominatore, ma il codice che lo implementa li inizializza rispettivamente ai valori 0 e 1. In generale, come abbiamo già avuto modo di vedere nel capitolo dedicato alle funzioni, la possibilità offerta dal linguaggio C++ di definire metodi distinti esclusivamente dal tipo e dal numero dei parametri è nota come ***overloading***; essa è particolarmente utile nel caso dei costruttori.

OSSERVAZIONE Sempre nell'esempio precedente si noti l'istruzione di assegnazione che calcola il prodotto delle frazioni *x* e *y* assegnando il risultato alla frazione *z*:

```
z = x.prodotto(y);
```

Il metodo di un oggetto viene invocato come una funzione, ma con la stessa sintassi di accesso ai campi di una struttura, utilizzando il simbolo «.». Inoltre la possibilità garantita al codice dei metodi di accedere direttamente agli attributi permette di fornire al metodo come parametro solo la «seconda» frazione (*y*) coinvolta nell'operazione, essendo la «prima» frazione l'oggetto stesso (*x*) che, oltre ai valori degli attributi che rappresentano il numeratore e il denominatore, rende disponibile anche il codice per eseguire l'operazione.

ESEMPIO

Il seguente programma di verifica delle funzionalità della classe `Frazione` esemplifica la dichiarazione e l'uso di oggetti istanze della classe stessa:

```
void main(void)
{
  Frazione x, y, z;
  int t;

  cout<<"Inserisci numeratore X: ";
  cin>>t;
  x.setNum(t);
  cout<<"Inserisci denominatore X: ";
  cin>>t;
  x.setDen(t);
  cout<<"X = "<<x.getNum()<<"/"<<x.getDen()<<endl;
  cout<<"Inserisci numeratore Y: ";
  cin>>t;
  y.setNum(t);
  cout<<"Inserisci denominatore Y: ";
  cin>>t;
  y.setDen(t);
  cout<<"Y = "<<y.getNum()<<"/"<<y.getDen()<<endl;

  z = x.somma(y);
  cout<<"X + Y = "<<z.getNum()<<"/"<<z.getDen()<<endl;
  z = x.differenza(y);
  cout<<"X - Y = "<<z.getNum()<<"/"<<z.getDen()<<endl;
  z = x.prodotto(y);
  cout<<"X * Y = "<<z.getNum()<<"/"<<z.getDen()<<endl;
  z = x.quoziente(y);
  cout<<"X / Y = "<<z.getNum()<<"/"<<z.getDen()<<endl;
  return;
}
```

OSSERVAZIONE Gli operatori di input/output del linguaggio C++ consentono di inserire e di visualizzare valori di tipo predefinito, di conseguenza nel programma dell'esempio precedente si ricorre estensivamente ai metodi di accesso della classe che permettono di impostare e recuperare i singoli valori del numeratore e del denominatore come valori interi.

L'appesantimento sintattico tipico dei programmi che rispettano le convenzioni della programmazione orientata agli oggetti è in particolare dovuto alla necessità di utilizzare in modo consistente i metodi di accesso a causa della natura «privata» degli attributi degli oggetti.

ESEMPIO

Nel caso in cui gli attributi della classe `Frazione` non fossero stati definiti come privati (è sufficiente spostarne la dichiarazione nella sezione della classe che segue la parola chiave `public`) il programma dell'esempio precedente poteva essere codificato nel seguente modo: ▶

```
void main(void)
{
  Frazione x, y, z;

  cout<<"Inserisci numeratore X: ";
  cin>>x.num;
  cout<<"Inserisci denominatore X: ";
  cin>>x.den;
  x.semplifica();
  cout<<"X = "<<x.num<<"/"<<x.den()<<endl;
  cout<<"Inserisci numeratore Y: ";
  cin>>y.num;
  cout<<"Inserisci denominatore Y: ";
  cin>>y.den;
  y.semplifica();
  cout<<"Y = "<<y.num<<"/"<<y.den<<endl;

  z = x.somma(y);
  cout<<"X + Y = "<<z.num<<"/"<<z.den<<endl;
  z = x.differenza(y);
  cout<<»X - Y = «<<z.num<<»/»<<z.den<<endl;
  z = x.prodotto(y);
  cout<<»X * Y = «<<z.num<<»/»<<z.den<<endl;
  z = x.quoziente(y);
  cout<<"X / Y = "<<z.num<<"/"<<z.den<<endl;
  return;
}
```

Principi della programmazione orientata agli oggetti

La programmazione orientata agli oggetti è basata su tre principi fondamentali:

- incapsulamento;
- ereditarietà;
- polimorfismo.

L'**incapsulamento** prevede il raggruppamento in un'unica entità (la classe) sia delle strutture dati (attributi) sia delle procedure (metodi) che operano su di esse.
L'**ereditarietà** consente di definire nuove classi a partire da altre classi ereditandone attributi e metodi, eventualmente aggiungendone di nuovi o ridefinendo quelli esistenti.
Il **polimorfismo** permette di manipolare oggetti di classi diverse in relazione di ereditarietà, ottenendo automaticamente in fase di esecuzione comportamenti diversi per specifici metodi in base alla classe di appartenenza dell'oggetto.

La pratica di definire gli attributi di una classe privati può apparire a prima vista inutilmente complessa, ma garantire che l'accesso ai valori degli attributi avvenga esclusivamente mediante i metodi (e quindi attraverso il filtro dell'esecuzione del codice) consente di gestire in modo coerente aspetti che sarebbero altrimenti demandati alla correttezza del codice che utilizza gli oggetti. Nella classe `Frazione`, per esempio, il problema del denominatore che deve essere diverso da zero è gestito in modo adeguato dal codice dei costruttori e dei metodi di accesso, ma l'eventuale accesso incontrollato ai valori degli attributi renderebbe vani i controlli operati.

La pratica di accedere agli attributi di una classe esclusivamente mediante i suoi metodi è un modo per mettere in pratica l'importante principio di ***information hiding*** (**occultamento dell'informazione**), che stabilisce la netta separazione tra l'implementazione e l'interfaccia di un oggetto.

Nel linguaggio di programmazione C++ un ulteriore modo di rispettare tale principio consiste nella separazione tra la definizione dei metodi di una classe e il codice che ne implementa le funzionalità. In questo modo, infatti, un eventuale cambiamento del codice dei metodi che ne rispetti la semantica non avrà nessuna ripercussione sul codice che dichiara e utilizza

gli oggetti istanze della classe. Questa tecnica di programmazione è spesso denominata **incapsulamento** ed è una delle tecniche base della programmazione orientata agli oggetti. Dal momento che nella programmazione orientata agli oggetti si mantiene una stretta relazione tra i dati e le procedure che agiscono su di essi, una forma di applicazione dell'incapsulamento è data dalla pratica di **comprendere in un unico oggetto i dati e le azioni a essi applicabili**.

ESEMPIO

I dati che caratterizzano un conto bancario (attributi) sono il numero del conto, la data di apertura e/o chiusura, le generalità del cliente, il saldo attuale, il tasso di interesse ecc. Le azioni (metodi) che su tali dati operano sono: apertura, chiusura, deposito, prelevamento, modifica dei dati del cliente, aggiornamento del tasso di interesse ecc.
Un oggetto «conto» *incapsula* in un'unica entità i dati e le azioni relative: in questo modo un qualsiasi aggiornamento al sistema informatico di esercizio della banca che comporti una modifica della gestione dei conti correnti richiederà di intervenire semplicemente sulla classe che descrive gli oggetti «conto», senza interessare il resto del sistema.

2 Code e pile come tipi di dato astratto

Dovendo realizzare un sistema informatico per gestire la fila di attesa di un servizio, è necessario implementare una *coda*.

▶ Una **coda** è una struttura dati che realizza una politica di tipo **FIFO**, ***First-In First-Out*** (il primo che entra è il primo a uscire).

ESEMPIO

Supponendo che una banca doti i propri clienti di una tessera elettronica di identificazione (per motivi di privacy i clienti saranno identificati da un codice numerico e non dai dati anagrafici) e posizioni ai propri ingressi un dispositivo di lettura delle tessere, è possibile fare in modo che sia visualizzato il codice numerico del cliente al momento che uno sportello si libera per riceverlo, gestendo una coda dei clienti in attesa.

Le operazioni fondamentali che si possono eseguire su una coda di elementi (che nel caso dell'esempio sono valori interi, i codici dei clienti della banca) sono due:

- inserire un nuovo elemento in fondo alla coda (*push*, spingere);
- estrarre il primo elemento dalla coda (*pop*, sfilare).

La seguente classe C++ definisce una coda di valori interi in cui possono essere inseriti al massimo 1000 elementi memorizzati in un vettore:

```
class Coda
{
  private:

  int elementi[1000];
  int primo;
  int ultimo;
  int lunghezza;

  public:

  Coda(void); // costruttore
  void push(int elemento);
  int pop(void);
  int getLunghezza(void);
};

Coda::Coda(void)
{
  primo = 0;
  ultimo = 0;
  lunghezza = 0;
}

void Coda::push(int elemento)
{
  // si inserisce l'elemento solo se la coda non e' piena
  if (lunghezza < 1000)
    {
      elementi[ultimo] = elemento;
      ultimo++;
      if (ultimo >= 1000)
      ultimo = 0;
      lunghezza++;
    }
}

int Coda::pop(void)
{
  int elemento;

  if (lunghezza > 0)
    {
      lunghezza--;
      elemento = elementi[primo];
      primo++;
      if (primo >= 1000)
        primo = 0;
```

▶

```
        return elemento;
      }
    // se la coda e' vuota si restituisce il valore zero
    return 0;
}

int Coda::getLunghezza(void)
{
    return lunghezza;
}
```

Il vettore *elementi* viene utilizzato per memorizzare gli elementi inseriti nella coda: gli attributi *primo* e *ultimo* rappresentano rispettivamente gli indici dell'elemento in testa alla coda (il prossimo da estrarre) e del prossimo elemento da inserire; l'attributo *lunghezza* contiene il numero di elementi presenti nella coda.
La FIGURA **1** mostra, a partire dalla situazione iniziale determinata dal costruttore, lo stato degli attributi dopo uno, due, tre inserimenti e una estrazione.

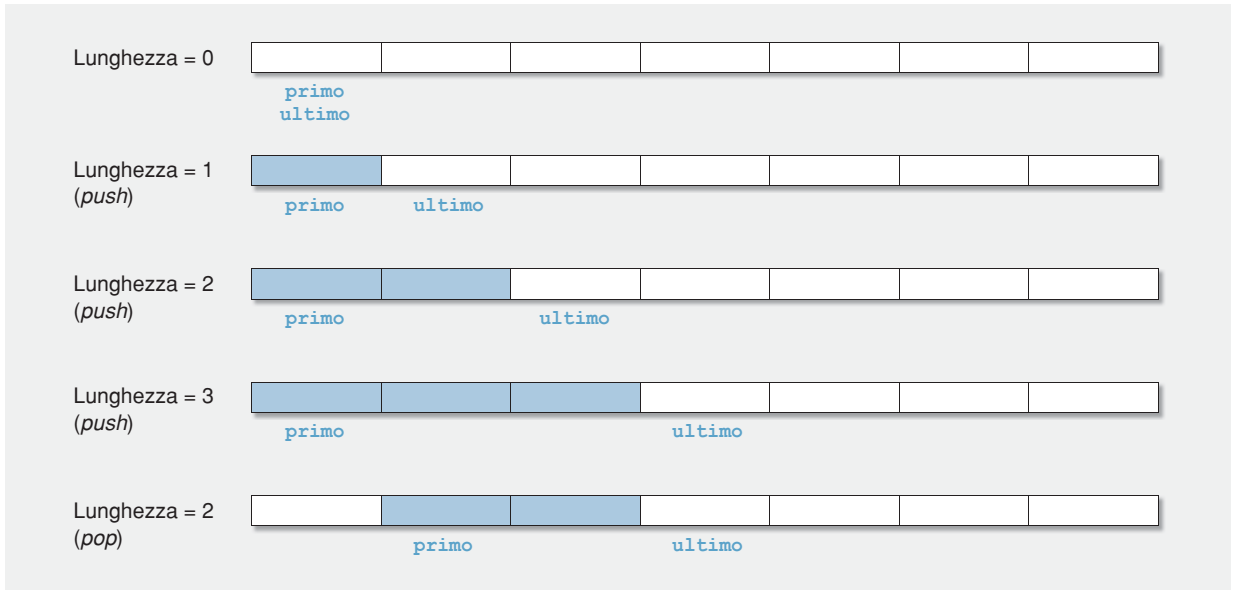


FIGURA **1**

Una volta raggiunta la dimensione del vettore *elementi* gli indici sono incrementati in modo da riutilizzare ciclicamente i primi elementi ormai disponibili (FIGURA **2**).

OSSERVAZIONE Il fatto che la coda sia realizzata come tipo di dato astratto mediante una classe C++ che assicura la privatezza degli attributi applicando il principio di *information hiding* è essenziale per garantirne la correttezza del funzionamento: infatti un accesso incontrollato agli indici *primo* e *ultimo* e alla variabile *lunghezza* avrebbe come conseguenza la corruzione dello stato della coda.

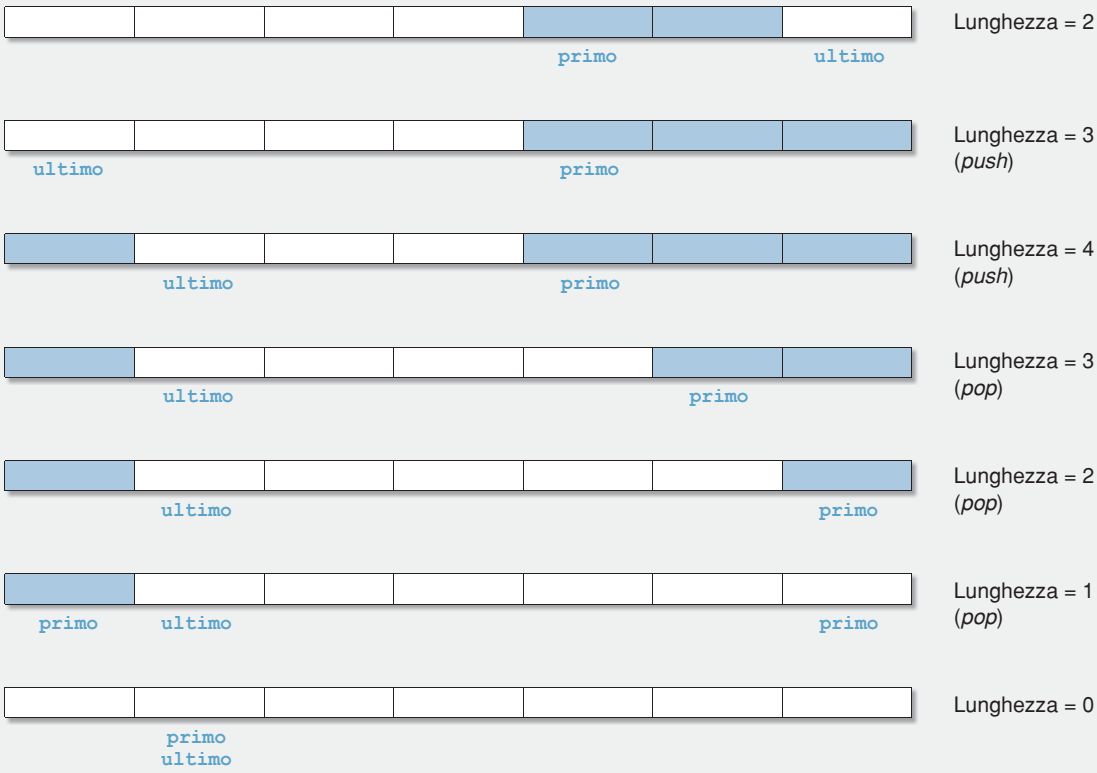


FIGURA **2**

ESEMPIO

Il seguente frammento di codice C++ esemplifica la dichiarazione e l'uso di un oggetto istanza della classe `Coda`:

```
...
Coda q;
...
q.push(…);
q.push(…);
...
while (q.getLunghezzza() > 0)
   {
     int e = q.pop();
     ...
   }
...
```

▶ Si definisce **pila** (***stack***) una struttura dati che implementa la politica **LIFO**, ***Last-In First-Out*** (l'ultimo a entrare è il primo a uscire).

