Dispense del corso di Informatica I

A.A. 2011/12

Corso di Laurea in Matematica

Marco Baioletti

1 Introduzione

Queste dispense sono divise in due parti.

Nella prima parte sarà svolta una breve introduzione ai concetti di base dell'informatica (architettura degli elaboratori, sistemi operativi, rappresentazione dell'informazione, elementi di base dell'algoritmica e della programmazione degli elaboratori).

La seconda parte tratterà dei principali elementi della programmazione imperativa, usando come linguaggio di programmazione quello presente in Matlab. In realtà si farà riferimento ad Octave, un programma gratuito molto simile a Matlab.

Queste dispense sono state scritte essenzialmente per il primo corso di informatica per una laurea triennale in matematica, ma possono essere utilizzate anche in altri corsi di laurea scientifici.

1.1 Una definizione di computer

Il computer può essere definito come un dispositivo programmabile di elaborazione dati. Inoltre opera mediante una tecnologia essenzialmente elettronica ed tratta dati rappresentati in forma digitale (preferibilmente binaria).

Lo scopo di un computer è essenzialmente quello di **elaborare dati**, ovvero partendo da dati in ingresso e, mediante un procedimento più o meno complesso, ottenere dei risultati in uscita. In generale i dati che si possono processare possono essere sia numerici sia non numerici. Un esempio di elaborazione numerica è il calcolo approssimato dell'integrale definito di una funzione ad una variabile reale, mentre un esempio di elaborazione non numerica potrebbe essere ordinare un elenco dei dati anagrafici di un insieme di persone in base al cognome.

Il computer è **programmabile**, ovvero è possibile fornire al computer dei programmi da eseguire. Anzi il suo unico scopo è proprio quello di eseguire programmi: infatti il computer non è in grado di prendere decisioni in proprio ed eseguire azioni al di là di quelle indicate nel programma. Inoltre il programma deve prevedere ogni possibile situazione in cui si può trovare il computer per indicargli come si deve comportare in quel frangente. E' importante capire che la presenza del programma da eseguire è essenziale: un computer senza programmi non serve a niente.

I computer attualmente usano una tecnologia **elettronica** basata sui transistor e sui circuiti integrati. Tale tecnologia consente di avere componenti estremamente veloci, di piccolissime dimensioni e molto economiche. In passato i computer usavano componenti elettromeccanici e prima ancora puramente meccaniche. In futuro forse si useranno componenti di tipo quantistico che potrebbero consentire dimensioni molto ridotte e tempi di calcolo minori di quelli attuali.

I dati elaborati dai computer moderni sono **digitali**, cioè rappresentati come sequenze di cifre, con un numero finito (e piccolo) di cifre possibili. La base comunemente usata nell'informatica è la base due, che usa solo due cifre: 0 e 1. Due è il numero minimo di cifre utilizzabile in un sistema di numerazione, l'uomo usa la base dieci per motivi storici (probabilmente perché ha dieci dita), ma in un computer due cifre sono più semplici da trattare che dieci (o un altro numero).

Uno degli esempi più convincenti della convenienza della base due è la scheda perforata, tra i più antichi sistemi di memorizzazione, ormai non più usato nell'informatica. Le cifre 0 e 1 possono essere rappresentate semplicemente come assenza o presenza di un foro nella scheda: rilevare se la scheda è bucata in un punto è abbastanza facile. Se ci fossero altre cifre bisognerebbe usare fori di grandezza o forma diversa, rendendo complicata la lettura dei dati e molto probabile un'errata rilevazione. In molte altre situazioni due cifre possono essere rappresentate facilmente, mentre basi maggiori sono decisamente più difficili da trattare.

E' evidente che i dati numerici sono facilmente esprimibili come sequenze di cifre. Se invece si vogliono trattare dati, come immagini, filmati, suoni, parlato, ecc., si devono usare particolari conversioni che trasformano i dati di luminosità, colore, frequenza ed altezza del suono, ecc. in

forma digitale. Agli albori dell'informatica i computer trattavano quantità numeriche in forma analogica, ma poi la maggiore versatilità della forma digitale ha nettamente prevalso, soppiantando completamente i calcolatori analogici.

Del resto in molti campi l'uso della tecnologia analogica si sta progressivamente sostituendo a quella digitale, ad esempio orologi, bilance e contachilometri che una volta erano solo analogici (usando lancette o altri strumenti) adesso sono anche digitali. In maniera simile si può osservare il passaggio dai supporti analogici a quelli digitali nella musica (dischi in vinile e musicassette sostituite dai CD) e nei film (DVD al posto di pellicole e videocassette).

1.2 Hardware e software

Un computer ha una serie di componenti fisiche, chiamate generalmente **hardware**, che sono studiate da una particolare branca dell'informatica., l'architettura degli elaboratori.

L'insieme dei programmi utilizzabili in un computer si chiama invece **software**. Come si vedrà esistono due categorie di software: quello di base e quello applicativo.

Chiaramente hardware e software hanno l'uno bisogno dell'altro e per certi versi si possono compensare. Ad esempio un'operazione può essere implementata in hardware (tramite un circuito logico, ad esempio) o in software (attraverso una funzione di libreria). La prima soluzione consente una maggiore efficienza, dato che l'operazione è eseguita direttamente dall'hardware, mentre la seconda deve essere scritta in termini di operazioni più elementari. Ma la seconda soluzione è decisamente più versatile: innanzitutto il software può essere aggiornato, modificato e inserito facilmente, l'hardware ovviamente no. Inoltre produrre nuove componenti software è possibile anche per un utente normale, mentre la produzione in proprio di nuove componenti hardware è un'impresa estremamente difficile, perché necessita di strumentazioni e materiali particolari.

2 Architettura degli elaboratori

In questo capitolo vedremo in dettaglio le componenti principali di un computer. Ma prima avremo bisogno di introdurre l'algebra di Boole.

2.1 L'algebra di Boole

Il computer lavora essenzialmente su sequenze di **bit**, ovvero di cifre binarie 0 e 1. Pertanto vediamo la cosiddetta algebra di Boole, la quale studia le operazioni algebriche sull'insieme B={0,1}. Le tre operazioni di base su B sono AND, OR e NOT.

2.1.1 Operazione AND

L'operazione di prodotto logico AND:$B^2 \rightarrow B$ applicata a due argomenti x e y produce come risultato un bit secondo la tabella

x	*y*	*x AND y*
0	0	0
0	1	0
1	0	0
1	1	1

Tale operazione è associativa e commutativa, ha 1 come elemento neutro e 0 come elemento assorbente (0 AND x=x AND 0=0).

2.1.2 Operazione OR

L'operazione di somma logica OR:$B^2 \rightarrow B$ applicata a due argomenti x e y produce come risultato un bit secondo la tabella

x	*y*	*x OR y*
0	0	0
0	1	1
1	0	1
1	1	1

Tale operazione è associativa e commutativa, ha 0 come elemento neutro e 1 come elemento assorbente (1 OR x=x OR 1=1).

Inoltre AND è distributiva rispetto a OR e viceversa, ovvero

- x AND (y OR z)=(x AND y) OR (x AND z)
- x OR (y AND z)=(x OR y) AND (x OR z)

2.1.3 Operazione NOT

L'operazione di complemento NOT:$B \rightarrow B$ applicata all'argomento x produce come risultato un bit secondo la tabella

x	*NOT x*
0	1
1	0

Tale operazione è involutiva (NOT NOT x=x), inoltre valgono le leggi di De Morgan

- NOT(x AND y)=(NOT x) OR (NOT y)
- NOT(x OR y)=(NOT x) AND (NOT y)

2.1.4 Connessioni con la teoria degli insiemi e la logica

Le operazioni AND, OR e NOT sono strettamente collegate alle operazioni insiemistiche di intersezione, unione e complementare, rispettivamente.

La connessione maggiore è però con i connettivi della logica proposizionale: AND corrisponde alla congiunzione, OR alla disgiunzione e NOT alla negazione. Infatti se x è il valore di verità della proposizione P e y quello della proposizione Q, allora

- x AND y è il valore di verità di $P \wedge Q$
- x OR y è il valore di verità di $P \vee Q$
- NOT x è il valore di verità di $\neg P$

2.1.5 Teorema di Shannon e implementazione circuitale

Le operazioni AND, OR e NOT sono le operazioni di base con le quali è possibile realizzare qualsiasi operazione sui bit. Infatti vale il seguente teorema, dovuto a Shannon:

Qualsiasi funzione binaria $f:B^n \to B^m$ *si può esprimere mediante una composizione finita di AND, OR e NOT.*

Ad esempio facciamo vedere come ottenere l'operazione XOR (o EX-OR), che ha due argomenti *x* e *y* e produce come risultato un bit secondo la tabella

x	*y*	*x XOR y*
0	0	0
0	1	1
1	0	1
1	1	0

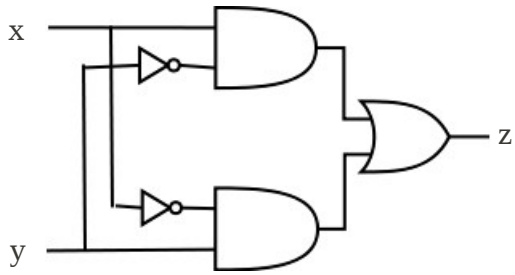
Tale operazione corrisponde alla somma nel campo $\mathbb{Z}_2$.

Applicando il teorema di Shannon si ottiene la seguente formula

x XOR y=(x AND (NOT y)) OR ((NOT x) AND y).

Una conseguenza pratica è la possibilità di implementare una funzione binaria f tramite un circuito logico, costituito dalle porte logiche AND, OR e NOT, che sono il corrispondente elettronico delle omonime operazioni. I bit 0 e 1 corrispondono a correnti di due voltaggi differenti (ad esempio, alto a +5 Volt e basso a 0 Volt). Ad esempio la porta AND ha due linee in ingresso ed una linea di uscita, che è alta se e solo se sono alte entrambi le linee di ingresso.

Ecco un circuito che calcola l'operazione di XOR tra due bit *x* e *y*



ove le porte a triangolo sono NOT, quelle a base piatta sono AND e quelle a base curva sono OR.

Le porte logiche sono realizzate attualmente tramite transistor e quindi i circuiti logici, anche con un numero molto alto di porte, possono essere costruiti in dimensioni piccolissime (*Very Large Scale Integrated*).

2.2 L'architettura di Von Neumann

I computer moderni sono organizzati internamente secondo una struttura che va sotto il nome di architettura di Von Neumann. Il computer è diviso in parti che svolgono funzioni diverse:

1. il processore
2. la memoria centrale
3. le memorie di massa
4. i dispositivi di I/O (Input/Output)

Le varie parti del computer comunicano tra di loro mediante delle linee di comunicazione chiamate bus, sul quale non forniremo ulteriori dettagli.

L'architettura di Von Neumann è stata utilizzata per la prima volta negli anni '50 e da allora l'evoluzione dell'informatica è stata impressionante. Ciò nonostante i computer convenzionali odierni seguono tuttora questa architettura, anche se ogni singola componente ha avuto uno sviluppo notevole: processori sempre più veloci, memorie sempre più capienti, ecc.

2.3 Il processore

Il processore è la parte del computer che esegue i programmi. Al suo interno comprende le seguenti componenti.

L'**unità di controllo** si occupa della gestione dell'esecuzione delle istruzioni. Le istruzioni sono codificate codificate mediante un codice numerico univoco, il quale condensa in un unico numero il tipo dell'istruzione e gli operandi su cui operare. Di solito questi possono costanti numeriche, registri o il contenuto di celle di memoria. Il programma da eseguire risiede nella memoria centrale.

L'unità di controllo quindi carica dalla memoria il codice dell'istruzione da eseguire (fase di FETCH), lo decodifica individuando l'operazione da eseguire ed i relativi operandi (fase di DECODE) ed infine predispone l'esecuzione dell'operazione (fase di EXECUTE). Poi passa all'istruzione successiva, compiendo le tre fasi, e così via.

I **registri** sono piccole memorie di un numero fissato di bit (comunemente 16, 32 o 64 bit). Un processore ha di solito un numero esiguo di registri, infatti tali memorie sono usate quasi esclusivamente per contenere i dati trattati in quel momento.

L'**unità logico-aritmetica** (ALU, Arithmetic Logic Unit) si occupa dell'esecuzione delle operazioni aritmetiche su numeri interi o reali (addizione, sottrazione, ecc.) e logiche su sequenze di bit (AND, OR, NOT, scorrimenti, ecc.)

L'insieme delle possibili istruzioni che un processore può eseguire si chiama linguaggio macchina. I principali tipi di istruzione sono

1. trasferimento interno: copiano dati da una parte di memoria o da un registro ad un'altra di memoria o su un altro registro;
2. logico-aritmetiche;
3. controllo: alterano la normale esecuzione sequenziale del programma, permettendo il "salto" ad un'istruzione arbitraria, anche in modo condizionato (il salto in tal caso avviene se è verificata una determinata condizione);
4. I/O: trasferiscono dati da o verso l'esterno.

Ogni processore ha un proprio linguaggio macchina, che può differire dal linguaggio macchina di un altro processore per la presenza di istruzioni diverse o in alcuni casi per un tipo di codifica diverso.

2.4 La memoria centrale

Innanzitutto sia la memoria centrale, sia le memorie di massa (trattate nella sezione successiva), sono dispositivi di memoria, ovvero servono a memorizzare dati e programmi. La capacità di memoria di un supporto si misura normalmente in byte (che è un insieme di 8 bit) o in suoi multipli.

Le unità di misura della memoria sono

1. il kilobyte (KB): 1 KB corrisponde a 1024 (2^{10}) byte,
2. il megabyte (MB): 1 MB corrisponde a 1024 KB, ovvero a 2^{20} byte,
3. il gigabyte (GB): 1 GB corrisponde a 1024 MB, ovvero a 2^{30} byte,
4. il terabyte (TB): 1 TB corrisponde a 1024 GB, ovvero a 2^{40} byte.

Le unità quindi anziché essere basate sulle potenze di 10 (come le unità del sistema metrico decimale), differiscono di un fattore pari a 2^{10} =1024, che però è vicino ad una potenza di 10.

La **memoria centrale** (o principale) è la memoria in cui sono memorizzati i programmi in esecuzione ed i relativi dati. La tipologia di memoria centrale più importante è la **RAM** (Random Access Memory) ed è una memoria a tecnologia elettronica.

Una memoria RAM è suddivisa in N celle (di solito N è una potenza di due o un suo multiplo, ad esempio $4x2^{20}$) di M bit l'una (normalmente M=8, 16, 32). Ogni cella è contrassegnata da un numero intero progressivo, detto indirizzo.

In una RAM si possono eseguire due operazioni elementari: la lettura e la scrittura di una singola cella.

L'operazione di **lettura** ha come operando un indirizzo *a* e restituisce come risultato il contenuto della cella *a*.

Ad esempio nella porzione di memoria

indirizzo	24	25	26	27	28	29	30	31	32
contenuto	10	24	90	101	244	101	89	90	0

l'operazione di lettura della cella di indirizzo 26 produce come risultato il numero 90.

L'operazione di **scrittura** ha come operandi un indirizzo *a* ed un numero *d* e modifica il contenuto della cella *a* memorizzando *d* al posto del contenuto che *a* aveva prima, il quale si perde

definitivamente.

Ad esempio nella porzione di memoria già vista in precedenza

indirizzo	24	25	26	27	28	29	30	31	32
contenuto	10	24	90	101	244	101	89	90	0

la scrittura del dato 44 sulla cella di indirizzo 26 produce come effetto

indirizzo	24	25	26	27	28	29	30	31	32
contenuto	10	24	**44**	101	244	101	89	90	0

La RAM è molto veloce, ossia i tempi di risposta per le operazioni di lettura e la scrittura sono molto bassi e paragonabili a quelli delle altre operazioni del processore.

D'altro canto per ottenere tale velocità, la RAM utilizza una tecnologia puramente elettronica che la rende relativamente costosa (se paragonata agli altri tipi di memoria), ma che soprattutto ha bisogno di un continuo passaggio di corrente. Infatti appena si toglie la corrente, la RAM perde completamente il proprio contenuto (**volatilità**). Questa caratteristica fa sì che la RAM possa essere usata solo come memoria temporanea, in cui memorizzare dati e programmi in corso, ma non come memoria a lungo termine. Del resto una memoria veloce come la RAM è indispensabile per avere un'esecuzione veloce dei programmi, infatti gli altri tipi di memoria non reggerebbero alla velocità di esecuzione del processore.

Un altro tipo di memoria centrale è la **ROM** (Read Only Memory), che consente solo operazioni di lettura, ma non di scrittura. Si tratta di una memoria pertanto di una memoria non volatile, ma che però non può essere usata che per memorizzare dati costanti e programmi fissi. Tale memoria viene riempita in fase di costruzione e non può essere modificata.

2.5 Le memorie di massa

Le memorie di massa superano il problema della volatilità della RAM e consentono di memorizzare in forma permanente i dati e in una quantità decisamente maggiore. Però hanno tempi di risposta molto più elevati (in termini di ordini di grandezza) e che, in molti casi, non sono nemmeno costanti.

Le tre tipologie di memoria di massa che attualmente sono utilizzate si differenziano in base alla tecnologia utilizzata per memorizzare i dati.

2.5.1 Dischi magnetici

I dischi magnetici usano superfici circolari metalliche i cui punti possono essere magnetizzati in due modi diversi. Mediante una testina è possibile controllare se un determinato punto è stato magnetizzato in un modo o nell'altro (rappresenta 0 o 1) e anche cambiare la magnetizzazione.

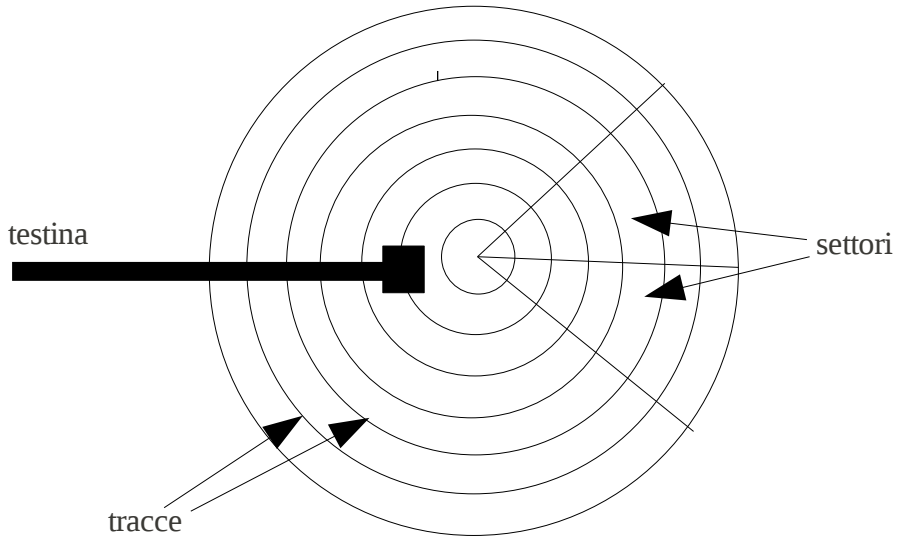
I dischi magnetici si suddividono in floppy disc e hard disc. I primi sono dischi estraibili di capienza limitata (intorno ai 720-1440 KB), composti da un'unica superficie. I secondi sono invece “pacchetti di dischi” alloggiati all'interno del computer che possono avere una grande capacità (anche diversi centinaia o migliaia di GB).

L'hard disc è di solito la memoria “principale” di massa, in cui risiedono il sistema operativo e i programmi “installati”, oltre che i dati più importanti.

Un disco magnetico è suddiviso in tracce (corone circolari) e settori (circolari) e ciascuna porzione che si ottiene intersecando una traccia con un settore, chiamata blocco, contiene un numero fisso di byte (di norma da qualche centinaio a qualche decina di migliaia), anche avendo una superficie

disponibile diversa (le tracce più vicine al centro sono ovviamente di lunghezza minore). Nei floppy disc e negli hard disc un blocco è individuato univocamente da tre coordinate: il numero di traccia, il numero di settore e il numero della superficie di riferimento. Quest'ultima coordinata, nei floppy distingue soltanto la superficie superiore o inferiore, negli hard disc comprende anche il numero del disco.

L'unità disco è dotata di una testina di lettura/scrittura per ogni superficie utile che si può spostare da una traccia all'altra mediante un movimento orizzontale. Nei floppy ci sono due testine in tutto, mentre negli hard disc ci sono due testine per ogni disco; in ogni caso tutte le testine si muovono insieme (si posizionano tutte sulla stessa traccia). L'insieme delle tracce alla stessa distanza dal centro si chiama cilindro.



Il disco ruota su stesso durante le operazioni di lettura/scrittura nei floppy disc, mentre i dischi ruotano sempre negli hard disc (quando è acceso il computer). Perciò per accedere al blocco posto nella traccia *t* e nel settore *s*, l'unità deve spostare la testina sulla traccia *t* e aspettare che il disco ruotando porti il settore *s* sotto alla testina. A questo punto può avvenire l'operazione di lettura o di scrittura dell'intero blocco.
Il tempo complessivo di svolgimento di tali operazioni dipende perciò anche dal tempo di posizionamento che si può scomporre in tempo di *seek* (che occorre per posizionare la testina sulla traccia *t*) e tempo di *latenza* (che si deve aspettare per arrivare al settore *s*). Ovviamente tali tempi dipendono dalla posizione della testina al momento in cui inizia l'operazione: si è fortunati i due tempi sono zero, mentre nel caso peggiore la testina deve percorrere l'intero raggio del disco e si deve aspettare un intero giro.
Infine, la lettura e la scrittura avvengono fisicamente con interazioni magnetiche tra la testina e la superficie del disco.

2.5.2 Dischi ottici

I dischi ottici sono simili ai floppy disc (unico disco rimovibile), con un disco ruotante e una testina che interagisce con esso. Si differenziano dai dischi magnetici perché si sfruttano le proprietà ottiche. I dischi ottici esistono in due formati principali: CD e DVD, i quali si differenziano nella capienza, i primi contengono circa 700 MB, mentre i secondi arrivano a circa 4 GB di dati.

Esistono unità di sola lettura (lettori CD/DVD) e unità di scrittura (masterizzatori CD/DVD); queste ultime sono comunque in grado anche di leggere.

Mentre la lettura di un CD o un DVD può essere svolta con l'ausilio di un raggio laser che colpendo la superficie del disco ne rivela la presenza di parti a potere riflettente diverso (“pit” o “land”), la

scrittura necessita da un lato di un supporto scrivibile, dall'altro di un raggio laser più potente.

Si distinguono infatti

- CD-ROM e DVD-ROM, dischi a sola lettura, il cui contenuto è memorizzato, durante la produzione del disco, in modo non modificabile;
- CD-R e DVD-R, dischi che essere letti un numero imprecisato di volte, ma possono essere scritti una sola volta (tramite un masterizzatore); non possono però essere cancellati o riscritti altre volte;
- CD-RW e DVD-RW, dischi che possono essere letti, scritti e cancellati più volte.

Una differenzia ulteriore con i dischi magnetici è che i dischi ottici, oltre ad essere più lenti, sono anche maggiormente soggetti ad errori di lettura e di scrittura.

2.5.3 Memorie flash

Le memorie flash sono essenzialmente delle memorie simili a delle ROM e che però possono essere cancellate e scritte mediante l'applicazione di opportune tensioni di corrente. La forma comunemente usata nei PC è quella delle cosiddette pen-drive (chiavette) che sono unità estraibili di capienza intorno a qualche GB, ma sono anche utilizzate nelle memorie delle macchine fotografiche e dei cellulari.

Le memorie flash non hanno parti meccaniche in movimento e quindi hanno tempi di risposta molto bassi, anche se non sono ancora paragonabili alla velocità di risposta della RAM.

2.6 I dispositivi di I/O

I dispositivi di Input/Output (I/O) servono a far comunicare il computer con il mondo esterno (utente, altri computer, ecc.) e a farlo interagire con gli oggetti circostanti. In teoria l'elenco dei dispositivi di I/O è una lista aperta, in cui in ogni periodo vi sono nuove invenzioni (e anche dispositivi che diventano obsoleti, come il perforatore di schede perforate).

Innanzitutto si distingue nettamente la classe dei dispositivi di solo input, da quella di solo output da quella ancora di input e output.

Dispositivi di solo input sono il mouse, la tastiera, il microfono, la videocamera.

Tra i dispositivi di solo output troviamo la scheda video congiunta con il monitor (anche se quest'ultimo con la funzione di *touch screen* diventa di input e output), la stampante, il plotter, le casse con la scheda audio.

Tra i dispositivi di input e output troviamo sicuramente i dispositivi di comunicazione via rete, come il modem e la scheda di rete.

Una caratteristica comune a tutti i dispositivi di I/O è che i dati in ingresso o in uscita dal computer devono essere in formato digitale e se all'origine non sono digitali (ad esempio le immagini) devono essere trasformati e codificati in tale formato.

Ad esempio il mouse invia al computer le coordinate, rispetto ad un piano cartesiano ideale (il tavolo), le coordinate x e y del mouse o gli spostamenti Δx e Δy rispetto alla posizione corrente.

Una videocamera invece invia al computer una matrice rappresentante per ogni unità di immagine (pixel) la codifica numerica del colore corrispondente, mediante una rappresentazione simile a quanto si vedrà nel prossimo capitolo.

2.7 Esercizi

1. Dimostrare il teorema di Shannon (suggerimento, usare il principio di induzione)
2. Far vedere che ogni porta logica si può scrivere usando soltanto la porta NAND, che è definita da x NAND y=NOT(x AND y). Enunciare il teorema di Shannon per il NAND.
3. Idem per la porta NOR, definita da x NOR y=NOT(x OR y).

4. Scrivere un circuito che dati tre bit x,y,z calcola il valore più frequente, ovvero 0 se almeno due bit sono 0, 1 altrimenti (in tal caso almeno due bit sono 1).

3 Rappresentazione dell'informazione

In questo capitolo vedremo come il computer è in grado di rappresentare i dati che elabora, ad iniziare dai dati numerici. Come si è già detto, i dati in un computer sono puramente digitali, ovvero sono rappresentati mediante sequenze di bit.

3.1 Rappresentazione dei numeri naturali

I numeri naturali sono rappresentati in base 2 in modo usuale con un numero N fissato di bit.

Di solito si usa N=16, 32 o 64. I numeri rappresentabili perciò vanno da 0 a 2^N-1. Ad esempio con N=8 il più grande numero rappresentabile è 255.

Per determinare le cifre binarie si calcolano i resti delle divisioni successive per 2, si dispongono nell'ordine inverso e si aggiungono degli zeri a sinistra fino ad avere N cifre binarie.

Ad esempio la rappresentazione di x=67 è 01000011. Infatti

67:2=33 con il resto di 1

33:2=16 con il resto di 1

16:2=8 con il resto di 0

8:2=4 con il resto di 0

4:2=2 con il resto di 0

2:2=1 con il resto di 0

1:2=0 con il resto di 1

Per convertire un numero in notazione decimale è sufficiente calcolare la somma delle potenze di 2 corrispondenti alle cifre pari a 1. Ad esempio il numero 01101101 corrisponde a $2^6+2^5+2^3+2^2+2^0=109$.

3.2 Rappresentazione dei numeri interi

I numeri interi sono comunemente rappresentati con la notazione in **complemento a due**. Scelto il numero di bit N (anche in questo caso, di solito N=16, 32 o 64), è possibile rappresentare tutti i numeri interi nell'intervallo $I_N=[-2^{N-1}, 2^{N-1}-1]$. Ad esempio con N=8 l'intervallo è [-128,127]. Tale asimmetria è dovuta al fatto che 0 è considerato positivo.

I numeri maggiori o uguali a zero sono rappresentati normalmente con esattamente N cifre binarie (aggiungendo degli zeri a sinistra per completare le eventuali cifre mancanti) ed avranno perciò il bit più a sinistra 0. Ad esempio con N=8, il numero x=78 corrisponde a 01001110.

I numeri negativi sono rappresentati invece aggiungendovi 2^N. Ad esempio con N=8, x=-78 diventa -78+256=178=10110010. I numeri negativi si distinguono quindi dagli altri numeri perché hanno 1 come bit più sinistra.

Per cambiare di segno si può eseguire la seguente procedura

1) si scambiano gli zeri e gli uno
2) si aggiunge uno
3) si trascura l'eventuale riporto sull'N-esimo bit.

Infatti partendo da 78=01001110, invertendo i bit si ottiene 10110001, poi aggiungendo 1 si ottiene

10110010.

L'operazione di addizione si svolge in modo analogo all'addizione in base 10, solo che essendo solo due le cifre possibili, tutto è molto più semplice.

Ad esempio 78+23=101

```
 01001110 +
 00010111 =
----------
 01100101
```

Si noti che 1+1 fa 0 con riporto di 1.

La sottrazione *x-y* è calcolata come *x+(-y)*, dato il complemento a due permette di addizionare con lo stesso metodo anche numeri aventi segni qualsiasi.

Ad esempio 49-24=49+(-24)=25

```
 00110001 +
 11101000 =
----------
 00011001
```

Il riporto finale di 1 può essere trascurato.

Si noti che la somma di due numeri interi non è sempre definita, poiché il risultato potrebbe uscire dall'intervallo di definizione. Ad esempio con N=8, 50+80 non si può fare, infatti svolgendo la somma si otterrebbe come risultato -126. Tale situazione si chiama overflow e non è superabile se non utilizzando un maggior numero di bit. Purtroppo non sempre il computer segnala questo tipo di errori.

La moltiplicazione tra due numeri interi si fa prima calcolando il prodotto tra i due valori assoluti e poi aggiustando il segno del risultato. Per moltiplicare due numeri positivi sono necessari solo le operazioni di addizione e di scorrimento verso sinistra (che corrisponde alla moltiplicazione per 2).

Ad esempio 9x12=108

```
      1001 x
      1100 =
----------
      0000 +
     00000 +
    100100 +
   1001000 =
----------
   1101100
```

La divisione invece si ottiene mediante operazioni di sottrazione e scorrimento verso destra e

sinistra.

In generale anche la moltiplicazione può generare overflow.

Una considerazione importante riguarda il numero di bit. Il computer predilige trattare numeri interi con un numero fissato di bit, ciò rende molto più semplici le operazioni, soprattutto in hardware. D'altro canto è proprio questo il motivo dell'overflow: se si usano interi con numero illimitato di bit (qualche linguaggio offre, almeno teoricamente, tale possibilità) non può accadere che la somma di due numeri interi non è rappresentabile. Il rovescio della medaglia è però una maggiore lentezza di esecuzione che dipende dal numero di bit coinvolti.