La seguente classe C++ realizza una pila di elementi numerici di tipo **`float`**:

```
class Pila
{
  private:

  float elementi[1000];
  int lunghezza;

  public:

  Pila(void); // costruttore
  void push(float elemento);
  float pop(void);
  int getLunghezza(void);
};

Pila::Pila(void)
{
  lunghezza = 0;
}

void Pila::push(float elemento)
{
  // si inserisce l'elemento solo se la pila non e' piena
  if (lunghezza < 1000)
    {
      elementi[lunghezza] = elemento;
      lunghezza++;
    }
}

float Pila::pop(void)
{
  if (lunghezza > 0)
    {
      lunghezza--;
      return elementi[lunghezza];
    }
  // se la coda e' vuota si restituisce il valore zero
  return 0;
}

int Pila::getLunghezza(void)
{
  return lunghezza;
}
```

Anche in questo il vettore *elementi* viene utilizzato per memorizzare gli elementi inseriti nella pila, ma non è necessario utilizzarlo in modo circo-

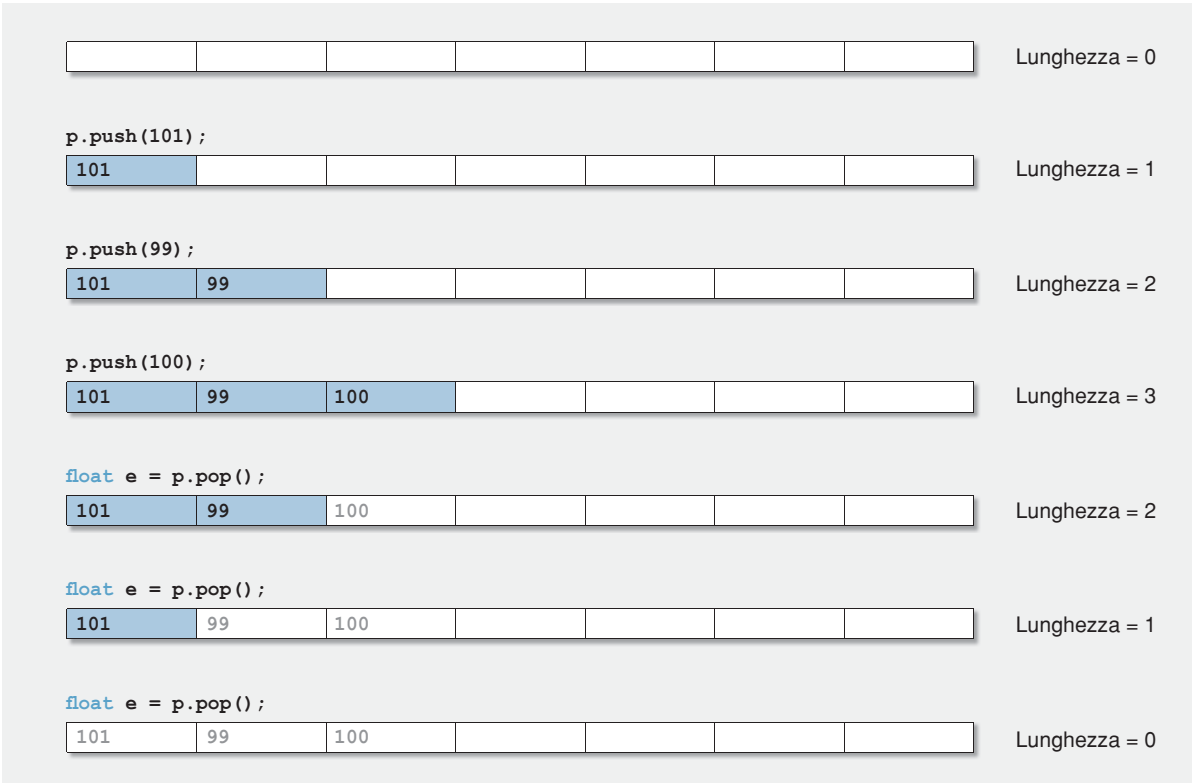


FIGURA **3**

lare, in quanto il primo elemento inserito rimane sempre nella posizione di indice 0 e l'ultimo elemento inserito (il primo da estrarre) nella posizione di indice *lunghezza* – 1.

La FIGURA **3** mostra, a partire dalla situazione iniziale determinata dal costruttore, lo stato della pila determinato dalle invocazioni dei metodi illustrate per un oggetto *p* istanza della classe `Pila`.

3 Il dimensionamento dinamico degli attributi in C++

In particolare, nelle classi che implementano strutture dati – come avviene nel caso delle code e delle pile – è utile poter definire al momento della dichiarazione dell'oggetto la dimensione massima in termini di elementi. È naturale pensare di passare la dimensione al costruttore come parametro:

```
Coda c(100); // oggetto di classe Coda che contiene 100 elementi
```

Per rendere possibile questa pratica all'interno della classe, l'*array* che implementa la struttura dati non può essere definito direttamente, ma solo come **puntatore**, utilizzando la seguente sintassi:

```
tipo* nome;
```

Indirizzi e puntatori in C/C++

I **puntatori** (identificati dal simbolo «*») sono una caratteristica importante del linguaggio C/C++.

Essi consentono di manipolare gli **indirizzi** di memoria delle variabili (l'operatore «&» premesso al nome di una variabile ne rappresenta l'indirizzo) e sono utilizzati a vari scopi, per esempio come alternativa al passaggio di parametri per riferimento, per consentire la variazione nel codice invocante dei valori delle variabili fornite come argomento a ▶

► una funzione che ne modifica il contenuto.

I progettisti del linguaggio C hanno scelto di identificare sintatticamente il nome di un puntatore con il nome di un *array* a esso associato (il puntatore in questo caso rappresenta l'indirizzo del primo elemento del vettore). Questa scelta è una delle caratteristiche originali del linguaggio, confermata in C++ dalla possibilità di usare l'operatore `new` per il dimensionamento dell'*array*.

ESEMPIO

La seguente dichiarazione è relativa a un puntatore per un *array* di valori di tipo `int`:

```
int* elementi;
```

Un *array* definito come puntatore non può essere utilizzato fino a che non viene inizializzato mediante l'operatore `new`, che associa al nome dell'*array* lo spazio di memoria necessario per la memorizzazione dei suoi elementi. L'operatore `new` viene utilizzato impiegando la seguente sintassi:

```
puntatore = new tipo[dimensione];
```

dove *puntatore* è il nome dell'*array* dichiarato in precedenza come puntatore, *tipo* è il tipo degli elementi dell'*array* e *dimensione* è la dimensione dell'*array*.

ESEMPIO

La classe `Coda` può essere modificata come nel seguito per consentire al codice che la utilizza di stabilire il numero massimo di elementi che essa può contenere:

```
class Coda
{
  private:

  int* elementi;
  int dimensione;
  int primo;
  int ultimo;
  int lunghezza;

  public:

  Coda(void); // costruttore per dimensione predefinita (100 elementi)
  Coda(int dim); // costruttore per specificare il numero di elementi
  void push(int elemento);
  int pop(void);
  int getLunghezza(void);
};

Coda::Coda(void)
{
  dimensione = 100;
  elementi = new int[100]; // inizializzazione vettore di 100 elementi
  primo = 0;
  ultimo = 0;
  lunghezza = 0;
}

Coda::Coda(int dim)
{
  dimensione = dim;
```

►

```
  elementi = new int[dim]; // inizializzazione vettore di DIM elementi
  primo = 0;
  ultimo = 0;
  lunghezza = 0;
}

void Coda::push(int elemento)
{
  // si inserisce l'elemento solo se la coda non e' piena
  if (lunghezza < dimensione)
    {
      elementi[ultimo] = elemento;
      ultimo++;
      if (ultimo >= dimensione)
      ultimo = 0;
      lunghezza++;
    }
}

int Coda::pop(void)
{
  int elemento;

  if (lunghezza > 0)
    {
      lunghezza--;
      elemento = elementi[primo];
      primo++;
      if (primo >= dimensione)
        primo = 0;
      return elemento;
    }
  // se la coda e' vuota si restituisce il valore zero
  return 0;
}

int Coda::getLunghezza(void)
{
  return lunghezza;
}
```

Il codice che dichiara oggetti di classe `Coda` utilizzerà la seguente sintassi:

```
Coda c1(10); // coda con 10 elementi
Coda c2; // coda con 100 elementi (valore predefinito)
Coda c3(1000); // coda con 1000 elementi
```

Il costruttore di un oggetto viene eseguito al momento della creazione dell'oggetto stesso: se l'oggetto viene distrutto – come accade per esempio alle variabili locali di una funzione quando questa termina l'esecuzione – lo spazio di memoria associato a un puntatore allocato con l'operatore **new** resta inutilizzato e inutilizzabile, causando uno spreco di memoria. In C++ la

memoria richiesta al sistema operativo dall'operatore **new** deve essere rilasciata esplicitamente utilizzando l'operatore **delete** con la seguente sintassi:

```
delete puntatore;
```

dove *puntatore* è il nome associato all'*array*.

ESEMPIO

La seguente istruzione rilascia la memoria associata al puntatore denominato *elementi*:

```
delete elementi;
```

L'operatore **delete** deve essere utilizzato quando si ha la certezza che l'*array* associato al puntatore non verrà più utilizzato; rimane da individuare il metodo più opportuno per l'invocazione di **delete**.

In C++ esiste un metodo – denominato **distruttore** – che, se definito, viene invocato al momento in cui l'oggetto istanziato dalla classe viene distrutto. Il metodo distruttore assume il nome della classe preceduto dal simbolo «~» e non ha argomenti.

ESEMPIO

L'aggiunta di un distruttore alla classe `Coda` produce la seguente interfaccia:

```
class Coda
{
  private:

  int* elementi;
  int dimensione;
  int primo;
  int ultimo;
  int lunghezza;

  public:

  Coda(void); // costruttore per dimensione predefinita (100 elementi)
  Coda(int dim); // costruttore per specificare il numero di elementi
  ~Coda(void); // distruttore
  void push(int elemento);
  int pop(void);
  int getLunghezza(void);
};
```

Il metodo distruttore ha il compito di rilasciare la memoria allocata dal costruttore e ha la seguente implementazione:

```
Coda::~Coda(void)
{
  delete elementi;
}
```

OSSERVAZIONE Il metodo distruttore non deve essere invocato esplicitamente; esso viene invocato automaticamente nel momento in cui l'oggetto viene distrutto.

ESEMPIO

Il seguente programma di verifica delle funzionalità della classe `Coda` esemplifica la dichiarazione e l'uso di oggetti istanze della classe stessa:

```
void main(void)
{
  int d;
  char s;
  int e;

  cout<<"Inserisci la dimensione della pila: ";
  cin>>d;

  Coda c(d); // dichiarazione di oggetto istanza della classe Coda

  do
    {
      cout<<"I = inserimento"<<endl;
      cout<<"E = estrazione"<<endl;
      cout<<"L = lunghezza"<<endl;
      cout<<"U = uscita"<<endl;
      cin>>s;

      switch (s)
        {
          case 'i':
          case 'I': cout<<"Elemento: ";
                    cin>>e;
                    c.push(e);
                    break;

          case 'e':
          case 'E': e = c.pop();
                    cout<<"Elemento = "<<e<<endl;
                    break;

          case 'l':
          case 'L': cout<<"Lunghezza = "<<c.getLunghezza()<<endl;
                    break;

          case 'u':
          case 'U': return;
        }
    } while(true);

  return;
}
```

4 Un esempio di genericità in C++

Strutture dati come le code e le pile sono utili in molti contesti diversi; naturalmente il tipo di elementi che vi devono essere memorizzati varia di caso in caso, passando da un tipo predefinito del linguaggio C/C++ (`char`, `int`, `float`, …) a una struttura definita dall'utente programmatore.

ESEMPIO

Volendo realizzare un programma di gestione della coda dei clienti di un servizio che attendono il proprio turno per essere serviti, l'elemento da memorizzare nella coda potrebbe assumere un tipo come il seguente:

```
struct CLIENTE
{
  char nome[32];
  char cognome[32];
  char codice[16];
};
```

In generale l'elemento di una coda o di una pila può avere un tipo qualsiasi e si rende necessario copiare e adattare il codice di definizione e di implementazione delle classi `Coda` e `Pila` per ogni diverso tipo di valore per cui intendiamo utilizzarle.

Nel linguaggio C++ questo problema è stato risolto con l'adozione dei ***template***: è possibile definire una classe «generica» lasciando indefiniti uno o più tipi utilizzati dalla classe stessa, per definirli al momento della dichiarazione di un oggetto istanza della classe stessa.

La definizione della classe deve essere preceduta da una dichiarazione di *template* come la seguente:

```
template <class T1, class T2>
```

dove *T*1, *T*2 sono i tipi indefiniti. Al momento di dichiarare un oggetto istanza della classe, il nome della classe sarà seguito da un elenco di tipi come, per esempio, il seguente:

```
...<int, float>
```

In questo modo l'oggetto istanzia la classe come se nella sua definizione le occorrenze del tipo *T*1 siano il tipo `int` e le occorrenze del tipo *T*2 il tipo `float`.

ESEMPIO

Una classe pila generica – in cui cioè il tipo degli elementi inseribili nella pila è indefinito – può essere così definita:

```
template <class T>
class Pila
{
  private:
```

▶

```
  T* elementi;
  int dimensione;
  int lunghezza;

  public:

  Pila(int dim) // costruttore
  {
    dimensione = dim;
    elementi = new T[dim];
    lunghezza = 0;
  }

  ~Pila(void) // distruttore
  {
    delete elementi;
  }

  void push(T elemento)
  {
    // si inserisce l'elemento solo se la pila non e' piena
    if (lunghezza < dimensione)
      {
        elementi[lunghezza] = elemento;
        lunghezza++;
      }
  }

  T pop(void)
  {
    if (lunghezza > 0)
      {
        lunghezza--;
        return elementi[lunghezza];
      }
    // se la coda e' vuota si restituisce il valore zero
    return 0;
  }

  int getLunghezza(void)
  {
    return lunghezza;
  }

};
```

La classe può essere utilizzata per istanziare pile che contengono elementi di tipo diverso, semplicemente indicando il tipo *T* nella dichiarazione:

```
Pila<int> p1(99);
Pila<float> p2(101);
Pila<CLIENTE> p3(100);
```

Sintesi

Tipo di dato astratto (ADT, *Abstract Data Type*). Insieme di *valori* e di *operazioni* definite su di essi in modo indipendente dalla loro implementazione. Tale indipendenza comporta la necessità di prevedere un'*interfaccia astratta*, che prescinda dalla soluzione che la implementa.

Classe. Una classe in C++ è una estensione del concetto di struttura (è quindi un «tipo» utilizzabile per la definizione di variabili che in questo caso prendono il nome di oggetti) che comprende, oltre ai campi denominati *attributi*, anche i *metodi*, ovvero le funzioni che consentono di operare su di essi.

Oggetto. È una istanza di una classe e cioè una variabile dichiarata di un tipo definito da una classe.

Attributi. Relativamente a una classe, gli attributi ne costituiscono il contenuto informativo.

Metodi. I metodi di una classe ne rappresentano il codice, ovvero l'insieme delle funzioni che operano sui suoi attributi.

Interfaccia. Meccanismo mediante il quale una classe espone i suoi attributi e i suoi metodi, nascondendo il codice che ne implementa le funzionalità.

Information hiding. Questo principio stabilisce la separazione tra l'interfaccia di una classe e la sua implementazione interna.

Incapsulamento. Caratteristica fondamentale base della programmazione orientata agli oggetti che prevede il raggruppamento in un'unica entità (la classe) sia delle strutture dati (attributi) sia delle procedure (metodi) che operano su di esse. In base al principio di *information hiding* l'incapsulamento viene applicato mediando con specifici metodi l'accesso agli attributi di una classe.

Costruttore. Il costruttore di una classe è un metodo speciale che assume il nome stesso della classe e non può restituire valori. Esso viene invocato automaticamente al momento della creazione di un oggetto istanza della classe con lo scopo di inizializzarne gli attributi.

Distruttore. Metodo di una classe che, se definito, viene invocato al momento in cui l'oggetto istanziato a partire dalla classe stessa viene distrutto. Il metodo distruttore assume il nome della classe preceduto dal simbolo «~» e non prevede argomenti.

Operatori *new/delete*. Un dato (tipicamente un *array*) definito come puntatore non può essere utilizzato fino a che non viene inizializzato mediante l'operatore *new*, che associa al nome del dato lo spazio di memoria necessario per la memorizzazione dei suoi elementi. L'operatore *delete* consente di liberare la memoria assegnata mediante l'uso dell'operatore *new*.

Coda. Struttura dati di tipo astratto che realizza una politica di tipo FIFO, *First-In First-Out* (il primo che entra è il primo a uscire).

Pila. Struttura dati di tipo astratto che realizza una politica di tipo LIFO, *Last-In First-Out* (l'ultimo che entra è il primo a uscire).

Template. Classe «generica» la cui definizione lascia indefiniti uno o più tipi utilizzati dalla classe stessa per definirli al momento della dichiarazione di un oggetto istanza della classe.

QUESITI

1 **Un ADT (*Abstract Data-Type*) è …**

A … l'insieme dei dati con le operazioni su di essi definite.

B … un tipo per il quale non possono essere definite variabili concrete.

C … una classe con tutti gli attributi dichiarati come «privati».

D … un oggetto che consente di definire altri oggetti.

2 **L'applicazione della tecnica dell'incapsulamento ha come conseguenza che …**

A … l'accesso agli attributi di una classe deve essere effettuato mediante specifici metodi.

B … i nomi degli attributi non devono essere comprensibili.

C … la sezione privata di una classe deve essere separata dalla sezione pubblica.

D … i nomi delle variabili locali dei metodi nascondono i nomi degli attributi omonimi.

3 **Nel linguaggio C++ una classe è …**

A … una collezione di oggetti.

B … l'implementazione di un tipo di dato astratto.

C … una estensione del concetto di struttura che prevede la definizione di attributi e metodi.

D Nessuna delle risposte precedenti.

4 **Il *costruttore* di una classe è …**

A … uno speciale metodo che assume il nome della classe stessa invocato automaticamente all'atto della creazione di un oggetto istanza della classe per inizializzarne il valore degli attributi.

B … un riferimento identificativo specificato all'interno della classe dall'autore che l'ha sviluppata per rivendicare il diritto d'autore.

C … uno speciale attributo che assume il nome della classe stessa il cui valore serve a identificare e distinguere tra loro vari oggetti istanziati da quella classe.

D … uno speciale metodo (il cui nome è dato dalla classe stessa) che il codice che definisce oggetti istanza della classe deve invocare per inizializzarne il valore degli attributi.

5 **Due oggetti diversi istanziati a partire dalla stessa classe condividono …**

A … il nome e il valore degli attributi.

B … il nome e il codice dei metodi.

C … una parte del nome che ha come prefisso il nome della classe.

D Nessuna delle risposte precedenti.

6 **Sono date le seguenti due dichiarazioni:**

```
class C                 void main(void)
{                       {
  public:                 C o1, o2;
                          …
  int a1, a2;           }
}
```

A cosa equivale l'istruzione `o1=o2`?

A `C.a1=C.a2;`

B `o1.a1=o1.a2; o2.a1=o2.a2;`

C `a1.o1=a1.o2; a2.o1=a2.o2;`

D `o1.a1=o2.a1; o1.a2=o2.a2;`

7 **Il valore degli attributi dichiarati nella sezione privata di una classe …**

A … può essere modificato direttamente con una espressione del tipo

```
oggetto.attributo = espressione;
```

B … non può essere modificato.

C … può essere modificato solo dal costruttore.

D … può essere modificato mediante specifici metodi della classe stessa.

8 **Lo spazio di memoria assegnato agli attributi creati con l'operatore `new` viene liberato …**

A … automaticamente dal supporto a tempo di esecuzione del linguaggio C++.

B … solo utilizzando l'operatore `delete`.

C … automaticamente dal distruttore della classe.

D … solo al termine dell'esecuzione del programma.

9 **Associare le strutture dati con la corretta politica che esse implementano:**

Coda	FIFO
	FILO
Pila	LILO
	LIFO

10 **Nel linguaggio C++ un *template* è ...**

A … una classe con attributi e/o metodi i cui tipi o parametri sono definiti al momento della dichiarazione di un oggetto istanza della classe stessa.

B … una classe da cui non si possono istanziare oggetti.

C … una classe da cui è possibile istanziare un unico oggetto.

D Nessuna delle risposte precedenti.

ESERCIZI

Nota. Gli esercizi che seguono richiedono la progettazione e realizzazione di classi in linguaggio C++. Volendone verificare il corretto funzionamento è necessario creare un programma principale che istanzi alcuni oggetti della classe e che ne permetta l'invocazione dei metodi, come illustrato in vari esempi del capitolo.

1 **Definire una classe in linguaggio C++ per la gestione di un punto nel piano cartesiano che, oltre a permettere di impostare e recuperare le coordinate, implementi le seguenti operazioni:**

- **determinazione delle coordinate in forma polare[2];**
- **calcolo della distanza tra due punti;**
- **restituzione del punto medio tra due punti.**

2. Le coordinate polari di un punto in un riferimento cartesiano sono date dalla sua distanza dall'origine del riferimento e dall'angolo (misurato in senso antiorario) tra la congiungente con l'origine del sistema di riferimento e l'asse delle ascisse.

2 **Definire una classe in linguaggio C++ per la gestione di numeri complessi che, oltre a permettere di impostare e recuperare i valori delle parti reale e immaginaria, implementi le seguenti operazioni:**

- **modulo;**
- **inverso;**
- **somma;**
- **differenza;**
- **prodotto;**
- **quoziente;**
- **potenza.**

3 **Definire una classe in linguaggio C++ che simuli una lampadina colorata il cui funzionamento è definito dai seguenti attributi, non direttamente modificabili dall'esterno: codice di colore (1 = bianco, 2 = rosso, 3 = blu, 4 = giallo), livello di luminosità (da 0 a un valore massimo), stato accesa/spenta. Oltre a un costruttore che accetti come parametri il colore e il valore massimo di luminosità della lampadina, si prevedano i seguenti metodi:**

- **aumento della luminosità di una lampadina accesa, senza superare il valore massimo;**
- **diminuzione della luminosità di una lampadina, fino a spegnerla;**
- **accensione di una lampadina spenta con impostazione della luminosità al valore minimo;**
- **spegnimento di una lampadina accesa;**
- **restituzione del colore della lampadina;**
- **restituzione del valore corrente della luminosità.**

4 **Definire una classe in linguaggio C++ per la gestione di dati di tipo orario nella forma H:M:S (H rappresenta le ore, M i minuti ed S i secondi con $0 \le H < 24$, $0 \le M < 60$, $0 \le S < 60$). Oltre a un costruttore adeguato e ai metodi di accesso agli attributi, la classe deve prevedere metodi per effettuare le seguenti operazioni:**

- **restituzione del valore dell'orario in ore decimali;**
- **restituzione del valore dell'orario in secondi trascorsi dalla mezzanotte;**
- **differenza tra due orari espressa in secondi.**

5 Definire una classe in linguaggio C++ per la gestione di dati di tipo angolo nella forma G° P′ S″ (G rappresenta i gradi, P i primi ed S i secondi con $0 \leq G < 360$, $0 \leq P < 60$, $0 \leq S < 60$). Oltre a un costruttore adeguato e ai metodi di accesso agli attributi, la classe deve prevedere metodi per effettuare le seguenti operazioni:

- restituzione del valore dell'angolo in gradi decimali;
- restituzione del valore dell'angolo in secondi;
- somma tra due angoli;
- differenza tra due angoli.

6 Un magazzino contiene un certo numero di prodotti distinti specificato nel costruttore. Il magazzino può essere rifornito di una data quantità di un prodotto specifico con l'acquisto di merce da un fornitore, o scaricato di una data quantità di un prodotto specifico per soddisfare l'ordine di un cliente.
Definire una classe C++ che consenta di gestire il funzionamento di un magazzino il cui costruttore, oltre a dimensionare un vettore per memorizzare le quantità dei vari prodotti, ne inizializzi a zero le quantità. Gli oggetti istanza della classe devono consentire le seguenti operazioni:

- carico nel magazzino di una data quantità di un certo prodotto (il prodotto viene indicato con un codice numerico che corrisponde all'indice della posizione del vettore in cui se ne memorizza la quantità disponibile);
- scarico dal magazzino di una data quantità di un certo prodotto; il metodo deve restituire il valore 0 se lo scarico è possibile (quantità in magazzino superiore alla quantità richiesta), il valore –1 se lo scarico è possibile solo parzialmente (quantità richiesta superiore alla quantità presente in magazzino) e il valore –2 se lo scarico è impossibile (quantità presente in magazzino nulla).

7 Definire una classe in linguaggio C++ per la gestione di una data nel formato G/M/A (G rappresenta il giorno, M il mese e A l'anno). Oltre a un costruttore adeguato e ai metodi di accesso agli attributi, la classe deve prevedere metodi per effettuare le seguenti operazioni:

- restituzione del numero di giorni trascorsi dal primo gennaio dell'anno in corso (considerare bisestili gli anni divisibili per quattro);
- restituzione mediante un codice numerico (0 = domenica, 1 = lunedì ecc.) del giorno della settimana corrispondente;
- differenza tra due date espressa in giorni.

LABORATORIO

1 A un ufficio postale accedono clienti che scelgono di usufruire di uno dei seguenti tipi di servizio:

- servizio di spedizione (S);
- servizio di ricezione (R);
- servizio finanziario (F).

I singoli sportelli dell'ufficio postale sono specializzati per effettuare uno specifico servizio e i clienti sono serviti nell'ordine in cui sono arrivati in base al servizio richiesto (può accadere quindi che un cliente X che richiede il servizio S sia servito prima del cliente Y che richiede un diverso servizio, anche se Y è giunto dopo X).

- Definire e implementare una classe C++ che modelli il funzionamento dell'ufficio postale consentendo di simulare le seguenti attività e identificando il singolo cliente con un codice numerico:
  - arrivo di un nuovo cliente che richiede uno specifico servizio;
  - chiamata di un cliente in attesa per un servizio da parte di uno sportello;
  - visualizzazione del numero dei clienti in attesa per ciascuno dei tre servizi.
- Scrivere un programma di test della classe.
- Modificare la classe in modo da gestire con priorità e per il solo servizio F i clienti in possesso di una carta *Poste-Pay*.

SEZIONE

B Pagine web

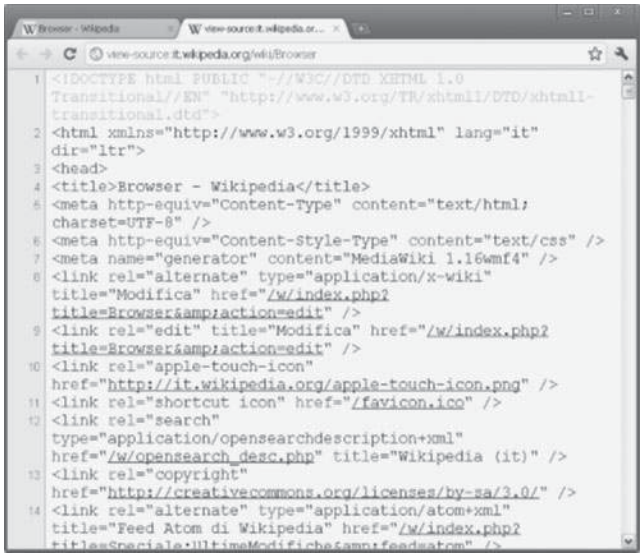
B1 Il linguaggio HTML

B2 CSS (*Cascading Style Sheet*) per pagine web

B1 Il linguaggio HTML

Molte ricerche scolastiche iniziano dalla seguente pagina web:

Questa è la «vista» della *home-page* del sito web `www.wikipedia.org` che il browser (in questo caso *Chrome*, ma non vi sarebbero sostanziali differenze utilizzando *Explorer*, *Firefox*, *Safari* o *Opera*) visualizza. In realtà il file ricevuto dal server che ospita il sito contiene il codice che segue, visualizzabile con lo strumento «visualizza sorgente»:



Pagine web

Le pagine web sono file di testo contenenti codice in linguaggio HTML che il browser interpreta per restituire all'utente la visualizzazione della pagina stessa.

Le pagine che costituiscono un sito sono memorizzate in un computer server che le rende disponibili in rete (la rete locale dei computer di una organizzazione, oppure la rete globale Internet).

Il browser si comporta da client richiedendo al computer server una specifica pagina identificata dall'URL (*Uniform Resource Locator*, l'«indirizzo» web): alla richiesta il computer server risponde inviando il contenuto della pagina, il codice HTML.

Le pagine web che siamo soliti vedere visualizzate nel browser sono definite mediante il **linguaggio HTML** (***Hyper-Text Mark-up Language***), la cui prima versione è stata ideata da Tim Berners-Lee nel 1990 presso i laboratori di ricerca del CERN di Ginevra. HTML non è un linguaggio di programmazione – non consente infatti di implementare algoritmi – ma un linguaggio che permette di definire il contenuto e il *layout* (cioè il formato di visualizzazione) di una pagina web.

> Il browser ha il compito di interpretare il linguaggio HTML con cui sono definite le pagine web e di visualizzarne il contenuto (testo e immagini, ma anche elementi multimediali quali suoni, musica, video ecc.) con l'aspetto che il codice HTML stesso definisce.

Nel seguito saranno introdotti gli elementi fondamentali della versione 5 del linguaggio HTML recentemente standardizzata.

L'evoluzione del linguaggio HTML

La prima versione del linguaggio HTML è stata definita e implementata da Tim Berners-Lee nei primi anni '90 del secolo scorso presso i laboratori di ricerca del CERN di Ginevra e prevedeva una ventina di *tag*, alcuni dei quali ancora presenti nel linguaggio attuale.
La versione 2 del linguaggio è stata standardizzata nel 1995, nella fase di rapida espansione della rete Internet alle utenze private, sia aziendali sia domestiche, causata dalla possibilità di realizzare «siti» utilizzando il linguaggio HTML.
Negli anni le tecnologie Internet legate in vario modo al linguaggio HTML si sono moltiplicate, ma il linguaggio stesso è rimasto fermo alla versione 4 – approvata nel 1997 come compromesso tra le funzionalità non standard implementate dai vari browser nel corso degli anni – per più di un decennio, fino all'avvento della innovativa versione 5 nel 2011.

1 Gli elementi fondamentali del linguaggio HTML

1.1 I *tag*

Osservando il codice sorgente di una qualsiasi pagina HTML è possibile notare come il testo sia racchiuso all'interno di numerosi elementi di contenimento caratterizzati da due parentesi angolari (i simboli «<» e «>»): essi prendono il nome di ***tag*** e permettono di definire la struttura della pagina web.
La maggior parte dei *tag* necessita di essere chiusa mediante un *tag* di chiusura, ovvero un *tag* identico a quello di apertura, eccezion fatta per il simbolo «/» iniziale; la visualizzazione di una sezione di testo di una pagina web è condizionata dal tipo di *tag* che la racchiude al proprio interno.

ESEMPIO

Il *tag* «html» è così formato:

```
<html>
```

Il corrispondente *tag* di chiusura è quindi il seguente:

```
</html>
```

La disposizione dei *tag* nella pagina HTML deve seguire uno schema di **annidamento** rigoroso, analogo a quello delle parentesi in un'espressione matematica: un *tag* aperto all'interno di un altro *tag* deve necessariamente essere chiuso prima del *tag* di chiusura del *tag* più esterno.

ESEMPIO

Il *tag* `<body>`, inserito all'interno del *tag* `<html>`, dovrà essere chiuso prima della chiusura del *tag* più esterno:

```
<html>
   <body>
   </body>
</html>
```

La seguente struttura risulta invece errata:

```
<html>
<body>
</html>
</body>
```

OSSERVAZIONE Una pagina web in linguaggio HTML può essere scritta utilizzando un qualsiasi editor di testo – come il programma «blocco note» del sistema operativo Windows – ma è preferibile ricorrere a strumenti che evidenzino la presenza dei *tag* nel testo della pagina e che ne verifichino la correttezza della chiusura e dell'annidamento. I file di testo che contengono codice HTML sono normalmente salvati con estensione *.html*.

In quasi tutti i tipi di *tag* è possibile inserire uno o più **attributi** utilizzati per definire alcune impostazioni relative al contenuto del *tag* stesso; gli attributi devono essere posti immediatamente dopo il nome del *tag* e sono composti da due parti: la prima, a sinistra del simbolo «=», è il nome dell'attributo, la seconda, a destra del simbolo «=» e compresa tra due simboli «"» è l'argomento:

```
<tag attributo="argomento">
```

ESEMPIO

Il *tag* `<img>`, utilizzato per incorporare immagini all'interno della pagina web, necessita dell'attributo `src` che specifica il nome (comprensivo del percorso, o dell'indirizzo in cui si trova) del file contenente l'immagine:

```
<img src="immagine.jpg">
```

OSSERVAZIONE Esistono due diversi tipi di attributi: gli **attributi standard** e gli **attributi di evento**. I primi definiscono dimensioni, collegamenti, misure, mentre i secondi specificano quali aspetti devono essere adottati in determinate situazioni, per esempio al click mediante il mouse o al passaggio del puntatore.

Il **contenuto** del *tag* è infine il testo inserito fra il *tag* di apertura e il corrispondente *tag* di chiusura[1]:

1. Il testo «Lorem ipsum…» è tratto da un'opera di Cicerone ed è utilizzato fin dall'avvento della stampa, nel XV secolo, come testo di riempimento casuale.

```
<body>
Lorem ipsum dolor sit amet
</body>
```

Esistono infine i ***tag* di commento**, che non sono presi in considerazione dal browser e che sono quindi utilizzati esclusivamente per aggiungere considerazioni o istruzioni all'interno del codice, in modo da semplificare l'organizzazione del lavoro e avere delle linee guida in una futura modifica della pagina. Essi hanno la seguente struttura:

```
<!-- commento non visualizzato dal browser -->
```

OSSERVAZIONE Il linguaggio HTML non è *case-sensitive*, il che significa che non vi è distinzione fra le lettere maiuscole e minuscole nella scrittura del codice. Inoltre lo spazio lasciato fra una parola e l'altra del testo viene interpretato dal browser come un singolo spazio, indipendentemente da quanti essi siano realmente nel codice. Nel caso sia necessario inserire più di uno spazio all'interno di una frase, si utilizza il simbolo speciale «` `»:

```
Lorem ipsum   dolor sit amet
```

1.2 La struttura della pagina HTML

Il codice di una pagina HTML è sempre formato da tre parti differenti:

- l'**intestazione**, costituita dal seguente *tag* che non necessita di essere chiuso:

```
<!doctype html>
```

- lo ***header***, compreso tra i *tag* `<head>` e `</head>`
- il ***body***, compreso tra i *tag* `<body>` e `</body>`

L'*intestazione* deve necessariamente essere posta all'inizio del codice, in quanto ha la funzione di comunicare al browser il linguaggio in cui la pagina web è scritta.
Lo *header* e il *body* sono compresi nel *tag* `<html>` e contengono rispettivamente la parte invisibile (titolo, descrizione ecc.) e la parte visibile della pagina web.
La struttura fondamentale di una pagina HTML è di conseguenza sempre la seguente:

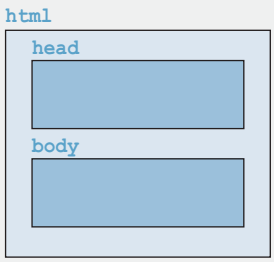
```
<!doctype html>
<html>
  <head>
  </head>
  <body>
  </body>
</html>
```

Motori di ricerca

Un motore di ricerca come *Google* mantiene sui propri server le associazioni tra il testo contenuto nelle pagine web e i loro URL in modo da poter fornire, a partire da una richiesta basata su alcune parole chiave, un elenco di pagine pertinenti opportunamente ordinate.