3.3 Rappresentazione dei numeri reali

I numeri reali sono problematici per un computer, in quanto la maggior parte di essi sono hanno un numero infinito di cifre, qualunque sia la base B. Infatti sia tutti i numeri irrazionali, sia i numeri razionali il cui denominatore non è una potenza di B hanno un'espansione B-aria illimitata, anche se i secondi sono in realtà periodici, per cui potrebbe essere rappresentati mediante qualche artificio con un numero finito di cifre.

La rappresentazione più utilizzata nell'informatica è quella in virgola mobile. Fissata una base *B*, ogni numero reale *x*, diverso da 0, si può scrivere come

$$x = s \cdot m \cdot 2^e$$

ove *s* è il segno (può essere +1 o -1), *m* è la mantissa, ovvero un numero reale nell'intervallo [1/*B*,1[ed *e* è un numero intero chiamato esponente.

La richiesta che *m* sia maggiore o uguale a 1/*B* fa sì che la rappresentazione di *x* sia unica e che *m* si possa scrivere in base *B* come

$$m = 0,abcd...$$

in cui *a*,*b*,*c*,*d*,... sono cifre della notazione B-aria ed, in particolare, *a* è diversa da zero.

Ad esempio con B=10 il numero *x*=34.789 si scrive come

$x = +1 \cdot 0.34789 \cdot 10^2$

mentre il numero *y*=347.89 ha la stessa mantissa, ma esponente diverso

$y = +1 \cdot 0.34789 \cdot 10^3$

infine il numero *z*=-0.0034789 si scrive come

$z = -1 \cdot 0.34789 \cdot 10^{-2}$

Il computer per rappresentare i numeri reali usa la base B=2 e si serve di *h* bit per rappresentare la mantissa e di *k* bit per l'esponente. L'esponente, essendo un numero intero, è rappresentato in complemento a due (o in forme equivalenti) con *k* bit.

La mantissa, che sarà un numero binario del tipo

0,1.....

con *h* cifre dopo la virgola è rappresentata tralasciando la prima cifra dopo la virgola (che è sempre 1) conservando le altre *h-1* cifre. Il segno di *x* è memorizzato con un ulteriore bit, ad esempio 0 se *x* è positivo, 1 se *x* è negativo.

Il numero zero viene memorizzato in un modo particolare, che in queste dispense non è descritto.

Ad esempio il numero 1.5 viene scomposto come

$+1 \cdot 0.75 \cdot 2^1$

che in binario con h=8 e k=4 avrà la forma (s=0, m=100000000, e=0001).

Infatti *m* sarebbe 0.11000000, ma la prima cifra dopo la virgola viene eliminata. Il bit di segno s è 0, dato che il numero è positivo.

Indicheremo con R_{hk} l'insieme dei numeri reali che si possono rappresentare esattamente nel modo precedente, ovvero che sono pari a zero, o che hanno una mantissa di *h* cifre e un esponente con *k* bit.

E' facile vedere che tali numeri sono tutti razionali ed in particolare sono esprimibili come frazioni il cui denominatore è una potenza di 2. Questi numeri formano un insieme “estremamente piccolo” rispetto a $\mathbb{R}$, anche perché R_{hk} è un insieme finito.

In realtà, indicando con $U=sup\ R_{hk}$ e $L=inf\ (R_{hk} \cap \mathbb{R}^+)$, preso un qualsiasi numero reale *x* esistono quattro possibilità:

1. se $|x|>U$, allora *x* non è rappresentabile in alcun modo;
2. se $L \leq |x| \leq U$ e $x \in R_{hk}$, allora x è rappresentabile esattamente;
3. se $L \leq |x| \leq U$ e $x \notin R_{hk}$, esiste un elemento indicato con $fl(x) \in R_{hk}$ che è più vicino possibile a *x* (a meno che *x* si trovi esattamente a metà strada tra due elementi consecutivi di R_{hk}, in tal caso per *fl(x)* si sceglie uno dei due elementi). Il computer quindi userà *fl(x)* al posto di *x*, commettendo un errore pari $|x-fl(x)|$, che in generale è molto piccolo (dipende da *h* e *k*).
4. se $|x| \leq L$, allora *x* è rappresentato come 0. Se *x* non è 0, tale approssimazione si chiama underflow.

Le operazioni aritmetiche in virgola mobile sono relativamente semplici da implementare.

L'addizione e la sottrazione per poter essere svolte, necessitano una fase simile all'incolonnamento: la virgola di uno dei due termini viene spostata in modo che l'esponente coincida con quello dell'altro termine. Solo a questo punto le due mantisse possono essere addizionate o sottratte.

Il risultato però potrebbe avere troppe cifre (cioè più di *h*) e quindi è necessaria una fase di arrotondamento per eliminare le cifre in più. Tale operazione perciò approssima il risultato, che non è quindi calcolato esattamente.

Ad esempio, con h=5 e k=2, la somma di x=0.11011 2^5 e y=0.10001 2^6 verrebbe calcolata come (0.011011 +0.100010) 2^6, che avrebbe come risultato 0.111101 2^6. La sesta cifra (il 1) deve essere eliminata, producendo come risultato (scorretto) 0.11110 2^6.

Come conseguenza di tale arrotondamento, l'addizione tra numeri in virgola mobile non rispetta nemmeno la proprietà associativa.

La moltiplicazione e la divisione sono invece svolte moltiplicando o dividendo le mantisse e addizionando o sottraendo i relativi esponenti. Potrebbe essere necessaria una fase di normalizzazione del risultato, in cui le eventuali cifre in più sono eliminate e arrotondate.

E' molto importante capire che l'uso dell'aritmetica in virgola mobile deve far fronte a due fonti di imprecisione. La prima è dovuta al fatto che la maggior parte degli infiniti numeri reali può essere rappresentata solo in forma approssimata. La seconda nasce invece dagli arrotondamenti che si rendono indispensabili nel calcolo delle operazioni per eliminare le cifre superflue. In caso di più operazioni eseguite, gli errori di arrotondamento per lo più si accumulano, rendendo il risultato sempre meno preciso.

Per ovviare a questo problema, si utilizzano numeri in virgola mobile con alta precisione. Tra le possibilità offerte dai computer moderni troviamo

- **numeri reali a precisione singola**: si usano 32 bit, di cui k=8 bit per l'esponente e h=24 bit

tra mantissa e segno;

- **numeri reali a precisione doppia**: 64 bit di cui k=11 e h=53 bit;
- **numeri reali a precisione quadrupla**: 128 bit di cui k=15 e h=113 bit.

E' evidente che maggiore è la precisione, più basso è l'errore di approssimazione e quindi più accurati sono i calcoli.

3.4 Rappresentazione di dati testuali

I dati di tipo testuale, come documenti, testi, singole frasi o più semplicemente sequenze di lettere e simboli possono essere facilmente rappresentati mediante sequenze di numeri. E' infatti sufficiente definire l'**alfabeto** di base del testo, cioè l'insieme dei caratteri di cui il testo può essere composto, e attribuire un codice numerico ad ogni elemento dell'alfabeto. Un insieme standard di caratteri è l'insieme ASCII, che comprende lettere, cifre e altri simboli, quali punteggiatura, operazioni matematiche e parentesi. Una parte significativa di tale insieme è quella formata dai caratteri nell'intervallo da 32 (spazio) a 126 (tilde), elencata nella tabella seguente.

	0	1	2	3	4	5	6	7	8	9
30				!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			

Con questa rappresentazione la parola “dieci” corrisponde al vettore (101,105,102,109,105), mentre la parola “Dieci” corrisponde a (68,105,102,109,105). Infine la sequenza “10” corrisponde al vettore (49,48).

3.5 Rappresentazione di immagini

Come esempio di rappresentazione di dati non numerici, prendiamo il caso delle immagini.

Le immagini in natura sono oggetti “analogici” e per essere trattati da un computer devono essere discretizzate. Una prima fase di discretizzazione è usare una matrice di $h \times v$ punti, detti **pixel** (abbreviazione di *picture element*) in cui h sono i punti in orizzontale e v in verticale. Gli infiniti punti dell'immagine che non cadono nella griglia non sono rappresentati. Comunque più la risoluzione, ovvero il rapporto tra numero di pixel per unità di superficie, è maggiore e più alta è la precisione della rappresentazione, dato che punti vicini potrebbero avere caratteristiche uguali o simili.

Per ogni pixel possono essere memorizzate varie informazioni. Tra le diverse possibilità si può scegliere di rappresentare per ogni punto

- un bit 0 o 1, se si vuole un'immagine monocromatica, ovvero con due soli colori: l'immagine

e lo sfondo (ad esempio 0 se il punto fa parte dello sfondo, 1 se fa parte dell'immagine);

- un numero intero che rappresenta il livello di grigio, ad esempio da 0 (nero) a 255 (bianco);
- una terna di numeri interi, che rappresentano i livelli dei tre colori fondamentali (rosso, verde e blu), ognuno, ad esempio, tra 0 e 255.

Quindi il colore è ottenuto con un secondo livello di discretizzazione in cui grandezze fisiche come la luminosità e la frequenza sono ridotte a numeri interi.

Le immagini occupano molto spazio e normalmente, sia in fase di memorizzazione su disco o di trasporto in rete, sono compresse, cioè si usano particolari tecniche per ridurre lo spazio occupato.

4 Sistemi operativi e software di base

4.1 Tipologie di software

Come abbiamo visto, l'insieme dei programmi che si usano in un computer si chiama **software**. Si distinguono due categorie di software: il software di base ed il software applicativo.

Il **software di base** è un insieme di programmi il cui scopo è quello di gestire le varie componenti del computer in modo da renderle più semplici da utilizzare (sia per l'utente, sia per gli altri programmi). La parte principale del software di base è il sistema operativo.

Il **software applicativo** è invece costituito da tutti quei programmi che sono utilizzati per scopi specifici: word processor, fogli elettronici, navigazione su internet, ecc.

4.2 Il sistema operativo

Il sistema operativo è un insieme di programmi che si occupano della gestione dell'intero computer e pertanto può essere scomposto in vari moduli, ognuno dei quali si occupa di una funzione diversa.

Il sistema operativo viene in parte caricato in memoria e mandato in esecuzione all'accensione del computer (fase di *bootstrap*), mentre altre parti sono eseguite nel momento in cui servono.

Un computer senza sistema operativo sarebbe praticamente inutilizzabile, dato che l'utente e i programmi sarebbero costretti ad interagire direttamente con le risorse hardware del computer. E' chiaro che lo svantaggio maggiore sarebbe per l'utente, ma anche i programmi sarebbero molto più difficili da scrivere e inoltre dipenderebbero in maniera preponderante dalla configurazione su cui devono girare.

Un sistema operativo è suddiviso in varie componenti, tra le quali vi sono

- gestore dei processi
- gestore della memoria centrale
- gestore della memoria secondaria
- gestore dei dispositivi
- interfaccia utente

4.2.1 Gestione dei processi

Un computer convenzionale ha un solo processore, con una sola unità di esecuzione. E' vero però che alcuni processori moderni (*dual-core* e *quad-core*) hanno due o quattro unità di esecuzione. Comunque è normale che si mandino in esecuzione molti programmi contemporaneamente: ad esempio navigare su internet, ascoltare la musica e scrivere un documento, mentre è in funzione l'anti-virus, oltre ad altri programmi.

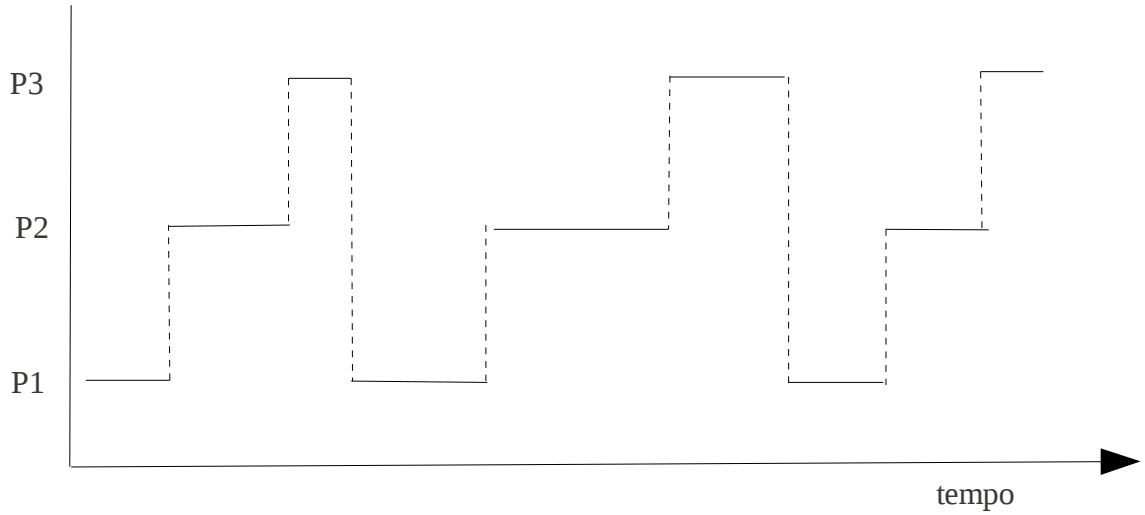
Il gestore dei processi rende possibile l'esecuzione di più programmi (processi) nello stesso intervallo di tempo.

La tecnica comunemente utilizzata è quella del time-slicing: il processore si alterna nell'esecuzione dei vari processi, eseguendo ognuno di essi “un po' per volta”, ovvero dedicando a ciascuno una parte del proprio tempo a rotazione.

Supponendo che i processi in corso siano P1, P2 e P3, il processore inizierà ad eseguire P1 per un certo periodo di tempo. Poi salverà in memoria le informazioni sullo stato di P1 e inizierà ad eseguire P2, sempre per un certo lasso di tempo. Poi, dopo aver registrato le informazioni di P2, passerà a P3, ancora per un certo periodo di tempo. Scaduto quest'ultimo intervallo temporale, farà

ripartire P1 recuperando dalla memoria le informazioni utili a tal scopo, e così via.

Il seguente grafico illustra una possibile esecuzione in time-slicing



Il passaggio da un processo all'altro si chiama *cambio di contesto* e comporta l'esecuzione di alcune istruzioni di salvataggio e di recupero dello stato dei due processi in gioco (quello che era in esecuzione e quello che passerà ad essere eseguito).

L'esecuzione può avvenire per semplice *round-robin*, dedicando la stessa quantità di tempo ad ogni processo. Oppure si possono avere processi a maggiore o minore priorità: i primi sono eseguiti interamente prima degli altri, oppure hanno una quantità di tempo maggiore a disposizione.

E' importante sottolineare che l'esecuzione concorrente dei processi avviene in maniera trasparente per chi scrive il programma: ci pensa il sistema operativo a sospendere e a riprendere l'esecuzione di ogni processo, salvando eventuali le informazioni necessarie per una ripresa corretta. Anche chi utilizza il programma può non rendersi conto dell'esecuzione in corso di altri programmi, anche se noterà una minore velocità di esecuzione.

4.2.2 Gestione della memoria centrale

La memoria centrale è una risorsa importante che deve essere condivisa, in qualche forma, da tutti i processi in esecuzione. Il fatto che il processore esegua più processi nello stesso intervallo di tempo fa sì che la memoria contenga (almeno in parte) sia i programmi in esecuzione, sia i rispettivi dati.

Un programma in generale può richiedere molta memoria, sia per contenere le proprie istruzioni, sia per i propri dati, e quindi con molti programmi il rischio di saturare la memoria è reale.

E' quindi necessario gestire la memoria in maniera ottimale. Da un lato è possibile sfruttare il fatto che un programma non ha bisogno in ogni momento di tutta la memoria che complessivamente userà in tutto il suo corso: è infatti probabile che l'esecuzione si concentri maggiormente su alcune parti del programma e su alcuni dati. Allora è sufficiente tenere in memoria centrale solo la parte di codice e di dati maggiormente usata. Il sistema operativo è in grado, usando alcune tecniche statistiche, di individuare la porzione giusta e lasciare il resto su disco (o altra unità di memoria di grandi dimensioni).

D'altro canto è possibile dotare il processore di meccanismi di traduzione automatica di indirizzi di memoria. Infatti una gestione efficiente necessita che la parte di memoria assegnata ad un processo non sia necessariamente né presente tutta nella memoria centrale (come si è appena visto), né sia memorizzata fisicamente su posizioni consecutive. Allora è possibile utilizzare una doppia serie di

indirizzi: gli **indirizzi fisici** (che sono gli indirizzi “veri” delle locazioni di memoria) e gli **indirizzi logici** (che sono quelli citati nelle istruzioni del programma).

Il programma è infatti scritto senza che sia possibile sapere quanta parte della memoria utilizzata si trova realmente in memoria centrale ed esattamente a quali locazioni corrisponda.

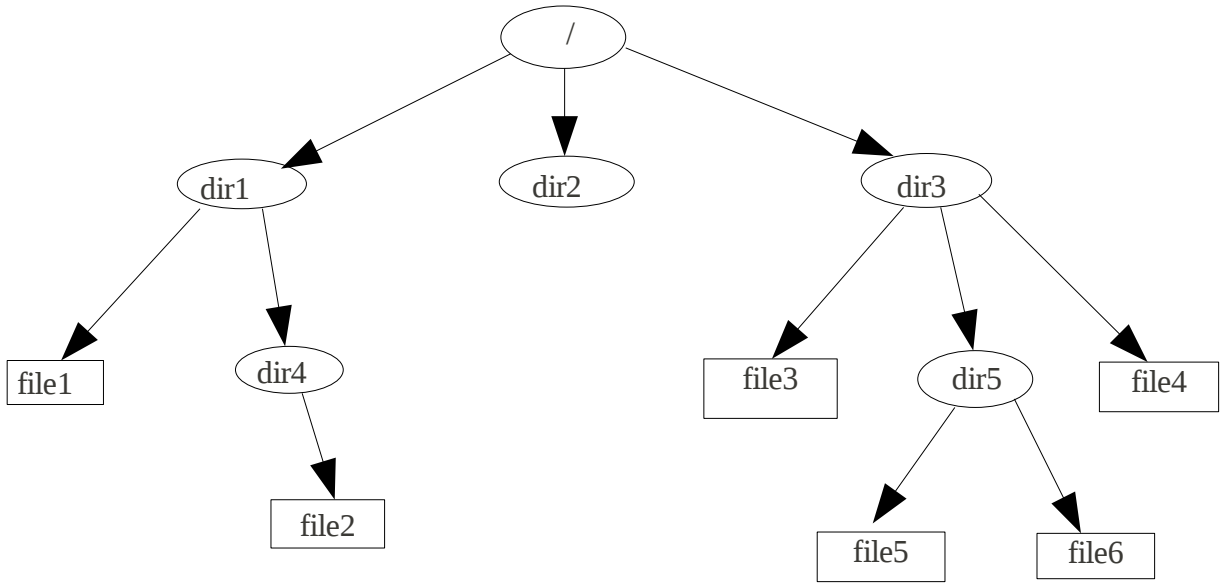
E' quindi compito del sistema operativo, con l'ausilio delle tecniche di **memoria virtuale,** di “dirottare” ogni accesso alla memoria svolto dal programma verso la locazione esatta, traducendo l'indirizzo logico in indirizzo fisico.

4.2.3 Gestione della memoria secondaria

La memoria secondaria (dischi di vario tipo, memorie flash, ecc.) è uno strumento fondamentale nell'informatica. Il sistema operativo rende estremamente semplice la gestione dei dati nelle memorie secondarie tramite l'organizzazione in file e directory.

I file sono contenitori di informazioni, solitamente omogenee e organizzate mediante una qualche struttura, e sono contraddistinti da un nome e da una posizione all'interno della gerarchia delle directory.

Ogni directory puo' contenere file e directory ed è identificata da un nome. Le directory sono organizzate in maniera gerarchica, formando la classica struttura ad albero.



Il vincolo è che in ogni directory non possono esistere due elementi (file o directory) con lo stesso nome, ma in directory diverse i nomi si possono ripetere, ad esempio file3 e file4 devono avere nomi diversi, ma file3 e file5 si possono chiamare allo stesso modo.
La directory denotata con “/” è la cosiddetta directory principale, che corrisponde al livello più alto della gerarchia. Al livello più basso si possono trovare solo file o directory vuote, come dir2.

I dati sulle memorie secondarie sono in realtà gestiti tenendo conto delle “coordinate” necessarie a raggiungere sul disco (o sul supporto utilizzato) le informazioni desiderate. Ad esempio un file memorizzato su un hard disc corrisponderà a vari blocchi, di ognuno dei quali è necessario sapere la traccia, il settore e la superficie in cui si trova.
Il sistema operativo mantiene quindi una tabella di corrispondenza che per ogni file e directory contiene le coordinate dei blocchi associati.
Ogni operazione di accesso ai file sarà pertanto convertito in operazioni di accesso ai blocchi corrispondenti.

4.2.4 Gestione dei dispositivi

Il sistema operativo consente di gestire i dispositivi di I/O mediante funzioni “generiche” che permettono un utilizzo “uniforme” del dispositivo, senza che vi siano apparenti differenze tra modelli e marche diverse. La componente fondamentale di tale meccanismo è il cosiddetto **driver** del dispositivo, che è un insieme di funzioni, rese normalmente disponibili dal costruttore del dispositivo, che si inseriscono nel sistema operativo e che devono essere usate in ogni accesso al dispositivo stesso.

La fase di installazione di un nuovo dispositivo prevede infatti l'inserimento del driver nel sistema operativo.

L'uso dei driver rende i programmi indipendenti dalle marche e dai modelli dei singoli dispositivi. Ad esempio un programma che vuole accedere alla stampante per stampare un documento non ha bisogno di conoscere i dettagli di comunicazione con la stampante (che dipenderanno dalla marca e dal modello della stessa). Il programma invia semplicemente i dati usando le funzioni del “driver” della stampante, le quali si occupano di dialogare fisicamente con il dispositivo stesso.

Questa caratteristica fa sì che il programma non debba essere acquistato o modificato in base alle caratteristiche della stampante posseduta.

4.2.5 Interfaccia utente

L'interfaccia utente è una componente essenziale di un sistema operativo e si occupa dell'interazione con l'utente: l'utente invia comandi ed il sistema operativo li esegue.

Esistono due tipi di interfaccia utente. Le interfacce testuali prevedono che l'utente digiti i comandi tramite un linguaggio di comandi, detto linguaggio di shell.

Le interfacce grafiche invece consentono di utilizzare le finestre ed il mouse, in modo da semplificare al massimo l'interazione con l'utente. Ad esempio in Linux, per cancellare un file con l'interfaccia testuale si deve digitare il comando `rm` seguito dal nome del file, mentre nell'interfaccia grafica è sufficiente selezionare l'icona del file e trascinarla con il mouse sul cestino.

Le operazioni comunemente svolte tramite l'interfaccia utente sono

- gestione di file e directory (copiare, spostare, cancellare, rinominare, ecc.)
- esecuzione e terminazione di programmi
- amministrazione del sistema (installazione e disinstallazione di programmi e dispositivi, aggiornamento del sistema operativo, configurazione del sistema, gestione degli utenti e dei permessi, ecc.)

L'interfaccia utente richiede anche l'autenticazione dell'utente, ad esempio mediante login e password.

4.3 Altre componenti del software di base

Tra le altre componenti del software di base, una particolare attenzione viene rivolta agli strumenti per la programmazione. Si tratta di programmi che consentono di creare, modificare ed eventualmente eseguire programmi. I più importanti strumenti per la programmazione sono

- **editor**: si usano per scrivere i programmi e registrarli come file testo;
- **compilatori** ed **interpreti**: traducono ed eseguono i programmi, saranno spiegati nel prossimo capitolo;
- **debugger**: servono per cercare errori di programmazione, consentendo ad esempio l'esecuzione rallentata di un programma;
- **ambienti integrati**: raccolgono insieme editor, compilatore o interprete e debugger, permettendo di lavorare in un unico ambiente.

4.4 I principali tipi di software applicativo per la matematica

Un elenco completo di possibili tipologie di software applicativo non è possibile: esistono un numero sterminato di applicazioni e nuove tipologie vengono introdotte continuamente.

E' però possibile brevemente illustrare alcune tipologie di software applicativo utili per un matematico.

4.4.1 Software di scrittura di testi scientifici

Mentre i comuni documenti (lettere, ecc.) possono essere scritti usando normali sistemi di word processing, come Writer di Open Office o Word di Microsoft ™, i testi scientifici devono essere scritti mediante programmi particolari. Infatti un testo scientifico contiene parti che non rientrano nei normali documenti: formule matematiche, schemi, diagrammi e disegni.

Le formule matematiche in particolare usano sia simboli non presenti sulla tastiera e raramente usati altrove (operatori, lettere greche, ecc.), sia una disposizione dei simboli nel testo che non è semplicemente sequenziale come un testo normale. Si pensi ad una formula come

$$\int_0^{\infty} \frac{e^x}{1+e^x}\,dx$$

in cui 0 e infinito sono posti in una certa posizione rispetto al simbolo di integrale, mentre numeratore e denominatore sono scritti, rispettivamente, sopra e sotto la linea di frazione.

Le tecniche usate per trattare le formule sono essenzialmente due.

La prima è quella di potenziare i word processor in modo da permettere l'inserimento e la modifica di formule e diagrammi all'interno del testo. Un esempio di tale soluzione è Equation Editor per Microsoft Word.

La seconda tecnica è quella adottata dal sistema TeX/LaTeX, in cui si usa un particolare linguaggio per rappresentare testi, formule e diagrammi. Il documento può essere visto sullo schermo o stampato solo dopo che è stato tradotto in formato grafico da un particolare programma traduttore. Da un lato la resa grafica è decisamente migliore e, una volta presa confidenza con il linguaggio, la scrittura di documenti scientifici è agevole, tant'è che il formato TeX è usato comunemente dalla comunità scientifica.

D'altro canto, però, mentre si digita il documento non si ha sempre idea di come apparirà sullo schermo o sulla carta e quando viene visualizzato in forma grafica non si ha la possibilità di modificarlo. In pratica per il TeX non vale la caratteristica dei moderni word processor sintetizzata dall'acronimo WYSIWYG (What You See Is What You Get), ovvero *quello che* (*digiti e*) *vedi sullo schermo è quello che ottieni* (*sulla carta*).

4.4.2 Software per il calcolo “numerico”

Altri programmi per il calcolo scientifico sono utilizzati per svolgere operazioni numeriche, anche molto complesse, quali calcolo di integrali, risoluzioni di equazioni differenziali, sistemi lineari, ed equazioni non lineari, calcolo vettoriale e matriciale. Si possono inoltre utilizzare anche per ottenere rappresentazioni grafiche, utili nello studio qualitativo di funzioni.

Tra questi segnaliamo Matlab (con i “cloni” gratuiti Octave e Scilab) e R (orientato alle applicazioni statistiche).

4.4.3 Software per il calcolo simbolico

Un particolare tipo di programmi utilizzati in ambito scientifico sono quelli destinati al calcolo simbolico. Esempi di tali programmi sono Mathematica, Maple, Derive, Macsyma.

Si tratta di programmi che sono in grado di manipolare espressioni, polinomi, funzioni e oggetti algebrici in modo simbolico, oltre che fornire le operazioni di tipo numerico e grafico, seppure in modo meno efficiente.

Ad esempio tali programmi sono in grado di calcolare la derivata simbolica, ad esempio, di x^2+3x ottenendo come risultato $2x+3$, o di semplificare $x+y+2x-5y$ in $3x-4y$.

5 Algoritmi, programmi e linguaggi

5.1 Introduzione

Nella programmazione è centrale il concetto di **programma**, che è un insieme (dotato di una certa struttura) di istruzioni, appartenenti ad un certo linguaggio di programmazione, che consente ad un computer di svolgere un dato compito.

La generalizzazione di programma conduce ad un concetto non legato all'informatica: il concetto di algoritmo.

Un **algoritmo** è un procedimento finito mediante il quale un ente (uomo, computer, robot, ecc.) è in grado di svolgere un compito complesso.

Esempi di algoritmi non informatici sono

- i metodi per sommare, sottrarre, moltiplicare e dividere numeri con carta e penna, che si imparano alle scuole elementari
- i metodi di calcolo di limiti, derivate, integrali, serie (alcuni metodi non sono completi, nel senso che non sempre danno risposta);
- le istruzioni di utilizzo di elettrodomestici e strumenti vari;
- le istruzioni di montaggio di vari oggetti;
- i protocolli medici per trattare i pazienti con determinate patologie e condizioni cliniche
- le ricette di cucina.

E' comunque evidente che lo studio degli algoritmi trova la sua collocazione naturale all'interno dell'informatica, anche perché un computer, a differenza dell'uomo, non può imparare a risolvere i compiti da solo, ma può solo eseguire programmi (e quindi algoritmi) creati dall'uomo.

5.2 Problemi ed Algoritmi

I problemi che si possono risolvere con un computer sono compiti di elaborazione dati e tecnicamente sono chiamati problemi computazionali.

Un **problema computazionale** può essere descritto da una terna (I,O,f), in cui

- I è l'insieme dei possibili dati di ingresso (input)
- O è l'insieme dei possibili risultati (output)
- $f:I \rightarrow O$ è una funzione che associa ad ogni input il corrispondente output.

Si noti che un elemento di I corrisponde a tutti gli input necessari per specificare completamente il problema (istanza del problema) e allo stesso modo un elemento di O consiste in tutti i risultati di una singola istanza.

Ad esempio il problema di calcolare il massimo comun divisore di due numeri naturali può essere descritto da

- $I=\mathbb{N}^2$
- $O=\mathbb{N}$
- $f(m,n)$=massimo comun divisore di m e n.

Infatti per specificare un'istanza del problema sono necessari i due numeri naturali, i quali formano una coppia; mentre il risultato può essere un qualsiasi numero naturale.

Altri esempi di problemi computazionali sono

1) dato un numero intero *n*, stabilire se *n* è un numero primo;
2) data una matrice A di dimensione *n*, calcolare il determinante;
3) dato un numero intero *n*, determinare il più piccolo divisore di *n* (ad esclusione di 1);
4) data una funzione *f* lineare di *n* variabili reali $x_1,...,x_n$ ed un insieme finito *D* di disequazioni lineari su $x_1,...,x_n$, trovare il vettore $(a_1,..,a_n)\in\mathbb{R}^n$ tale che le disequazioni *D* sono soddisfatte con $x_i=a_i$, per $i=1,...,n$ e $f(a_1,....,a_n)$ è massimo.