L'aggiornamento delle associazioni è demandato a programmi (noti come *web crawler* o *spider*) che visitano automaticamente i siti presenti in Internet, aggiornando gli indici delle parole chiave utilizzati dal motore di ricerca.

OSSERVAZIONE Come nei linguaggi di programmazione, l'indentazione del codice HTML viene utilizzata per evidenziarne la struttura di nidificazione che, in questo caso è la seguente:



All'interno dello *header* è possibile indicare il titolo della pagina web che sarà visibile nella «barra del titolo» del browser o all'interno dei risultati di un motore di ricerca; per farlo è sufficiente utilizzare il *tag* `<title>` contenente il titolo desiderato all'interno dello *header*:

```
<head>
  <title>Titolo</title>
</head>
```

Utilizzando il *tag* `<meta>` nello *header* si ha la possibilità di aggiungere alla pagina anche alcune parole chiave e una breve descrizione del contenuto: questi sono elementi essenziali per il posizionamento della pagina nei risultati forniti dai motori di ricerca. Il *tag* `<meta>` ha due attributi principali – *name* e *content* – e non necessita di essere chiuso, in quanto presenta già il simbolo «/» all'interno del *tag* di apertura.
Per inserire la descrizione è necessario assegnare il valore `"description"` all'attributo *name* e assegnare la stringa di descrizione all'attributo *content*.
Per aggiungere le parole chiave è necessario assegnare il valore `"keywords"` all'attributo *name* e assegnare l'elenco delle parole chiave – separate dal carattere «,» – all'attributo *content*.

ESEMPIO

La *home-page* di un sito dedicato alla storia del computer potrebbe avere uno *header* come il seguente:

```
<head>
  <title>La storia del computer</title>
  <meta name="description" content="Il computer dal 1945 ad oggi"/>
  <meta name="keywords" content="computer, storia, Von Neumann, PC"/>
</head>
```

OSSERVAZIONE A eccezione del titolo (*tag* `<title>`), ciò che viene inserito nello *header* di una pagina HTML non viene visualizzato quando essa viene aperta dal browser.

Il *body* è la parte visibile della pagina HTML; infatti è all'interno del *tag* *body* che deve essere inserito il contenuto della pagina (testi, immagini, video, audio ecc.).

1.3 Il testo

Il testo all'interno della pagina HTML viene solitamente separato in paragrafi contenuti all'interno del *tag* `<p>`:

```
<body>
  <p>testo del primo paragrafo</p>
  <p>testo del secondo paragrafo</p>
</body>
```

Per andare a capo è quindi sufficiente chiudere un paragrafo e aprirne uno nuovo; se invece si desidera andare a capo all'interno dello stesso paragrafo, è possibile farlo utilizzando il *tag* `<br/>`, che non necessita di essere chiuso:

```
<body>
  <p>prima linea di testo<br/>seconda linea di testo</p>
</body>
```

È possibile aggiungere un titolo al paragrafo inserendolo all'interno del *tag* `<h1>`:

```
<body>
  <h1>Titolo del paragrafo</h1>
  <p>testo del paragrafo</p>
</body>
```

Esistono sei diversi livelli di titolazione, individuati dai *tag* `<h1>`, `<h2>`, …, `<h6>`, che consentono di inserire sottotitoli di vario livello. Titoli e sottotitoli devono essere raggruppati all'interno del *tag* `<hgroup>`:

```
<body>
  <hgroup>
    <h1>Titolo del paragrafo</h1>
    <h2>Sottotitolo del paragrafo</h2>
  </hgroup>
  <p>testo del paragrafo</p>
</body>
```

Per evidenziare il testo in modo che appaia in grassetto è necessario racchiuderlo all'interno del *tag* `<strong>`, mentre il testo enfatizzato (che molti browser restituiscono in corsivo) si ottiene inserendo il testo nel *tag* `<em>`:

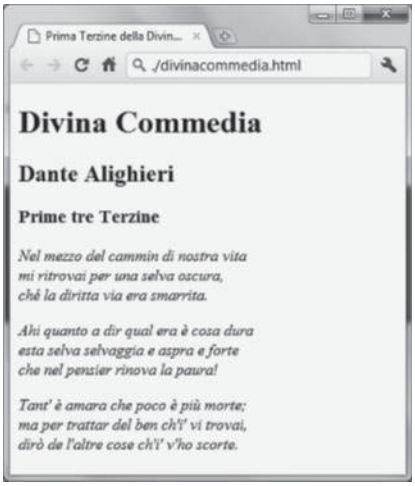
```
<body>
  <p>testo normale<br/>
    <strong>testo in grassetto</strong><br/>
    <em>testo in corsivo</em><br/>
    <strong><em>testo in grassetto e corsivo</em></strong>
  </p>
</body>
```

ESEMPIO

Il seguente codice HTML:

```
<!doctype html>
  <html>
    <head>
      <title>Prima Terzine della Divina Commedia</title>
    </head>
    <body>
      <hgroup>
        <h1>Divina Commedia</h1>
        <h2>Dante Alighieri</h2>
        <h3>Prime tre Terzine</h3>
      </hgroup>
      <p>
        <em>
          Nel mezzo del cammin di nostra vita<br/>
          mi ritrovai per una selva oscura,<br/>
          ch&eacute; la diritta via era smarrita.
        </em>
      </p>
      <p>
        <em>
          Ahi quanto a dir qual era &egrave; cosa dura<br/>
          esta selva selvaggia e aspra e forte<br/>
          che nel pensier rinova la paura!
        </em>
      </p>
      <p>
        <em>
          Tant'&egrave; amara che poco &egrave; pi&ugrave;
          morte;<br/>
          ma per trattar del ben ch'i' vi trovai,<br/>
          dir&ograve; de l'altre cose ch'i' v'ho scorte.
        </em>
      </p>
    </body>
  </html>
```

viene visualizzato dal browser nel seguente modo:



OSSERVAZIONE I caratteri accentati della lingua italiana non fanno parte dell'insieme dei caratteri fondamentali utilizzabili in una pagina HTML, per cui è necessario utilizzare i simboli che codificano i caratteri speciali, necessari anche nel caso in cui il testo debba contenere simboli normalmente usati per la codifica dei *tag* stessi, come per esempio «<» e «>».

La TABELLA **1** riporta la codifica[2] dei principali caratteri HTML speciali.

2. A differenza dei *tag*, le codifiche dei caratteri speciali sono *case-sensitive*.

TABELLA **1**

Carattere	Codice HTML	Carattere	Codice HTML	Carattere	Codice HTML
"	"	¾	¾	à	à
'	'	¿	¿	á	á
&	&	×	×	â	â
<	<	÷	÷	ã	ã
>	>	À	À	ä	ä
		Á	Á	å	å
¡	¡	Â	Â	æ	æ
¢	¢	Ã	Ã	ç	ç
£	£	Ä	Ä	è	è
¤	¤	Å	Å	é	é
¥	¥	Æ	Æ	ê	ê
¦	¦	Ç	Ç	ë	ë
§	§	È	È	ì	ì
¨	¨	É	É	í	í
©	©	Ê	Ê	î	î
ª	ª	Ë	Ë	ï	ï
«	«	Ì	Ì	ð	ð
¬	¬	Í	Í	ñ	ñ
		Î	Î	ò	ò
®	®	Ï	Ï	ó	ó
¯	¯	Ð	Ð	ô	ô
°	°	Ñ	Ñ	õ	õ
±	±	Ò	Ò	ö	ö
²	²	Ó	Ó	ø	ø
³	³	Ô	Ô	ù	ù
´	´	Õ	Õ	ú	ú
µ	µ	Ö	Ö	û	û
¶	¶	Ø	Ø	ü	ü
·	·	Ù	Ù	ú	ú
¸	¸	Ú	Ú	û	û
¹	¹	Û	Û	ü	ü
º	º	Ü	Ü	ý	ý
»	»	Ý	Ý	þ	þ
¼	¼	Þ	Þ	ÿ	ÿ
½	½	ß	ß		

1.4 Gli elenchi e le tabelle

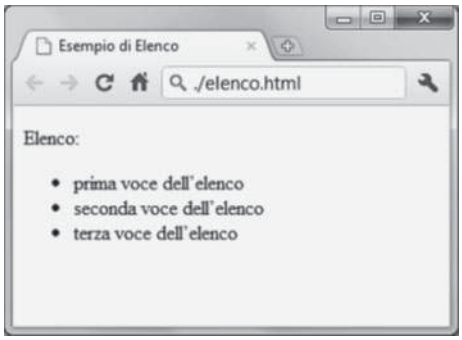
È possibile organizzare il testo, o comunque il contenuto di una pagina, in elenchi e/o tabelle. Per realizzare un elenco testuale è necessario utilizzare il *tag* `<ul>` e inserire ogni singola voce dell'elenco all'interno del *tag* `<li>`.

ESEMPIO

Il seguente codice crea una pagina contenente un elenco di tre voci

```
<!doctype html>
<html>
  <head>
    <title>Esempio di Elenco</title>
  </head>
  <body>
    <p>Elenco:</p>
    <ul>
      <li>prima voce dell'elenco</li>
      <li>seconda voce dell'elenco</li>
      <li>terza voce dell'elenco</li>
    </ul>
  </body>
</html>
```

che il browser visualizza nel seguente modo:



Le tabelle sono organizzate per righe e colonne e si costruiscono utilizzando il *tag* `<table>`. Il numero di colonne della tabella è dato dal numero di celle inserite in ogni riga, perciò il numero di celle – e quindi il numero di colonne – dovrà essere lo stesso in ogni riga. Le righe sono individuate dal *tag* `<tr>` (*table row*), mentre le celle sono definite dal *tag* `<td>` (*table data*). Per esempio, una tabella di due righe per due colonne può essere definita mediante il seguente codice:

```
<body>
  <table>
    <tr>
      <td>contenuto cella [1,1]</td>
      <td>contenuto cella [1,2]</td>
    </tr>
    <tr>
```

▶

```
      <td>contenuto cella [2,1]</td>
      <td>contenuto cella [2,2]</td>
    </tr>
  </table>
</body>
```

OSSERVAZIONE Con le precedenti versioni del linguaggio HTML le tabelle venivano spesso usate per strutturare graficamente l'intera pagina. Si tratta di un modo errato di formattare il layout grafico che deve invece essere sempre realizzato con l'uso dei CSS (*Cascading Style Sheet*) come vedremo nel prossimo capitolo.

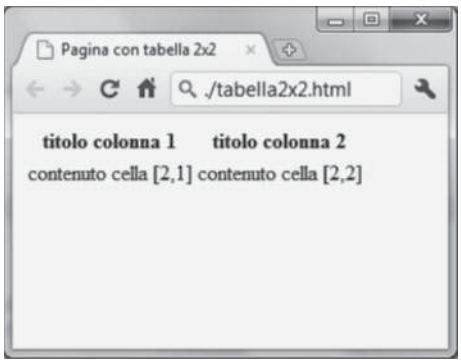
Le celle della tabella destinate a contenere l'intestazione di una colonna, o di una riga, devono essere marcate con il *tag* `<th>` invece che con il *tag* `<td>`.

ESEMPIO

Il codice HTML che definisce una tabella di due righe per due colonne con le celle della prima riga contenenti i titoli delle rispettive colonne è così strutturato:

```
<!doctype html>
  <html>
    <head>
      <title>Pagina con tabella 2x2</title>
    </head>
    <body>
      <table>
        <tr>
          <th>titolo colonna 1</th>
          <th>titolo colonna 2</th>
        </tr>
        <tr>
          <td>contenuto cella [2,1]</td>
          <td>contenuto cella [2,2]</td>
        </tr>
      </table>
    </body>
  </html>
```

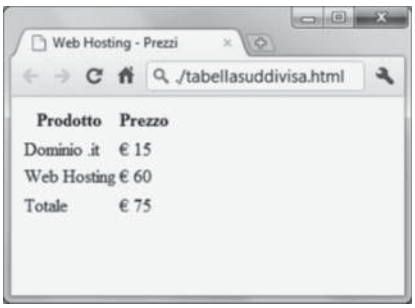
La tabella viene visualizzata dal browser in questo modo:



Talvolta risulta utile suddividere la tabella in tre sezioni: una sezione iniziale di intestazione (*header*), una sezione centrale che costituisce il corpo della tabella (*body*) e una sezione terminale costituita normalmente dalle ultime righe (*footer*): in HTML è possibile definire le sezioni utilizzando rispettivamente i *tag* `<thead>`, `<tbody>`, `<tfoot>`.

ESEMPIO

Per realizzare una tabella come la seguente, suddivisa in uno *header* con l'intestazione delle colonne, in un *footer* riportante la somma totale e il *body* con l'elenco dei prodotti:



è necessaria una pagina HTML così impostata:

```
<!doctype html>
  <html>
    <head>
      <title>Web Hosting - Prezzi</title>
    </head>
    <body>
      <table>
        <thead>
          <tr>
            <th>Prodotto</th>
            <th>Prezzo</th>
          </tr>
        </thead>
        <tfoot>
          <tr>
            <td>Totale</td>
            <td>&euro; 75</td>
          </tr>
        </tfoot>
        <tbody>
          <tr>
            <td>Dominio .it</td>
            <td>&euro; 15</td>
          </tr>
          <tr>
            <td>Web Hosting</td>
            <td>&euro; 60</td>
          </tr>
        </tbody>
      </table>
    </body>
  </html>
```

OSSERVAZIONE Come è evidente dall'esempio precedente, il corretto ordinamento delle sezioni della tabella viene effettuato automaticamente dal browser in fase di visualizzazione.

2 I collegamenti ipertestuali (*link*)

I **collegamenti ipertestuali** o ***link*** sono elementi del testo[3] che, se selezionati con il puntatore del mouse, indirizzano il browser verso una sezione diversa del corpo della stessa pagina, verso un'altra pagina del sito, oppure un diverso sito o direttamente al contenuto di un file.
L'elemento di testo che si desidera utilizzare come collegamento deve essere collocato all'interno del *tag* `<a>`, mentre la destinazione del collegamento stesso deve essere definita come valore dell'attributo ***href*** del *tag*, utilizzando la seguente forma:

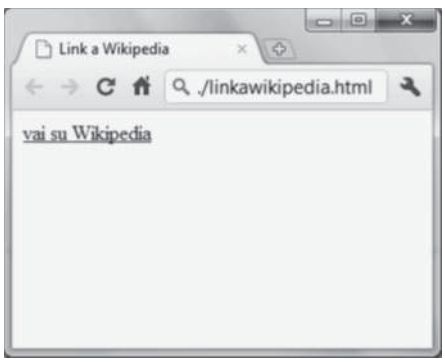
```
<a href="destinazione">testo del collegamento</a>
```

3. Vedremo in seguito che anche un elemento non testuale – per esempio un'immagine – può essere utilizzato come *link*.

ESEMPIO

Il seguente codice HTML definisce una pagina contenente un collegamento con la dicitura «vai su Wikipedia» che, se selezionato con il puntatore del mouse, visualizzerà nel browser il contenuto della pagina web «http://www.wikipedia.org»:

```
<!doctype html>
  <html>
    <head>
      <title>Link a Wikipedia</title>
    </head>
    <body>
      <a href="http://www.wikipedia.org">vai su Wikipedia</a>
    </body>
  </html>
```



Se la destinazione del *link* è una pagina interna allo stesso sito, all'attributo *href* può essere assegnato il *pathname* assoluto o relativo del file che contiene il codice HTML della pagina:

```
<a href="./pagina.html">vai alla pagina</a>
```

Se invece la destinazione del *link* è un sito web esterno, è necessario assegnare l'URL all'attributo *href* e, anche se non è obbligatorio, indicare che il collegamento è di tipo esterno utilizzando l'attributo *rel* con valore `"external"`:

```
<a href="http://www.website.com" rel="external">vai al sito</a>
```

OSSERVAZIONE In ogni caso è possibile fare in modo che il browser apra il collegamento in una nuova scheda assegnando il valore `"_blank"` all'attributo *target* del *tag* `<a>`:

```
<a href="destinazione" target="_blank">testo del collegamento</a>
```

3 Le immagini

Le immagini – a differenza del testo – non sono mai direttamente presenti nel codice HTML di una pagina web. Per inserire un'immagine all'interno di una pagina si ricorre al *tag* `<img>`, assegnando all'attributo ***src*** il *pathname* del file che contiene l'immagine da incorporare nella pagina (il nome del file deve sempre essere comprensivo dell'estensione):

```
<img src="./fotografia.jpg"/>
```

OSSERVAZIONE Il *tag* `<img>` non necessita del corrispondente *tag* di chiusura, in quanto il simbolo «/» è già compreso al suo interno.

È importante aggiungere l'attributo ***alt*** a cui assegnare un testo che viene visualizzato dal browser nel caso risulti impossibile caricare l'immagine[4]:

```
<img src="./fotografia.jpg" alt="testo descrittivo della foto"/>
```

4. Il testo alternativo viene letto automaticamente dai browser utilizzati da ipovedenti: questo è uno degli accorgimenti fondamentali per rendere una pagina web «accessibile» per utenti che necessitano di una diversa presentazione della pagina.

OSSERVAZIONE È fortemente consigliato utilizzare solamente immagini in formato PNG, GIF o JPG. Inoltre, le immagini devono essere di dimensioni contenute (infatti file di grandi dimensioni rallentano non poco il caricamento della pagina da parte del *browser*).

Per specificare le misure di larghezza e lunghezza si utilizzano rispettivamente gli attributi ***width*** ed ***height***, assegnando a ciascuno di essi un valore espresso in pixel. Se viene specificato uno solo dei due valori, l'altro sarà impostato automaticamente dal browser rispettando le proporzioni geometriche dell'immagine.

ESEMPIO

La pagina HTML che segue visualizza una fotografia, ridimensionata a 200 pixel di larghezza, contenuta nel file «spiaggia.jpg» collocato nella stessa directory della pagina stessa:

```
<!doctype html>
  <html>
```

▶

```
    <head>
      <title>Pagina con Immagine</title>
    </head>
    <body>
      <img src="spiaggia.jpg" alt="Spiaggia con palma"
      width="200"/>
    </body>
  </html>
```

Un'immagine può anche essere utilizzata come collegamento ipertestuale (*link*); è sufficiente a questo scopo inserire il *tag* `<img>` all'interno di un *tag* `<a>`:

```
<a href="destinazione">
  <img src="immagine" alt="testo"/>
</a>
```

Altrimenti è possibile fare in modo che aree distinte di una stessa immagine si comportino come *link* con destinazioni diverse. Per ottenere questo risultato è necessario creare una ***usemap*** – ovvero una suddivisione virtuale dell'immagine – inserendo l'attributo *usemap* all'interno del *tag* `<img>` e assegnando a essa un nome preceduto dal simbolo «#»:

```
<img src="immagine" usemap="#mappa"/>
```

Una volta definita, la mappa virtuale deve essere creata utilizzando il *tag* `<map>` e assegnando all'attributo ***name*** il nome precedentemente scelto:

```
<img src="./fotografia.jpg" usemap="#mappafotografia"/>
<map name="#mappafotografia">
</map>
```

All'interno del *tag* `<map>` devono essere inseriti i vari ***tag <area>*** utilizzati per definire i diversi settori dell'immagine che si intendono associare ai *link*. Il *tag* `<area>` ha quattro attributi:

- ***shape***: determina se la forma del settore è un rettangolo (`"rect"`), oppure un cerchio (`"circle"`);

Formati delle immagini digitali

I file contenenti immagini digitali si presentano in vari formati.

Anche se alcuni browser sono in grado di visualizzare altri formati, i formati standard previsti dal linguaggio HTML sono 3:

- GIF
- JPG
- PNG

Sono tutti formati «compressi» perché file-immagine di grandi dimensione rallenterebbero anche in modo evidente i tempi di attesa per la visualizzazione da parte dell'utente mentre il file viene ricevuto dal computer server.

I formati GIF e PNG adottano algoritmi di compressione di tipo *loseless*, che non alterano la qualità dell'immagine originale, ma il formato GIF è in grado di supportare solo un numero limitato (256) di colori diversi.

Il formato JPG realizza uno schema di compressione *lose* che altera l'immagine originale, ma che garantisce al tempo stesso dimensioni più contenute del file.

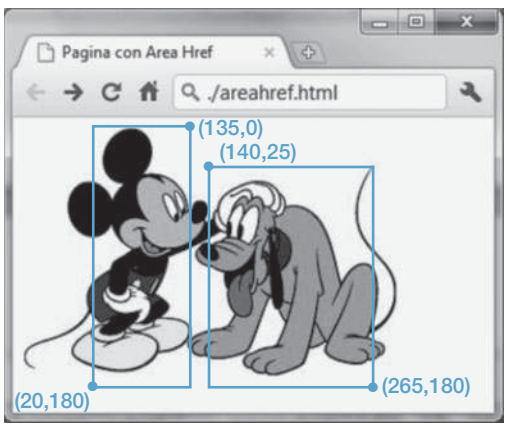
- *coords*: se *shape* definisce un rettangolo, imposta le coordinate cartesiane relative ai pixel dell'immagine di due vertici opposti del rettangolo; se invece definisce un cerchio, imposta le coordinate cartesiane del centro e la lunghezza del raggio espressa in pixel;
- *href*: come nel *tag* `<a>` definisce la destinazione del collegamento;
- *alt*: come nel *tag* `<img>` contiene il testo alternativo relativo all'immagine.

OSSERVAZIONE Contrariamente al normale sistema di riferimento cartesiano utilizzato in matematica, l'asse delle ordinate è definito sulla superficie di un'immagine digitale in modo invertito: le coordinate crescono procedendo dall'alto verso il basso.

ESEMPIO

Il seguente codice HTML consente di realizzare una pagina contenente un'immagine in cui sono presenti sia Topolino (nella parte sinistra dell'immagine), sia Pluto (nella parte destra): se l'utente seleziona il settore dell'immagine in cui si trova Topolino verrà indirizzato verso la pagina web «topolino.html», mentre se seleziona la sezione dell'immagine con Pluto verso la pagina «pluto.html»:

```
<!doctype html>
  <html>
    <head>
      <title>Topolino e Pluto</title>
    </head>
    <body>
      <img src="topolino.jpg" usemap="#topolinopluto"
           alt="Topolino e Pluto"/>
        <map name="topolinopluto">
          <area shape="rect" coords="20,180,135,0"
          href="topolino.html"
            alt="Topolino" />
          <area shape="rect" coords="140,25,265,180"
          href="pluto.html"
            alt="Pluto" />
        </map>
    </body>
  </html>
```



4 Suddivisione della pagina in funzione del contenuto

Nel caso di una pagina web complessa che presenta tipologie di contenuto differenziate, è importante dividere quest'ultimo utilizzando i ***tag* semantici**: la suddivisione risulta essenziale non solo perché permette al browser di presentare i contenuti a seconda della loro importanza e coerenza, ma – come vedremo nel seguito – anche perché consente all'autore della pagina web di gestire in modo ordinato il *layout* grafico della pagina utilizzando i CSS (*Cascading Style Sheet*).

OSSERVAZIONE I *tag* semantici sono stati introdotti dalla recente versione 5 del linguaggio HTML; fino alla precedente versione 4 il contenuto della pagina veniva suddiviso solo in previsione del *layout* grafico utilizzando il *tag* `<div>`. Mentre è fortemente consigliato utilizzare questo *tag* esclusivamente nel caso in cui si abbia bisogno di posizionare in un certo punto della pagina un elemento puramente grafico.

I principali *tag* semantici mediante i quali è possibile strutturare il contenuto della pagina sono quelli riportati nella TABELLA **2**.

TABELLA **2**

`<header>`	Intestazione della pagina (titolo, sottotitolo, *banner*[5], …) o del documento
`<nav>`	Collegamenti che consentono la navigazione all'interno della pagina, del documento o del sito
`<article>`	Contenuto principale della pagina o del documento
`<section>`	Parti specifiche di una pagina, o di un documento, oppure sezioni generiche di contenuto non definibili con *tag* specifici
`<aside>`	Testo non direttamente collegato al contenuto della pagina o del documento, anche sotto l'aspetto della formattazione grafica
`<footer>`	Contenuto da collocare in fondo alla pagina, o alla fine di un documento

5. Immagine che, nella maggior parte dei casi, comprende il logo, il titolo e il motto del sito web.

ESEMPIO

La seguente pagina HTML è strutturata mediante l'utilizzo dei *tag* semantici e costituisce un esempio di pagina web completa.

```
<!doctype html>
  <html>
    <head>
      <title>Deep Purple</title>
      <meta name="description" content="Sito dedicato ai Deep Purple"/>
      <meta name="keywords" content="deep, purple, italia, fansite"/>
    </head>
    <body>
      <header>
        <img src="./deeppurple.jpg" height="200" alt="foto della band">
        <h1>Deep Purple</h1>
      </header>
      <nav>
        <a href="home.html">Home</a>  
```

▶

```
      <a href="discografia.html">Discografia</a>  
      <a href="band.html">La Band</a>  
      <a href="concerti.html">Concerti</a>
    </nav>
    <article>
      <p>I <strong>Deep Purple</strong> sono un gruppo rock britannico
         nato nel 1968 a Hertford, Inghilterra.
         Insieme a gruppi come Led Zeppelin e Black Sabbath, sono
         considerati i primi gruppi hard rock della storia.
         Si possono considerare i Deep Purple come una delle band
         musicalmente più complete degli anni 70, con un
         sostrato musicale che spazia dal <em>blues</em> al <em>rock
         n'roll</em>, dal <em>funky</em> al <em>jazz</em> e al
         <em>folk</em>, dalla <em>musica orientale</em>
         alla <em>musica classica</em>. <br />Ai Deep Purple va
         attribuita la prima commistione tra umori e temi musicali
         neoclassici
         al tempo e al ritmo tipici del blues e del rock and roll. Il
         suono della band comprende anche elementi pop e
         progressive rock.
      </p>
      <p>Hanno venduto più di 100 milioni di copie nel mondo e sono
         stati presenti sul Guinness dei primati come la band
         più rumorosa del mondo. A differenza di molti gruppi coetanei,
         nati dall'incontro dei musicisti, sono nati
         principalmente per la volontà di due manager: John Coletta e
         Tony Edwards.
      </p>
      <p>Da <a href="http://www.wikipedia.it"
              target="_blank"><em>Wikipedia</em></a></p>
    </article>
    <footer>
      <p><em>Tutti i contenuti presenti sono di proprietà dei siti
        web di provenienza</em></p>
    </footer>
  </body>
</html>
```

Sintesi

WWW. Acronimo di *World Wide Web*. È un servizio di Internet che permette l'accesso a un insieme vastissimo di contenuti in formato ipertestuale e multimediale. Il Web è stato inizialmente progettato da Tim Berners-Lee (ricercatore al CERN di Ginevra). Oggi gli standard su cui è basato sono mantenuti dal World Wide Web Consortium (W3C). L'elemento informativo base del WWW è la pagina web.

HTML. Acronimo di *Hyper Text Markup Language*. È un particolare linguaggio, basato su marcatori denominati ***tag***, che permette di descrivere le modalità di impaginazione, formattazione e visualizzazione del contenuto di una pagina web. Esso non è un vero e proprio linguaggio di programmazione, ma supporta l'inserimento di script e oggetti esterni quali immagini o filmati.

Browser. Sono applicazioni che consentono di visualizzare i contenuti delle pagine dei siti web e navigare in Internet interagendo con i collegamenti ipertestuali che permettono di passare da una pagina all'altra. Le loro funzionalità sono consentite dalla capacità di interpretare il linguaggio HTML, eseguire *script* realizzati con particolari linguaggi di programmazione e integrare componenti per la visualizzazione di immagini, filmati e riproduzioni audio.

URL. Acronimo di *Uniform Resource Locator*. È una stringa di caratteri che identifica univocamente l'indirizzo di una risorsa in Internet, come una pagina web. Per esempio l'URL `http://www.google.it` permette l'accesso alla pagina base del sito del motore di ricerca Google.

Web server. Con questo termine si indica un servizio e, per estensione, anche il computer su cui esso è in esecuzione che fornisce, tramite un software dedicato su richiesta dell'utente, file di qualsiasi tipo, tra cui pagine web.

Tag. I *tag* sono i **marcatori** che costituiscono la base del linguaggio HTML e che permettono di definire la struttura della pagina web. Sintatticamente un *tag* è una stringa di caratteri racchiusa tra due parentesi angolari («<» e «>»). La maggior parte dei *tag* necessita di essere chiusa mediante un ***tag* di chiusura** identico a quello di apertura, eccezion fatta per il simbolo «/» iniziale. I *tag* definiscono e condizionano la modalità di visualizzazione di una pagina web. La disposizione dei *tag* nella pagina HTML deve seguire uno schema di annidamento rigoroso, analogo a quello delle parentesi in un'espressione matematica: un *tag* aperto all'interno di un altro *tag* deve necessariamente essere chiuso prima del *tag* di chiusura del *tag* più esterno.

Attributi. In quasi tutti i tipi di *tag* è possibile inserire uno o più attributi per definire impostazioni relative al contenuto nel *tag* stesso; gli attributi devono essere posti dopo il nome del *tag* nella forma *nome-attributo* = "*valore-attributo*". Per esempio, il *tag* `<img>`, utilizzato per incorporare immagini all'interno di una pagina web, necessita dell'attributo **src**, che specifica il nome (comprensivo del percorso o dell'indirizzo in cui si trova) del file contenente l'immagine; con il *tag* `<img src="prova.jpg">` si incorpora nella pagina l'immagine `prova.jpg`.

Pagina web. Una pagina web è l'elemento base attraverso cui le informazioni del WWW vengono rese disponibili all'utente finale. Un insieme di pagine web, tra loro relazionate secondo una struttura ipertestuale e riferibili, di norma, a un unico web server, costituiscono un sito web. Una pagina web è articolata in più sezioni delimitate da opportuni *tag*, le principali delle quali sono: intestazione (costituita dal *tag* `<!doctype html>`), *header* (compreso tra i *tag* `<head>` e `</head>`) e *body* (compreso tra i *tag* `<body>` e `</body>`). Il *body* costituisce la parte visibile della pagina web.

Pagine web ed elementi multimediali. Le pagine web contengono al loro interno solo testo; qualsiasi altro elemento multimediale (immagine, filmato, suono) non è contenuto direttamente nella pagina, che invece ne specifica solo il riferimento con un opportuno *tag*. Quando una pagina viene scaricata per essere visualizzata dal browser, gli elementi multimediali vengono anch'essi scaricati come file a sé stanti dal server che li contiene. Per non rendere pesante il trasferimento delle pagine è consigliabile utilizzare file multimediali di dimensioni contenute (nel caso delle immagini i formati consigliati sono JPG, PNG e GIF).

Riferimento ipertestuale. Lo stesso acronimo HTML fa esplicitamente riferimento a un

sistema ipertestuale dove i collegamenti tra pagine rappresentano una funzionalità essenziale dei siti web. In genere, un riferimento ipertestuale (o *link*) è un meccanismo che permette di passare da un pagina web a un'altra. Per esempio, il *tag*

```
<A HREF="http://www.google.it"> Homepage di Google </A>
```

posizionato all'interno di una qualsiasi pagina web, permette di definire un riferimento ipertestuale sul testo «Homepage di Google» che permette di passare dalla pagina corrente alla pagina base del motore di ricerca Google. Un *link* può essere definito su un qualsiasi elemento di una pagina web (testo, immagine ecc.).

Tag `<meta>`. Utilizzando il *tag* `<meta>` nello *header* si ha la possibilità di aggiungere alla pagina alcune parole chiave e una breve descrizione del contenuto. Questi sono elementi essenziali per il posizionamento della pagina nei risultati forniti dai motori di ricerca.

Web spider/Web crawler. Sono programmi che esaminano in maniera automatica e sistematica le pagine presenti sul World Wide Web navigando tra esse. Questi vengono prevalentemente usati per creare elenchi di URL indicizzati in funzione di parole chiave relative al contenuto delle singole pagine visitate, elenchi che saranno poi resi disponibili a motori di ricerca per velocizzare le ricerche sul web.

Tag semantici e CSS. Nel caso di pagine web complesse è possibile organizzarne le tipologie di contenuto utilizzando i *tag* semantici, facendo sì che il browser presenti le informazioni a seconda della loro importanza e coerenza. Essi consentono inoltre all'autore della pagina web di gestire in modo ordinato il *layout* grafico della pagina stessa utilizzando i CSS (*Cascading Style Sheet*).

QUESITI

1 Il significato della sigla HTML è …

A … *Hyper Text Markup Language*.
B … *Hot Text Mail Language*.
C … *Hyper Textual Makeup Language*.
D Nessuna delle risposte precedenti.

2 L'HTML serve a …

A … creare slide di presentazione.
B … creare programmi javascript.
C … creare pagine web.
D Nessuna delle risposte precedenti.

3 Un *tag* è …

A … un programma per scrivere testi HTML.
B … un componente del linguaggio HTML.
C … un errore in un programma HTML.
D Nessuna delle risposte precedenti.

4 Il *tag* `<tr>` identifica una …

A … *task row*.
B … *title row*.
C … *table row*.
D Nessuna delle risposte precedenti.

5 Il *tag* `<td>` identifica una …

A … *task data*.
B … *title data*.
C … *table data*.
D Nessuna delle risposte precedenti.

6 Una tabella con una riga e due celle si crea con …

A … `< table >< tr >< td >< /td >< td >< /td >< /tr >< /table >`
B … `< table >< td >< tr >< /tr >< tr >< /tr >< /td >< /table >`
C … `< table >< td >< /td >< td >< /td >< /table >`
D Nessuna delle risposte precedenti.

7 L'attributo `width` si riferisce …

A … alla larghezza dell'area che contiene un'immagine.
B … all'altezza dell'area che contiene un'immagine.
C … alla diagonale dell'area che contiene un'immagine.
D Nessuna delle risposte precedenti.

8 **Per realizzare un elenco si usa …**

A … un *tag* `<ul>` e tanti *tag* `<li>` quante sono le voci dell'elenco.

B … un *tag* `<li>` e tanti *tag* `<ul>` quante sono le voci dell'elenco.

C … un *tag* `<ul>` e un *tag* `<li>`.

D Nessuna delle risposte precedenti.

9 **Il testo in grassetto si imposta col *tag* …**

A … *strong*.

B … *grassett*.

C … *underline*.

D Nessuna delle risposte precedenti.

10 **Un *link* è …**

A … una pagina web.

B … un collegamento ipertestuale.

C … un componente funzionale del browser.

D Nessuna delle risposte precedenti.

11 **Il titolo di una pagina HTML si trova nella sezione …**

A … *head*.

B … *body*.

C … *title*.

D Nessuna delle risposte precedenti.