Nel problema 1) l'insieme O è {sì,no} (oppure {0,1}) e *f*(*n*) vale "sì" (o 1) se n è primo, "no" (oppure 0) altrimenti. I problemi in cui O={sì,no} si chiamano **problemi decisionali**.

Nel problema 3), che è strettamente collegato al 1), è un **problema di ricerca**. Si noti che se il risultato è *n*, il numero è primo.

Il problema 4) è un classico **problema di ottimizzazione** (problema della programmazione lineare).

Quindi un problema computazionale consiste nel calcolare una funzione *f* ed un algoritmo è un metodo per calcolare *f(x)* a partire da un qualsiasi valore di *x*.

La differenza tra problema e algoritmo è che nel problema si definisce soltanto la funzione *f*, senza specificare come calcolare *f(x)*.

Una definizione di funzione potrebbe infatti essere non costruttiva. Ad esempio per la funzione *f* che dati *n* e *k* vale 1 se e solo se nell'espansione decimale di π compaiono almeno *n* cifre pari a *k*, 0 altrimenti, non si ha nessun modo evidente per calcolare *f*, soprattutto nei casi in cui *f* vale *0*.

Infatti come si fa a sapere quando *f(n,k)=0* ? Bisognerebbe calcolare tutte le cifre di π e verificare che non ci sono *n* cifre pari a *k*, ma questo metodo non è realizzabile nella pratica, perché richiederebbe un processo infinito di calcolo.

Anche una descrizione di una funzione in modo costruttivo potrebbe comunque non essere praticabile come algoritmo. Ad esempio è noto che la definizione di determinante richiede di calcolare una somma per tutte le permutazioni di indici. Pertanto un algoritmo basato sulla definizione di determinante non sarebbe utilizzabile per matrici di dimensione medio-grandi, perché le permutazioni di *n* elementi sono *n*!, che cresce molto rapidamente rispetto a *n*.

5.3 Metodi di descrizione degli algoritmi

Gli algoritmi possono essere descritti a parole, ad esempio per risolvere il problema 1 (calcolare il MCD di due numeri interi) si può utilizzare il procedimento descritto da Euclide nel settimo libro "Elementi":

> *Si sottragga ripetutamente il più piccolo dei due numeri al più grande, fino a che i due numeri non diventano uguali. Il numero trovato è il massimo comun divisore dei due numeri di partenza.*

Per avere una descrizione più precisa e e con una terminologia uniforme si preferiscono usare metodi più formalizzati. Tra questi i più diffusi sono i diagrammi di flusso e lo pseudo-codice.

Infine un algoritmo può essere rappresentato anche tramite un linguaggio di programmazione, diventando a tutti gli effetti un programma. Come abbiamo già detto, questa forma di descrizione di un algoritmo è però troppo dettagliata se lo scopo è quello di sintetizzare o analizzare il procedimento o illustrarlo ad altri esseri umani.

5.4 Diagrammi di flusso

Un **diagramma di flusso** è un grafo orientato, ovvero è costituito da un insieme finito di nodi e da

un insieme di frecce che collegano i nodi tra di loro.

I nodi sono di quattro tipi

1. Ellisse: inizio e fine del diagramma
2. Rettangolo: istruzioni di calcolo (o istruzioni descritte in altro modo, ad esempio a parole)
3. Rombo: biforcazioni del diagramma associate a condizioni (vero/falso)
4. Parallelogramma: istruzioni di ingresso o di uscita.

Sono ammessi punti di raccordo a cui si può arrivare seguendo percorsi diversi.

Ogni nodo del diagramma, ad eccezione dei rombi e del nodo finale, ha un'unica freccia uscente che lo collega al nodo che lo segue nell'ordine di esecuzione.

Il nodo finale non ha frecce uscenti, mentre il nodo iniziale non ha frecce entranti. I nodi rombo hanno due frecce uscenti: una etichettata con SÌ e una con NO.

L'esecuzione di un diagramma di flusso parte con il nodo INIZIO e sono eseguite le istruzioni che si incontrano durante il percorso, passando da un nodo a quello successivo seguendo le frecce. Arrivando ad un nodo rombo, il percorso prosegue percorrendo l'arco etichettato con SÌ se la condizione è vera, quello con NO se la condizione è falsa.

I nodi di calcolo richiedono di effettuare dei calcoli e di assegnare il risultato alle variabili. La forma più utilizzata di calcolo è l'assegnamento, che è indicato con la notazione :

$$x \leftarrow e$$

in cui *x* è una variabile ed *e* è un'espressione. Tale operazione assegna alla variabile *x* il valore corrente dell'espressione *e*.

Le istruzioni di calcolo possono anche istruzioni non elementari, descritte a parole o tramite altri diagrammi di flusso, o comunque ritenute eseguibili in qualche modo.

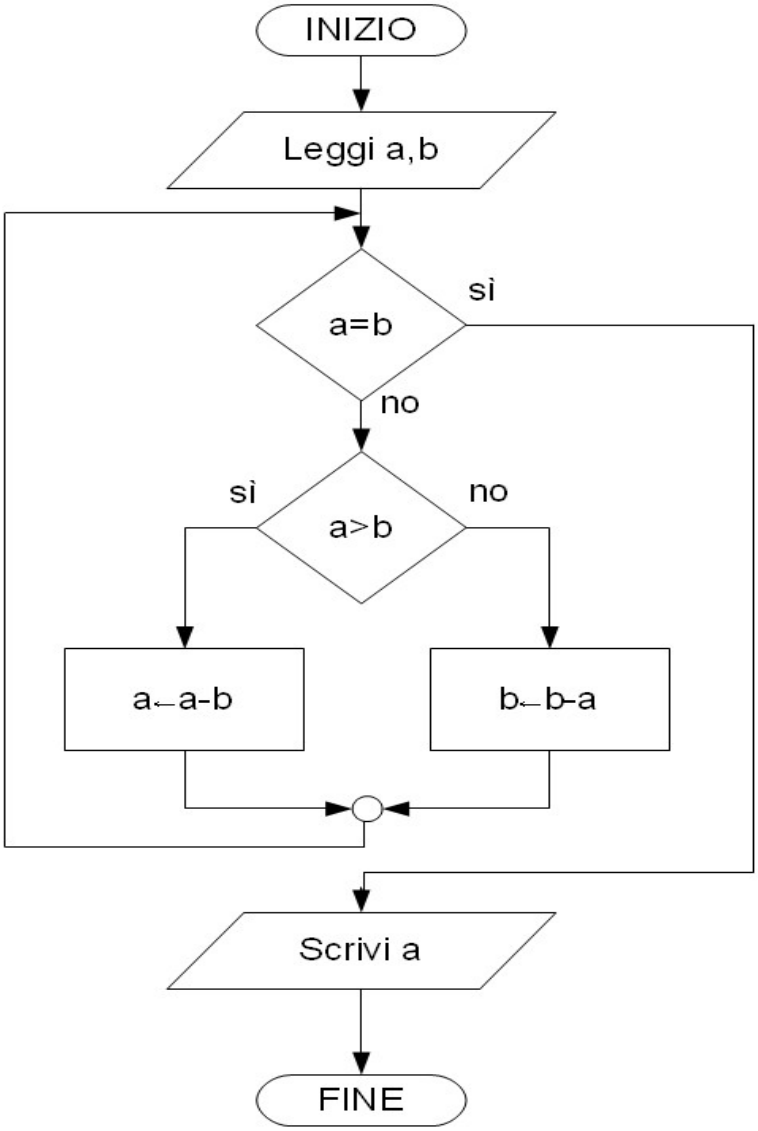
Le istruzioni di ingresso (contrassegnate dalla parola *Leggi*) consentono all'esecutore di ricevere dei dati dall'esterno, mentre quelle di uscita (contrassegnate dalla parola *Scrivi*) indicano all'esecutore di produrre all'esterno dei risultati.

L'esecuzione termina quando si arriva al nodo FINE.

Come esempio di diagramma di flusso nella pagina seguente vedremo l'algoritmo di Euclide per il MCD.

Mentre è facile seguire l'esecuzione di un diagramma di flusso, è meno ovvio capire qual è lo scopo dell'algoritmo, soprattutto quando il diagramma è molto intrigato. L'utilizzo delle frecce inoltre crea problemi quando l'algoritmo deve essere tradotto in un linguaggio di programmazione moderno, dato che l'istruzione equivalente della freccia (GOTO) non è più tenuta in considerazione e si evita di usarla.

Per questi motivi si preferisce usare la pseudo-codifica.



5.5 Pseudo-codifica

La **pseudo-codifica** utilizza un linguaggio testuale abbastanza simile (almeno come struttura) ad un linguaggio di programmazione: infatti le versioni "inglesi" della pseudo-codifica hanno molte istruzioni pressoché identiche a quelle presenti nei veri linguaggi di programmazione, come Algol, Pascal o C. Noi però utilizzeremo una versione italiana, avente le seguenti istruzioni:

1. **Inizio**, che segna l'inizio dell'algoritmo.
2. **Fine**, che segna la fine dell'algoritmo.
3. **Leggi**, che ottiene dati dall'esterno.
4. **Scrivi**, che invia dati all'esterno.
5. **Assegnamento**, in una forma simile a quella usata nei diagrammi di flusso.
6. **Se** *condizione* **Allora**
   istruzioni_1
   Altrimenti
   istruzioni_2

Fine-se,
che indica di eseguire le istruzioni tra **Allora** e **Altrimenti** se la *condizione* è vera, le istruzioni tra **Altrimenti** e **Fine-Se** se la *condizione* è falsa. In ogni caso l'esecuzione continuerà con l'istruzione successiva alla **Fine-Se**.

7. **Mentre** *condizione*
   Ripeti
   istruzioni
   Fine-Ripeti,
   che indica di eseguire ripetutamente le istruzioni tra **Ripeti** e **Fine-Ripeti** fintantoché la condizione resta vera, quando questa diventa falsa il ciclo termina e l'esecuzione continua con l'istruzione successiva alla **Fine-Ripeti**.
8. Altre istruzioni saranno illustrate nei prossimi capitoli.

Come esempio di algoritmo in pseudo-codifica presentiamo ancora l'algoritmo di Euclide per il M.C.D.:

```
Inizio
    Leggi a,b
    Mentre a ≠ b
    Ripeti
        Se a>b Allora
            a ← a-b
        Altrimenti
            b ← b-a
        Fine-se
    Fine-Ripeti
    Scrivi a
Fine
```

Anche in uno pseudo-codice possono comparire operazioni non elementari, descritte ad esempio a parole, fermo restando che deve essere noto come possano essere eseguite.

5.6 Proprietà degli algoritmi

Gli algoritmi hanno tre proprietà obbligatorie:

- Eseguibilità: l'algoritmo deve essere scritto in modo tale che l'esecutore sia in grado di eseguirlo autonomamente senza l'aiuto di altre entità (a parte chi inserisce i dati in ingresso e legge le risposte). L'algoritmo non deve pertanto richiedere all'esecutore di compiere azioni al di là delle proprie capacità.
- Finitezza: l'algoritmo deve terminare sempre, qualsiasi siano i dati forniti. Un procedimento che non termina è inutile, perché in tal caso la risposta non arriva mai.
- Correttezza: l'algoritmo deve produrre sempre il risultato corrispondente ai dati. Ovviamente un procedimento che sbaglia è inutile in quanto non calcola veramente la funzione per cui è stato ideato.

Un procedimento che viola una di queste proprietà non è ritenuto un algoritmo valido per il problema da risolvere. Solo sulla correttezza ci possono essere situazioni in cui si accettano algoritmi non corretti, ad esempio gli algoritmi probabilistici, che possono commettere errori con una probabilità nota (o stimata) a priori.

Una quarta proprietà importante ma non obbligatoria è l'efficienza: l'algoritmo deve consumare la minor quantità possibile di risorse di calcolo (tempo, spazio in memoria, ecc.).

Per valutare il costo di un algoritmo in termini di consumo di risorse, in particolari quelle temporali e spaziali, si rimanda il lettore al capitolo apposito.

A titolo di esempio analizziamo l'algoritmo di Euclide.

L'eseguibilità è ovvia in quanto il procedimento richiede solo di compiere confronti e sottrazioni tra numeri interi.

La terminazione è garantita dal fatto che ad ogni passo il più grande tra i due numeri diminuisce e tuttavia i due numeri rimangono positivi. E' evidente che in questo modo prima o poi i due numeri diventeranno uguali, nel caso peggiore arrivando al valore uno, e così termina l'algoritmo.

La correttezza deriva dalla proprietà che se a e b sono due numeri naturali e $a>b$ allora

$$\mathrm{MCD}(a,b)=\mathrm{MCD}(a-b,b),$$

invece se $a<b$ allora

$$\mathrm{MCD}(a,b)=\mathrm{MCD}(a,b-a).$$

Infine vale l'ovvia proprietà

$$\mathrm{MCD}(a,a)=a.$$

Quindi ad ogni passo l'algoritmo sostituisce i due numeri con altri due che hanno lo stesso massimo comun divisore. Alla fine, $a=b$ e perciò $\mathrm{MCD}(a,b)=a$. Ma tale valore è quindi uguale al massimo comun divisore dei due numeri in ingresso.

Per quanto riguarda il costo di esecuzione, è stato calcolato che il numero di passi dipende in modo proporzionale dal numero di cifre decimali dei due numeri.

5.7 Programmi e linguaggi di programmazione

Una volta delineato un algoritmo, il passaggio successivo è l'implementazione, ovvero la traduzione in programma. Questo passaggio non è banale e può essere addirittura più difficile che non trovare un algoritmo. Infatti un programma ha bisogno di un numero di dettagli, in parte dovuti al linguaggio di programmazione scelto.

Da un certo punto di vista l'unico linguaggio di programmazione che un computer è direttamente in grado di comprendere è il linguaggio macchina. Ma programmare in tale linguaggio presenta due gravi problemi.

Innanzitutto il linguaggio macchina è estremamente rudimentale: ci sono poche istruzioni che hanno come operandi solo registri o locazioni di memoria. Perciò è veramente difficile scrivere programmi significativi ed è necessaria una lunga fase di apprendimento e anche tempi lunghi per scrivere e correggere programmi. Un aspetto poco piacevole, soprattutto nelle applicazioni scientifiche, è che il linguaggio macchina non usa una notazione algebrica, per cui il concetto di espressione non esiste. Quindi quando si programma in tale linguaggio bisogna convertire tutte le espressioni in comandi del tipo addiziona il contenuto R1 con quello del registro R2.

Inoltre un programma in linguaggio macchina è per forza di cose legato al processore e ad altre componenti hardware, nonché anche al software di base presente nel computer, per cui va riscritto nel caso in cui lo si voglia usare in piattaforme diverse da quella originale.

Per ovviare a questi due problemi, fin dagli anni 50 sono stati ideati i linguaggi di programmazione ad alto livello. Tali linguaggi sono per certi versi intermedi tra l'uomo (che usa un linguaggio naturale) e il computer (che usa il linguaggio macchina).

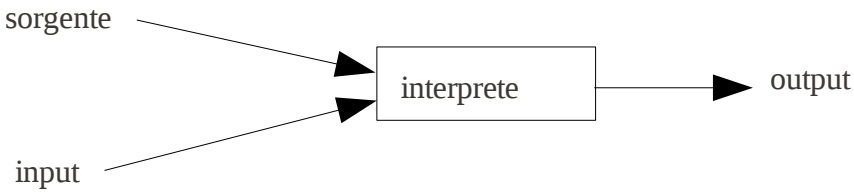
Naturalmente è necessario qualche accorgimento perché il computer sia in grado di eseguire programmi scritti nei linguaggi di programmazione ad alto livello: il programma deve essere tradotto automaticamente in linguaggio macchina.

Esistono a tal scopo due tecniche di base: l'interpretazione e la compilazione.

Nell'**interpretazione** il programma scritto nel linguaggio di programmazione (programma sorgente) viene tradotto durante l'esecuzione. Si usa un apposito programma, chiamato interprete, il quale legge il sorgente un'istruzione alla volta, la decodifica e la esegue, in modo simile a quanto fa un

processore per eseguire un programma in linguaggio macchina.

Nella fase di decodifica di un'istruzione, il risultato della traduzione non viene in alcun modo salvato, infatti se un'istruzione deve essere eseguita più volte, viene decodificata più volte.



Nella **compilazione** invece il programma sorgente viene tradotto completamente in linguaggio macchina, creando un programma eseguibile, il quale potrà essere eseguito anche più volte, senza bisogno di ulteriori traduzioni. La compilazione, producendo un programma in linguaggio macchina, consente un'esecuzione decisamente più efficiente.

D'altro canto nella compilazione vi è un certo ritardo tra quando si finisce di scrivere un programma e quando viene eseguito: tale ritardo è dovuto alla traduzione di tutto il codice. La compilazione quindi non è adatta in tutte quelle situazioni in cui si vuole un'esecuzione immediata, ad esempio di poche istruzioni o addirittura di una singola istruzione per volta.

Perciò negli ambiente interattivi, come i pacchetti matematici o le shell dei sistemi operativi si usa quasi esclusivamente l'interpretazione.

5.8 I linguaggi di programmazione

Tra i linguaggi usati nelle applicazioni scientifiche bisogna innanzitutto citare il Fortran, che è stato per l'appunto il primo linguaggio di programmazione introdotto, nato proprio per tali tipi di applicazioni (il nome è l'acronimo di FORmula TRANslation). Il linguaggio Fortran essendo il più antico linguaggio di programmazione ha subito durante gli anni una serie di modifiche (Fortran II, III, IV, 77, 90, 95, 2000, ecc.) per restare al passo con i tempi. Sono ancora molto diffusi programmi scritti con le versioni IV e 77, che oggi sono un po' obsolete, soprattutto il Fortran IV.

A livello scientifico e didattico ha avuto un discreto successo il linguaggio Pascal, nato da un precedente linguaggio (ALGOL), che ha avuto una grande influenza anche su altri linguaggi. Da Pascal e ALGOL è nato ADA, che però ha avuto una scarsa diffusione, nonostante sia stato commissionato dal dipartimento della difesa americana.

Un linguaggio *sui generis* è sicuramente APL (A Programming Language o Array Programming Language), che non ha istruzioni “a parole” ma solo simboli di tipo matematico e logico e che è in grado di trattare vettori, matrici e array multidimensionali. Ormai tale linguaggio è comunque sempre meno utilizzato.

Infine a livello scientifico sono molto utilizzati C e C++, nonostante il primo sia nato espressamente per applicazioni di sistema ed il secondo sia un estensione del primo verso la programmazione ad

oggetti.

I linguaggi di programmazione si distinguono innanzitutto in tre grandi categorie: i linguaggi imperativi, funzionali e logici.

In queste dispense ci occuperemo solamente della programmazione di tipo imperativo. Il lettore interessato ad approfondire gli aspetti degli altri paradigmi di programmazione (funzionale e logica), molto più legati ad aspetti matematici, è invitato a consultare dei testi specifici sull'argomento.

5.9 Esercizi

1. Scrivere un diagramma di flusso per descrivere le azioni da compiere per preparare la colazione.
2. Scrivere un diagramma di flusso per rappresentare il procedimento risolutivo di un'equazione di II grado.
3. Scrivere un diagramma di flusso o uno pseudo-codice per rappresentare il metodo di addizione tra due numeri naturali.
4. Scrivere un diagramma di flusso o uno pseudo-codice per rappresentare il metodo di moltiplicazione tra due numeri naturali.
5. Scrivere un diagramma di flusso o uno pseudo-codice per rappresentare il metodo di estrazione di radice quadrata.
6. Scrivere un diagramma di flusso o uno pseudo-codice per rappresentare il metodo di calcolo della derivata di un polinomio.
7. Scrivere un diagramma di flusso o uno pseudo-codice per rappresentare il metodo di moltiplicazione di due polinomi.

6 Linguaggio Matlab e istruzioni elementari

6.1 Introduzione a Matlab

Matlab è uno dei suoi software più diffusi per il calcolo “numerico”. In queste dispense in realtà descriveremo un programma gratuito molto simile a Matlab, chiamato Octave, ma continueremo a riferirci al termine Matlab.

Matlab si comporta come una sorta di “super-calcolatrice”, ovvero è possibile digitare un'espressione ed ottenere sullo schermo il risultato.

Ad esempio digitando al prompt *1+2* l'espressione

```
octave:1> 1+2
```

otterremo come risposta

```
ans = 3
```

La stringa “`octave: 1>`” è appunto il **prompt** e “1” è il numero progressivo dell'espressione inserita. “ans =” indica la risposta (abbreviazione di *answer*).

Matlab, al pari di altri sistemi interattivi di calcolo o di vari linguaggi interpretati, opera un ciclo “*read-eval-print*”, ovvero esegue continuamente le seguente tre fasi

1. legge da tastiera un'espressione;
2. la valuta;
3. scrive sullo schermo il risultato.

Matlab può lavorare sia con numeri interi, numeri reali e numeri complessi.

I numeri interi sono digitati usualmente in base 10 e con essi si possono usare le operazioni +, -, * (prodotto). Il quoziente della divisione si ottiene con la funzione *fix* applicata alla divisione /, ad esempio digitando

```
fix(17/5)
```

si ottiene 3, mentre se si calcola direttamente 17/5 si ottiene come risultato il numero reale 3.4.

Il resto della divisione si ottiene con la funzione *rem*: ad esempio con

```
rem(17,5)
```

si ottiene 2.

I numeri reali si possono introdurre in notazione decimale (con il punto al posto della virgola). Ma si può anche usare la notazione scientifica, comoda per esprimere numeri molto grandi, come $3.7 \cdot 10^{56}$ che si scrive 3.7e+56, o molto piccoli, come $2 \cdot 10^{-17}$, che scrive 2e-17.

I numeri reali ammettono le quattro operazioni +, -, * e /. Inoltre è possibile usare l'elevamento a potenza sia con l'operatore ** che con l'operatore ^. Ad esempio 7^9 si calcola con 7**9 o con 7^9.

Le principali funzioni con numeri reali già presenti in Matlab sono

- *exp*, che calcola e^x
- *log*, logaritmo naturale
- *sin, cos, tan*, con argomento in radianti

- *sqrt*, che calcola la radice quadrata
- *floor, fix e ceil* che calcolano l'intero più vicino all'argomento partendo rispettivamente da sotto e da sopra: *floor(3.4)* e *fix(3.4)* danno come risultato 3, mentre *ceil(3.4)* fa 4.

Queste funzioni richiedono obbligatoriamente le parentesi che racchiudono l'argomento.

Il valore di π è disponibile con `pi`.

Nelle espressioni si possono usare anche le parentesi tonde per alterare l'ordine di valutazione. Ad esempio mentre 3+4*5 fa 23, (3+4)*5 fa 35. Si noti che si possono solo usare le parentesi tonde, anche in forma ripetuta, ciò infatti non crea problemi di ambiguità.

6.2 Variabili ed assegnamento

Le variabili sono piccole parti di memoria su cui un programma è in grado di memorizzare un valore per poi riutilizzarlo in seguito.

Le variabili sono identificate da un nome, che deve iniziare con una lettera alfabetica e deve essere composto solo da lettere, cifre e il simbolo di sottolineatura ("_").

Ad esempio sono nomi validi di variabili "z", "x1", "a2b", "tasso_di_interesse".

I nomi possono essere usati per identificare anche altri oggetti all'interno di un programma: in generale si chiamano **identificatori**.

In Matlab una variabile può essere utilizzata appena è stata soggetto di un assegnamento. Ad esempio l'istruzione

```
a=7
```

crea una variabile di nome il cui valore è 7.

La proprietà fondamentale delle variabili è la persistenza. La variabile *a* continuerà a contenere 7, fintantoché non sarà soggetta ad un nuovo assegnamento. Ad esempio con

```
a+8
```

si ottiene come risultato 15, ma a contiene sempre 7. Infatti digitando `a`, si ottiene 7.

Se poi si cambia valore ad *a* con l'istruzione

```
a=9
```

adesso a+8 varrà 17.

E' importante notare che al contrario di quanto avviene negli usuali linguaggi di programmazione, in Matlab non esiste il concetto di "dichiarazione di variabile". Infatti mentre in Matlab una variabile inizia ad esistere dopo il primo assegnamento, in Pascal, in C e in molti altri linguaggi il programmatore deve indicare esplicitamente e preventivamente quali variabili ha intenzione di usare nel programma. Perciò in tali linguaggi usare una variabile non dichiarata è un errore.

Inoltre nella dichiarazione delle variabili è obbligatorio specificare il tipo della variabile, ad esempio in Pascal con

```
var a:integer
```

o in C con

```
int a
```

si dichiara che *a* è una variabile intera, ovvero potrà contenere solo numeri interi.

In Matlab invece le variabili sono generali e possono contenere dati arbitrari (in momenti diversi).

Ad esempio *a* all'inizio contiene il numero 7 (intero), ma in seguito potrebbe contenere un vettore di numeri reali.

L'assegnamento ha come forma elementare

x=e

in cui *x* è l'identificatore di una variabile ed *e* è un'espressione.

Si noti che in *e* può comparire anche *x* stesso. Ad esempio, partendo da

```
a=5
```

se si esegue l'istruzione

```
a=a+1
```

a diventa 6, dato che l'espressione *a+1* ha proprio tale valore.

Una caratteristica importante da capire è che l'assegnamento non fa altro che assegnare alla variabile *x* il valore che in quel momento possiede l'espressione *e*. Qualunque successiva operazione che altera il valore di *e* non avrà più alcun effetto su *x*.

Ad esempio, partendo da

```
a=7
```

se si assegna a *b* il valore di *a+1*

```
b=a+1
```

quando si richiede il valore di *b* si ottiene 8.

Anche quando si cambia il valore di *a* con

```
a=5
```

b rimane inalterato al valore 8.

Un'altra proprietà interessante è che l'assegnamento non è commutativo. Sia infatti l'espressione assegnata composta da una variabile. Ad esempio si prendano due variabili *a* e *b*, allora *a=b* e *b=a* hanno effetti diversi. Infatti, partendo da

```
a=7
b=4
```

l'istruzione `a=b` modifica *a* facendola diventare uguale a *b*: ora *a* e *b* hanno entrambi il valore 4.

Mentre, partendo dalla stessa situazione precedente (*a=7* e *b=4*)

con l'istruzione

```
b=a
```

si ottiene che sia *a* che *b* assumano il valore 7.

6.3 Dichiarazione e chiamata di funzioni elementari

In Matlab è possibile definire delle funzioni che consentono di calcolare uno o più valori in funzione di alcuni valori, detti parametri.

Si voglia ad esempio calcolare il valore *ip* dell'ipotenusa a partire dal valore dei due cateti *c1* e *c2*.

In Matlab si può definire la funzione ipotenusa nel seguente modo

```
function ip=ipotenusa(c1,c2)
     s=c1**2+c2**2;
     ip=sqrt(s);
endfunction
```

In tale definizione *c1* e *c2* sono i parametri della funzione, mentre *ip* è il risultato. Le istruzioni interne alla funzione sono eseguite al momento in cui la funzione viene chiamata.

Si noti che gli assegnamenti sono conclusi con il punto e virgola per evitare che il risultato dell'assegnamento sia scritto (inutilmente) sullo schermo.

Per evitare che ogni volta che Matlab riparte sia necessario ridigitare la funzione, è possibile scrivere la definizione all'interno di un file. Il nome di tale file deve essere uguale all'identificatore della funzione, seguito dal suffisso ".m". La funzione *ipotenusa* deve essere quindi salvata sul file `ipotenusa.m` in un'apposita directory.

La funzione ipotenusa può essere richiamata sia da riga di comando, sia all'interno di un'altra funzione. La chiamata di una funzione deve specificare i valori da attribuire a ciascun parametro: tali valori si chiamano argomenti e ognuno di essi è il risultato di un'espressione.

Ad esempio con

```
ipotenusa(3,4)
```

si ottiene 5 dato che la funzione viene chiamata imposta 3 come valore di *c1* e 4 come valore di *c2*.

L'esecuzione delle due istruzioni interne produce come effetto che *s* vale 25 e *ip* vale 5. Questo è il risultato finale della funzione.

Se *a* vale 3 e *b* vale 2, allora

```
b=ipotenusa(a,b*2)
```

si assegna 5 alla variabile, dato che nuovamente *c1* vale 3 e *c2* vale 4.

E' importante notare che sia *s* che *ip* sono variabili locali alla funzione ipotenusa e che quando la funzione termina, esse spariscono dalla memoria.

Una funzione può anche produrre più risultati contemporaneamente. In tal caso si usa la notazione

function [ris1,ris2,...]=funzione(argomenti)

    istruzioni

endfunction

ove ris1, ris2, …, sono le variabili che conterranno i risultati.

Ad esempio una funzione che calcola le soluzioni dell'equazione di secondo grado

$$ax^2+bx+c=0$$

a partire dai coefficienti a,b,c e nell'ipotesi che il delta sia maggiore o uguale a zero è la seguente

```
function [x1,x2]=risolvi_eq2_grado(a,b,c)
     delta=b*b-4*a*c;
     x1=(-b-sqrt(delta))/(2*a);
```

```
    x2=(-b+sqrt(delta))/(2*a);
endfunction
```

Una descrizione precisa del funzionamento delle funzioni sarà fornita in un capitolo successivo.

6.4 Convenzioni lessicali

Quando si scrivono le istruzioni o i programmi in Matlab, occorre tenere in mente le seguenti convenzioni.

Innanzitutto Matlab è un linguaggio *case-sensitive*, ovvero la differenza tra lettere maiuscole e minuscole è significativa. Ad esempio le variabili *a* e A sono distinte, perciò se si assegna un valore ad *a*, si ottiene un messaggio di errore quando si tenta di valutare A.

Anche le istruzioni e i nomi delle funzioni tengono conto di maiuscole e minuscole. Ad esempio la funzione coseno si chiama `cos` scritta in minuscolo, non esiste una funzione chiamata Cos, COS o coS. Allo stesso modo l'istruzione `if`, ad esempio, deve essere scritta solo in minuscolo, altrimenti si genera un errore di sintassi.