12 **Il corpo di una pagina HTML viene definito nella sezione …**

A … *head*.

B … *body*.

C … *title*.

D Nessuna delle risposte precedenti.

13 **Un *tag* si chiude col simbolo …**

A … /

B … >

C … !

D Nessuna delle risposte precedenti.

14 **Un browser è …**

A … un programma per scrivere pagine web.

B … un programma per visualizzare pagine web.

C … un insieme di pagine web collegate tra loro.

D Nessuna delle risposte precedenti.

15 **Web server …**

A … è l'equivalente di un browser.

B … è un computer su cui risiedono pagine web.

C … è un servizio e, per estensione, il computer su cui esso è in esecuzione che fornisce, tramite un software dedicato e su richiesta dell'utente, file di qualsiasi tipo, tra cui pagine web.

D Nessuna delle risposte precedenti.

16 **Il *tag* `<a href="http://alfa.html">` `alfa` `</a>` …**

A … crea un *link* ipertestuale verso la pagina `alfa.html` sulla stringa `alfa`.

B … crea un *link* ipertestuale verso la pagina `alfa` sulla stringa `http://alfa.html`.

C … annulla un *link* ipertestuale verso la pagina `alfa.html` sulla stringa `alfa`.

D Nessuna delle risposte precedenti.

17 **Il *tag* `<a href="http://alfa.html"> <img src="pict.jpg"> </a>` …**

A … crea un *link* ipertestuale verso l'immagine `pic.jpg` sulla stringa `http://alfa.html`.

B … crea un *link* ipertestuale verso la pagina `alfa.html` sull'immagine `pict.jpg`.

C … annulla un *link* ipertestuale verso la pagina `alfa.html` sull'immagine `pict.jpg`.

D Nessuna delle risposte precedenti.

18 **Quali dei seguenti sono formati consigliati di immagini da utilizzare per la realizzazione di immagini web?**

A BMP.

B JPG.

C PNG.

D PIF.

CERN
Organizzazione Europea per la Ricerca Nucleare

hypertext
ipertesto: corpus informativo composto da un insieme di testi collegati fra loro in base a legami associativi (link), consente letture non sequenziali che l'utente sceglie liberamente

NeXT
Computer prodotto dall'omonima società fondata nel 1985 da Steve Jobs

WebServer
è un servizio, e per estensione il computer su cui esso è in esecuzione, che fornisce, tramite un software dedicato, su richiesta dell'utente, file di qualsiasi tipo, tra cui pagine web

Mosaic
è stato un browser sviluppato al National Center for Supercomputing Applications

1990 was a momentous year in world events. In February, Nelson Mandela was freed after 27 years in prison. In April, the space shuttle Discovery carried the Hubble Space Telescope into orbit. And in October, Germany was reunified. Then at the end of 1990, a revolution took place that changed the way we live today.

CERN, the European Organization for Nuclear Research, is where it all began in March 1989. A physicist, Tim Berners-Lee, wrote a proposal for information management showing how information could be transferred easily over the Internet by using hypertext, the now familiar point-and-click system of navigating through information. The following year, Robert Cailliau, a systems engineer, joined in and soon became its number one advocate.

The idea was to connect hypertext with the Internet and personal computers, thereby having a single information network to help CERN physicists share all the computer-stored information at the laboratory. Hypertext would enable users to browse easily between texts on web pages using links. The first examples were developed on NeXT computers.

Berners-Lee created a browser-editor with the goal of developing a tool to make the Web a creative space to share and edit information and build a common hypertext. What should they call this new browser: The Mine of Information? The Information Mesh? When they settled on a name in May 1990, it was the WorldWideWeb.

Info.cern.ch was the address of the world's first-ever web site and web server, running on a NeXT computer at CERN.

The first web page address was http://info.cern.ch/hypertext/WWW/TheProject.html, which centred on information regarding the WWW project. Visitors could learn more about hypertext, technical details for creating their own webpage, and even an explanation on how to search the Web for information. There are no screenshots of this original page and, in any case, changes were made daily to the information available on the page as the WWW project developed. You may find a later copy (1992) on the World Wide Web Consortium website.

However, a website is like a telephone; if there's just one it's not much use. Berners-Lee's team needed to send out server and browser software. The NeXT systems however were far advanced over the computers people generally had at their disposal: a far less sophisticated piece of software was needed for distribution.

By spring of 1991, testing was underway on a universal line mode browser, which would be able to run on any computer or terminal. It was designed to work simply by typing commands. There was no mouse, no graphics, just plain text, but it allowed anyone with an Internet connection access to the information on the Web.

The historic NeXT computer used by Tim Berners-Lee in 1990; is on display in the Microcosm exhibition at CERN. It was the first web server, hypermedia browser and web editor.

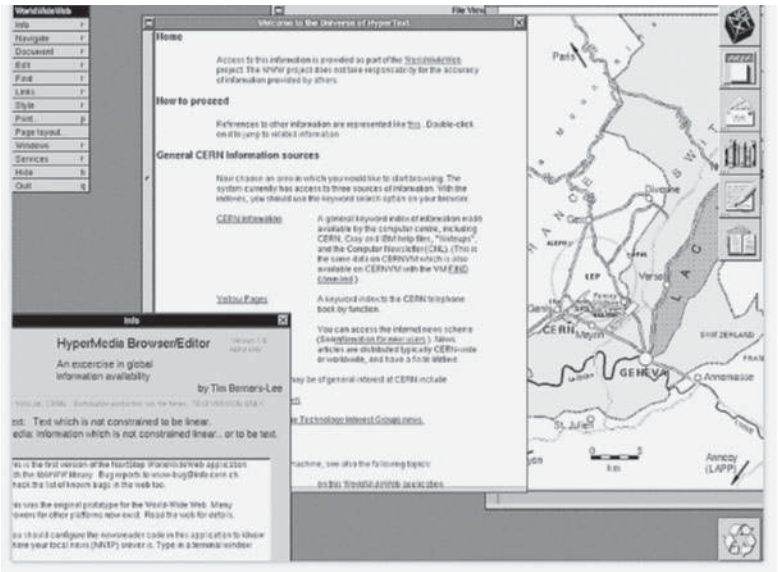
During 1991 servers appeared in other institutions in Europe and in December 1991, the first server outside the continent was installed in the US at SLAC (Stanford Linear Accelerator Center). By November 1992, there were 26 servers in the world, and by October 1993 the figure had increased to over 200 known web servers. In February 1993, the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign released the first version of Mosaic, which was to make the Web available to people using PCs and Apple Macintoshes.... and the rest is Web history.

Although the Web's conception began as a tool to aid physicists answer tough questions about the Universe, today its usage applies to various aspects of the global community and affects our daily lives. Today there are upwards of 80 million websites, with many more computers connected to the Internet, and hundreds of millions of users. If households nowadays want a computer, it is not to compute, but to go on the Web.

Tim Berners-Lee's original World Wide Web browser

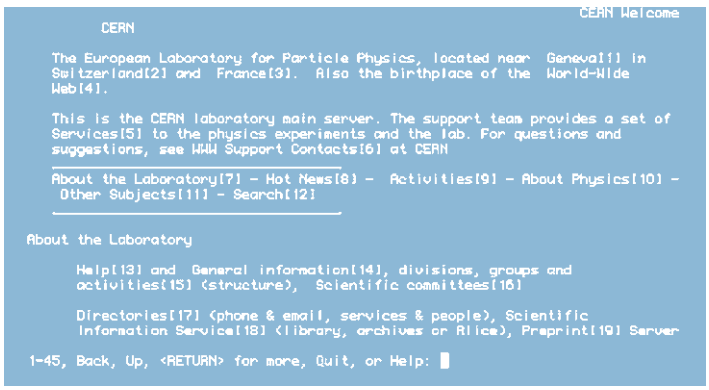
A screen shot taken from a NeXT computer running Tim Berners-Lee's original WorldWideWeb browser. It has taken a long time for technology to catch up with Berners-Lee's original vision. The first ever web browser was also an editor, making the web an interactive medium, the problem was that it only ran on the NeXTStep operating system. With recent

phenomena like blogs and wikis, the web is beginning to develop the kind of collaborative nature that its inventor envisaged from the start.



The first universal line-mode browser

During the 1990s a visitor to CERN's early homepage would have seen something like this. This looks much more primitive than the NeXT screen shots, and it was! But it was the first universal line mode browser that allowed anyone access to the Web, regardless of the kind of computer system used. When compared with the current homepage for CERN, most of the information provided is the same.



[Welcome to info.cern.ch - http://info.cern.ch/]

QUESTIONS

- **a** What does the acronym WWW means?
- **b** What idea laid behind the WWW?
- **c** Who are Tim Berners-Lee and Robert Cailliau?
- **d** What happened in February 1993?

B2 CSS (*Cascading Style Sheet*) per pagine web

L'esempio conclusivo del capitolo precedente, dedicato al linguaggio HTML, viene così visualizzato dal browser:

Sebbene il testo sia correttamente suddiviso in paragrafi, e in particolare l'*header* sia caratterizzato dalla presenza di un'immagine, la pagina risulta comunque priva di una qualsiasi impostazione grafica: ogni elemento è allineato a sinistra, non vi sono oggetti posti uno a fianco dell'altro (ma solo uno sotto l'altro) e non è presente alcuno sfondo.
Con il ricorso ai CSS (*Cascading Style Sheet*) è possibile impostare il *layout* grafico della pagina web; per esempio la pagina HTML precedente integrata con codice CSS può essere così visualizzata dal browser:

L'evoluzione dei «fogli di stile»

Fin dalla proposta iniziale, quasi coeva del linguaggio HTML, lo scopo dei «fogli di stile» è stato quello di separare il **contenuto** di una pagina web – definito utilizzando il linguaggio HTML – dalla sua **presentazione**, consentendo al browser di adattare quest'ultima alle impostazioni dell'utente e alle esigenze del dispositivo di visualizzazione.
La prima versione ufficiale dei *Cascading Style Sheet* (CSS-1), rilasciata nel 1996, offriva soluzioni per la formattazione del testo e il posizionamento degli elementi nella pagina, ma – dato lo scarso supporto da parte dei browser – incontrò poco successo.
La seconda versione (CSS-2) fu rilasciata nel 1998 e, pur essendo una naturale evoluzione della prima, si impose progressivamente come standard per l'impostazione grafica delle pagine web. L'innovativa versione 3 (CSS-3) non è ancora stata a oggi ufficialmente rilasciata, ma – dato il progressivo anche se parziale supporto da parte dei browser – è già diffusamente utilizzata.

1 Struttura del codice CSS

1.1 Inserimento di codice CSS all'interno della pagina HTML

Il codice CSS può essere inserito all'interno di una pagina HTML in tre diversi modi:

- mediante un file esterno;
- all'interno dell'intestazione della pagina (*tag* `<head>`);
- come attributo *style* in un qualsiasi *tag*.

Per inserire del codice CSS in un file esterno alla pagina è necessario porre all'interno del *tag* `<head>` il seguente *tag*:

```
<link rel="stylesheet" type="text/css" href="filename.css"/>
```

dove nell'attributo *href* è specificato il nome, o il percorso, del file avente estensione «.css» e contenente il codice CSS per la pagina.
In alternativa è possibile incorporare il codice CSS direttamente all'interno del *tag* `<head>` della pagina utilizzando il *tag* `<style>`:

```
<head>
  <style>
    …
    codice CSS
    …
  </style>
</head>
```

Infine è possibile inserire codice CSS direttamente come valore dell'attributo *style* in un qualsiasi *tag*:

```
<tag style="…codice CSS…">Lorem ipsum dolor sit amet</tag>
```

OSSERVAZIONE La scelta del metodo da utilizzare dipende fondamentalmente dalla complessità della pagina web da realizzare. Nel caso essa richieda codice CSS solamente per pochi elementi, si preferiscono gli ultimi due metodi, mentre il primo metodo è indicato per pagine in cui il codice CSS necessario per la formattazione è complesso ed è applicato a numerosi elementi della pagina.

Nella stessa pagina HTML è quindi possibile riferire un qualsiasi numero di file esterni contenenti codice CSS e inserire codice CSS sia all'interno del *tag* `<head>` sia come valore dell'attributo *style* di altri *tag*.

Il codice CSS verrà interpretato secondo il seguente **ordine a cascata** (da cui la denominazione *Cascading Style Sheet*):

- codice CSS predefinito del browser;
- codice CSS contenuto in file esterni alla pagina;
- codice CSS inserito all'interno del *tag* `<head>`;
- codice CSS assegnato come valore agli attributi *style* dei singoli *tag* della pagina.

OSSERVAZIONE Un frammento di codice CSS assegnato come valore all'attributo *style* di un *tag* della pagina prevarrà sul codice CSS contenuto in un file esterno, o definito all'interno dell'intestazione della pagina.

1.2 Sintassi del codice CSS

La sintassi del codice CSS è piuttosto semplice; essa è infatti formata da selettori che identificano l'oggetto sul quale si applicano le proprietà e da conseguenti dichiarazioni contenenti gli attributi da applicare all'oggetto con i relativi valori:

```
selettore {proprietà: valore;}
```

ESEMPIO

Il seguente codice CSS viene utilizzato per rendere rossi i caratteri di tutti i titoli della pagina definiti utilizzando il *tag* `<h1>`:

```
h1 {color: red;}
```

OSSERVAZIONE Se il codice CSS viene inserito come valore dell'attributo *style* di un *tag*, non vi è necessità di inserire alcun selettore (e di conseguenza sono omessi anche i simboli «{» e «}»): in questo caso l'elemento al quale si applica il codice CSS è quello definito dal *tag* stesso. Per esempio, se si intende visualizzare in rosso un particolare titolo definito con il *tag* `<h1>`, all'interno del corpo della pagina si inserirà il seguente codice HTML:

```
<h1 style="color: red;">Titolo</h1>
```

Eventuali commenti sono inseriti nel codice CSS comprendendoli tra i simboli «/*» e «*/», come di seguito:

```
/* commento */
```

1.3 I tipi di selettori CSS: ID e *class*

Non sempre si desidera utilizzare come selettore per l'applicazione del codice CSS un *tag* HTML con la conseguente applicazione delle proprietà specificate dal codice a tutti i *tag* dello stesso tipo presente nella pagina HTML; spesso è necessario applicare alcune caratteristiche di formattazione solamente a uno specifico *tag*, o a numero limitato di essi.

Per applicare una caratteristica di formattazione a un singolo *tag* è possibile – come si è visto in precedenza – inserire il codice CSS all'interno del *tag* stesso, oppure utilizzare un selettore di tipo ID.

Per fare questo è necessario assegnare un ID al *tag* e, all'interno del codice CSS, impiegare come selettore l'ID associato al *tag* preceduto dal simbolo «#».

ESEMPIO

Il codice HTML/CSS che segue visualizza il titolo definito dal *tag* `<h1>` in rosso:

```
<html>
  <head>
    <style>#titolorosso {color: red;}</style>
  </head>
  <body>
    <h1 ID="titolorosso">Titolo</h1>
  </body>
</html>
```

Se risulta invece necessario specificare delle caratteristiche di formattazione comuni a più elementi (*tag*) della pagina, si impiega un selettore di tipo *class* (il cui nome è invece preceduto dal simbolo «.» nel codice CSS).

ESEMPIO

Il codice HTML/CSS che segue visualizza il titolo, il sottotitolo e il testo del paragrafo definiti dai *tag* `<h1>`, `<h2>` e `<p>` in rosso perché appartenenti alla classe *rosso*:

```
<html>
  <head>
    <style>.rosso {color: red;}</style>
  </head>
  <body>
    <h1 class="rosso">Titolo</h1>
    <h2 class="rosso">Sottotitolo</h2>
    <p class="rosso">Lorem ipsum dolor sit amet</p>
  </body>
</html>
```

OSSERVAZIONE Nel caso si desideri applicare alcune caratteristiche di formattazione solamente a un tipo di *tag* tra quelli appartenenti alla classe, è possibile specificarlo inserendo nel selettore CSS il nome del *tag* prima del nome della classe, come nel seguente esempio:

```
p.rosso {color: red;}
```

Il precedente codice CSS applicherà la proprietà esclusivamente ai *tag* di tipo `<p>` appartenenti alla classe *rosso*.

In ogni caso una proprietà può essere associata a più selettori semplicemente elencandoli con il simbolo «,» come separatore:

```
selettore-1, …, selettore-N {proprietà: valore;}
```

In modo analogo a un selettore possono essere applicate più proprietà utilizzando la seguente sintassi:

```
selettore {proprietà-1: valore-1; …; proprietà-N: valore-N}
```

2 CSS: formattazione del testo

2.1 Gestire le proprietà del testo con i CSS

Utilizzando esclusivamente codice HTML è possibile solo stabilire se il testo debba essere formattato in corsivo o in grassetto, mentre non vi è modo di specificarne l'allineamento, il *font* del carattere, il colore o la dimensione. Ricorrendo al codice CSS è possibile specificare altre importanti caratteristiche di formattazione del testo:

Proprietà	Caratteristica
`color`	Colore
`text-align`	Allineamento
`text-decoration`	Decorazione

I principali colori assegnabili al testo come valori della proprietà `color` sono riportati nella TABELLA **1**.

OSSERVAZIONE I codici numerici esadecimali rappresentano il colore in formato RGB (*Red*, *Green*, *Blue*): le 6 cifre esadecimali devono essere interpretate come tre valori numerici di due cifre (da 00 a FF esadecimale, cioè da 0 a 255 decimale) corrispondenti ai tre colori primari della sintesi additiva. Nel codice CSS essi possono essere usati al posto della denominazione dei colori, consentendo di rappresentare anche i colori per i quali non esiste una denominazione standard.

TABELLA **1**

Denominazione	Codice
AliceBlue	#F0F8FF
AntiqueWhite	#FAEBD7
Aqua	#00FFFF
Aquamarine	#7FFFD4
Azure	#F0FFFF
Beige	#F5F5DC
Bisque	#FFE4C4
Black	#000000
BlanchedAlmond	#FFEBCD
Blue	#0000FF
BlueViolet	#8A2BE2
Brown	#A52A2A
BurlyWood	#DEB887
CadetBlue	#5F9EA0
Chartreuse	#7FFF00
Chocolate	#D2691E
Coral	#FF7F50
CornflowerBlue	#6495ED
Cornsilk	#FFF8DC
Crimson	#DC143C
Cyan	#00FFFF
DarkBlue	#00008B
DarkCyan	#008B8B
DarkGoldenRod	#B8860B
DarkGray	#A9A9A9
DarkGreen	#006400
DarkKhaki	#BDB76B
DarkMagenta	#8B008B
DarkOliveGreen	#556B2F
DarkOrange	#FF8C00
DarkOrchid	#9932CC
DarkRed	#8B0000
DarkSalmon	#E9967A
DarkSeaGreen	#8FBC8F
DarkSlateBlue	#483D8B
DarkSlateGray	#2F4F4F
DarkTurquoise	#00CED1
DarkViolet	#9400D3
DeepPink	#FF1493
DeepSkyBlue	#00BFFF
DimGrey	#696969
DodgerBlue	#1E90FF
FireBrick	#B22222
FloralWhite	#FFFAF0
ForestGreen	#228B22
Fuchsia	#FF00FF
Gainsboro	#DCDCDC
GhostWhite	#F8F8FF
Gold	#FFD700
GoldenRod	#DAA520
Grey	#808080
Green	#008000
GreenYellow	#ADFF2F
HoneyDew	#F0FFF0
HotPink	#FF69B4
IndianRed	#CD5C5C
Indigo	#4B0082
Ivory	#FFFFF0
Khaki	#F0E68C
Lavender	#E6E6FA
LavenderBlush	#FFF0F5
LawnGreen	#7CFC00
LemonChiffon	#FFFACD
LightBlue	#ADD8E6
LightCoral	#F08080
LightCyan	#E0FFFF
LightGoldenRodYellow	#FAFAD2
LightGrey	#D3D3D3
LightGreen	#90EE90
LightPink	#FFB6C1
LightSalmon	#FFA07A
LightSeaGreen	#20B2AA
LightSkyBlue	#87CEFA
LightSlateGrey	#778899
LightSteelBlue	#B0C4DE
LightYellow	#FFFFE0
Lime	#00FF00
LimeGreen	#32CD32
Linen	#FAF0E6
Magenta	#FF00FF
Maroon	#800000
MediumAquaMarine	#66CDAA
MediumBlue	#0000CD
MediumOrchid	#BA55D3
MediumPurple	#9370D8
MediumSeaGreen	#3CB371
MediumSlateBlue	#7B68EE
MediumSpringGreen	#00FA9A
MediumTurquoise	#48D1CC
MediumVioletRed	#C71585
MidnightBlue	#191970
MintCream	#F5FFFA
MistyRose	#FFE4E1
Moccasin	#FFE4B5
NavajoWhite	#FFDEAD
Navy	#000080
OldLace	#FDF5E6
Olive	#808000
OliveDrab	#6B8E23
Orange	#FFA500
OrangeRed	#FF4500
Orchid	#DA70D6
PaleGoldenRod	#EEE8AA
PaleGreen	#98FB98
PaleTurquoise	#AFEEEE
PaleVioletRed	#D87093
PapayaWhip	#FFEFD5
PeachPuff	#FFDAB9
Peru	#CD853F
Pink	#FFC0CB
Plum	#DDA0DD
PowderBlue	#B0E0E6
Purple	#800080
Red	#FF0000
RosyBrown	#BC8F8F
RoyalBlue	#4169E1
SaddleBrown	#8B4513
Salmon	#FA8072
SandyBrown	#F4A460
SeaGreen	#2E8B57
SeaShell	#FFF5EE
Sienna	#A0522D
Silver	#C0C0C0
SkyBlue	#87CEEB
SlateBlue	#6A5ACD
SlateGrey	#708090
Snow	#FFFAFA
SpringGreen	#00FF7F
SteelBlue	#4682B4
Tan	#D2B48C
Teal	#008080
Thistle	#D8BFD8
Tomato	#FF6347
Turquoise	#40E0D0
Violet	#EE82EE
Wheat	#F5DEB3
White	#FFFFFF
WhiteSmoke	#F5F5F5
Yellow	#FFFF00
YellowGreen	#9ACD32

I possibili valori per la proprietà `text-align` sono riportati nella TABELLA **2**.

TABELLA **2**

Valori proprietà	Tipo di allineamento
`left`	Sinistra (valore predefinito)
`right`	Destra
`center`	Centrato
`justify`	Giustificato

ESEMPIO

Per giustificare il testo dei paragrafi e centrare i titoli di tipo `<h1>` si definisce il seguente codice CSS:

```
p {text-align: justify;}
h1 {text-align: center;}
```

I possibili valori per la proprietà `text-decoration` sono elencati nella TABELLA **3**.

TABELLA **3**

Valori proprietà	Tipo di decorazione
`none`	Nessuna (valore predefinito)
`underline`	Sottolineato
`line-through`	Barrato
`blink`	Lampeggiante
`overline`	Sopralineato

ESEMPIO

Per sottolineare il testo dei paragrafi di classe *sottolineato* e rendere lampeggianti i titoli di tipo `<h1>` si definisce il seguente codice CSS:

```
.sottolineato {text-decoration: underline;}
h2 {text-decoration: blink;}
```

ESEMPIO

Per definire la pagina web che il browser visualizza nel seguente modo

è stato utilizzato il seguente codice HTML/CSS:

```
<!doctype html>
  <html>
    <head>
      <title>Testo gestito con CSS</title>
      <style>
        body {text-align: center;}
        h2 {text-decoration: underline;}
        .rosso {color: red;}
      </style>
    </head>
    <body>
      <h1>Titolo</h1>
      <h2>sottotitolo</h2>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
         Mauris et ipsum quis nunc euismod aliquet.
         Quisque at rhoncus sapien.</p>
      <p class="rosso">
         Lorem ipsum dolor sit amet, consectetur adipiscing elit.
         Mauris et ipsum quis nunc euismod aliquet.
         Quisque at rhoncus sapien.</p>
    </body>
  </html>
```

Impiegando i CSS è anche possibile operare sulla selezione del tipo e della dimensione del carattere del testo, in particolare valorizzando le seguenti proprietà:

Proprietà	Caratteristica
`font-family`	Tipo di carattere
`font-size`	Dimensione del testo

All'attributo `font-family` è possibile assegnare un valore generico corrispondente a una categoria di caratteri, oppure il nome di uno specifico *font*. Le tre categorie generiche sono le seguenti:

`Serif`	*Font* di caratteri con grazie[1]
`Sans-serif`	*Font* di caratteri senza grazie[2]
`Monospace`	*Font* in cui tutti i caratteri hanno la stessa dimensione orizzontale

Nel caso si desideri formattare il testo utilizzando uno specifico *font* di caratteri, è necessario importare il *font* stesso nel codice CSS assegnandogli un nome di riferimento e utilizzando la seguente sintassi:

```
@font-face
{
  font-family: nome_font;
  src: url(nome_file.ttf);
}
```

1. I caratteri tipografici con le grazie presentano alle estremità dei tratti terminali ortogonali al segno del carattere denominate *grazie*.

2. I caratteri tipografici senza grazie, o a *bastone*, non presentano alle estremità i tratti terminali noti come *grazie*.

OSSERVAZIONE Il file di definizione del *font* di caratteri, avente estensione «.ttf», deve necessariamente essere posizionato nella stessa directory che contiene il file HTML della pagina web, o il file CSS esterno alla pagina.

Per esempio, volendo applicare il *font* importato al testo dei paragrafi, si impiega la seguente sintassi CSS:

```
p {font-family: nome_font;}
```

La dimensione del testo viene invece specificata attraverso l'attributo `font-size`, indicando un valore corrispondente alla larghezza di ogni singolo carattere.

OSSERVAZIONE Sebbene la dimensione dei caratteri possa essere indicata in pixel (*px*), si preferisce usare come unità di misura *em*[3], in quanto unità di misura relativa e non assoluta. La dimensione standard – assegnata dal browser – è solitamente di 1 *em*, corrispondente nella maggior parte dei casi a 16 pixel. In ogni caso, per assicurarsi che ogni browser visualizzi il testo esattamente nella dimensione indicata, è buona regola applicare anche al *tag* `<body>` la proprietà `font-size` indicando come valore «100%»:

```
body {font-size: 100%;}
```

3. Il nome *em* deriva dalla pratica tipografica di considerare come riferimento la larghezza del carattere «M».

ESEMPIO

Per assegnare la dimensione di 2,10 *em* ai titoli di tipo `<h1>` deve essere usato il seguente codice CSS:

```
p {font-size: 2.10em;}
```

ESEMPIO

Il seguente esempio mostra come assegnare una dimensione di 3 *em* e un particolare *font* (definito nel file «Whiskey_Town-Buzzed.ttf») al titolo di tipo `<h1>` della pagina:

```
<!doctype html>
  <html>
    <head>
      <title>Assegnazione font al titolo</title>
      <style>
        @font-face
        {
          font-family: WhiskeyTown;
          src: url(Whiskey_Town-Buzzed.ttf);
        }
        body {text-align: center; font-size: 100%;}
```

▶

```
        h1 {font-family: WhiskeyTown; font-size: 3em;}
        .rosso {color: red;}
      </style>
    </head>
    <body>
      <h1>Titolo</h1>
      <p class="rosso">
         Lorem ipsum dolor sit amet, consectetur adipiscing elit.
         Mauris et ipsum quis nunc euismod aliquet.
         Quisque at rhoncus sapien.</p>
    </body>
  </html>
```

2.2 Gestire lo stile dei *link* testuali

Per un *link* HTML (realizzato con il *tag* `<a>`) si possono identificare i seguenti stati:

- non visitato (`link`);
- visitato (`visited`);
- cursore del mouse sopra il *link* (`hover`);
- click del mouse sopra il *link* (`active`).

I quattro diversi stati sono utilizzabili come selettori CSS mediante la seguente sintassi:

- `a:link`
- `a:visited`
- `a:hover`
- `a:active`

È possibile specificare per ciascuno stato alcune proprietà riguardanti lo stile: `color`, `font-decoration`[4] e `background-color`, che indica il colore di sfondo del link.

4. Viene spesso utilizzato con valore *none* per eliminare la sottolineatura predefinita.

ESEMPIO

Il codice CSS incorporato nella pagina HTML che segue illustra come assegnare uno stile diverso da quello predefinito ai *link* presenti nella pagina, in particolare:

- bianco con sfondo nero in condizioni normali;
- verde con sfondo nero una volta visitato;
- bianco con sfondo arancione con il cursore del mouse sovrapposto;
- arancione con sfondo nero al momento dell'attivazione.

```
<!doctype html>
  <html>
    <head>
      <title>Personalizzazione link</title>
      <style>
      a:link      {
                    color: white;
                    font-decoration: none;
                    background-color: black;
                  }
      a:visited   {
                    color: green;
                    font-decoration: none;
                    background-color: black;
                  }
      a:hover     {
                    color: white;
                    font-decoration: none;
                    background-color: orange;
                  }
      a:active    {
                    color: orange;
                    font-decoration: none;
                    background-color: black;
                  }
      </style>
    </head>
    <body>
      <p>
        <a href="http://www.google.com">Vai a Google</a>
      </p>
    </body>
  </html>
```

3 Sfondi

A numerosi elementi HTML è possibile assegnare uno sfondo, sia esso un colore o un'immagine. Per applicare un colore di sfondo all'intero corpo della pagina web si utilizza la proprietà `background-color`, specificando il colore mediante il nome o il codice esadecimale[5].

ESEMPIO

Per visualizzare una pagina con sfondo nero è possibile utilizzare il seguente codice CSS:

```
body {background-color: #000000;}
```

5. I nomi e i codici esadecimali dei colori sono quelli elencati nella TABELLA **1**.

Per utilizzare invece un'immagine come sfondo della pagina web si ricorre alla proprietà `background-image`, specificando il nome/percorso o l'indirizzo del file contenente l'immagine.

ESEMPIO

Per visualizzare una pagina avente come sfondo l'immagine contenuta nel file «immagine.jpg» posizionato nella stessa directory del file contenente il codice HTML/CSS è possibile utilizzare il seguente codice CSS:

```
body {background-image: url(immagine.jpg);}
```

È possibile determinare se l'immagine di sfondo debba essere o meno ripetuta e, in caso di ripetizione, se solo in orizzontale, in verticale o in entrambe le direzioni. La proprietà coinvolta è `background-repeat`, che può assumere i valori indicati in TABELLA **4**.

TABELLA **4**

Valori proprietà	Tipo di ripetizione
`no-repeat`	Nessuna ripetizione
`repeat-x`	Ripetizione orizzontale
`repeat-y`	Ripetizione verticale

Se l'immagine di sfondo non viene ripetuta, diventa importante il suo posizionamento nella pagina web visualizzata dal browser; la proprietà `background-position` consente di specificare i valori indicati nella TABELLA **5**.

TABELLA **5**

Valori proprietà	Tipo di posizionamento
`left top`	A sinistra in alto
`left center`	A sinistra al centro
`left bottom`	A sinistra in basso
`center top`	Al centro in alto
`center center`	Al centro
`center bottom`	Al centro in basso
`right top`	A destra in alto
`right center`	A destra al centro
`right bottom`	A destra in basso

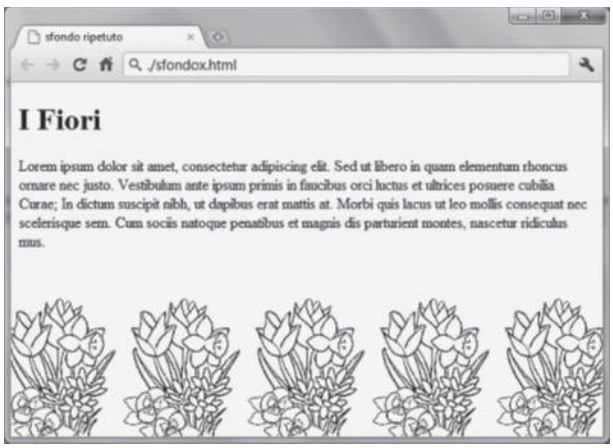
OSSERVAZIONE Come impostazione predefinita un'immagine di sfondo viene posizionata in alto a sinistra e ripetuta sia in verticale sia in orizzontale. Per fare in modo che lo sfondo di una pagina web non sia ripetuto e sia centrato nella pagina deve essere utilizzato il seguente codice CSS:

```
body{background-image: url(immagine.jpg);
     background-repeat: no-repeat;
     background-position: center;
    }
```

ESEMPIO

In questo caso l'immagine di sfondo associata al corpo della pagina web è ripetuta solo in direzione orizzontale ed è posizionata in basso:

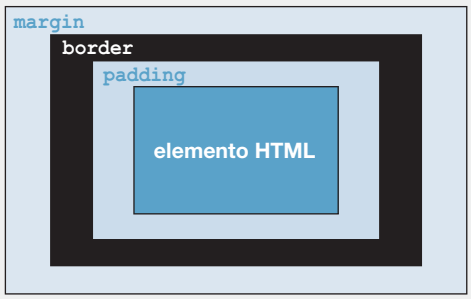
```
<!doctype html>
  <html>
    <head>
      <title>sfondo ripetuto</title>
      <style>
        body {
              background-image: url(fiori.gif);
              background-repeat: repeat-x;
              background-position: bottom;
            }
      </style>
    </head>
    <body>
      <h1>I Fiori</h1>
      <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed
         ut libero in quam elementum rhoncus ornare nec justo.
         Vestibulum ante ipsum primis in faucibus orci luctus et
         ultrices posuere cubilia Curae; In dictum suscipit nibh, ut
         dapibus erat mattis at. Morbi quis lacus ut leo mollis
         consequat nec scelerisque sem. Cum sociis natoque penatibus et
         magnis dis parturient montes, nascetur ridiculus mus.</p>
      <!-- I seguenti paragrafi vuoti servono solo per aumentare
           la lunghezza della pagina-->
      <p> </p> <p> </p> <p> </p> <p> </p>
      <p> </p>
    </body>
  </html>
```



4 Il modello «a scatola»

L'allineamento degli oggetti è ottenuto in CSS tramite i valori assegnati alle proprietà `margin`, `border` e `padding` che, insieme, costituiscono il cosiddetto **modello «a scatola»** (*box model*). I valori assegnati a ognuna di

queste proprietà consentono infatti di posizionare un qualsiasi elemento HTML in una precisa posizione – assoluta o relativa – all'interno della pagina. Lo schema che segue identifica il significato associato a queste proprietà:



I valori delle proprietà `margin`, `border` e `padding` possono essere assegnati in pixel, in percentuale, o in *em*, ma di questi solo il bordo è visibile (predefinito di colore nero).