In una funzione normalmente si deve mettere un'istruzione per riga e, nel caso di assegnamenti, si deve mettere alla fine il punto e virgola, altrimenti quando la funzione viene eseguita viene visualizzato sullo schermo il risultato dell'assegnamento e ciò è particolare dannoso all'interno di cicli in cui si riempe lo schermo (e si rallenta l'esecuzione del programma) con questi risultati intermedi, per la maggior parte inutili.

Se però accade che un'istruzione è troppo lunga per entrare in una riga dello schermo, la si può spezzare con tre puntini (…) come nel seguente esempio

```
x=a**2 + b**2 + c**2 + d**2 +e**2 + ...
   f**2 + g**2 + h** 2;
```

Infine è possibile inserire un commento all'interno di una funzione o in generale di un file contenente istruzioni Matlab, mediante il simbolo %.

Un commento può contenere qualsiasi sequenza di caratteri e termina a fine riga. Di solito nei commenti sono inserite delle frasi di descrizione o di chiarimento del programma. I commenti possono essere scritti anche in italiano (o in qualsiasi lingua), dato che l'interprete Matlab ignora completamente il contenuto, passando direttamente alla riga successiva.

Ad esempio

```
% funzione che calcola l'ipotenusa di un triangolo rettangolo
% dati i due cateti c1 e c2
function ip=ipotenusa(c1,c2)
     s=c1**2+c2**2;
     ip=sqrt(s);
endfunction
```

6.5 Introduzione alla semantica denotazionale*

Una formalizzazione matematica del processo di calcolo delle espressioni e dell'esecuzione dei programmi da parte di Matlab può essere fornita mediante la semantica denotazionale.

Sia V l'insieme di tutte le variabili usate in un certo ambito. Il contenuto corrente delle variabili forma lo stato, che formalmente è definito mediante un'applicazione σ che ad ogni variabile $x \in V$ assegna il valore corrispondente $\sigma(x)$. Indicando con U l'insieme di tutti i possibili valori che una variabile Matlab può assumere (U quindi contiene come elementi tutti i numeri reali e interi

rappresentabili, tutti i vettori e le matrici, ecc.), allora uno stato è una funzione σ:V→U.

Data un'espressione *e*, indicheremo con V(e,σ) il valore che e assume nello stato σ. Tale valore sarà calcolato da un processo di valutazione che può essere formalizzato dalle seguenti equazioni di base

V(k,σ)=k, se k è una costante

V(x,σ)=σ(x), se x è una variabile

e dalle seguenti equazioni ricorsive

V(e1+e2,σ)=V(e1,σ)+V(e2,σ)

V(e1-e2,σ)=V(e1,σ)-V(e2,σ)

V(e1*e2,σ)=V(e1,σ)×V(e2,σ)

ecc.

in cui e1 e e2 sono espressioni.

Considerando gli stati σ;τ e ρ tali che

- σ(x)=2 e σ(y)=3,
- τ(x)=1 e τ(y)=-1 e
- ρ(x)=-2.1 e ρ(y)=3.4

l'espressione e = x*3+2*y+5 assume i valori

nello stato V(e,σ)=2*3+2*3+5=17

nello stato V(e,τ)=1*3+2*(-1)+5=6

infine nello stato V(e,ρ)=(-2.1)*3+2*3.4+5=5.5

Un'istruzione di assegnamento del tipo

x=e

può essere allora pensato come una funzione che altera lo stato corrente, producendo come risultato un nuovo stato in cui la variabile *x* assume come valore quello di *e*, mentre le altre variabili conservano il valore assunto precedentemente.

In termini formali, indicando con σ'=E(x=e,σ) lo stato che si ottiene eseguendo l'istruzione “x=e” nello stato σ, si avrà che

σ'(x)=V(e,σ)

mentre per ogni variabile y distinta da x

σ'(y)=σ(y).

Ad esempio se σ è tale che σ(x)=2 e σ(y)=3, allora E(x=y*2,σ) produce lo stato σ' in cui σ'(x)=6 e σ'(y)=3, mentre se in τ vale τ(x)=1 e τ(y)=-1, allora E(x=y*2,σ) produce lo stato τ' in τ'(x)=-2 e τ'(y)=-1.

In generale, per ogni istruzione *s* indicheremo con E(s,σ) lo stato che si ottiene eseguendo *s* nello stato σ.

Una sequenza di due istruzioni *s1* e *s2* produrrà uno stato che dipende sia da σ sia dall'ordine di esecuzione. Indicando con “s1; s2” la sequenza composta da s1 e poi da s2, avremo la seguente equazione

$E(s1;s2 , \sigma)=E(s2, E(s1,\sigma))$

mentre l'esecuzione nell'ordine inversa è regolata dall'equazione

$E(s2;s1 , \sigma)=E(s1, E(s2,\sigma))$.

La generalizzazione al caso di una sequenza di *n* istruzioni $s_1, s_2, \ldots, s_n$ è ovvia e lasciata al lettore.

6.6 Esercizi

1. Scrivere una funzione che calcola l'area e il perimetro di un triangolo a partire dalle misure dei tre lati. Suggerimento, per l'area usare la formula di Erone.
2. Scrivere una funzione che calcola il determinante di una matrice 2x2 a partire dati i suoi 4 elementi
3. Scrivere una funzione che calcola il determinante di una matrice 3x3 a partire dati i suoi 9 elementi
4. Scrivere una funzione che calcola le soluzioni di un sistema lineare di due equazioni in due incognite a partire dai 4 coefficienti e dai 2 termini noti.
5. Scrivere una funzione che calcola il prodotto u+iv di due numeri complessi a+ib e c+id (senza usare l'aritmetica complessa di Matlab), prendendo come input a,b,c,d e producendo come risultato u,v

7 Strutture condizionali

7.1 Condizioni e tipo di dato logico

Un dato logico (o Booleano) può assumere solo due valori: *false* e *true*. Matlab rappresenta *false* con il numero 0 e *true* con 1. Una condizione è un'espressione il cui risultato è di tipo logico.

Le condizioni elementari si possono formare mediante gli operatori di confronto, i quali si possono applicare sia a numeri, sia a stringhe:

- >
- <
- >=, che corrisponde a ≥, dato che sulla tastiera non è presente tale simbolo
- <=, che corrisponde a ≤
- ==, che è il simbolo dell'uguaglianza. Il doppio uguale serve a distinguerlo dall'operazione di assegnamento che viene indicata con l'uguale singolo. Infatti *a=b* significa che la variabile *a* diventa uguale alla variabile *b* (assume il valore corrente di *b*), mentre *a==b* si chiede se *a* abbia lo stesso valore di *b*.
- !=, che corrisponde a ≠, dato che ! indica il NOT. Al posto di != si può usare anche ~=.

Ad esempio se *a* vale 3 e *b* vale 7, allora la condizione *a<b* è vera e quindi vale true, mentre la condizione *a*b==4* è falsa (ossia vale false).

Le condizioni possono essere composte con gli operatori booleani && (che corrisponde a AND), || (che corrisponde a OR) e ! (che corrisponde a NOT). Riportiamo qui le tabelle di verità corrispondenti, che sono perfettamente analoghe a quelle dell'algebra di Boole

C1	C2	C1 && C2	C1 || C2	! C1
false	*false*	*false*	*false*	*true*
false	*true*	*false*	*true*	*true*
true	*false*	*false*	*true*	*false*
true	*true*	*true*	*true*	*false*

Quindi

- C1 && C2 è vera solo se entrambe le condizioni C1 e C2 sono vere (congiunzione), ad esempio *(a>b) && (b>3)* è falsa, mentre *(a<b) && (b==7)* è vera
- C1 || C2 è vera se almeno una delle due condizioni C1 e C2 è vera (disgiunzione), ad esempio *(a>b) || (b>3)* è vera, mentre *(a+b<5) || (b<7)* è falsa.
- ! C1 è vera solo se C1 è falsa (negazione), ad esempio *!(a>5)* è vera, mentre *!(b-a==4)* è falsa.

Dato che && e || sono associative, è possibile usare congiunzioni e disgiunzioni multiple

- C1 && C2 && … Cn è vera se solo se tutte le condizioni C_i sono vere
- C1 || C2 || … || Cn è vera se almeno una delle condizioni C_i è vera

Ad esempio per controllare se un anno *a* è bisestile, è necessario controllare che

1. *a* sia divisibile per 4 e

1. *a* non sia divisibile per 100 oppure
2. *a* sia divisibile per 400

Infatti la regola dice che gli anni non divisibile per 4 non sono bisestili, ad esempio 2011 e invece generalmente quelli divisibili per 4 lo sono, ad esempio 2012. Ma gli anni che iniziano il secolo non lo sono, ad esempio 1900, tranne quelli divisibili per 400, come 2000, che lo sono.

La condizione che controlla se *a* è bisestile è quindi

```
(rem(a,4)==0) && (rem(a,100)!=0 || rem(a,400)==0)
```

Per controllare se un numero *x* appartiene ad un intervallo chiuso [*a*,*b*], con *a*<*b*, si può usare la seguente condizione

(x>=a) && (x<=b)

Si noti che a<=x<=b non funziona correttamente perché verrebbe interpretata come (a<=x)<=b, cioè controlla se il risultato del confronto a<=x (che è 0 o 1) è minore di b. Ad esempio per a=1, b=4 e x=0 darebbe come risultato 1 (cioe true), anziché 0.

7.2 If-else e if

L'istruzione condizionale di base, presente in tutti i linguaggi, è l'istruzione If-Then-Else. In Matlab, come del resto in molti linguaggi assume due forme, con e senza parte *else*.

7.2.1 If-then-else

In Matlab l'istruzione *if-then-else* ha la seguente sintassi

if (C)

    S1

else

    S2

endif

ove C è una condizione e S1, S2 sono due sequenze di istruzioni. La parola chiave **endif** può essere sostituita dal semplice **end**.

L'esecuzione di tale istruzione consiste nei seguenti passi

1. viene valutata la condizione C
2. se il risultato è true, è eseguita la sequenza S1
3. se il risultato è false, è eseguita la sequenza S2.

Ad esempio per calcolare il massimo m di due numeri reali a,b si può usare il seguente schema

$$m=\begin{cases} a & \text{se } a\geq b \\ b & \text{se } a<b \end{cases}$$

che corrisponde alla funzione

```
function m=massimo(a,b)
```

```
     if (a>b)
          m=a;
     else
          m=b;
     endif
endfunction
```

Un altro esempio è una funzione che calcola il valore assoluto di un numero reale x (senza usare la funzione abs) potremo sfruttare il seguente schema

$$|x| = \begin{cases} x & \text{se } x \geq 0 \\ -x & \text{se } x < 0 \end{cases}$$

cioè la funzione

```
function a=assoluto(x)
     if(x>=0)
          a=x;
     else
          a=-x;
     endif
endfunction
```

Nell'istruzione if-then-else sia S1 che S2 possono essere sequenze composte da più istruzioni. Ad esempio il seguente codice calcola contemporaneamente il massimo e il minimo di due numeri reali

```
function [mass,min]=massimo_minimo(a,b)
     if (a>b)
          mass=a;
          min=b;
     else
          mass=b;
          min=a;
     endif
endfunction
```

7.2.2 If-then

L'istruzione *if* ha anche una versione senza la parte *else*. Tale istruzione è chiamata If-Then ed

assume in Matlab la seguente sintassi

if(C)

S1

endif

ove C è una condizione e S1 è una sequenza di istruzioni.

Anche in questo caso **endif** si può abbreviare in **end**.

L'esecuzione di tale istruzione corrisponde ai seguenti passi

1. è valutata la condizione C
2. se il risultato è true, è eseguita la sequenza S1
3. se il risultato è false, non si fa nulla.

Questa forma si distingue sintatticamente dalla forma completa (con **else**) perché alla fine della prima sequenza non compare **else**, ma un'altra istruzione.

Ad esempio un modo alternativo per scrivere la funzione massimo è

```
function m=massimo(a,b)
     m=b;
     if (a>b)
          m=a;
     endif
endfunction
```

Questa soluzione si può facilmente generalizzare al calcolo del massimo su tre (o più argomenti):

```
function m=massimo(a,b,c)
     m=a;
     if (b>m)
          m=b;
     endif
     if (c>m)
          m=c;
     endif
endfunction
```

In tal caso, infatti, la variabile *m* agisce da massimo parziale: prima solo su *a*, dopo il primo contiene il massimo tra *a* e *b*, poi il massimo di tutti e tre i numeri.

Un altro esempio interessante è il seguente: dati tre numeri reali, contare quanti sono gli elementi uguali tra di loro.

```
function u=uguali(a,b,c)
     u=0;
```

```
     if (a==b)
          u=u+1;
     endif
     if (a==c)
          u=u+1;
     endif
     if (b==c)
          u=u+1;
     endif
endfunction
```

Ad esempio uguali(4,4,4) vale 3, uguali (4,4,2) vale 2, infine uguali(4,2,1) vale 0.

In questo esempio la variabile *u* agisce da contatore: è inizializzata a zero e per ogni coppia di numeri è incrementata se i due numeri sono uguali, mediante l'istruzione *u=u+1*.

Si noti che un'istruzione If-Then-Else si può sempre ricondurre a due If-Then relativi a condizioni complementari. Ad esempio la funzione massimo si potrebbe anche scrivere

```
function m=massimo(a,b)
     if (a>b)
          m=a;
     endif
     if (a<=b)
          m=b;
     endif
endfunction
```

Tale soluzione è però da ritenersi inferiore rispetto alle altre, dato che richiede di valutare le due condizioni *a>b* e *a<=b*, pur sapendo che la seconda è l'opposto della prima.

7.3 Semantica di if

Intuitivamente i costrutti if-then-else e if-then possono illustrati dai seguenti diagrammi di flusso.

Il diagramma di flusso per

if(c)

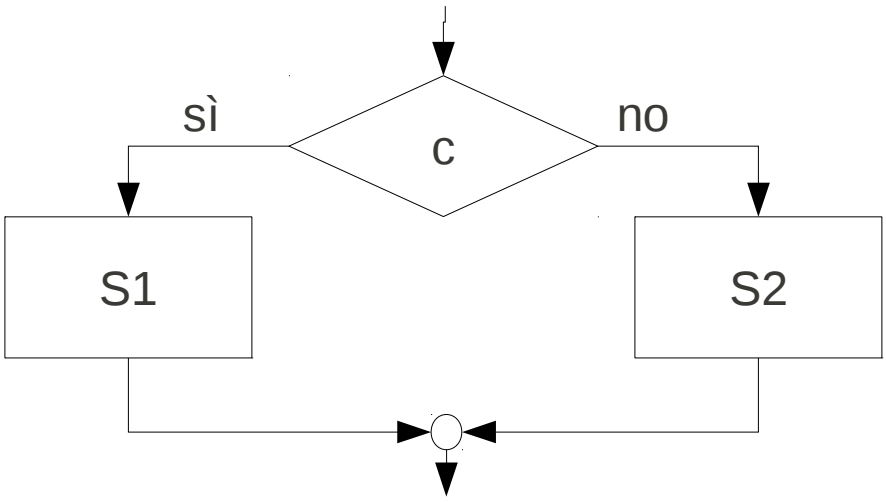
S1

else

S2

endif

è



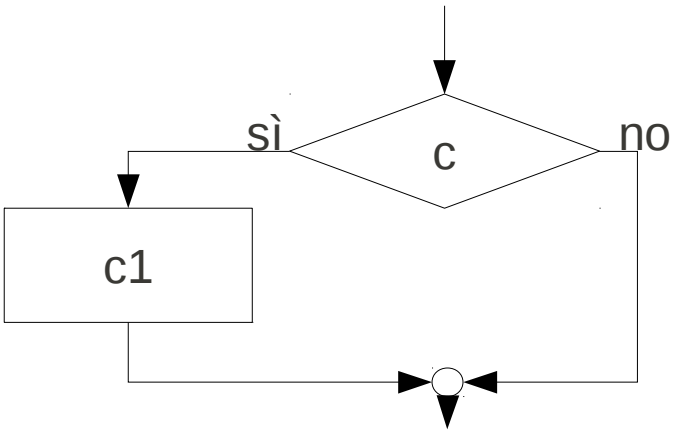
mentre quello per

if(c)

S1

endif

è



La semantica denotazionale dell'istruzione if-then-else è facilmente derivabile a partire dalla semantica intuitiva.

Infatti vale la seguente legge

$$E(\text{ if(c) S1 else S2 endif}, \sigma)) = \begin{cases} E(S1, \sigma), & \text{se } V(c,\sigma)=1 \\ E(S2, \sigma), & \text{se } V(c,\sigma)=0 \end{cases}$$

Se non è presente la parte else, la semantica si semplifica in

$$E(\text{ if(c) S1 endif}, \sigma)) = \begin{cases} E(S1, \sigma), & \text{se } V(c,\sigma)=1 \\ \sigma, & \text{se } V(c,\sigma)=0 \end{cases}$$

7.4 If in cascata

Chiaramente si può sempre inserire un'istruzione If dentro un'altra istruzione If. Ad esempio

```
if(a>5)
     if(b!=3)
          c=c+1;
     endif
     d=d-2;
endif
```

in cui quando *a* è maggiore di 5, *d* viene decrementato di 2, e se anche *b* è diverso da 3, *c* è incrementato di 1.

Una situazione frequente nella programmazione è quando si deve prendere una decisione tra più di due alternative. Mentre una decisione binaria (cioè tra due alternative) è facilmente riconducibile ad un If-Then-Else, una decisione multivoca si può ricondurre ad una serie di If-Then-Else in cascata, cioè in cui ogni If (tranne il primo) è dentro la parte Else di quello precedente.

La struttura è quindi del tipo

if(alternativa1)
    S1
else
    if(alternativa2)
        S2
    else
        if(alternativa3)
            S3
        else
            S4
        endif
    endif
endif

Ad esempio per classificare un triangolo in equilatero, isoscele o scaleno a partire dalle lunghezze dei tre lati *a,b,c* si può usare la seguente funzione

```
function tipo=classifica_triangolo(a,b,c)
     if(a==b && b==c)
          tipo=”equilatero”;
     else
          if(a!=b && a!=c && b!=c)
               tipo=”scaleno”;
```

```
        else
            tipo=”isoscele”;
        endif
    endif
endfunction
```

L'utilizzo di tale costruzione è così frequente che Matlab, al pari di alcuni altri linguaggi (ad esempio il linguaggio Fortran), possiede un'istruzione apposita, chiamata If-Then-ElseIf-Else. La sua sintassi in Matlab è

if(c1)

S1

elseif(c2)

S2

elseif(c3)

S3

…

else

Sn

endif

in cui c1,c2,... sono condizioni e S1,S2,...,Sn sono sequenze di istruzioni.

L'esecuzione di tale istruzione consiste nei seguenti passi

1. viene valutata la conduzione c1, se è vera è eseguita la sequenza S1
2. in caso contrario, viene valutata la conduzione c2, se è vera è eseguita la sequenza S2
3. viene valutata la conduzione c3, se è vera è eseguita la sequenza S3
4. …
5. se nessuna condizione è vera, viene eseguita la sequenza Sn.

In pratica con questa istruzione si evita di di dover “aprire” e successivamente “chiudere” n istruzioni if una dentro l'altra.

L'esempio della classificazione dei triangoli diventa così un po' più semplice

```
function tipo=classifica_triangolo(a,b,c)
    if(a==b && b==c)
        tipo=”equilatero”;
    elseif(a!=b && a!=c && b!=c)
        tipo=”scaleno”;
    else
        tipo=”isoscele”;
```

```
    endif
endfunction
```

Un altro esempio semplice è fornito dal seguente problema: si vuole definire una funzione che dato il numero *m*, compreso tra 1 e 12 e corrispondente ad un mese dell'anno, deve restituire come risultato il numero di giorni di tale mese (nell'ipotesi che l'anno non sia bisestile). Per risolvere tale problema si può tradurre in Matlab un famoso proverbio ottenendo

```
function g=giorni_mese(m)
    if(m==4 || m==6 || m==9 || m==11)
        g=30;
    elseif(m==2)
        g=28;
    else
        g=31;
    endif
endfunction
```

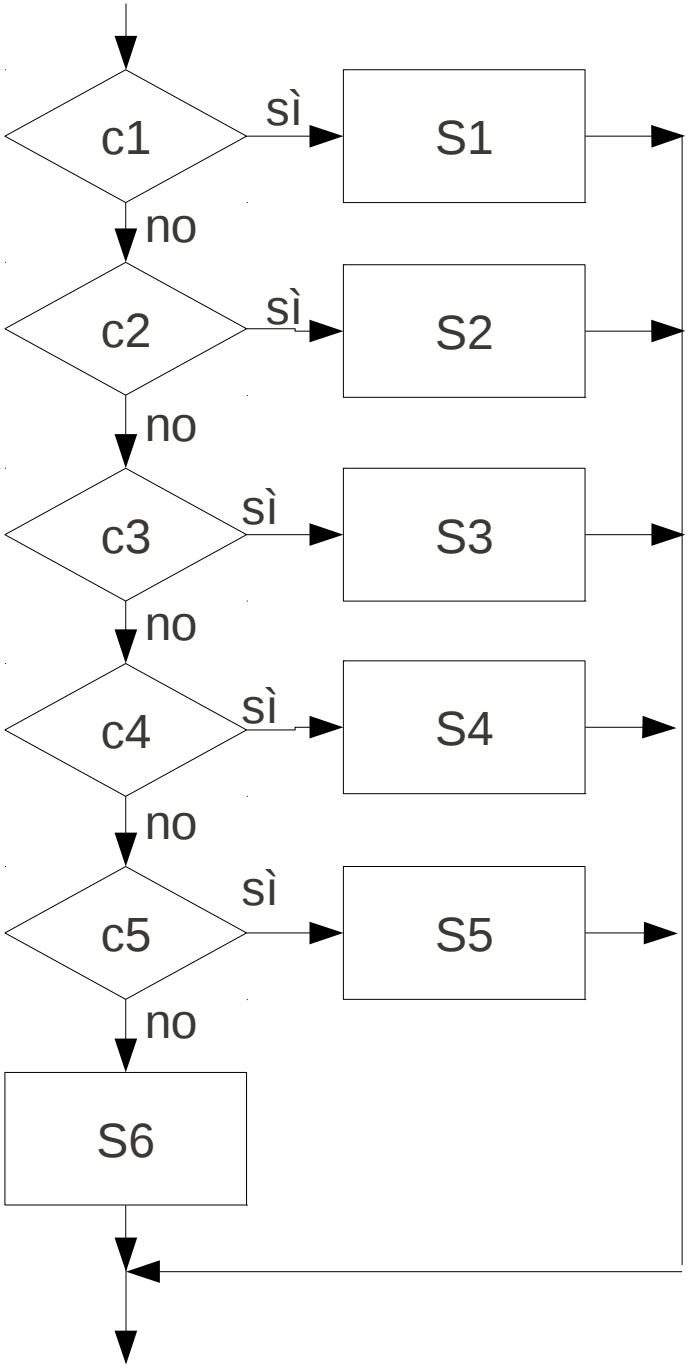
L'istruzione If-Then-ElseIf-Else corrisponde al seguente diagramma di flusso presentato nella pagina seguente.

Matlab, al pari di altri linguaggi, possiede l'istruzione *switch* che viene utilizzata in alcune situazioni per ottenere un costrutto condizionale con più alternative. Il lettore interessato può consultare a tal scopo un qualsiasi manuale.

7.5 Esercizi

1. Scrivere una funzione che dati i coefficienti a,b,c di un'equazione di secondo grado $ax^2+bx+c=0$ trova le radici reali, o nel caso in cui il delta è <0, restituisce la coppia di stringhe (“no”,”no”)
2. Scrivere una funzione che calcola il massimo di 3 numeri reali a,b,c.
3. Scrivere una funzione che calcola il massimo di 4 numeri reali a,b,c,d.
4. Scrivere una funzione che dati tre numeri naturali g,m,a, decide se formano una data valida
5. Scrivere una funzione che dati i coefficienti a,b,c,d e i termini noti e,f di un sistema di due equazioni in due incognite
   ax+by=e
   cx+dy=f
   trova le soluzioni o, quando il sistema non è compatibile, restituisce la coppia di stringhe (“no”,”no”)
6. Scrivere una funzione che dati tre coppie di punti P1=(x1,y1), P2=(x2,y2) e P3=(x3,y3) decide se P1 è all'interno del rettangolo i cui vertici opposti sono P2 e P3.
7. Scrivere una funzione che dato un numero naturale n compreso tra 1 e 999999 restituisce come risultato il numero delle cifre decimali di n (senza usare il logaritmo).

Diagramma di flusso di If-ElseIf-Else



8 Strutture iterative limitate

8.1 Iterazione limitata e illimitata

I costrutti iterativi consentono di eseguire delle istruzioni per più volte ed hanno un'importanza fondamentale nella programmazione: in pratica ogni programma significativo ha almeno un costrutto iterativo (o ciclo).
Esistono due motivi principali per usare i cicli, oltre alla semplice ripetizione *tout court* di un programma:

- algoritmi di tipo iterativo: un algoritmo ha bisogno di più iterazioni per convergere alla soluzione cercata, ad esempio l'algoritmo di Euclide per calcolare il M.C.D. di due numeri naturali;
- operazioni su collezioni di elementi: si vuole svolgere la stessa operazione su tutti gli elementi di un vettore, un insieme o altra collezione di dati, ad esempio calcolare la somma di due vettori o la media aritmetica di un insieme di dati numerici.

Un ciclo valido deve ripetere le istruzioni per un numero finito di volte (numero di iterazioni).
Distinguiamo perciò due casi:
1) Il numero di iterazioni è noto all'inizio del ciclo. Si tratta dell'**iterazione limitata**, per usare la quale bisogna rispondere alla domanda “quante volte ripetere ?”.
2) Il numero di iterazioni non è noto all'inizio del ciclo. In tal caso si tratta dell'**iterazione non limitata**. Il ciclo va avanti, controllando ad ogni iterazione se è il caso di continuare. Pertanto la domanda da porsi è “fino a quando ripetere ?”.

In questo capitolo vedremo il principale costrutto di iterazione limitata, studieremo invece nel prossimo capitolo i costrutti di iterazione non limitata.

8.2 Il ciclo for

Il ciclo for nella sua forma più utilizzata in Matlab ha la seguente sintassi

for I=A:B

S

endfor

in cui I è una variabile chiamata **indice del ciclo**, A e B sono gli estremi di un intervallo di numeri, solitamente interi, e S è una sequenza di istruzioni. La parola chiave **endfor** può essere abbreviata in **end**.

E' molto frequente, sia in matematica, che nella programmazione, di usare come indici *i* e *j*. Dato che entrambi in Matlab indicano l'unità immaginaria, se si usano i numeri complessi, *i* e *j* non possono essere utilizzati come indici del ciclo for. In queste dispense eviteremo pertanto di usare *i* nel ciclo for.

L'esecuzione del ciclo for consiste nell'eseguire S per ognuno dei numeri interi nell'intervallo [A,B]. L'indice I viene fatto variare in [A,B], ovvero assume ad ogni iterazione un valore crescente di tale intervallo: A, A+1, A+2, …., B-1, B.

S viene pertanto eseguita B-A+1 volte.

La semantica intuitiva del ciclo for è

1) valuta A e B, siano *a* e *b* i valori ottenuti
2) inizializza l'indice I al valore *a*
3) se I è maggiore di *b*, termina il ciclo

4) (altrimenti) esegui S
5) incrementa I di 1
6) torna al punto 3.

L'esecuzione di un ciclo corrisponde perciò ad eseguire le seguenti istruzioni in sequenza

I=A;

S

I=A+1;

S

…

I=B-1;

S

I=B;

S

con gli evidenti vantaggi di compattezza (S è scritto una sola volta) e generalità (A e B possono essere espressioni non costanti).

Ad esempio per calcolare la somma dei numeri naturali da 1 a n (con n>1)

```
function s=somma_numeri(n)
     s=0;
     for j=1:n
          s = s + j;
     endfor
endfunction
```

Con n=5 si avrebbe il seguente andamento

j	somma
?	0
1	1
2	3
3	6
4	10
5	15

Tale esempio si può generalizzare al calcolo di una sommatoria generica $\sum_{j=1}^{n} f(j)$ in cui f è una funzione di *j*, anche espressa sotto forma di espressione.

Ad esempio per calcolare la somma dei quadrati dei numeri da 1 a *n* si può scrivere

```
function s=somma_quadrati(n)
     s=0;
     for j=1:n
          s = s + j^2;
     endfor
endfunction
```

In maniera molto simile è possibile calcolare il fattoriale di un numero naturale *n* come prodotto di tutti i numeri naturali compresi tra 1 e *n*

```
function f=fattoriale(n)
     f=1;
     for j=1:n
          f=f*j;
     endfor
endfunction
```

Tale funzione è corretta anche nel caso *n=0*. Infatti un ciclo da 1 a 0 non ha effetto e il risultato è 1, che corrisponde a 0! per definizione.

Anche in questo caso una generalizzazione possibile è quella del calcolo della produttoria generica $\prod_{j=1}^{n} f(j)$ in cui, come sopra, è una funzione di *j*.

Supponiamo come ulteriore esempio di voler calcolare il coefficiente binomiale $\binom{n}{k}$ di due numeri naturali n e k, con $0 \leq k \leq n$. Il coefficiente binomiale si può calcolare mediante la seguente formula

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

che porta alla seguente funzione

```
function c=coeff_bin(n,k)
     f1=1;
     for j=1:n
          f1=f1*j;
     endfor
     f2=1;
     for j=1:k
          f2=f2*j;
     endfor
```

```
     f3=1;
     for j=1:n-k
          f3=f3*j;
     endfor
     c=f1/(f2*f3);
endfunction
```

Si noti che i tre cicli for usano tutti lo stesso indice *j* e ciò è perfettamente legale.

Una soluzione più compatta (ma non più efficiente) si ottiene usando la funzione fattoriale già definita sopra

```
function c=coeff_bin(n,k)
     c=fattoriale(n)/(fattoriale(k)*fattoriale(n-k));
endfunction
```

Una soluzione invece più efficiente si ottiene usando la formula alternativa

$$\binom{n}{k} = \frac{\prod_{j=n-k+1}^{n} j}{k!}$$

Una prima soluzione usa due cicli for separati

```
function c=coeff_bin(n,k)
     f1=1;
     for j=n-k+1:n
          f1=f1*j;
     endfor
     f2=1;
     for j=1:k
          f2=f2*j;
     endfor
     c=f1/f2;
endfunction
```

Poiché i due cicli for hanno lo stesso numero di iterazioni, si possono accorpare in un solo ciclo for. L'unica difficoltà è che bisogna aggiustare la formula in modo da usare lo stesso indice.

Una possibilità è quindi

```
function c=coeff_bin(n,k)
     f1=1;
     f2=1;
```

```
     for j=1:k
          f1=f1*(n+1-j);
          f2=f2*j;
     endfor
     c=f1/f2;
endfunction
```

Come nuovo esempio facciamo vedere come calcolare in maniera approssimata il massimo di una funzione *f* definita su un intervallo [*a*,*b*]. Un'idea molto semplice è quella di calcolare *f* su un insieme finito di punti di [*a*,*b*] e trovare il valore maggiore di *f* su tali punti.

Si potrebbe per esempio prendere i punti

$$x_0=a,\ x_1=a+h,\ x_2=a+2h,\ldots,\ x_{n-1}=b-h,\ x_n=b$$

ove $h=\frac{b-a}{n}$

la funzione che si ottiene è quindi

```
function m=massimo_f(a,b,n)
     h=(b-a)/n;
     x=a;
     m=f(a);
     for j=1:n
          x=x+h;
          y=f(x);
          if(y>m)
               m=y;
          endif
     endfor
endfunction
```

Il funzionamento dell'algoritmo è il seguente. La variabile *m* all'inizio è pari a *f(a)*, mentre *x* è uguale ad *a*. Ad ogni passo *x* è spostato al punto successivo, tramite un incremento di *h*, e si valuta la funzione *f* sul punto *x*, ottenendo *y*. Se *y* è maggiore di *m*, *m* è aggiornato con *y*. E' facile vedere che alla fine del ciclo

$$m=max\ \{\ f(x_j):\ per\ j=0,1,\ldots,n-1,n\ \}$$

Con semplici modifiche è possibile ottenere, anziché il valore massimo di *f*, il (o uno dei) punto x_j in cui tale valore è ottenuto.

Come ultimo esempio facciamo vedere come verificare se un numero naturale *n* è un numero primo oppure no. A tal scopo applichiamo la definizione di numero primo: *n* è primo se e solo se non è

divisibile per alcun numero *d* compreso tra 2 e *n-1*. Il numero *d* è detto divisore proprio di *n*.

In realtà è facile vedere che è sufficiente fermarsi alla parte intera, arrotondata per eccesso, della radice quadrata di *n*, dato che se *n* fosse divisibile per un numero $d>\sqrt{n}$, allora lo sarebbe anche per il numero *n/d*, che è $\leq \sqrt{n}$.

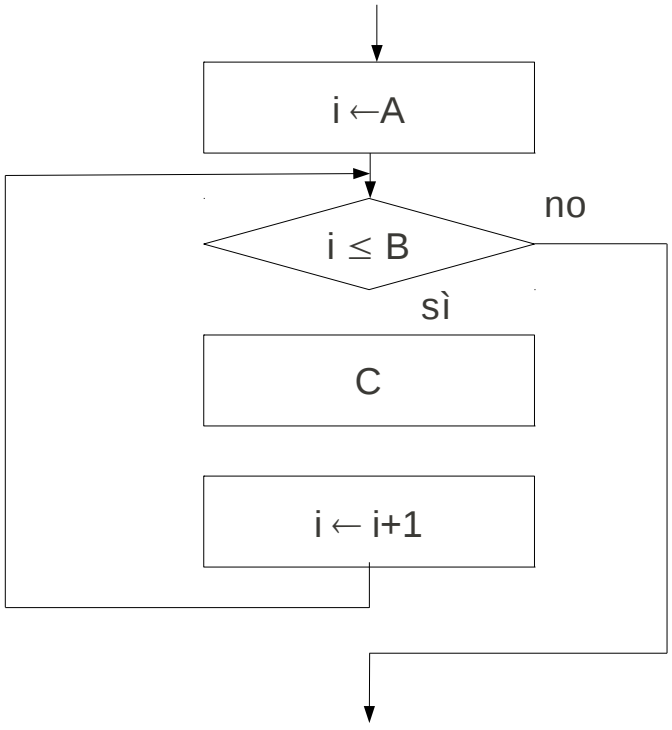
Possiamo quindi usare un ciclo for che controlla se n possiede un divisore compreso tra *2* e $\sqrt{n}$ e per ottenere il risultato corretto (true se il numero è primo, false altrimenti), ci serviremo di una variabile "flag". All'inizio tale variabile assume il valore true. Se si trova un divisore proprio di *n* la variabile è impostata a false. Alla fine del ciclo, quindi la variabile vale true se e solo se il numero n non ha divisori propri.

```
function p=primo(n)
     p=true;
     r=ceil(sqrt(n));
     for d=2:r
          if(rem(n,d)==0)
               p=false;
          endif
     endfor
endfunction
```

Vedremo una versione un po' più efficiente nel capitolo seguente.

8.3 Semantica del ciclo for

Un'ulteriore modo di avere semantica intuitiva del ciclo for del tipo for i=a:b S end, in cui *a* e *b* sono numeri interi è illustrata dal seguente diagramma di flusso



Dal punto di vista formale, la semantica denotazionale dell'intero ciclo è

$$E(\text{for i=a:b S endfor},\sigma)=\sigma_b,$$

ove $\sigma_{a-1}=\sigma$

e per ogni k=a,...,b

$$\sigma_k=E(S, E(i=k, \sigma_{k-1})).$$

8.4 Cicli annidati

Supponiamo di voler calcolare il valore di *e* mediante la somma parziale n-esima della serie

$$\sum_{k=0}^{\infty} \frac{1}{k!}$$

arrestandosi cioè all'elemento di ordine *n*.

Un modo diretto per calcolare di tale sommatoria è di generare ad uno ad uno i fattoriali dei numeri da 0 a n e di sommare i loro inversi. Ciò può essere svolto dal seguente programma

```
function e=calcola_e(n)
     somma=0;
     for k=0:n
          fattoriale=1;
          for j=1:k
               fattoriale=fattoriale*j;
          endfor
          somma=somma+1/fattoriale;
     endfor
     e=somma;
endfunction
```

In questa funzione si nota una costruzione particolare in cui si hanno due cicli for, il primo dei quali contiene il secondo. Questa situazione è indicata mediante il termine di “cicli for annidati” ed è molto utilizzata nell'informatica.

Il ciclo for con indice *k* è chiamato **ciclo for esterno**, mentre il ciclo for con indice *j* è il **ciclo for interno**.

Ovviamente l'annidamento può essere anche esteso a più di due cicli.

```
for j=1:1000
     for k=1:100
          for h=1:200
...
```

In tal caso si introduce il concetto di profondità di annidamento: il ciclo esterno (indice *j*) ha profondità 0, quello mediano (indice *k*) ha profondità 1 e quello più interno (indice *h*) ha profondità 2.

Il corretto funzionamento dei cicli annidati richiede che i for coinvolti usino indici diversi. E' però ammesso che l'intervallo associato ad un ciclo dipenda dal valore di indici di cicli. Ad esempio nella funzione che calcola *e* l'intervallo del ciclo su *j* dipende dal valore di *k*.

Per capire meglio cosa succede scriviamo i valori assunti da *j* e *k* nel caso n=4 :

k=0, j=?
k=1, j=1
k=2, j=1
k=2, j=2
k=3, j=1
k=3, j=2
k=3, j=3
k=4, j=1
k=4, j=2
k=4, j=3
k=4, j=4

Si noti che per k=0, il ciclo for interno non ha effetto e *j* non assume alcun valore.

8.5 Varianti del ciclo for di base

Alcune varianti utili del ciclo for visto in precedenza sono le seguenti.

8.5.1 For all'indietro

Per creare un ciclo for che “va all'indietro” si usa

for I=A:-1:B
    S
endfor

Ad esempio

```
for h=10:-1:1
    disp(h);
endfor
```

scrive sullo schermo (tramite la funzione disp) i numeri da 1 a 10 in ordine decrescente.

8.5.2 For a passo non unitario

In generale nel ciclo

for I=A:D:B
    S
endfor

l'indice viene incrementato di D ad ogni passo e l'iterazione termina quando I ha raggiunto o superato B. Ad esempio

```
somma=0;
for h=1:2:n
    somma=somma+h;
endfor
```

calcola la somma dei numeri dispari compresi tra 1 e *n*, infatti *h* assume i valori 1,3,5,...,. L'ultimo valore assunto da *h* è *n*, se *n* è dispari, se invece *n* è pari, il ciclo termina con *h=n-1*.

8.6 Esercizi

1. Scrivere una funzione che dati *n* e *k*, calcola il numero di disposizioni semplici di n elementi a *k* a *k*, che è pari a n (n-1) (n-2)...(n-k+1).
2. Scrivere una funzione che, dati *x* numero reale e *n* numero naturale, calcola x^n tramite moltiplicazioni successive.
3. Estendere la soluzione del precedente esercizio al caso in cui *n* è un numero relativo.
4. Scrivere una funzione che, dati *m,n,k* numeri naturali, calcola m^n modulo *k* tramite moltiplicazioni successive modulo *k*.
5. Scrivere una funzione che, dato un numero naturale *n*, calcola la somma dei divisori propri di *n*.
6. Scrivere una funzione che controlla se un numero naturale *n* è perfetto, ovvero è pari alla somma dei suoi divisori (incluso 1, ma escluso *n* stesso).
7. Scrivere una funzione che dato n calcola l'n-esimo numero di Fibonacci. I primi due numeri di Fibonacci sono 1, ogni altro si ottiene sommando i due numeri di Fibonacci precedenti. Perciò i primi numeri di Fibonacci sono 1,1,2,3,5,8,13,21,34,55.
8. Scrivere una funzione che, dati x e n, calcola *log(x+1)* mediante la formula di Taylor arrestata all'*n*-esimo termine.
9. Scrivere una funzione che, dati x e n, calcola e^x mediante la formula di Taylor arrestata all'n-esimo termine.
10. Scrivere una funzione che, dato n, stampa (usando disp) tutti i numeri primi compresi tra 2 e n.
11. Scrivere una funzione che, dati un numero naturale m ed un numero primo k, trova l'inverso moltiplicativo di m modulo k.
12. Scrivere una funzione che, dati a,b e n, calcola l'integrale approssimato di una funzione f, di una sola variabile x, nell'intervallo [a,b] dividendo l'intervallo in n sottointervalli, calcolando per ognuno di questi sottointervalli $[x_i,x_{i+1}]$ l'area del trapezio avente come basi $f(x_i)$ e $f(x_{i+1})$, ed infine sommando tali aree.

9 Strutture iterative non limitate

Come abbiamo visto, esistono due forme di iterazione, quella limitata, che è stata illustrata nel capitolo precedente, e quella non limitata, che sarà descritta in questo capitolo

9.1 Ciclo while

L'iterazione non limitata in Matlab può essere creata con le istruzioni while, che è presente in molti linguaggi di programmazione (Pascal, C, ecc.). In Octave è presente anche l'istruzione do-until che non verrà descritta in queste dispense.

Il ciclo while ha la sintassi

while(C)

    S

endwhile

in cui C è una condizione e S una sequenza di istruzioni. La parola chiave **endwhile** si può abbreviare in **end** .

La semantica “intuitiva” del ciclo while è la seguente

1) valuta la condizione C
2) se è falsa, il ciclo termina
3) (altrimenti) esegui S
4) torna al punto 1

Ad esempio per calcolare il massimo comun divisore mediante il metodo di Euclide si può usare un ciclo while nel seguente modo

```
function m=mcd(a,b)
     while(a!=b)
          if(a>b)
               a=a-b;
          else
               b=b-a;
          endif
     endwhile
     m=a;
endfunction
```

Un altro esempio concerne il calcolo della radice quadrata di un numero reale x non negativo tramite il metodo di Newton.

Si parte con $r_0=x$ e poi ad ogni passo si pone

$$r_{n+1}=\frac{1}{2}\left(r_n+\frac{x}{r_n}\right)$$

fino a che i due valori consecutivi r_n e r_{n+1} sono sufficientemente vicini, ad esempio quando $|r_n - r_{n+1}| < \varepsilon$.

Dato che il ciclo while continua quando la condizione è vera, è necessario usare condizione l'opposto di quella scritta in precedenza, ovvero $|r_n - r_{n+1}| \geq \varepsilon$.

Quindi avremo una funzione che dipende da x e da ε, abbreviato in eps.

```
function r=radice(x,eps)
     r=a;
     r_prec=0;
     while (abs(r-r_prec)>=eps)
          r_prec=r;
          r=0.5*(r+x/r);
     endwhile
endfunction
```

Come ulteriore esempio di ciclo while, vediamo un metodo per trovare uno zero di una funzione continua f:[a,b]→ℝ nell'ipotesi che f assuma agli estremi di $[a,b]$ valori di segno opposto. Per un noto teorema esiste almeno un elemento $z \in [a,b]$, detto zero di f, tale che $f(z)=0$.

Se *m* è il punto medio di [a,b] e f(a)<0 (e quindi f(b)>0), allora i casi sono tre

1. f(m)=0, allora m è uno zero di *f;*
2. f(m)>0, allora f ha almeno uno zero in $[a,m]$;
3. f(m)<0, allora f ha almeno uno zero in $[m,b]$.

In maniera analoga è il caso in cui f(a)>0. In sintesi basta controllare se f(a) e f(m) hanno lo stesso segno oppure no: un modo semplice è verificare se f(a) f(m)>0.

Si può quindi impostare un ciclo while che continua fino a che non si trova uno zero oppure l'intervallo in cui f ammette uno zero è sufficientemente piccolo, di modo che il suo punto medio è un'approssimazione accettabile di uno zero di f.

L'eventualità che l'algoritmo trovi proprio uno zero di f è molto remota anche perché a causa degli errori di arrotondamento è improbabile che pur essendo m il vero zero di f, accada che f(m) sia esattamente 0.

Perciò si usa il ciclo while con una condizione che controlla solo l'ampiezza dell'intervallo

```
function z=zero_f(a,b,eps)
     while(b-a>=eps)
          m=(a+b)/2;
          if(f(m)*f(a)<0)
               b=m;
          else
               a=m;
          endif
     endwhile
```

```
    z=(a+b)/2;
endfunction
```

9.2 Semantica del ciclo while

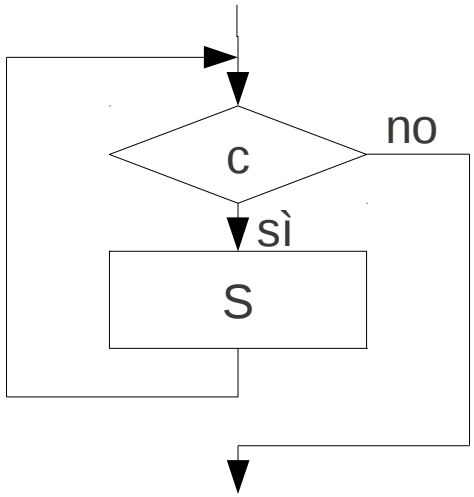
La semantica del ciclo

while(C)

S

endwhile

può essere spiegata anche mediante il seguente diagramma di flusso



Invece dal punto di vista formale, la semantica denotazionale di un ciclo while, di cui si è certi della terminazione, è $E(\text{while(C) S endwhile},\sigma)=\sigma_T$, ove

$\sigma_0=\sigma$

mentre per k=1,2,...

$\sigma_k=E(S, \sigma_{k-1})$

Invece T è il più piccolo indice per $V(C, \sigma_T)=0$, ovvero $V(C, \sigma_0)=V(C, \sigma_1)=...=V(C, \sigma_{T-1})=1$.

Se invece il ciclo while(C) S end non termina a partire dallo stato σ, ovvero quando per ogni k=1,2,... accade che $V(C, \sigma_k)=1$, allora $E(\text{while(C) S end},\sigma)$ non è definito.

9.3 Confronto tra while e for

Un ciclo for del tipo

for I=A:B

S

endfor

può essere sempre emulato con un ciclo while nel seguente modo

I=A;

while(I<=B)

S

I=I+1;

endwhile

Ad esempio il ciclo for che calcola la somma dei primi n numeri naturali

```
somma=0;
for i=1:n
     somma=somma+i;
endfor
```

si può riscrivere come

```
somma=0;
i=1;
while(i<=n)
     somma=somma+i;
     i=i+1;
endwhile
```

Pertanto possiamo dire che il ciclo for è un caso particolare del ciclo while e che i linguaggi che possiedono solo il ciclo while (come ad esempio il linguaggio C che apparentemente possiede anche un ciclo for, ma che in sostanza è a tutti gli effetti una variante del ciclo while) possono essere utilizzati senza problemi (anche se con qualche scomodità).

La traduzione da for a while risulta utile se vuole creare una variante del ciclo for con la possibilità di uscire dal ciclo anticipatamente.

Nell'esempio di funzione che controlla se un numero *n* è primo, descritto nel capitolo precedente, si può notare che una volta che *p* è diventato false, è inutile andare avanti nel ciclo for, perché il risultato della funzione è sicuramente false. Per sfruttare questa proprietà innanzitutto sostituiamo il ciclo for con un ciclo while

```
function p=primo(n)
     p=true;
     r=ceil(sqrt(n);
     d=2;
     while(d<=r)
          if(rem(n,d)==0)
               p=false;
          endif
          d=d+1;
     endwhile
```

```
endfunction
```

e poi facciamo sì che il ciclo termini anche quando *p* diventa false. Poiché la condizione del ciclo while indica invece quando il ciclo deve continuare, bisogna aggiungere (mediante la congiunzione AND) che *p* deve essere true, ottenendo una versione più efficiente:

```
function p=primo(n)
     p=true;
     r=floor(sqrt(n);
     d=2;
     while(d<=r && p==true)
          if(rem(n,d)==0)
               p=false;
          endif
          d=d+1;
     endwhile
endfunction
```

Perciò il ciclo while termina quando la sua condizione è falsa, ovvero quando

1. d>r, oppure
2. p!=true (ossia p==false).

Nel primo caso, *n* è primo, nel secondo caso, *n* non è primo.

Una considerazione finale riguarda la relazione inversa tra while e for: un ciclo while si può esprimere come ciclo for ? La risposta è in generale negativa, infatti la condizione di un ciclo while potrebbe dipendere da dati che mano a mano arrivano al computer ma che non sono disponibili all'inizio del ciclo, per cui in tali situazioni è materialmente impossibile determinare a priori il numero di iterazioni necessarie. Ma anche nel caso in cui i dati fossero già tutti disponibili, esistono esempi di funzioni che non si possono calcolare con cicli for, ma che necessitano di cicli while. Un esempio di funzione con questa caratteristica è la funzione di Ackermann (in termini tecnici si dice che è *ricorsiva generale*, ma non *ricorsiva primitiva*). Il lettore interessato può trovare la definizione su qualsiasi testo di teoria della calcolabilità.

L'uso dei cicli while però solleva il problema della terminazione: è possibile scrivere cicli while che non terminano mai quando la condizione non diventa mai falsa. Un uso corretto dell'istruzione while impone che il programmatore si accerti, ad esempio con una dimostrazione matematica o mediante un'argomentazione convincente, che prima o poi la condizione diventa falsa.

Si noti che un programma che contiene un ciclo while è quindi potenzialmente non terminante, mentre l'uso dei cicli for garantisce la terminazione in qualsiasi situazione. Perciò da un lato sarebbe auspicabile non usare i cicli while, ma d'altro canto alcune situazioni necessitano proprio di tale tipo di istruzione.

Allora la conclusione a cui si giunge è la seguente: un linguaggio di programmazione può essere universale (cioè permettere il calcolo di tutte le funzioni calcolabili) se e solo se consente di scrivere programmi non terminanti.

9.4 Esercizi (da risolvere con il ciclo while)

1. Scrivere una funzione che, dato n, trova il più piccolo divisore di n.
2. Scrivere una funzione che, dati un numero naturale m ed un numero primo k, trova l'inverso moltiplicativo di m modulo k.
3. Scrivere una funzione che, dati m è un numero naturale e k è un numero primo, trova una radice quadrata di m modulo k, o “no” se m non è un quadrato modulo k.
4. Scrivere una funzione che, dato n, trova il più piccolo numero primo maggiore o uguale a n. Suggerimento: trovare un numero primo tra n, n+1, n+2, …
5. Scrivere una funzione che, dato n, scrive sullo schermo (con disp) i primi n numeri primi. Ad esempio con n=7 la funzione deve scrivere 2,3,5,7,11,13,17.
6. Scrivere una funzione che, dati due numeri naturali *a* e *b*, implementa in modo più veloce l'algoritmo di Euclide per il calcolo del massimo comun divisore. L'idea è che se *a* è (molto) più grande di *b*, anziché sottrarre *b* da *a*, conviene calcolare il resto della divisione di *a* per *b* e sostituirlo al posto di *a*. A questo punto *a* è sicuramente minore di *b* (perché ?) e quindi...
7. Scrivere una funzione che, dato un numero naturale n, trova la somma delle cifre decimali di n. Suggerimento: partendo ad esempio da n=12345, si noti che rem(n,10)=5 mentre fix(n/10) è 1234...
8. Scrivere una funzione che, dati un numero naturale *m* ed un numero primo *k*, trova l'ordine di *m* modulo *k*, ovvero il più piccolo numero *n* tale che m^n modulo *k* è 1.

10 Tipi di dati strutturati

Matlab, al pari dei comuni linguaggi di programmazione e di software per il calcolo scientifico, offre la possibilità di lavorare con array ed altri tipi di dati strutturati. Un tipo di dato si dice **strutturato** se è ottenuto mediante aggregazione di altri dati. La forma più semplice di aggregazione è l'**array**, in cui si aggregano dati dello stesso tipo.

Un array costituisce una generalizzazione del concetto matematico di vettore. Innanzitutto gli elementi di un array non sono necessariamente numeri, ma è possibile avere array di caratteri, di stringhe o altri tipi. Inoltre un array può essere disposto su più dimensioni e quindi essere indiciato con più indici.

In queste dispense useremo solo array ad una ed a due dimensioni con elementi numerici, che corrispondono ai concetti matematici di vettore e matrice.

Un'altra forma di aggregazione usata nella programmazione è quella in cui i dati aggregati sono di tipi diversi. Tale forma si chiama **record** (o **struttura**) e non verrà descritta in queste dispense.

Tra le più diffuse altre forme di aggregazioni vi sono le liste e gli insiemi. Si tratta di tipologie di aggregazione omogenea, ma con modalità diverse da quelle dei vettori. In particolare le liste sono strutture in cui è possibile aggiungere e togliere elementi in maniera veloce, mentre nei vettori tali operazioni sono veloci solo se svolte nella posizione finale. Gli insiemi si differenziano dalle liste perché in queste ultime l'ordine (e la posizione) in cui gli elementi sono stati inseriti all'interno dell'aggregazione conta, mentre nei primi no. Inoltre gli insiemi sono organizzati in modo da rendere veloce la ricerca di un elemento (appartenenza). Anche queste strutture dati non saranno illustrate in queste dispense.

10.1 Array unidimensionali

Chiameremo vettori gli array ad una dimensione.

Un vettore in Matlab può essere utilizzato come vettore riga o vettore colonna. Gli elementi di un vettore colonna sono indicati tra parentesi quadre e sono separati da punti e virgola. Ad esempio

```
[1;2;3]
```

definisce il vettore colonna

1

2

3

In un vettore riga gli elementi sono separati da virgole o da spazi. Ad esempio con

```
[1,2,3,4]
```

si crea il vettore riga

1 2 3 4

Il numero di elementi di un vettore si ottiene mediante la funzione *length*. Ad esempio l*ength*([*1,2,3*]) dà come risultato 3, mentre *length*([*1;4;5;8*]) dà come risultato 4.

E' possibile assegnare un vettore ad una variabile mediante l'usuale istruzione di assegnamento. Ad esempio

```
u=[4;9;7]
```

o

```
v=[10,11,12,15]
```

In Matlab sono supportate molte operazioni classiche dell'algebra lineare. Ad esempio la somma e la differenza di vettori della stessa dimensione e della stessa posizione (per riga o per colonna).

Quindi definendo

```
w=[-1,5,10,-4]
```

v+w calcola la somma dei due vettori riga, ottenendo come risultato il vettore riga, ottenendo come risultato il vettore riga

9 16 22 11

mentre *v-w* calcola la differenza tra vettori, ottenendo come risultato il vettore riga

11 6 2 19

L'operatore postfisso ' (apice) calcola il trasposto e, per i vettori, cambia il vettore da riga a colonna o viceversa. Per cui *u'* vale

4 9 7

mentre *w'* vale

-1

5

10

-4

L'operatore * applicato ad array denota il classico prodotto righe per colonne. Nel caso dei vettori, *v * z* denota il prodotto scalare: *v* deve essere un vettore riga e *z* un vettore colonna della stessa dimensione di *v*. Se z=[1;0;2;-1] allora *v*z* vale 10+0+24-15=19.

La moltiplicazione è possibile anche tra uno scalare e un vettore, quindi sia 3*v che v*3 producono il vettore riga 30 33 36 45.

Sono ammesse anche molte operazioni sui vettori che non hanno equivalente nell'algebra lineare.

E' possibile svolgere un'operazione binaria tra ogni elemento di un vettore ed il corrispondente elemento di un secondo vettore, creando come risultato un terzo vettore. L'operazione deve essere preceduta dal punto (che fa ricordare che l'operazione è svolta “puntualmente”). Ad esempio *v .* w* calcola il prodotto componente per componente ottenendo come risultato il vettore riga -10 55 120 -60.

La funzione `sum` calcola la somma di tutti gli elementi di un vettore, mentre la funzione `prod` calcola il prodotto di un vettore, infine `max` e `min` determinano, rispettivamente, il più grande e il più piccolo elemento di un vettore. Ad esempio sum(w) fa 10, prod(w) fa 200, max(w) fa 10 e min(w) fa -4.

L'accesso alla *i*-esima componente di un vettore *v*, che in matematica si indica con v_i, in Matlab si indica con la notazione *v(i)*. Ovviamente tale notazione ha senso in un'espressione solo se *i* è un numero intero compreso tra *1* e *n*.

Ad esempio *w(2)* vale 5.

Molte funzioni scalari di Matlab sono vettorizzate, cioè possono essere applicate a vettori e producono come risultato un vettore, i cui elementi sono ottenuti applicando la funzione elemento per elemento. Ad esempio *exp(w)* dà il vettore

3.6788e-01 1.4841e+02 2.2026e+04 1.8316e-02

ovvero il vettore (*exp(-1)*, *exp(5)*, *exp(10)*, *exp(-4)*).

Le principali funzioni di Matlab (exp, log, sqrt, sin, cos, ecc.) sono vettorizzate.

E' possibile creare vettori particolari mediante opportune operazioni

- *ones(1,n)* crea un vettore riga di n elementi tutti pari a 1
- *ones(n,1)* crea un vettore colonna di n elementi tutti pari a 1
- *zeros(1,n)* crea un vettore riga di n elementi tutti pari a 0
- *zeros(n,1)* crea un vettore colonna di n elementi tutti pari a 0
- *a:b*, in cui *a* e *b* sono numeri interi con *a*<*b*, crea un vettore riga di tutti i numeri interi compresi tra *a* e *b*. Ad esempio 5:11 crea il vettore 5 6 7 8 9 10 11. Funziona anche se *a* e *b* sono numeri reali (in tal caso è come se fosse a:1:b)
- *a:b:c*, in cui *a*, *b* e *c* sono numeri reali, crea un vettore riga che contiene, in ordine, tutti i numeri reali a partire da *a* e con incremento *c* fino ad arrivare a *b*, senza superarlo. Ad esempio 1:0.5:3.1 restituisce il vettore riga 1.0 1.5 2.0 2.5 3.0.

10.2 Uso del ciclo for con i vettori

L'uso dell'indice dei vettori e di un opportuno ciclo for permette di implementare una qualsiasi operazione sui vettori.

Ad esempio una funzione che calcola la somma di un vettore che è l'equivalente di `sum` è

```
function s=somma_vettore(v)
     n=length(v);
     s=0;
     for j=1:n
          s=s+v(j);
     endfor
endfunction
```

mentre una funzione equivalente a *max* è

```
function m=massimo_vettore(v)
     n=length(v);
     m=v(1);
     for j=2:n
          if(v(j)>m)
               m=v(j);
```

```
          endif
     endfor
endfunction
```

Una funzione può anche restituire un array come risultato. In tal caso si sfrutta la caratteristica di Matlab che se in un assegnamento che riguarda l'elemento *v(i)* di un vettore *v*, *i* è superiore alla dimensione di *v*, allora *v* viene automaticamente ridimensionato, aggiungendo tante componenti (inizializzate a zero) in modo che la nuova dimensione di *v* sia pari ad *i*.

In particolare se *i* supera di uno la dimensione di *v*, allora a *v* viene aggiunta la componente i-esima, mentre una variabile vettore inesistente viene trattata come se avesse dimensione 0.

Ad esempio se *v* è il vettore (1,2,3), con l'istruzione *v(5)=7* si ottiene il vettore (1,2,3,0,7).

Ecco a titolo di esempio una funzione che calcola la somma di due vettori della stessa dimensione.

```
function w=somma_vettori(u,v)
     n=length(v);
     for j=1:n
          w(j)=u(j)+v(j);
     endfor
endfunction
```

In pratica tale funzione costruisce un vettore *w* aggiungendo una componente alla volta. Dato che tale operazione è computazionalmente pesante, conviene costruire un vettore *w* di *n* elementi, inizialmente zeri, e poi mano a mano riempirlo.

```
function w=somma_vettori(u,v)
     n=length(v);
     w=zeros(n,1);
     for j=1:n
          w(j)=u(j)+v(j);
     endfor
endfunction
```

I vettori possono essere usati anche per rappresentare insiemi di numeri. Ad esempio una funzione che implementa il controllo di appartenenza di un numero *x* ad un vettore-insieme *v* è la seguente

```
function a=appartiene(x,v)
     a=false;
     n=length(v);
     j=1;
     while (j<1 && a==false)
          if(v(j)==x)
```

```
            a=true;
        endif
        j=j+1;
    endwhile
endfunction
```

Quando si usano i vettori con i cicli for è però importante vedere se è possibile risolvere lo stesso problema senza i cicli for. Infatti una delle caratteristiche più interessanti di Matlab è quella che spesso si riesce a svolgere la stessa operazione sfruttando al massimo le funzioni vettoriali.

Ad esempio la somma dei quadrati da 1 a n si può calcolare anche con

```
sum((1:n).^2)
```

dato che `1:n` crea il vettore di tutti i numeri da 1 a *n*, "`.^2`" eleva ogni elemento al quadrato ed infine `sum` ne calcola la somma.