ESEMPIO

I paragrafi di classe *contenuto* potranno essere allineati in base alla seguente proprietà:

```
p.contenuto {border: 2px; padding: 3px; margin: 12em;}
```

Nel caso si desideri applicare valori diversi ai quattro lati dell'elemento (destro, sinistro, alto e basso) è possibile combinare le tre proprietà con le parole chiave `right`, `left`, `top` e `bottom`.

ESEMPIO

Per assegnare al margine sinistro il valore di 10 *em*, mentre al margine destro il valore è di 15 *em*, è necessario il seguente codice:

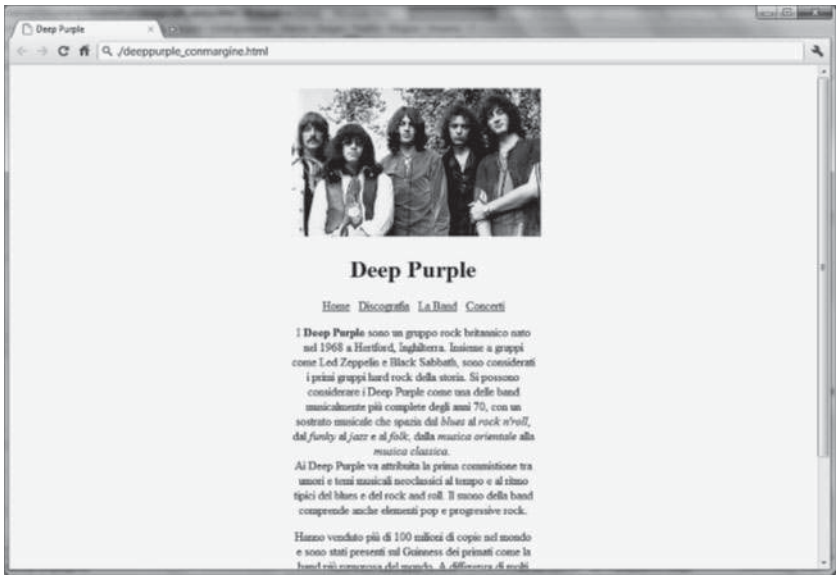
```
p.contenuto {
              border: 2px; padding: 3px;
              margin-left: 10em; margin-right: 15em;
            }
```

ESEMPIO

Aggiungendo il semplice codice CSS riportato nel seguito all'esempio conclusivo del capitolo relativo al linguaggio HTML, si ottiene una pagina web dall'aspetto grafico molto piacevole, dove ogni elemento della pagina è allineato al centro:

```
body {
       margin-left: 33%; margin-right: 33%; margin-top: 3%;
       text-align: center;
     }
```

▶



4.1 Formattazione dei bordi delle tabelle

La definizione delle tabelle mediante i soli *tag* HTML comporta l'assenza di bordi – sia esterni sia interni – nella visualizzazione prodotta dal browser. È possibile visualizzare tabelle dotate di bordi impostando opportunamente la proprietà `border`[6] (TABELLA 6).

6. La proprietà `border` è in realtà una «scorciatoia» per le proprietà `border-style`, `border-width`, `border-color`, …

TABELLA 6

Valori proprietà	Tipo di bordo
`solid thin`	Sottile continuo
`solid medium`	Medio continuo
`solid thick`	Spesso continuo

ESEMPIO

La seguente pagina HTML

```
<!doctype html>
  <html>
    <head>
      <title>Pagina con tabella 2x2</title>
      <style>
        td {border: solid medium;}
      </style>
    </head>
    <body>
      <table>
        <tr>
          <td>cella [1,1]</td>
          <td>cella [1,2]</td>
```

▶

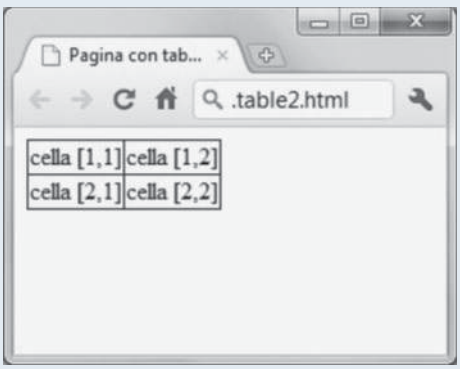
```
        </tr>
        <tr>
          <td>cella [2,1]</td>
          <td>cella [2,2]</td>
        </tr>
      </table>
    </body>
  </html>
```

viene visualizzata dal browser nel seguente modo:

OSSERVAZIONE Nell'esempio precedente le singole celle della tabella hanno bordi separati e indipendenti; per ottenere la visualizzazione di una tabella in cui i bordi interni sono comuni a due celle contigue è necessario aggiungere nel *tag* `<style>` il seguente codice CSS:

```
table {border-collapse: collapse;}
```

Con questa integrazione, e la definizione del bordo di tipo `solid thin`, la pagina viene visualizzata nel seguente modo:



4.2 Combinare tra loro varie proprietà CSS

Una pagina HTML di elevata complessità richiede tipi diversi di proprietà CSS combinate fra loro. Infatti, come abbiamo visto, è possibile inserire

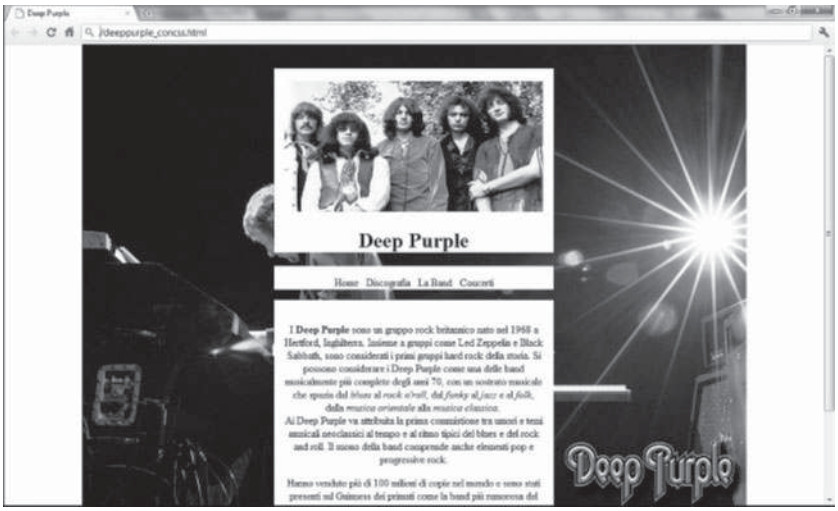
all'interno della stessa pagina HTML proprietà CSS senza alcun limite, o rischio di incompatibilità fra di esse.

ESEMPIO

Il codice CSS associato alla pagina dell'esempio conclusivo del capitolo relativo al linguaggio HTML è di seguito integrato con altre proprietà relative allo sfondo e allo stile dei collegamenti ipertestuali (*link*):

```
body {
       margin-left: 33%;
       margin-right: 33%;
       margin-top: 3%;
       text-align: center;
       background-image: url("deepback.jpg");
       background-position: top;
       background-repeat: no-repeat;
     }
header, h1, nav, article {
                              background-color: white;
                              padding-left:1em;padding-right:1em;
                            }
nav {padding-top: 1em;}
a:link {text-decoration: none; color: black;}
```

Con queste integrazioni la pagina viene così visualizzata dal browser:



OSSERVAZIONE Come abbiamo visto CSS non è un linguaggio complesso, ma le funzionalità che rende disponibili sono numerosissime: l'immagine della pagina web dedicata ai Deep Purple in apertura di questo capitolo impiega alcune funzionalità CSS (come la gestione della trasparenza) che vanno al di là dello scopo di questa trattazione introduttiva. Ormai, più che scrivere direttamente codice CSS, i creatori di siti web utilizzano degli editor grafici che permettono di definire il *layout*, componenti e caratteristiche estetiche delle singole pagine e che generano automaticamente il codice CSS sotteso.

Sintesi

CSS. I CSS (*Cascading Style Sheet*) o *fogli di stile* per pagine web, permettono di separare il contenuto di una pagina web – definito utilizzando il linguaggio HTML – dalla sua presentazione grafica, consentendo al browser di adattare quest'ultima alle impostazioni dell'utente e alle esigenze del dispositivo di visualizzazione.

CSS e pagine HTML. Il codice CSS può essere associato a una pagina HTML in tre modi diversi:

- mediante un file esterno;
- all'interno dell'intestazione della pagina (*tag* `<head>`);
- come attributo *style* di un qualsiasi *tag*.

La scelta del metodo da utilizzare dipende dalla complessità della pagina web da realizzare. Nel caso essa richieda codice CSS solo per pochi elementi, si preferiscono gli ultimi due metodi, mentre il primo metodo è indicato per pagine in cui il codice CSS necessario per la formattazione sia complesso e applicato a numerosi elementi della pagina.

Interpretazione del codice CSS. In una stessa pagina HTML è possibile riferire un qualsiasi numero di file esterni contenenti codice CSS e inserire codice CSS sia all'interno del *tag* `<head>` sia come valore dell'attributo *style* di altri *tag*. Il codice CSS verrà interpretato secondo il seguente ordine «a cascata» (da cui la denominazione *Cascading Style Sheet*):

- codice CSS predefinito del browser;
- codice CSS contenuto in file esterni alla pagina;
- codice CSS inserito all'interno del *tag* `<head>`;
- codice CSS assegnato come valore agli attributi *style* dei singoli *tag* della pagina.

Un frammento di codice CSS assegnato come valore all'attributo *style* di un *tag* della pagina prevarrà sul codice CSS contenuto in un file esterno, o definito all'interno dell'intestazione della pagina.

Sintassi del codice CSS. La sintassi del codice CSS è formata da selettori, che identificano l'oggetto sul quale si applicano le proprietà, e da conseguenti dichiarazioni contenenti gli attributi da applicare all'oggetto con i relativi valori: `selettore {proprietà: valore;}`. Per esempio, il seguente codice CSS viene utilizzato per rendere rossi i caratteri di tutti i titoli della pagina definiti utilizzando il *tag* `<h1>`: `h1 {color: red;}`.

I tipi di selettori CSS. I CSS utilizzano due diversi tipi di selettori: ID e *class*. Nel primo caso si tratta di usare il selettore ID per applicare una caratteristica di formattazione a un singolo *tag* (in alternativa all'inserimento di tale caratteristica direttamente nel *tag* stesso). È necessario assegnare un ID (per esempio una stringa) al *tag* e, all'interno del codice CSS, impiegare come selettore l'ID associato al *tag* preceduto dal simbolo «#». L'uso del selettore *class* (il cui nome è preceduto dal simbolo «.» nel codice CSS) risulta invece più indicato per associare delle caratteristiche di formattazione comuni a più elementi (*tag*) della pagina.

Proprietà del testo con i CSS. L'HTML permette solo di stabilire se il testo debba essere formattato in corsivo o in grassetto. I CSS permettono di specificare diverse altre caratteristiche con cui il testo delle pagine web deve essere visualizzato: *font*, dimensione, colore, allineamento, decorazione.

Stile dei *link* testuali. Per un *link* HTML (realizzato con il tag `<a>`) si possono identificare i seguenti stati:

- non visitato (`link`);
- visitato (`visited`);
- cursore del mouse sopra il *link* (`hover`);
- click del mouse sopra il *link* (`active`).

I quattro diversi stati sono utilizzabili come selettori CSS mediante la seguente sintassi:

- `a:link`;
- `a:visited`;
- `a:hover`;
- `a:active`

con la possibilità di specificare per ciascuno stato alcune proprietà riguardanti lo stile: `color`, `font-decoration` e `background-color` che indica il colore di sfondo del *link*.

Sfondi. A numerosi elementi HTML è possibile assegnare uno sfondo, sia esso un co-

lore o un'immagine. Per applicare un colore di sfondo all'intero corpo della pagina web si utilizza la proprietà `background-color`, applicata al *body*, specificando il colore mediante il nome o il corrispondente codice esadecimale. Per utilizzare invece un'immagine come sfondo della pagina web, si ricorre alla proprietà `background-image`, specificando il nome/percorso o l'indirizzo del file contenente l'immagine. È possibile determinare se l'immagine di sfondo debba essere o meno ripetuta e, in caso di ripetizione, se solo in orizzontale, in verticale o in entrambe le direzioni tramite la proprietà `background-repeat`.

Allineamento degli oggetti. L'allineamento degli oggetti è ottenuto in CSS tramite i valori assegnati alle proprietà `margin`, `border` e `padding` che, insieme, costituiscono il cosiddetto modello «a scatola» (*box model*). I valori assegnati a ognuna di queste proprietà consentono di posizionare un qualsiasi elemento HTML in una precisa posizione – assoluta o relativa – all'interno della pagina.

QUESITI

1 Il significato della sigla CSS è …

A … *Cascading Symbolic Sheet*.
B … *Cascade Style System*.
C … *Component Style Sheet*.
D Nessuna delle risposte precedenti.

2 Il codice CSS può riferirsi a una pagina HTML …

A … mediante un file esterno.
B … mediante un file interno.
C … come attributo *style* in un qualsiasi *tag*.
D Nessuna delle risposte precedenti.

3 Il termine *Cascading* si riferisce al fatto che il codice CSS viene interpretato …

A … in cascata secondo la priorità: browser, file esterni alla pagina, *tag* `<head>`, attributi *style* dei singoli *tag*.
B … in cascata secondo la priorità: file esterni alla pagina, *tag* `<head>`, attributi *style* dei singoli *tag*, browser.
C … in cascata secondo la priorità: *tag* `<head>`, file esterni alla pagina, attributi *style* dei singoli *tag*, browser.
D Nessuna delle risposte precedenti.

4 L'attributo *style* di un qualsiasi *tag* costituisce …

A … un elemento sintattico che identifica il *tag* con relativo valore.
B … una modalità per specificare del codice CSS all'interno di una pagina HTML.
C … una modalità per specificare l'intestazione di una pagina HTML.
D Nessuna delle risposte precedenti.

5 Il codice CSS `h1 {color: green;}` permette di …

A … visualizzare in verde il testo di una pagina.
B … visualizzare in verde lo sfondo di una pagina.
C … visualizzare in verde il primo titolo di una pagina.
D Nessuna delle risposte precedenti.

6 Un selettore di tipo ID permette di …

A … applicare una caratteristica di formattazione a un singolo *tag*.
B … applicare una caratteristica di formattazione a tutti i *tag* di una pagina.
C … applicare una caratteristica di formattazione ai *tag* di tipo `<identification>`.
D Nessuna delle risposte precedenti.

7 Un selettore di tipo *class* permette di …

A … applicare caratteristiche di formattazione a più elementi di una pagina.
B … applicare caratteristiche di formattazione a tutti i *tag* di una pagina.
C … applicare caratteristiche di formattazione ai *tag* di tipo `<classification>`.
D Nessuna delle risposte precedenti.

8 Per avere tutto il testo di una pagina centrato rispetto alla stessa si può usare …

A … `body{text-align:center}`.

B … `body{align:center}`.

C … `body{font-align:center}`.

D Nessuna delle risposte precedenti.

9 Quali dei seguenti sono stati dei *link* testuali …

A … `link`.

B … `over`.

C … `visited`.

D Nessuna delle risposte precedenti.

10 Quali dei seguenti sono selettori CSS per gli stati dei *link* testuali …

A … `a.link`.

B … `a.active`.

C … `a.blinked`.

D Nessuna delle risposte precedenti.

11 Il seguente codice CSS

```
a:visited {
            color: green;
            font-decoration: none;
            background-color: black;
          }
```

permette di fare in modo che …

A … un *link* testuale già visitato venga visualizzato in verde su fondo nero.

B … un *link* testuale da visitare venga visualizzato in verde su fondo nero.

C … un *link* testuale già visitato venga visualizzato in nero su fondo verde.

D Nessuna delle risposte precedenti.

12 L'allineamento degli oggetti in CSS è ottenuto con un modello a scatola le cui caratteristiche sono definite tramite gli attributi …

A … `box`, `padding`, `border`.

B … `margin`, `padding`, `border`.

C … `box`, `margin`, `border`.

D Nessuna delle risposte precedenti.

13 L'impostazione predefinita di un'immagine di sfondo è …

A … in basso a destra e ripetuta solo in orizzontale.

B … in alto a sinistra e ripetuta sia in orizzontale sia in verticale.

C … in alto a sinistra e ripetuta solo in verticale.

D Nessuna delle risposte precedenti.

14 Il codice CSS

```
<style>
  td {border: solid thin;}
  table {border-collapse: collapse;}
```

inserito

```
</style>
```

permette di …

A … definire una tabella di una riga e una colonna.

B … definire lo stile per una tabella i cui elementi siano separati da linee contigue a tratto fine.

C … definire lo stile per una tabella i cui elementi siano separati da linee non contigue a tratto fine.

D Nessuna delle risposte precedenti.

Indice analitico

D

E

F

G

H

Fiorenzo Formichi Giorgio Meini

Corso di informatica

per Informatica

Algoritmi e linguaggio C++
Pagine web

Un corso di informatica focalizzato sulla pratica di laboratorio, mediante esempi graduali e attinenti ad aspetti professionalmente rilevanti. I contenuti proposti e gli esempi applicativi sono illustrati attraverso i linguaggi di programmazione più diffusi nella pratica professionale.

Nel libro

- Il testo è diviso in due sezioni: la prima dedicata a un tema fondamentale (*Algoritmi e linguaggio C++*), la seconda a una tematica web (*Pagine web*).
- Una **sintesi di fine capitolo** introduce i quesiti e gli esercizi.
- **Brani in inglese**, tratti da testi di riferimento della disciplina, sono accompagnati da un glossario.

NOVITÀ 2012