10.3 Array bidimensionali

Chiameremo matrici gli array a due dimensioni. Una matrice in Matlab si può introdurre mediante la notazione

[a_{11}, a_{12}, …, a_{1n}; a_{21}, a_{22}, …, a_{2n}; …; a_{m1}, a_{m2}, …, a_{mn}]

in cui le righe sono separate da punti e virgole e gli elementi di ogni riga sono separati da virgole (o spazi).

Ad esempio

```
[ 1, 3, 7; 5, 9, 11; -1, 7, 3; 4, -8, 0 ]
```

produce la matrice 4×3

1 3 7

5 9 11

-1 7 3

4 -8 0

Le funzioni *rows* e *columns* restituiscono, rispettivamente, il numero di righe e il numero di colonne di una matrice. Ad esempio se

```
a=[ 1, 3, 7; 5, 9, 11; -1, 7, 3; 4, -8, 0 ]
```

allora *rows(a)* fa 4 e *columns(a)* fa 3.

L'operatore ' si può applicare anche a matrici, ad esempio a' produce la matrice trasposta

1 5 -1 4

3 9 7 -8

7 11 3 0

L'operatore * calcola l'usuale prodotto righe per colonne. Ad esempio se

`b=[1,2;5,-4;3,-2]`

allora a*b produce la matrice

37 -24
83 -48
43 -36
-36 40

Ovviamente il prodotto * si può usare anche tra uno scalare e una matrice, ad esempio 3*a produce

3 9 21
15 27 33
-3 21 9
12 -24 0

e tra una matrice ed un vettore colonna, ad esempio `a*[1;2;-1]` produce

0
12
10
-12

La funzione det calcola il determinante di una matrice quadrata. Ad esempio se

`C=[4,5,-1;7,8,11;3,-2,-1]`

det(C) produce 294

La funzione **inv** calcola l'inversa di una matrice quadrata invertibile (ovvero che abbia determinante non nullo). Ad esempio inv(C) produce la matrice

0.0476190 0.0238095 0.2142857
0.1360544 -0.0034014 -0.1734694
-0.1292517 0.0782313 -0.0102041

L'operatore barra inversa "\" di Matlab è molto famoso e serve, tra l'altro, a risolvere i sistemi lineari. Se *u* è un vettore colonna di dimensione *m* e C una matrice *m*×*n*, allora allora C\u è la soluzione dell'equazione matriciale C*x*=*u*, in cui *x* è un vettore incognito di dimensione n.

Ad esempio se u è il vettore [23,26,6], allora C\u è la soluzione del sistema

$$4x+5y-z=23$$
$$7x+8y+11z=26$$
$$3x-2y-z=6$$

ovvero il vettore

3
2
-1

Alcune matrici speciali sono

- *zeros*(*m*,*n*), crea una matrice m×n di zeri
- *ones*(*m*,*n*), crea una matrice m×n di uni
- *eye*(*n*), crea la matrice identità di ordine n.

L'elemento generico a_{ij} di una matrice a si può ottenere in Matlab con la notazione a(i,j).

Ad esempio a(3,2) è 7.

Data una matrice è possibile estrarre sottomatrici in vario modo. Ad esempio

- a(1,:) è la prima riga di a: *1 3 7*
- a(:,1) è la prima colonna di a:
  1
  5
  -1
  4
- a(1:2,:) è formata dalle prime due righe
  1 3 7
  5 9 11
- a(1:2,2:3) è la sottomatrice formata dalle prime due righe e dalle colonne 2,3
  3 7
  9 11

Due matrici o due vettori sono possono "concatenare", sia per riga che per colonna, solo però quando il risultato è una matrice rettangolare o un vettore.

Ad esempio se u=[1,2] e v=[3,4], allora la notazione [u,v] produce il vettore

1 2 3 4

mentre [u;v] produce la matrice

1 2

3 4

10.4 Uso del ciclo for con le matrici

Analogamente a quanto visto per i vettori, è possibile usare i cicli for anche per trattare con le matrici. Facciamo vedere ora alcuni esempi di tale utilizzo.

Una funzione che calcola la somma dei quadrati degli elementi di una matrice è

```
function s=somma_quadrati_matrice(a)
    nr=rows(a);
    nc=columns(a);
    s=0;
    for h=1:nr
```

```
          for k=1:nc
               s=s+a(h,k)^2;
          endfor
     endfor
endfunction
```

Una funzione che calcola la somma di due matrici, nell'ipotesi in cui abbiano lo stesso numero di righe e di colonne è

```
function c=somma_matrici(a,b)
     nr=rows(a);
     nc=columns(a);
     for h=1:nr
          for k=1:nc
               c(h,k)=a(h,k)+b(h,k);
          endfor
     endfor
endfunction
```

Infine una funzione che implementa il prodotto righe per colonne tra due matrici, nell'ipotesi che tale prodotto abbia senso (il numero delle colonne della prima matrice deve essere uguale al numero di righe della seconda) è

```
function c=prodotto_matrici(a,b)
     nr=rows(a);
     nc=columns(b);
     nn=columns(a);
     % nn deve essere uguale anche a rows(b)
     for h=1:nr
          for k=1:nc
               s=0;
               for r=1:nn
                    s=s+a(h,r)+b(r,k);
               endfor
               c(h,k)=s;
          endfor
     endfor
endfunction
```

Infatti tale funzione è basata sulla formula

$$c_{hk} = \sum_{r=1}^{nn} a_{hr} b_{rk}$$

per *h=1,...,nr* e *k=1,...,nc*.

Anche in questo caso però si invita a sfruttare al massimo le potenzialità di Matlab nel trattamento di vettori e matrici, evitando, quando ciò è possibile, di usare i cicli for.

10.5 Esercizi (da svolgere con cicli for e while)

1. Scrivere una funzione che dato un vettore x e un numero intero m, minore o uguale alla dimensione di x, calcola la somma degli elementi dal m-esimo elemento in poi.
2. Scrivere una funzione che dato un vettore x, trova il più grande elemento tra quelli di posizione dispari.
3. Scrivere una funzione che calcola la norma in L_p di un vettore x, ovvero $\sqrt[p]{\frac{\sum_{i=1}^{n} |x_i|^p}{n}}$
4. Scrivere una funzione che concatena due vettori riga x e y.
5. Scrivere una funzione che dato due vettori x e y di ugual lunghezza, crea un vettore che prende gli elementi da x e da y in modo alternato, ad esempio da [1,2,3] e [4,5,6] crea [1,4,2,5,3,6].
6. Scrivere una funzione che dato un vettore x, restituisce come risultato un vettore identico ad x, senza però l'ultimo elemento.
7. Scrivere una funzione che dato un vettore x, restituisce il vettore con gli elementi posizionati nell'ordine opposto: ad esempio da [1,2,3,4] restituire [4,3,2,1].
8. Scrivere una funzione che dato un vettore x, restituisce come risultato un vettore contenente solo gli elementi di x che sono pari.
9. Scrivere una funzione, che dati due vettori x e y, rappresentanti due insiemi, calcola l'intersezione, ovvero gli elementi in comune. Suggerimento: sfruttare la funzione appartiene e traendo spunto dalla soluzione dell'esercizio precedente, dato che $A \cap B = \{x \in A : x \in B\}$
10. Nelle ipotesi dell'esercizio 9, calcolare l'unione di A e B.
11. Nelle ipotesi dell'esercizio 9, calcolare la differenza simmetrica.
12. Scrivere una funzione che dato un vettore contenente i coefficienti di un polinomio p (ad esempio [1,2,3] rappresenta il polinomio $p(x)=1+2x+3x^2$) e un numero reale t, calcola p(t).
13. Nelle ipotesi dell'esercizio precedente, calcolare p'(t).
14. Scrivere una funzione che controlla se un dato vettore è palindromo, ovvero è uguale a se stesso anche rovesciando l'ordine dei suoi elementi. Ad esempio [1,2,3,2,1] e [1,2,3,3,2,1] sono palindromi.
15. Scrivere una funzione che calcola il determinante di una matrice 3x3.
16. Scrivere una funzione che dato n crea la matrice identità di dimensione n
17. Scrivere una funzione che data una matrice quadrata calcola la traccia, ovvero la somma degli elementi sulla diagonale principale.
18. Scrivere una funzione che data una matrice A, trova l'elemento più grande di A in valore assoluto

19. Scrivere una funzione che data una matrice A, controlla se A è triangolare superiore, ovvero che ogni elemento al di sotto della diagonale principale è nullo.
20. Scrivere una funzione che data una matrice A restituisce come risultato il vettore contenente la somma degli elementi di ciascuna colonna.
21. Scrivere una funzione che data una matrice A, restituisce il numero della riga con il maggior numero di zeri (o -1, se ogni riga non ha zeri).

11 Funzioni

In questo capitolo vedremo in dettaglio la sintassi, la semantica (intuitiva) e le principali caratteristiche delle funzioni definite dall'utente, fornendo alla fine una serie di motivazioni per il loro impiego.

Il termine *funzione definita dall'utente* viene utilizzato in questo capitolo in opposizione al concetto di *funzione predefinita*, come ad esempio *cos, sin* o *log*, cioè funzioni già presenti nel linguaggio Matlab.

11.1 Sintassi

La gestione di una funzione distingue due momenti differenti: la **definizione** e la **chiamata**. Nella definizione si crea una nuova funzione indicando (in qualche modo) dominio, codominio e comportamento. Nella chiamata la funzione viene valutata su determinati argomenti. E' evidente che una funzione viene definita una sola volta, ma può essere chiamata tante volte, anche all'interno di altre funzioni.

Una funzione viene definita con la seguente sintassi

function *risultato=nomeFunzione*(*parametri*)

    istruzioni

endfunction

in cui

- *nomeFunzione* è il nome della funzione;
- *parametri* è un elenco di variabili, separate da virgole;
- *risultato* è una singola variabile o un elenco di variabili tra parentesi quadre e separate da virgola; è anche possibile definire funzioni senza risultato esplicito, omettendo completamente questa parte;
- **endfunction** si può abbreviare in **end**.

La chiamata di una funzione ha la sintassi

nomeFunzione(*argomenti*)

o, nel caso in cui questa fornisca più risultati, la sintassi

[*risultato1,...,risultatoN***]=***nomeFunzione*(*argomenti*)

in cui *argomenti* è un elenco di espressioni separate da virgole.

Una chiamata ad una funzione è valida solo quando il numero degli argomenti forniti è pari al numero dei parametri nella definizione della funzione. Infatti al momento della chiamata si stabilisce una corrispondenza biunivoca (temporanea) tra argomenti e parametri. Inoltre il corretto funzionamento presuppone che ogni argomento corrisponda ad un valore sensato per il corrispondente parametro. Ad esempio se una funzione ha un parametro di tipo matrice, allora in ogni chiamata l'argomento associato deve essere una matrice, altrimenti si genera una situazione di errore.

Avendo la seguente definizione di funzione

```
function p=potenza(x,n)
     p=1;
     for h=1:n
          p=p*x;
     endfor
endfunction
```

una chiamata corretta è

```
z=2;
potenza(3.1*z, 4)
```

in cui l'argomento 3.1*z, cioè 6.2, corrisponde al parametro x e l'argomento 4 corrisponde al parametro n.

Invece esempi di chiamate scorrette sono

```
potenza(z/2, 4, -17)
potenza(z)
```

La prima infatti ha troppi argomenti, mentre la seconda ne ha pochi.

Si noti che il parametro *x* può assumere qualsiasi valore scalare (anche complesso), mentre *n* deve essere un numero intero non negativo, altrimenti la funzione potenza non opera correttamente, quindi anche

```
potenza(z, 4.5)
```

non rappresenta una chiamata corretta.

11.2 Semantica

Supponiamo che una funzione *f* chiama una funzione *g* con argomenti $a_1,...,a_n$, ovvero con la chiamata $g(a_1,...,a_n)$, mentre siano $p_1,...,p_n$ i parametri di *g*, cioè è definita nel seguente modo

function r=g(p_1,...,p_n)

…

endfunction

Avvengono allora nell'ordine le seguenti azioni:

1. Gli argomenti della chiamata $a_1,...,a_n$ sono valutati e vengono memorizzati nei corrispondenti parametri di *g*: il valore di a_1 viene memorizzato nel primo parametro p_1, il valore di a_2 in p_2, e così via.
2. L'esecuzione di *f* è arrestata e in qualche modo Matlab si ricorda del punto in cui *f* è stata interrotta
3. Inizia l'esecuzione delle istruzioni di *g*.
4. L'esecuzione di *g* finisce.
5. Le variabili create in *g*, compresi i parametri $p_1,...,p_n$ e i risultati *r*, vengono eliminate dalla memoria.

6. L'esecuzione di *f* riprende dal punto in cui si era interrotta, usufruendo del risultato prodotto da *g*, cioè del valore che era contenuto nella variabile risultato *r*.

Si noti che la chiamata ad una funzione sospende momentaneamente l'esecuzione di *f*, ovvero Matlab non prevede l'esecuzione di più funzioni contemporaneamente. Del resto se *f* ha bisogno subito del risultato di *g*, non può far altro che aspettare che *g* sia finita.

11.3 File .m e funzioni

Una funzione può essere memorizzata in un file, il cui nome è uguale a quella della funzione a cui si aggiunge il suffisso (estensione) ".m" e che deve essere salvato in una directory particolare.

Quando una funzione *f* viene chiamata, Matlab controlla se ha già letto in precedenza il file associato e, in caso affermativo, se tale file non è stato nel frattempo alterato. Se ciò si verifica, Matlab usa la definizione di *f* che ha letto a suo tempo, altrimenti legge (o rilegge) il file ed acquisisce la (nuova) definizione di *f*. In pratica Matlab legge la definizione da file solo quando è necessario, evitando perciò inutili operazioni di lettura da disco, che potrebbero rallentare in maniera enorme l'esecuzione delle funzioni.

Un file con estensione ".m" si chiama in generale **script** e può contenere anche istruzioni qualsiasi, senza che siano inserite nella definizione di funzioni. Per eseguire uno script è sufficiente digitare il nome (senza suffisso ".m").

11.4 Parametri e variabili locali

Una funzione *f* può avere tre tipi di variabili locali

1. i parametri,
2. la o le variabili risultato,
3. le variabili interne.

Le variabili di quest'ultimo tipo sono tutte quelle variabili create, e quindi utilizzate, all'interno della funzioni. Di solito le variabili interne sono usate per conservare risultati intermedi.

E' importante capire che da un lato questi tre tipi di variabili sono trattate allo stesso modo:

1. sono utilizzabili solo all'interno della funzione di appartenenza;
2. tali variabili iniziano ad esistere in memoria solo quando la funzione viene chiamata;
3. quando la funzione termina, spariscono dalla memoria.

La prima caratteristica fa sì che una variabile interna a *f* resta confinata solo dentro *f* e nessuna funzione, anche quelle che vengono chiamate da *f*, vi può accedere.

La seconda e terza caratteristica lega la vita delle variabili interne alla funzione stessa: le variabili esistono in memoria solo e quando *f* è in esecuzione o è sospesa durante una chiamata che *f* effettua verso altre funzioni. Ciò comporta un'occupazione di memoria ottimale, nel senso che sono presenti in memoria solo le variabili che realmente servono a Matlab.

D'altro canto parametri, risultati e variabili interne sono completamente diverse se si guarda il loro utilizzo:

1. i parametri servono per ricevere i valori degli argomenti su cui calcolare il valore di *f*;
2. i risultati servono per indicare il valore di *f*;
3. le variabili interne hanno solo significato all'interno di *f*.

E' evidente che i parametri sono strettamente collegati al dominio di *f*, mentre i risultati lo sono al codominio. Invece le variabili interne non hanno alcun corrispettivo sulla definizione matematica di *f*.

11.5 Passaggio per valore

Matlab implementa solo il passaggio dei parametri **per valore**: se una funzione modifica un parametro non si hanno ripercussioni sul corrispondente argomento (ovviamente se si tratta di una variabile).

Ad esempio nella funzione che calcola il massimo comun divisore con l'algoritmo di Euclide, che qui riportiamo per comodità del lettore

```
function m=mcd(a,b)
     while(a!=b)
          if(a>b)
               a=a-b;
          else
               b=b-a;
          endif
     endwhile
     m=a;
endfunction
```

una chiamata del tipo

```
x=42;
y=70;
z=mcd(x,y);
```

non altera il valore degli argomenti *x* o *y*, nonostante alla fine dell'esecuzione della funzione *mcd* i corrispondenti parametri *a* e *b* sono diventati entrambi 14.

Il meccanismo di passaggio si chiama appunto per valore, in quanto la corrispondenza tra argomento *a* e parametro *p* comporta solo il passaggio del valore di *a* in *p* con una sorta di assegnamento, ma non vi è alcun passaggio inverso da *p* ad *a*.

Molti linguaggi di programmazione (Pascal, C, ecc.) permettono, oltre al passaggio per valore, anche il passaggio **per riferimento** (detto anche **per variabile** o **per indirizzo**). Tale tipo di passaggio, che non è consentito in Matlab, permette di scrivere funzioni che hanno *effetti collaterali*, ovvero funzioni che sono in grado di alterare volutamente gli argomenti della chiamata. Si rimanda il lettore interessato alla lettura di manuali dei rispettivi linguaggi di programmazione.

11.6 Utilizzo delle funzioni

La creazione di funzioni è un aspetto importante nella programmazione perché consente di creare una sorta di linguaggio di programmazione esteso, come se fosse una versione personale di Matlab, con cui poter ragionare e programmare a livello superiore.

Un contributo notevole delle funzioni è quello di consentire la risoluzione modulare di un problema computazionale. In pratica ciò corrisponde a

1. dividere il problema di partenza P in alcuni sotto-problemi $P_1,...,P_k$
2. trovare una soluzione algoritmica per ciascuno di essi mediante delle funzioni $f_1,...,f_k$
3. creare una funzione f che raccoglie i risultati intermedi e calcola il risultato complessivo.

Questa suddivisione è importante perché facilita in maniera considerevole la scrittura del codice,

evitando di risolvere un problema difficile tutto insieme, ma spingendo a ragionare per passi successivi.

Si può anche adottare una tecnica top-down che consiste nel pensare che per risolvere P è sufficiente risolvere in qualche modo (da specificare successivamente) i sotto-problemi P_1,...,P_k, poi nel ottenere il risultato finale mettendo insieme i risultati intermedi.

Il passaggio successivo è quello di affrontare nello stesso modo i sotto-problemi P_1,...,P_k, cercando per ognuno di loro una suddivisione in sotto-sotto-problemi, e così via, fino ad arrivare a problemi che possono essere risolti in maniera elementare.

Si noti inoltre che, mentre è praticamente impossibile vedere se una parte di una funzione è scritta correttamente (ovvero produce i risultati desiderati), è possibile farlo con una funzione intera. Pertanto se un problema è decomposto in sotto-problemi, ognuno dei quali fa a capo ad una funzione distinta, è possibile controllare la correttezza di ognuna delle parti controllando se la corrispondente funzione lavora in modo corretto.

Una caratteristica positiva della creazione di funzioni legate a sotto-problemi è che può capitare che un sotto-problema faccia parte anche di un nuovo problema che si affronterà in seguito. In tal caso si può usare la funzione già definita in precedenza. Ad esempio una volta definita la funzione *appartiene* questa può essere usata sia nel caso di funzioni insiemistiche, sia nel caso in cui si vuole controllare la presenza di un certo elemento in un vettore che non è necessariamente visto come insieme.

Infine l'uso delle funzioni può produrre anche una cospicua riduzione del codice, se le stesse operazioni sono necessarie in più punti di un programma: è infatti possibile definire una funzione che svolge tali operazioni e richiamare la funzione tutte le volte che serve. Usare una funzione con tale scopo ha come vantaggio quello di non dover scrivere più volte le stesse istruzioni e, nel caso in cui sia necessario modificarle, di doverle correggere in tutti i punti in cui compaiono. Ma è evidente che non si ha alcun vantaggio nel tempo di esecuzione, anzi addirittura si ha un lieve aggravio, dovuto alle operazioni che Matlab deve svolgere quando una funzione viene richiamata.

12 Ricorsione

12.1 Definizioni ricorsive

12.1.1 Ricorsione in matematica

Una funzione *f* è definita in modo ricorsivo se *f* compare nella propria definizione.

Una definizione ricorsiva può avere **casi ricorsivi** e **casi base**. Il primo tipo di legge lega il valore di *f* su un determinato argomento al valore di *f* calcolata su altri argomenti. Nel secondo tipo di legge il valore di *f* su un determinato argomento è fornito in modo diretto, senza che far ricorso ad altri valori di *f*.

Una tale definizione è ben posta se esiste una sola funzione che la rispetta, ovvero se la definizione univocamente determina la funzione.

Ad esempio la seguente definizione ricorsiva

1. $f(n)=n\ f(n-1)$ se n>0
2. $f(n)=1$ se n=0

ha un solo caso ricorsivo e un solo caso base, ed è soddisfatta soltanto dalla funzione $f(n)=n!$ per ogni $n\in\mathbb{N}$ (se ci limitiamo solo a funzioni da $\mathbb{N}$ a $\mathbb{N}$). Infatti se *h* è una qualsiasi funzione che soddisfa le equazioni seguenti, allora si dimostra facilmente per induzione che $h(n)=n!$ per ogni $n\in\mathbb{N}$. Infatti

1. se n=0, h(0)=1=0!
2. se n>0, sia per ipotesi induttiva che h(n-1)=(n-1)!, allora h(n)=n h(n-1)=n (n-1)!=n!

Le definizioni ricorsive giocano un ruolo molto importante in varie branche della matematica. Ad esempio tramite gli assiomi di Peano, i quali hanno come unica operazione primitiva il successore di un numero naturale, è possibile definire in modo ricorsivo le operazioni di addizione, moltiplicazione ed elevamento a potenza su $\mathbb{N}$.

12.1.2 Ricorsione nella programmazione

Anche nella programmazione le definizioni ricorsive sono molto utili. Esistono infatti una serie di problemi computazionali che possono essere risolti in modo più semplice, anche se non necessariamente più efficiente, tramite la ricorsione.

La ricorsione è una tecnica che consente di definire funzioni (informatiche, come quelle di Matlab) in modo ricorsivo, ovvero che richiamano sé stesse.

La ricorsione è uno strumento potente che consente di trovare soluzioni eleganti e compatte e di risolvere problemi difficili da risolvere con le tecniche tradizionali. Inoltre consente l'utilizzo della dimostrazione per induzione, per provare la correttezza, e delle equazioni alle ricorrenze, per calcolare il costo computazionale.

C'è però da dire che la ricorsione ha un suo costo intrinseco, come vedremo in seguito.

12.2 Un esempio: il fattoriale

In questa sezione facciamo vedere uno dei classici esempi di funzione definita ricorsivamente: il fattoriale. In Matlab è

```
function f=fattoriale(n)
    if(n==0)
```

```
        f=1;
    else
        f=n*fattoriale(n-1);
    endif
endfunction
```

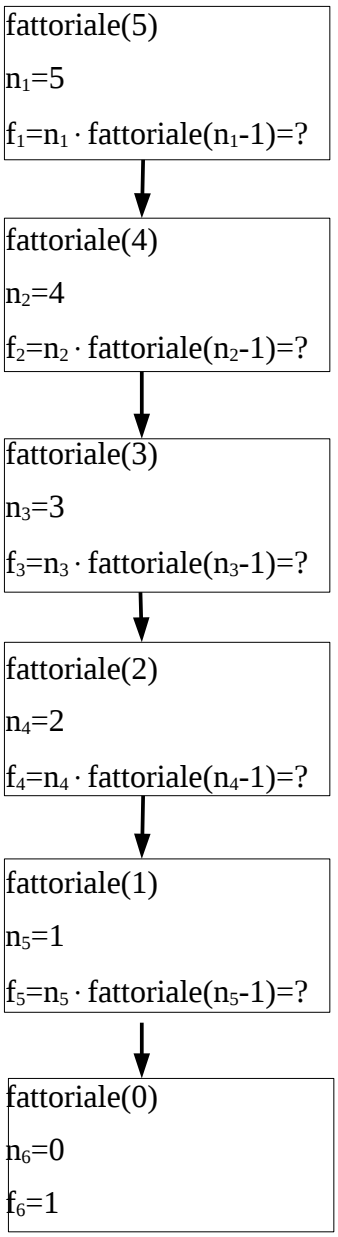
Tale definizione ricalca fedelmente la definizione matematica ricorsiva data nella precedente sezione.

Per capire il corretto funzionamento della ricorsione nella programmazione è indispensabile sapere che quando una funzione richiama sé stessa, si crea una nuova copia di tutte le variabili locali (parametri, risultati e variabili interne).

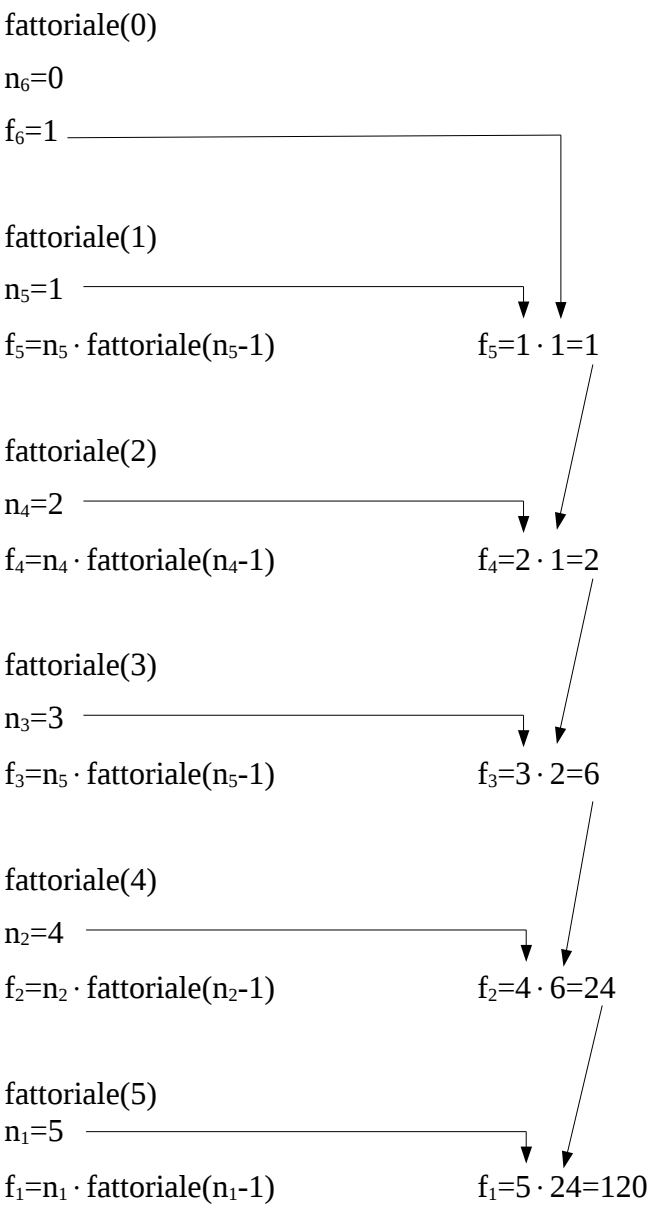
Ad esempio calcoliamo il fattoriale di 5.

La chiamata *fattoriale(5)* crea due variabili n e f, che indichiamo con n_1 e f_1, di cui la prima vale 5, mentre la seconda per ora non ha valore, dato che ha bisogno di calcolare fattoriale(4).

A sua volta la chiamata *fattoriale(4)* crea due nuove versioni delle variabili, indicate con n_2 e f_2 e poi chiama fattoriale(3). Andando avanti si produrrà il seguente schema



In quest'ultima chiamata a *f* viene assegnato subito 1 è ciò è il fattoriale di 0. A questo punto il risultato viene restituito a *fattoriale*(1) che ottiene il risultato $1 \cdot 1=1$, che viene restituito a *fattoriale*(2), che produce il risultato $2 \cdot 1=2$, e così via. Questi passaggi a ritroso possono essere seguiti in questo schema



Alla fine il risultato finale è 120, tutte le chiamate di fattoriale sono terminate e le variabili interne sono eliminate dalla memoria.

Si noti che è fondamentale che ogni chiamata ricorsiva abbia le proprie copie di *n* e *f*, infatti il prodotto tra *n* e il risultato della chiamata ricorsiva su *n-1*, deve essere calcolato usando il corrispondente valore di *n*. Se le variabili fossero sempre le stesse, *n* diventerebbe 0 in tutte le chiamate e la funzione restituirebbe sempre 0.

Per essere sicuri che la funzione calcola veramente il fattoriale di *n*, per ogni numero naturale *n* (entro i limiti imposti dall'aritmetica del computer) si può usare il principio di induzione.

Nel caso *n=0*, la funzione restituisce come risultato 1, che è uguale a 0!.

Dato un numero *n>0*, supponiamo per ipotesi induttiva che la funzione sia corretta su *n-1*, ovvero

che *fattoriale(n-1)* restituisca *(n-1)!*, allora *fattoriale(n)* restituisce come risultato il valore di *n*fattoriale(n-1)*, che per ipotesi induttiva è proprio uguale a *n (n-1)!=n!*.

12.3 Regole generali sulla ricorsione nella programmazione

Una definizione ricorsiva di una funzione in Matlab è necessariamente divisa in casi, tra i quali alcuni sono **regole ricorsive**, in cui la funzione richiama sé stessa, anche più volte, e gli altri sono **casi base**, in cui la funzione non richiama sé stessa, ma arriva al risultato tramite altre operazioni e chiamate ad altre funzioni.

Una funzione ricorsiva (pura) è organizzata tramite un'istruzione if-then-else o if-then-elseif-else, in cui si distinguono i casi base e le regole ricorsive. Ogni caso sarà contraddistinto da una condizione ed una serie di operazione da compiere.

Un fatto abbastanza evidente è che una funzione definita ricorsivamente può terminare, a partire da un determinato argomento, solo se, dopo un numero finito di passaggi tramite regole ricorsive, si arriva ad un caso base.

I casi base infatti hanno l'importante compito di “interrompere la catena di chiamate ricorsive”. La mancanza di uno di essi potrebbe infatti portare a funzioni non terminanti.

In realtà una funzione ricorsiva, poiché crea nuove copie delle variabili locali ad ogni chiamata, non può entrare in un ciclo infinito, ma dopo un numero finito di chiamate ricorsive successive riempe l'intera memoria RAM (o almeno la parte di memoria destinata al programma) e l'esecuzione in qualche modo si ferma.

Octave, in particolare, interrompe in maniera forzata la ricorsione dopo un numero finito di passaggi, producendo il messaggio di errore “*max_recursion_limit exceeded*”.

Vediamo ora come si programma in modo ricorsivo. Questo compito comporta una forma di ragionamento sostanzialmente differente dal modo in cui si approccia un problema computazionale in modo usuale.

E' difficile spiegare come si definisce una funzione ricorsiva partendo dalle caratteristiche del problema da risolvere. L'idea di base, comunque, è quella di analizzare il problema alla ricerca di regole ricorsive, in cui il valore della funzione *f* su un determinato argomento *x* si riesce a calcolare, in modo più o meno diretto, avendo a disposizione i valori di *f* su uno o più argomenti *y*, *y'*, *y''*, … distinti da *x*.

Bisogna trovare un numero sufficiente di regole ricorsive in modo da affrontare ogni possibile situazione. Infine è indispensabile definire un numero congruo di casi base che devono “chiudere” la ricorsione.

Per verificare che la definizione ricorsiva sia corretta è necessario innanzitutto provarne la terminazione e poi l'equivalenza funzionale con la funzione da calcolare, entrambe per induzione.

Spesso la ricorsione è svolta su un parametro $n \in \mathbb{N}$.

In tali situazioni i casi ricorsivi riconducono il valore di *f(n)* ai valori di *f(m')*, *f(m'')*,.. , per alcuni argomenti *m',m''*,... tutti strettamente minori di *n*. E' molto frequente il caso di *m=n-1*.

I casi basi allora corrispondono ai valori di *f* per n piccolo, ad esempio 0, 1,...

La presenza di ulteriori parametri in genere non crea problemi. Nelle prossime sezioni vedremo alcuni problemi di questo tipo risolti in maniera ricorsiva.

12.4 Esempi di ricorsione su $\mathbb{N}$

12.4.1 Elevamento a potenza, prima soluzione

Supponiamo di voler calcolare x^n con $x \in \mathbb{R}$ e $n \in \mathbb{N}$. Una regola ricorsiva molto semplice è

$$x^n = x \cdot x^{n-1}, \text{ per } n>0$$

ed il caso base è

$$x^0 = 1$$

E' facile vedere che questi casi producono una funzione terminante, dato che per qualsiasi $x \in \mathbb{R}$ e $n \in \mathbb{N}$, in *n* passaggi si arriva al caso base.

In Matlab tale funzione è

```
function p=potenza_ricorsiva(x,n)
     if (n==0)
          p=1;
     else
          p=x*potenza_ricorsiva(x,n-1);
     endif
endfunction
```

Si lascia al lettore la dimostrazione, per induzione, della correttezza della funzione.

12.4.2 Elevamento a potenza, seconda soluzione

Un secondo modo di calcolare x^n è quello di osservare che dimezzando l'esponente bisogna elevare al quadrato la base per ottenere lo stesso risultato. La regola ricorsiva è quindi

$$x^n = (x^2)^{\frac{n}{2}}$$

ovviamente tale legge vale per *n* pari e maggiore di 0.

Per *n* dispari, bisogna fare una piccola correzione

$$x^n = x(x^2)^{\frac{n-1}{2}}$$

Rimane escluso il caso *n=0*, che diventa il caso base come nell'esempio precedente.

In Matlab avremo

```
function p=potenza_ricorsiva_veloce(x,n)
     if (n==0)
          p=1;
     elseif (rem(n,2)==0)
          p=potenza_ricorsiva_veloce(x*x,n/2);
     else
```

```
        p=x*potenza_ricorsiva_veloce(x*x,(n-1)/2);
    endif
endfunction
```

Si noti che questa funzione è stata chiamata “potenza_ricorsiva_veloce” perché arriva al caso base in $log_2 n$ passaggi, anziché n. Per dimostrare la correttezza di tale funzione è necessario usare un principio di induzione più generale, ovvero usare come ipotesi induttiva che la funzione è corretta per tutti i valori minori di n (e non solo per $n-1$).

12.4.3 Numeri di Fibonacci

I numeri di Fibonacci sono già stati proposti come esercizio nel capitolo 8. Notiamo subito che la legge definitoria è ricorsiva in quanto, indicando con F_n l'n-esimo numero della successione, si ha la regola ricorsiva

$$F_n=F_{n-1}+F_{n-2} \text{ per } n>2$$

con i casi base $F_1=1$ e $F_2=1$.

Una funzione ricorsiva Matlab che dato $n>0$ calcola F_n è perciò

```
function f=fibonacci(n)
    if (n<2)
        f=1;
    else
        f=fibonacci(n-1)+fibonacci(n-2);
    endif
endfunction
```

La correttezza è intrinseca nel fatto che la funzione, anche matematicamente, è definita in modo ricorsivo.

Un aspetto interessante è che la funzione, nel caso ricorsivo, effettua due chiamate ricorsive. Questa situazione va sotto il nome di **ricorsione binaria**.

E' importante notare che questa definizione ricorsiva è comunque molto onerosa dal punto di vista computazionale. Infatti le chiamate ricorsive spesso sono costrette a calcolare più volte lo stesso valore. Ad esempio per calcolare *fibonacci*(*7*) serve calcolare *fibonacci*(*6*) e *fibonacci*(*5*). A sua volta *fibonacci*(*6*) ha di nuovo bisogno di sapere quanto vale *fibonacci*(*5*), ma non può far altro che ricalcolarlo. Questo fenomeno non è un'eccezione, anzi mano a mano che n dimiuusce, il valore di *fibonacci* è ricalcolato sempre un maggiore di volte. E' ovvio che ricalcolare la stessa quantità è inutile e in tale problema si ha un effetto disastroso: il numero di addizioni necessarie a calcolare *fibonacci(n)* cresce in modo esponenziale in n, mentre una soluzione con un ciclo for ne usa solo $n-2$.

12.4.4 Conteggio del numero delle cifre

Una funzione ricorsiva che conta di quante cifre decimale è composto un numero naturale n si può definire in base alla semplice constatazione che si divide n per 10, prendendone il quoziente, si elimina l'ultima cifra. Ad esempio se n=12783, allora $\lfloor n/10 \rfloor$ è 1278.

Da ciò nasce la regola ricorsiva, in cui *c* indica la funzione da calcolare,

$c(n)=1+c(\lfloor n/10 \rfloor)$, se $n>9$

La legge non vale per numeri inferiori a 10, ma questi hanno una sola cifra e pertanto costituiscono un ottimo caso base

$c(n)=1$, se $n<10$.

Da ciò si ottiene

```
function c=conta_cifre(n)
     if(n<10)
          c=1;
     else
          c=1+conta_cifre(fix(n/10));
     endif
endfunction
```

12.5 Ricorsione sugli array

E' possibile definire funzioni ricorsive anche sui vettori o su collezioni di dati. Un modo diretto, ma un po' inefficiente, è quello di usare leggi ricorsive che legano il valore della funzione *f* sull'intero vettore A al valore della funzione calcolata su vettori più piccoli, che si ottengono da A eliminando alcuni elementi.

Un modo decisamente meno oneroso è quello di trovare relazioni ricorsive che operano sui prefissi di un vettore, ovvero sugli elementi $[A_1,A_2,...,A_k]$, con $k \leq n$, ove n è la dimensione di A.

Infatti così operando il parametro vettore A rimane inalterato, mentre la ricorsione si svolge su k.

Un caso ricorsivo si ottiene legando il valore della funzione calcolata su $[A_1,A_2,...,A_k]$ al valore su $[A_1,A_2,...,A_{k-1}]$, mentre il caso base è normalmente k=1, ovvero un prefisso formato dal solo elemento A_1.

In maniera analoga si può operare ricorsivamente sui suffissi, ovvero porzioni del tipo $[A_k,A_{k+1},...,A_n]$, o su "intervalli" del tipo $[A_h,A_{h+1},...,A_k]$, con $1 \leq h \leq k \leq n$. Non faremo vedere esempi di queste tipologie di ricorsione.

12.5.1 Somma degli elementi di un vettore

Come primo esempio di ricorsione "a prefisso" su un vettore, definiamo una funzione ricorsiva che calcola la somma degli elementi di vettore.

Le legge ricorsiva è ovvia. Indicando con *s* la funzione "somma di un vettore" si ha che

$$s([A_1,A_2,...,A_k])=s([A_1,A_2,...,A_{k-1}])+A_k, \text{ con } k>1$$

Il caso base è ancora più semplice: $s([A_1])=A_1$.

In Matlab si ha

```
function s=somma_prefisso_vettore(a,k)
     if(k==1)
```

```
        s=a(1);
    else
        s=somma_prefisso_vettore(a,k-1)+a(k);
    endif
endfunction
```

Per calcolare la somma di un vettore *v* si usa la chiamata

```
somma_prefisso_vettore(v,length(v))
```

12.5.2 Massimo di un vettore

Se si vuole calcolare il massimo elemento di un vettore usando la tecnica della ricorsione "a prefisso" è necessario trovare una relazione ricorsiva che lega $M([A_1,A_2,...,A_k])$ a $M([A_1,A_2,...,A_{k-1}])$, ove M è la funzione da calcolare.

Esistono in realtà due leggi ricorsive

$M([A_1,A_2,...,A_k])=M([A_1,A_2,...,A_{k-1}])$ se $k>1$ e $M([A_1,A_2,...,A_{k-1}])>A_k$

e

$M([A_1,A_2,...,A_k])=A_k$ se $k>1$ e $M([A_1,A_2,...,A_{k-1}])\leq A_k$

In realtà le due leggi ricorsive indicano che $M([A_1,A_2,...,A_k])$ è il massimo tra A_k e $M([A_1,A_2,...,A_{k-1}])$. Il caso base è ancora $M([A_1])=A_1$.

In Matlab si ha

```
function m=massimo_prefisso_vettore(a,k)
    if(k==1)
        m=a(1);
    else
        m1=massimo_prefisso_vettore(a,k-1);
        if(a(k)>m1)
            m=a(k);
        else
            m=m1;
        endif
    endif
endfunction
```

La variabile *m1* serve a contenere il risultato della chiamata ricorsiva, in modo da evitare di ricalcolarne il valore quando lo si confronta con *a*(*k*) e tra questi si prende il più grande.

12.5.3 Ricerca di un elemento in un vettore

Vediamo ora una versione ricorsiva “a prefisso” della funzione “appartiene”, descritta nel capitolo 10, che controlla se un elemento *x* è presente in un vettore *v*. Questa funzione ha una regola ricorsiva banale

appartiene(x,[v_1,v_2,...,v_k])=appartiene(x,[v_1,v_2,...,v_{k-1}]) se k>1 e $v_k \neq x$

Il caso in cui $v_k=x$ è banalmente un caso base, dato che il risultato in questa situazione deve essere *true*. Inoltre è facile vedere che tale risultato vale anche quando *k=1*.

Allora possiamo estendere la regola ricorsiva anche quando *k=1*.

Resta quindi da capire qual è l'altro caso base, quello che dà come risultato *false*. E' chiaro che se la chiamata ricorsiva avviene anche quando *k=1* si ottiene un vettore vuoto. A tale situazione si perviene solo quando *x* non è presente in *v*, quindi l'altro caso base è proprio *k=0*, in cui il risultato è *false*.

Raccogliendo i vari casi, si ottiene la seguente definizione in Matlab

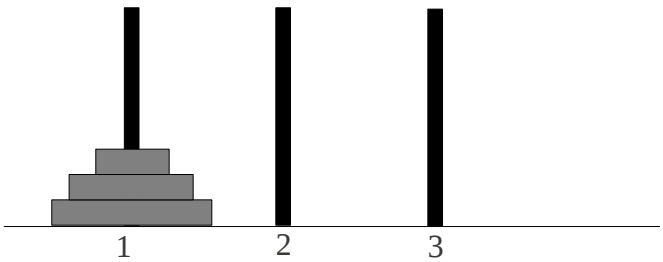
```
function a=appartiene(x,v,k)
     if(k==0)
          a=false;
     elseif(v(k)==x)
          a=true;
     else
          a=appartiene(x,v,k-1);
     endif
endfunction
```

12.6 Alcuni esempi notevoli di ricorsione

In questa sezione mostriamo due esempi notevoli di funzioni ricorsive.

12.6.1 Le torri di Hanoi

Le Torri di Hanoi sono un classico esempio di problema “naturalmente” ricorsivo. Supponiamo di avere tre pioli, numerati da 1 a 3, e che sul piolo 1 ci sono impilati *n* dischi di grandezza crescente. Lo scopo del gioco è spostare tutti i dischi sul piolo 3, potendo spostare un solo disco alla volta e non potendo mettere un disco sopra un disco più piccolo.



Vi è una soluzione ricorsiva molto semplice per spostare n dischi dal piolo 1 al piolo 3:

1. spostare ricorsivamente i primi n-1 dischi dal piolo 1 al piolo 2
2. spostare l'unico disco rimasto dal piolo 1 al piolo 3
3. spostare ricorsivamente i primi n-1 dischi dal piolo 2 al piolo 3

Il piolo 2 viene usato come piolo di appoggio ed infatti i primi n-1 dischi sono “parcheggiati” sul piolo 2, potendo così spostare il disco più grande dal 1 al 3.

La generalizzazione è facile, la funzione ha 4 parametri: i dischi da spostare, il piolo di partenza, quello di arrivo e quello di appoggio.

Il caso base è quando c'è un solo disco da spostare.

La funzione Matlab produce una matrice, le cui righe sono le mosse del gioco, rappresentate come coppie di numeri, il cui primo elemento indica il piolo da cui prendere il disco ed il secondo elemento il piolo su cui spostare il disco in questione.

```
function m=hanoi(nd,part,arr,app)
     if(nd==1)
          m=[part,arr];
     else
          m1=hanoi(nd-1,part,app,arr);
          m2=[part,arr];
          m3=hanoi(nd-1,app,arr,part);
          m=[m1;m2;m3];
     endif
endfunction
```

Ad esempio *hanoi(3,1,2,3)* produce la matrice

1 2
1 3
2 3
1 2
3 1
3 2
1 2

che si legge nel seguente modo

1. sposta il disco più in alto dal piolo 1 al piolo 2
2. sposta il disco più in alto dal piolo 1 al piolo 3
3. …

Si può dimostrare per induzione che la soluzione trovata dalla funzione ricorsiva *hanoi* è effettivamente una soluzione valida, perché non viola il vincolo sulle grandezze dei dischi da spostare.

12.6.2 Il calcolo del determinante

Come ultimo esempio, notiamo che la regola di calcolo del determinante conosciuta come regola di Laplace è di fatto ricorsiva. Infatti se A è una matrice di dimensione *n*, si sceglie una qualsiasi riga *r* di A (conviene prendere quella con il maggior numero di zeri) e per ogni elemento *x* di *r*, si calcola il determinante della sottomatrice, che si ottiene eliminando la riga *r* e la colonna in cui si trova *x*, e lo si moltiplica per *x*. La somma a segni alterni di tali prodotti dà il determinante di A.

Ad esempio, usando come riga *r* la prima

$$\begin{vmatrix} 1 & 2 & 3 & -5 \\ 7 & 8 & 4 & 3 \\ 11 & 0 & 3 & 2 \\ 8 & 1 & 7 & 4 \end{vmatrix} = 1\begin{vmatrix} 8 & 4 & 3 \\ 0 & 3 & 2 \\ 1 & 7 & 4 \end{vmatrix} - 2\begin{vmatrix} 7 & 4 & 3 \\ 11 & 3 & 2 \\ 8 & 7 & 4 \end{vmatrix} + 3\begin{vmatrix} 7 & 8 & 3 \\ 11 & 0 & 2 \\ 8 & 1 & 4 \end{vmatrix} - (-5)\begin{vmatrix} 7 & 8 & 4 \\ 11 & 0 & 3 \\ 8 & 1 & 7 \end{vmatrix}$$

Ogni determinante di ordine 3 andrà calcolato con lo stesso schema ricorsivo, e così via.

Un caso base semplice è quando *n=1*, in cui il determinante di A è l'unico elemento.

L'unico problema è come fare a togliere una riga *r* e una colonna *c* da A. Usando l'operatore di concatenazione e presupponendo che per semplicità r=1, la matrice ridotta è

[a[2:n,1:c-1],a[2:n,c+1:n]].

Infatti a[2:n,1:c-1] è la parte di matrice a sinistra della colonna c, mentre a[2:n,c+1:n] è quella a destra di c ed entrambe non hanno la prima riga. Non si hanno problemi nei casi in cui c=1 o c=n, ove una delle due parti è vuota.

Avremo perciò il seguente codice Matlab

```
function d=det_ric(a)
     n=rows(a);
     if(n==1)
          det=a(1,1);
     else
          det=0;
          segno=1;
          for c=1:n
               b=[a[2:n,1:c-1],a[2:n,c+1:n]];
               det=det+segno*a(1,c)*det_ric(b);
               segno=(-1)*segno;
          endfor
     endif
endfunction
```

Una considerazione finale è che questa funzione ricorsiva per calcolare il determinante non è assolutamente utilizzabile nella pratica. Con un po' di conti si può far vedere che il numero di operazioni necessarie per calcolare il determinante di una matrice di ordine *n* cresce come il

fattoriale di *n* (numero delle permutazioni dell'insieme {*1,2,...,n*}), e quindi è equivalente alla definizione del determinante come somma di *n!* prodotti di elementi di A.

Perciò nelle applicazioni non si usa la regola di Laplace, ma si preferisce, ad esempio, ridurre A a gradini e poi calcolare il prodotto degli elementi della diagonale principale.

12.7 Un confronto tra iterazione e ricorsione

Facciamo ora alcune considerazioni finali sull'uso della ricorsione.

E' abbastanza evidente che la ricorsione è alternativa all'iterazione, ovvero all'uso dei cicli. Da un lato è semplice dimostrare che ogni ciclo si può simulare con la ricorsione ed infatti quasi tutti gli esempi erano già stati risolti iterativamente. Il passaggio da ricorsione a iterazione è invece un po' più complicato e nei casi più difficili si deve simulare in qualche modo la presenza delle copie delle variabili locali (tramite una struttura chiamata *stack*).

Anche se sono equivalenti dal punto di vista delle funzioni che si riescono a calcolare, iterazione e ricorsione non sono però equivalenti sotto altri aspetti. Se da un lato la ricorsione ci aiuta a risolvere problemi in modo più semplice ed elegante (si pensi a quanto possa essere complicata una versione iterativa delle torri di Hanoi), ha comunque un costo di esecuzione maggiore, sia in termini di memoria, sia in termini di tempo. Infatti ogni chiamata ricorsiva consuma memoria e tempo per creare delle nuove variabili locali.

Quindi la ricorsione, almeno nei linguaggi di programmazione tradizionali o negli ambienti scientifici come Matlab, andrebbe usata solo quando realmente serve. Bisogna comunque notare che alcuni compilatori sono in grado di "eliminare la ricorsione" in molte situazioni e quindi sono in grado di produrre codice macchina efficiente anche se il programma originario è scritto in modo ricorsivo. Infine, molti linguaggi di programmazione funzionale (come Lisp, ML, Haskell) e logici (Prolog) richiedono l'uso della ricorsione, in modo esclusivo o preponderante, al posto dei costrutti iterativi.

12.8 Esercizi (da svolgere solo con la ricorsione)

1. Scrivere una funzione che dato un numero naturale *a*, calcola *2a* usando solo la funzione successore S(n)=n+1.
2. Scrivere una funzione che dati due numeri naturali a e b, calcola a+b usando solo la funzione successore S(n)=n+1.
3. Scrivere una funzione che dati due numeri naturali a e b, calcola a•b usando solo la funzione successore S(n)=n+1 e la funzione dell'esercizio precedente.
4. Scrivere una funzione che dati due numeri naturali *a* e *b*, calcola a^b usando solo la funzione successore S(n)=n+1 e la funzione dell'esercizio precedente.
5. Scrivere una funzione che calcola il massimo comun divisore di due numeri naturali *a* e *b*, tramite l'algoritmo di Euclide espresso in modo ricorsivo.
6. Scrivere una funzione che calcola la somma delle cifre decimali di un numero naturale.
7. Scrivere una funzione che calcola la media di un vettore.
8. Scrivere una funzione che controlla se tutti gli elementi di un vettore sono positivi.
9. Scrivere una funzione che dato un vettore, restituisce la posizione che contiene l'elemento più piccolo (se ce n'è più di uno, considerare quello più a sinistra).
10. Scrivere una funzione che dati due vettori *a* e *b* di uguale lunghezza, conta quante sono le

componenti omologhe di *a* e *b* (cioè quelle con lo stesso indice) che hanno lo stesso contenuto. Ad esempio dati [5,6,1,4,3] e [5,8,1,4,9] la funzione restituisce 3.

11. Scrivere una funzione che controlla se un vettore è palindromo, ovvero se il primo elemento è uguale all'ultimo, il secondo è uguale al penultimo, ecc. (suggerimento: usare la tecnica ad intervallo, con i due estremi del vettore)
12. Scrivere una funzione che dato un vettore *a* ed un numero intero *n*, restituisce un vettore *b* che contiene tutti gli elementi di *a* che sono divisibili per *n*.

13 Grafica bidimensionale

Matlab offre una serie di funzioni predefinite per disegnare punti e linee sullo schermo in grafica bi- e tridimensionale. In queste dispense non tratteremo gli aspetti di grafica tridimensionale, il lettore interessato può consultare il manuale del linguaggio.

13.1 Comando plot

Il comando di base per disegnare è **plot**. La sua forma più semplice è

plot(*x*,*y*)

in cui *x* e *y* sono vettori della stessa lunghezza, che indicheremo con *n*.

Il comando disegna in una nuova finestra i punti (x_1,y_1), (x_2,y_2), … (x_n,y_n) e collega ogni coppia di punti consecutivi (x_i,y_i), (x_{i+1},y_{i+1}) con un segmento di retta, ottenendo una spezzata.

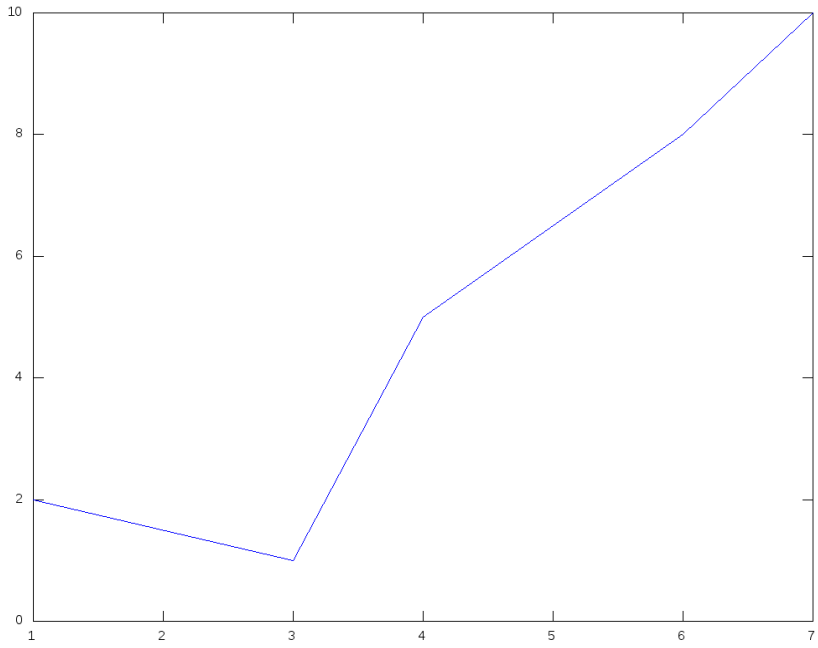
Ad esempio se

x=[1,3,4,6,7]

e

y=[2,1,5,8,10]

allora *plot(x,y)* produce il grafico

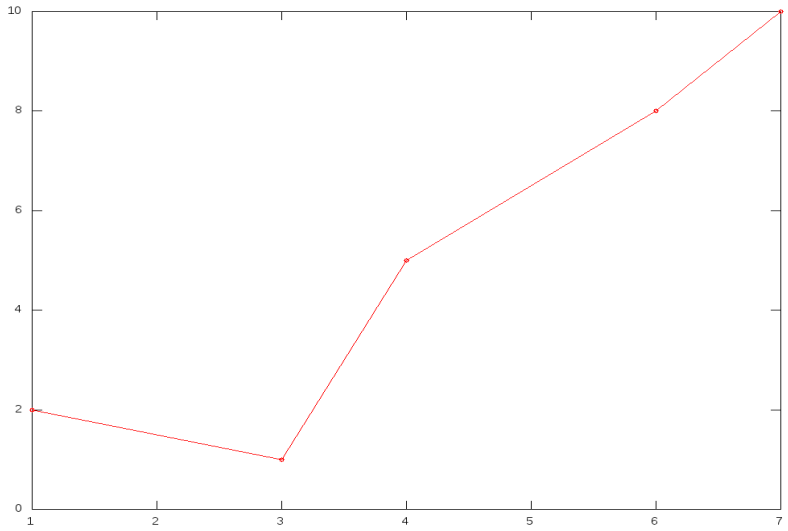


Tramite un terzo parametro, chiamato formato, è possibile scegliere

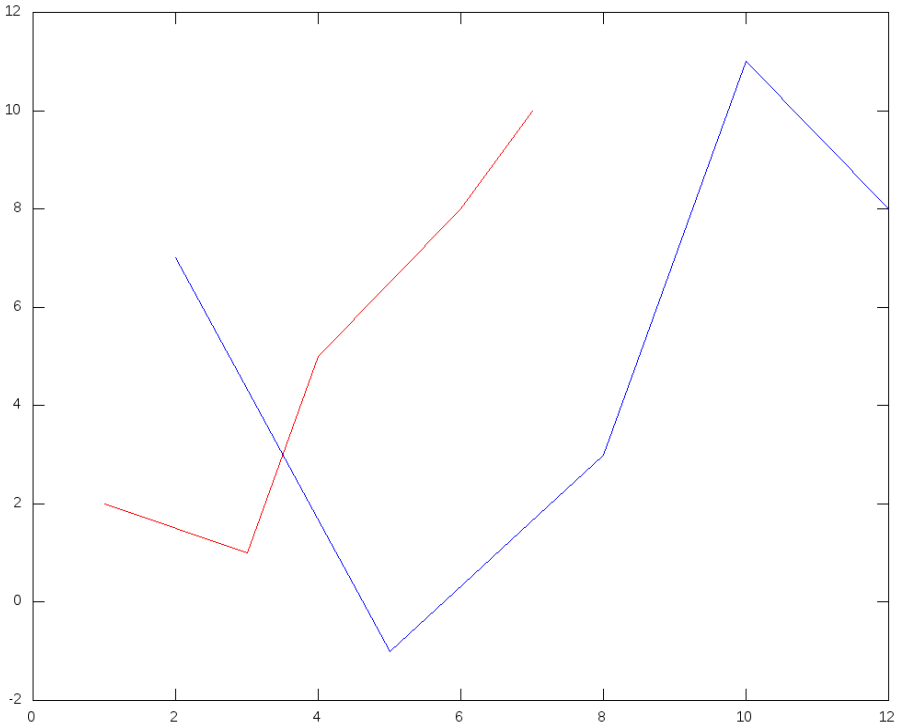
- se disegnare solo i punti o anche le linee,
- eventualmente quale forma di punti usare,
- quale colore usare.

Per disegnare solo i punti si usa “.”, “+”, “o”, “x” oppure “^”, a seconda della forma del punto che si desidera. I colori sono numerati da 1 a 6 o indicati con le lettere “k” (black), “r” (red), “g”

(green), “b” (blue), “m” (magenta) e “c” (cyan). Per disegnare anche le linee si usa “.-”,”x-”, ecc..
Ad esempio con plot(x,y,"o-r") si ottiene il grafico



con i punti tondi e le linee rosse.
Con plot è anche possibile disegnare più spezzate insieme nello stesso grafico.
Ad esempio se x1=[2,5,8,10,12] e y1=[7,-1,3,11,8], allora
plot(x,y,"r",x1,y1,"b")
produce il seguente grafico



in cui la prima spezzata è rossa e la seconda è blu.

13.2 Grafico di funzioni

Un'applicazione molto utile di plot è quella di disegnare il grafico (approssimativo) di una funzione di variabile reale.

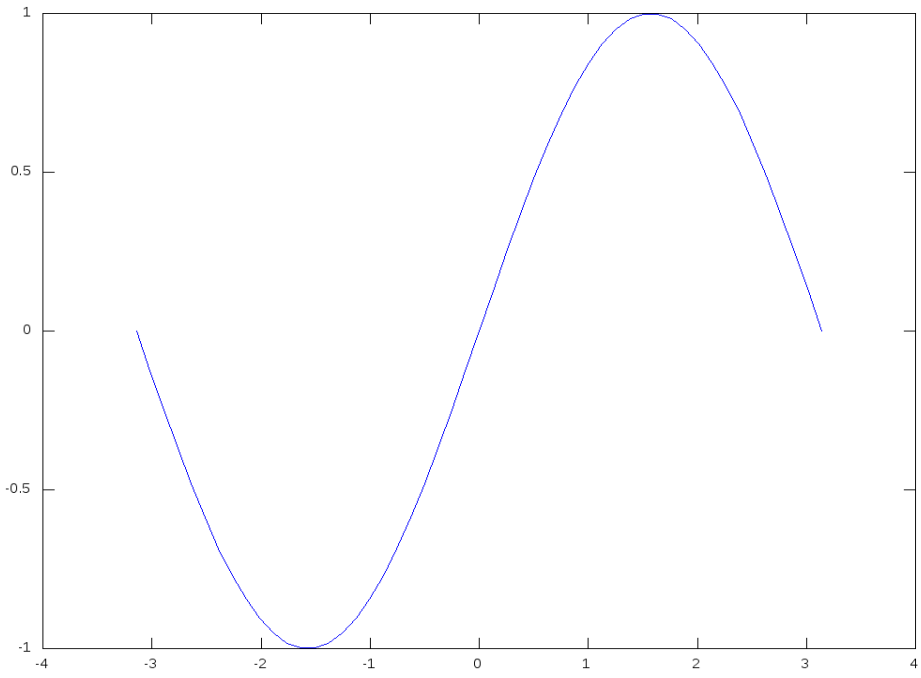
Sia f:[a,b]→ℝ una funzione di cui si vuole tracciare il grafico, allora prendiamo nell'intervallo [a,b] *n* punti $x_1,...,x_n$, per semplicità equispaziati, e per ognuno di essi calcoliamo $y_i=f(x_i)$. Otteniamo così due vettori *x* e *y* da utilizzare con plot.

Ad esempio otteniamo il grafico della funzione f(*x*)=*sen x* nell'intervallo [-π, π] con n=50 punti. La distanza tra un punto e l'altro si ottiene dividendo 2π per 49.

Perciò se diamo i seguenti comandi

```
h=2*pi/49;
x=-pi:h:pi;
y=sin(x);
plot(x,y)
```

si ottiene il seguente grafico



In questo primo esempio abbiamo sfruttato il fatto che in Matlab è possibile calcolare il seno di un vettore, componente per componente. Se la funzione di cui fare il grafico è più complicata oppure non esprimibile in forma analitica, allora il vettore *y* deve essere riempito tramite un ciclo for.

Facciamo ora vedere nello stesso grafico il confronto tra la funzione seno e il suo sviluppo con la formula di Taylor con *h=4* termini

$$g(x)=\sum_{k=0}^{h-1}\frac{(-1)^k x^{2k+1}}{(2k+1)!}$$

che calcoliamo mediante la funzione, in cui il numero di termini è *h*

```
function y=sin_approx(x,h)
```

```
     somma=0;
     termine=x;
     for k=0:h-1
          somma=somma+termine;
          termine=-termine*x^2/((2*k+3)*(2*k+2));
     endfor
     y=somma;
endfunction
```

Si è sfruttata la proprietà che il rapporto tra due termini consecutivi della sommatoria è $\frac{-x^2}{(2k+3)(2k+2)}$

Allora il grafico di confronto si ottiene con la funzione

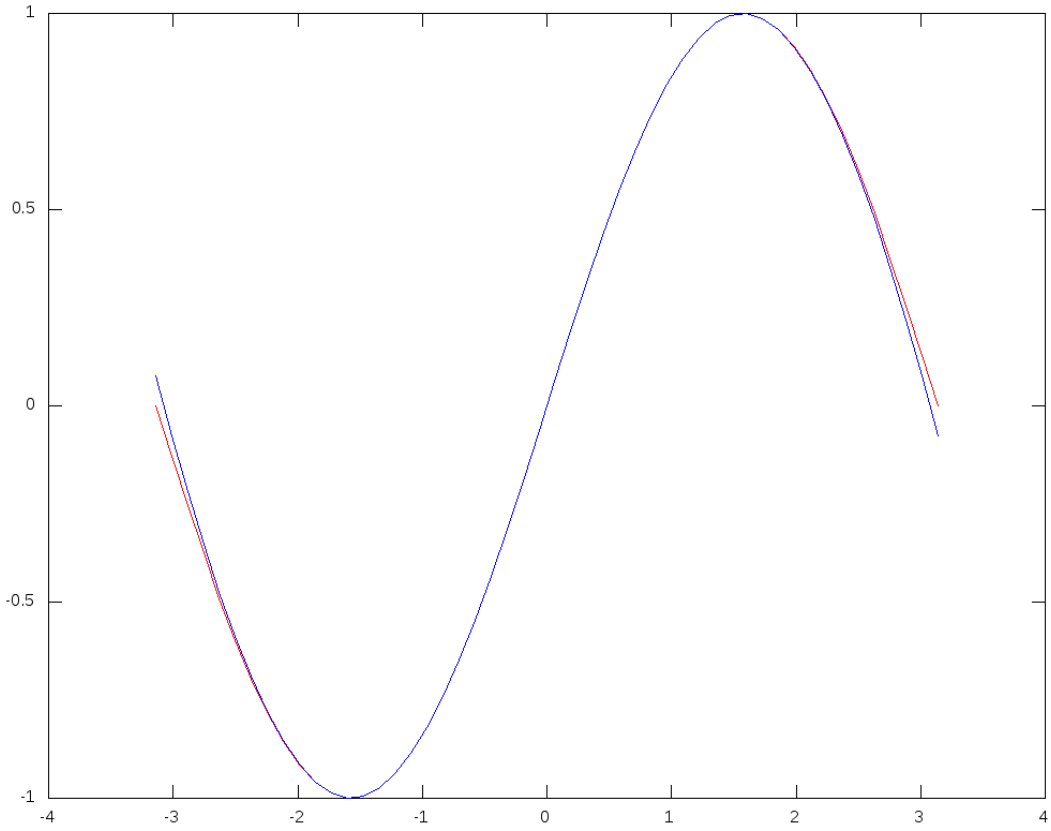
```
function confronta_sin(n,a,b,k)
     h=(b-a)/(n-1);
     x=a:h:b;
     y=sin(x);
     y1=zeros(1,n);
     for h=1:n
          y1(h)=sin_approx(x(h),k);
     endfor
     plot(x,y,"r",x,y1,"b");
endfunction
```

in cui [*a*,*b*] è l'intervallo, *n* è il numero di punti e *k* il numero di termini da prendere.

Ad esempio con `confronta_sin(50,-pi,pi,4)` si ottiene il grafico mostrato nella pagina seguente, in cui la funzione seno è rappresentata in rosso, mentre quella approssimata a 4 termini è in blu. Come si può notare, le due funzioni sono quasi sovrapposte, a dimostrazione della velocità di convergenza della formula di Taylor.

Il comando *plot* ha diversi comandi collegati con i quali è possibile cambiare gli assi, inserire una griglia, mettere un titolo al grafico, inserire più grafici nella stessa finestra, ecc.

Il lettore interessato può consultare il manuale per ulteriori dettagli.

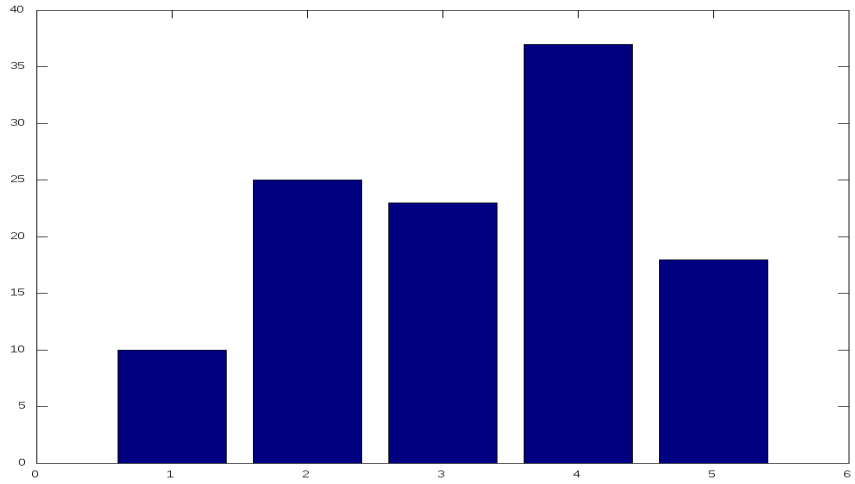


13.3 Altri comandi grafici

Altri comandi grafici utili sono **bar**, **pie**, **hist** e **stairs**.

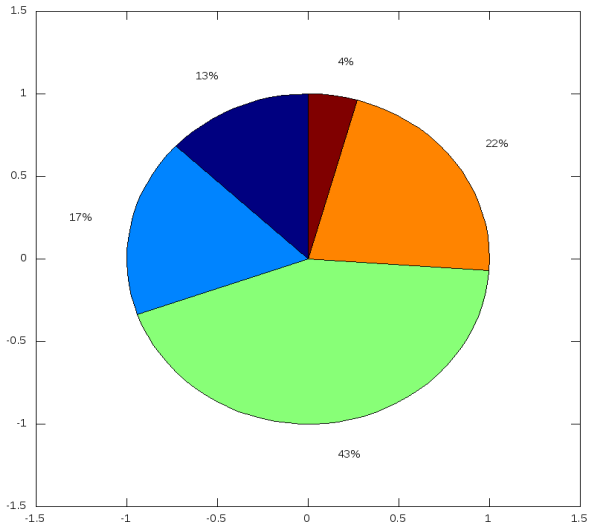
Il comando **bar** crea un grafico a barre.

Ad esempio se y=[10,25,23,37,18], allora *bar(y)* crea il grafico



Il comando **pie** visualizza un grafico a torta.

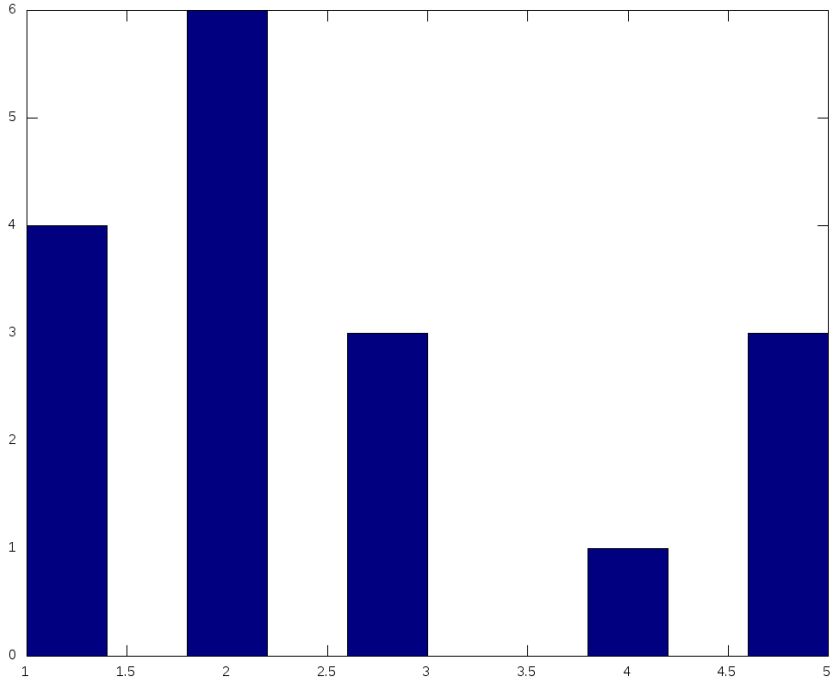
Ad esempio se x=[3,4,10,5,1], *pie(x)* crea il grafico



in cui la “torta” è suddivisa in settori circolari, con angoli proporzionali ai valori di x. E' utile per rappresentare percentuali.

Il comando **hist** crea un grafico ad istogramma.

Ad esempio se d=[1,2,5,5,3,2,1,3,2,5,1,4,2,2,3,1,2], *hist(d)* produce il grafico



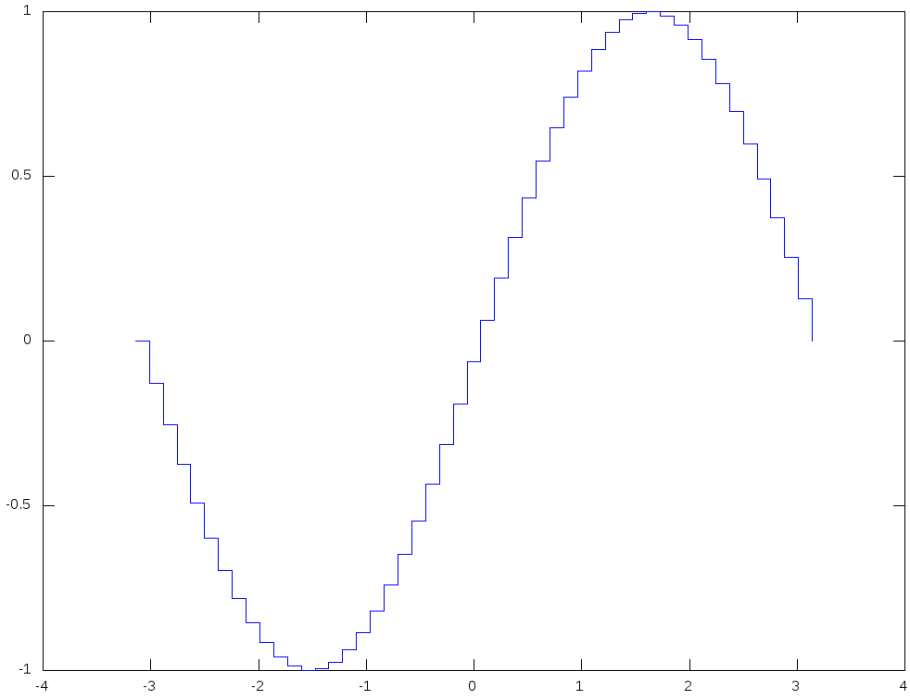
in cui si vedono le frequenze per ciascun valore di *d*, ad esempio 1 compare 4 volte.

Infine **stairs** produce un grafico a “gradini”. Ad esempio con

```
h=2*pi/49;
x=-pi:h:pi;
y=sin(x);
```

`stairs(x,y)`

si ottiene il grafico



14 Un confronto tra Matlab e C

In questo capitolo tracciamo un breve confronto tra il linguaggio di programmazione di Matlab ed uno dei più famosi ed utilizzati linguaggi di programmazione tradizionali, il linguaggio C.

Una prima differenza sostanziale tra una funzione Matlab ed un programma C è che la prima opera in un ambiente interattivo e quindi non ha bisogno, in generale, di effettuare operazioni di ingresso o di uscita, in quanto i parametri sono forniti direttamente dall'utente da linea di comando e i risultati sono visualizzati automaticamente sullo schermo. In C invece un programma viene eseguito in modalità “stand-alone”, ovvero senza alcun supporto esterno, se non le funzionalità minime del sistema operativo, pertanto è compito del programmatore inserire le opportune istruzioni di ingresso e di uscita dati.

Un programma scritto in C (come del resto in Pascal, Fortran, ecc.) deve essere tradotto in linguaggio macchina, creando un file eseguibile. L'artefice di tale traduzione è un programma apposito chiamato compilatore. Una volta tradotto, il programma può essere eseguito, dal sistema operativo o talvolta da un ambiente di esecuzione, avendo a disposizione solo l'eseguibile, senza far più riferimento al programma originario C.

14.1 Un primo esempio

In questo primo esempio, illustrato nella pagina seguente, mettiamo a confronto una funzione Matlab che calcola il MCD di due numeri naturali con il programma equivalente scritto in C. Anche quest'ultimo contiene una funzione “mcd”, ma a differenza della prima, il programma ha bisogno di una parte accessoria, la funzione “main”, che serve ad eseguire la funzione “mcd”.

Iniziamo ora a vedere le differenze tra i due linguaggi. Innanzitutto notiamo la prima riga del programma C

```
#include <stdio.h>
```

che serve per rendere utilizzabili le istruzioni di ingresso/uscita (Input/Output Standard) ed è tipica del linguaggio C.

Nella funzione scritta in C si notano le seguenti caratteristiche

1. Sono presenti molte parentesi graffe: la parentesi graffa chiusa corrisponde alle varie istruzioni *end* di Matlab, ogni graffa chiusa avrà poi una corrispondente graffa aperta.
2. Ogni istruzione è conclusa obbligatoriamente dal punto e virgola.
3. La linea di intestazione della funzione indica sia il tipo del risultato (**int**, ovvero un numero intero) sia i tipi dei due parametri (anch'essi int): in C è obbligatorio dichiarare il tipo di ogni elemento di programmazione, sia esso una variabile, una funzione, o altri oggetti.
4. Il risultato della funzione è indicato con l'istruzione **return**, anziché tramite una variabile apposita.

Notiamo anche notevoli somiglianze tra Matlab e C: le istruzioni *while* e *if* sono pressochè identiche, anche nella semantica, mentre l'assegnamento e molti operatori (quelli di confronto e quelli aritmetici) sono esattamente uguali.

La parte “main” troviamo

1. la dichiarazione di tre variabili intere *p*, *q* e *r*
2. l'istruzione **printf** che scrive sullo schermo la frase “Introduci due numeri interi”
3. l'istruzione **scanf** che legge dei dati da tastiera, in questo caso si aspetta che l'utente digiti due numeri interi in base dieci (la *d* di “*%d*” significa *decimal*) che verranno memorizzati nelle variabili *p* e *q*

Esempio 1: l'algoritmo di Euclide

Matlab	C

`function r=mcd(a,b)`
`  while(a!=b)`
`  if(a>b)`
`  a=a-b;`
`  else`
`  b=b-a;`
`  endif`
`  endwhile`
`  r=a;`
`endfunction`
	`#include <stdio.h>`
	`int mcd(int a,int b) {`
	`  while(a!=b) {`
	`  if(a>b) {`
	`  a=a-b;`
	`  } else {`
	`  b=b-a;`
	`  }`
	`  }`
	`  return a;`
	`}`
	
	`int main( ) {`
	`  int p,q,r;`
	`  printf(“Introduci due numeri interi “);`
	`  scanf(“%d%d”,&p,&q);`
	`  r=mcd(p,q);`
	`  printf(“il loro mcd e' %d\n”,r);`
	`}`

4. le variabili citate nella scanf devono essere precedute dall'operatore “&”
5. il calcolo del massimo comun divisore e la sua memorizzazione nella variabile *r*
6. la scrittura sullo schermo del risultato mediante la seconda istruzione di printf, in cui %d significa “scrivi un numero intero” (cioè *r*) e \n “vai a capo”

Due osservazioni sono importanti. La prima è l'obbligatorietà della regola “definire prima di utilizzare”, ciò vale sia per le variabili *p*, *q* e *r*, sia per la funzione *mcd* stessa. Usare in C un oggetto di programmazione che non è stato definito in precedenza comporta un errore che viene segnalato dal compilatore.

La seconda è il messaggio “`Introduci due numeri interi`”: tale messaggio è indispensabile, perchè l'utente, avviando il programma, deve sapere cosa deve fare, in tal caso digitare due numeri interi.

14.2 Un secondo esempio

Nel prossimo esempio facciamo vedere una funzione che calcola la media di un vettore di numeri reali. Anche in questo caso ad una singola funzione Matlab, corrisponde un programma C con due funzioni: *media* e il solito *main*.

Esempio 2: la media di un vettore

Matlab	**C**

`function m=media(x)`
`  n=length(x);`
 	
`  somma=0;`
`  for j=1:n`
`  somma=somma+x(j);`
`  endfor`
`  m=somma/n;`
`endfunction`
	`#include <stdio.h>`
	
	`double media(double x[],int n) {`
	`  int j;`
	`  double somma;`
	`  somma=0;`
	`  for(j=0;j<n;j++) {`
	`  somma=somma+x[j];`
	`  }`
	`  return somma/n;`
	`}`
	
	`int main() {`
	`  double vett[10],m;`
	`  int i;`
	`  printf(“inserisci gli elementi “);`
	`  for(i=0;i<10;i++) {`
	`  scanf(“%lf”,&vett[i]);`
	`  }`
	`  m=media(vett,10);`
	`  printf(“La media e' %lf\n”,m);`
	`}`

Nella funzione media si notano le seguenti caratteristiche:

1. La funzione restituisce un numero reale (in C si chiamano **double** i numeri reali a doppia precisione, invece i **float** sono quelli a precisione singola) e ciò è indicato prima del nome della funzione *media*.
2. I parametri della funzione sono un vettore *x*, con un numero non specificato di elementi di tipo *double*, ed un numero intero *n*. *n* deve contenere la dimensione di *x*, perchè a differenza di Matlab, in C non esiste l'equivalente di *length*, ovvero non vi è modo di ottenere la lunghezza di un vettore.
3. La dichiarazione delle due variabili *somma*, di tipo *double*, e *j*, di tipo *int*. Quest'ultimo sarà l'indice del ciclo for.
4. Il ciclo **for** assume una forma tipica in C (che ritroviamo anche in altri linguaggi quali Java o C++)
   i. la prima parte *j=0* indica che il ciclo parte da 0 (e non da 1 come in Matlab, perchè gli indici degli array in C iniziano da 0)
   ii. la seconda parte *j<n* indica che il ciclo va avanti fintantoché *j* non arriva a *n*, quando *j* diventa maggiore o uguale a *n*, il ciclo termina
   iii. la terza parte *j++* indica che *j* deve essere aumentato di 1 alla fine di ogni iterazione
   iv. tra parentesi graffe si indicano le istruzioni da eseguire.
5. Gli indici dei vettori e delle matrici vanno messi tra parentesi quadre e partono da 0, anziché da 1.

6. Come nell'esempio precedente, **return** indica il risultato della funzione.

Nel *main* si notano invece le seguenti caratteristiche

1. La dichiarazione del vettore vett, di dimensione 10 e tipo *double*. A differenza dell'enorme versatilità di Matlab, in C la dimensione di un array va data esplicitamente, non può essere alterata durante il corso del programma ed addirittura deve essere nota al momento in cui si scrive il programma: quindi deve essere necessariamente una costante.
2. La necessità di un ciclo for per acquisire gli elementi di un array. Infatti in C non è possibile effettuare alcuna operazione su un array completo, nemmeno quelle più elementari, come quelle di ingresso/uscita o gli assegnamenti, per cui ogni operazione va scomposta in termini di operazioni sui singoli elementi.
3. main deve passare la dimensione di *vett* come secondo argomento alla funzione *media*, come già spiegato precedentemente.
4. La lettura e la scrittura di numeri reali double usa il codice *%lf* nella *scanf* e nella *printf* .

14.3 Considerazioni finali

Il C è un linguaggio di programmazione compilato che consente di scrivere programmi molto efficienti e il suo uso è diffuso ampiamente ed è utilizzato sia in ambito scientifico, sia per scrivere software di base (ad esempio Windows e Linux sono scritti in gran parte in C) e applicativo (Octave stesso è scritto in C).

Il linguaggio C è un linguaggio in certi aspetti complicato e con una sintassi criptica e difficile per chi si avvicina per la prima volta alla programmazione. Mentre Matlab consente di scrivere programmi senza grossi problemi di “ambientamento”.

D'altra parte il C consente di lavorare in modo sofisticato e di creare programmi estremamente potenti, che non sarebbero alla portata di Matlab, anche perché quest'ultimo non è stato pensato per tale scopo.

Questo breve capitolo ha fornito solo uno sguardo sul linguaggio C. Si invita il lettore interessato a leggere uno dei tanti manuali disponibili, anche in rete, su questo linguaggio di programmazione.

14.4 Esercizi

1. Procurarsi un compilatore C e provare i programmi illustrati nel capitolo.
2. Scrivere un programma C che risolve un'equazione di II grado (attenzione: in C non c'è l'istruzione *elseif*, usare una serie di if-else annidati).
3. Scrivere una funzione C che calcola il massimo elemento di un vettore e scrivere un *main* associato.
4. Scrivere una funzione C che calcola la varianza di un vettore e scrivere un *main* associato.
5. Scrivere una funzione ricorsiva C che calcola il fattoriale di un numero naturale e scrivere un *main* associato.

15 Analisi del costo degli algoritmi

Un aspetto importante quando si definisce un algoritmo per risolvere un problema computazionale, ovvero per calcolare una funzione $f:I \rightarrow O$, è determinare il costo computazionale.

Infatti un algoritmo, quando sarà tradotto in un programma, avrà bisogno di una certa quantità di risorse per poter essere eseguito: tempo di calcolo, spazio occupato sui vari tipi di memoria, energia consumata, quantità di dati inviati o ricevuti dalla rete (se l'algoritmo opera su più computer in rete), ecc. Si può quindi definire un costo computazionale in base ad una o più risorse combinate insieme in qualche modo.

Se il costo è eccessivo, l'algoritmo diventa inutilizzabile: è come se non rappresentasse una soluzione ammissibile.

Si noti che il costo computazionale, tranne casi patologici, è sempre crescente rispetto alle dimensioni dell'input e tende ad infinito. Perciò l'analisi di un algoritmo ha bisogno di calcolare l'andamento di crescita del costo e stabilire se questo andamento è accettabile oppure è eccessivo.

Le risorse di calcolo su cui normalmente ci si concentra sono il tempo e la memoria. Nelle prossime sezioni parleremo solo di costo temporale.

L'analisi dell'occupazione della memoria da parte di un algoritmo è molto semplice e viene di solito svolta solo in casi in cui si evidenzia un uso particolare della memoria.

15.1 Costo temporale

Se si parla di costo temporale di un programma, viene subito in mente l'idea di determinare empiricamente il tempo di esecuzione, cronometrando la durata del programma a partire da un determinato input.

Tale analisi empirica ha svariati difetti.

Innanzitutto il risultato così ottenuto dipende fortemente dall'ambiente di esecuzione, principalmente dal processore, ma anche dalla quantità di memoria e dal sistema operativo.

E' dispendiosa se si vuole analizzare l'andamento del costo, perché sono necessarie un gran numero di misurazioni, ognuna su input diverso. In realtà così facendo, l'andamento non viene determinato in maniera analitica, ma solo stimato attraverso tecniche statistiche che non possono in generale determinare la forma funzionale esatta.

L'oggetto analizzato è un'implementazione dell'algoritmo e non l'algoritmo stesso, per cui i risultati ottenuti dipendono in parte dall'algoritmo e in parte da come è stato implementato, oltre che dalle caratteristiche del computer in cui si svolgono gli esperimenti.

Per analizzare un algoritmo A dal punto di vista del costo temporale, è necessario seguire un'altra strada.

L'idea iniziale è quella di contare il numero di istruzioni elementari in funzione dell'input I. Si ottiene quindi una prima funzione di costo $t_A:I \rightarrow \mathbb{N}$, in cui per ogni $i \in I$, $t_A(i)$ rappresenta il numero di istruzioni elementari che A deve eseguire per rispondere all'input *i*.

Tale analisi si basa sull'ipotesi che tutte le istruzioni elementari di A abbiano lo stesso costo computazionale. Se ciò non risultasse plausibile, si può eseguire un'analisi più fine, suddividendo le operazioni elementari in *k* classi di operazioni di costo uguale o paragonabile e conteggiando quante operazioni sono necessarie per ciascuna classe. Si ottiene così una nuova funzione, che indicheremo

sempre con t_A, definita da

$$t_A(i)=\sum_{c=1}^{k} \tau_c N_{A,c}(i)$$

in cui per c=1,...,k, τ_c è il costo di esecuzione di un'operazione della classe *c* e $N_{A,c}(i)$ è il numero di operazioni della classe *c* eseguite da A per rispondere all'input *i*.

In questa analisi i costi $\tau_1,...,\tau_k$ non sono in generale noti (né possono essere determinati in qualche modo ragionevole), ma non ci sono grossi problemi se vengono trattati come parametri, almeno se si studia il comportamento asintotico di t_A.

Anche senza avere i valori precisi per $\tau_1,...,\tau_k$, spesso si hanno idee ragionevoli, basate sulla conoscenza del funzionamento dei computer convenzionali, sulle loro grandezze relative. Ad esempio se le operazioni di una classe hanno un costo τ_c molto maggiore o minore rispetto a quello di un'altra classe, si può semplificare il calcolo del costo t_A eliminando i termini più piccoli e ottenendo comunque una buona approssimazione.

15.2 Costo nel caso peggiore

L'insieme degli input I può in genere avere una struttura arbitraria su cui è difficile compiere analisi asintotiche. Inoltre I ha in genere un numero enorme di elementi (in teoria infinito) e quindi è impensabile che si possa calcolare t_A per ogni elemento i∈I. Ed è per questi motivi che si introduce un'ulteriore funzione di costo temporale.

Sia d:I→ℕ una funzione che ad ogni input *i* associa un numero intero *d*(*i*) che indica la grandezza o dimensione di *i*. Un valore di *d*(*i*) può essere ottenuto codificando *i* tramite un qualche alfabeto, ad esempio come sequenze di bit, e contando il numero di simboli.

La grandezza di un numero intero *n* può essere calcolata facilmente tramite la formula

$$d(n)=\lceil log_B(n) \rceil$$

ove B è la base scelta per rappresentare *n* (di solito B=2). La grandezza di una collezione finita di dati si può calcolare come somma delle grandezze dei singoli elementi, e ciò si può utilizzare per vettori e matrici. Comunque è possibile anche considerare come grandezza di un vettore il numero di elementi (se gli elementi si possono considerare più o meno della stessa grandezza) e per le matrici sia il numero di elementi, sia nel caso di matrici quadrate l'ordine.

Definiamo perciò il costo temporale di un algoritmo A nel caso peggiore come

$$T_A(n)=\max \{ t_A(i) : d(i)=n \}$$

In sostanza per calcolare $T_A(n)$ per un certo valore di *n*, è necessario trovare tra tutti gli input che hanno dimensione *n*, determinare il caso peggiore possibile (dal punto di vista del tempo) e calcolare il costo temporale di A su quell'input.

L'utilizzo della funzione $T_A(n)$ presenta diversi vantaggi.

Innanzitutto è più facile da calcolare che non l'intera funzione t_A, di cui rappresenta una sintesi. Infatti in molti algoritmi è relativamente facile trovare, dato *n*, l'input peggiore di dimensione *n*, cioè quello con il costo maggiore. Spesso è sufficiente creare degli input che cerchino di eseguire al massimo le operazioni di A. Non di rado accade che $t_A(i)$ è uguale o abbastanza simile per tutti gli input *i* di una certa dimensione *n*: in questo caso il calcolo è ulteriormente semplificato.

E' facile vedere che per ogni input i∈I con d(*i*)=*n* accade che

$$t_A(i) \leqslant T_A(n)$$

ovvero che $T_A(n)$ rappresenta sempre una stima per eccesso del costo temporale su un input di dimensione n.

E' poi possibile determinare il comportamento asintotico di T_A confrontandolo con varie funzioni di n. In generale si dice che T_A è dell'ordine di f(n), in simboli O(*f*), se esiste una costante C e un numero naturale m tale che per ogni n>m accade che

$$T_A(n) \leqslant C\, f(n)$$

Calcolare l'andamento asintotico di T_A equivale a trovare l'ordine di infinito ed è molto più semplice che non calcolare il valore effettivo di T_A. Infatti costanti moltiplicative e termini di grado minore o, più in generale, di ordine di infinito minore possono essere trascurati senza alterare il risultato. Ad esempio $3n^3+2n-5$ è comunque dell'ordine di n^3.

L'analisi asintotica ha anche l'ulteriore vantaggio che i parametri $\tau_1,\ldots, \tau_k$ spariscono, essendo conglobati nella costante C, per cui diventano inessenziali.

15.3 Alcuni esempi di calcolo del costo temporale

Nel primo esempio analizziamo l'algoritmo classico che calcola il prodotto C di due matrici quadrate A e B di ordine n tramite la formula

$$c_{hk} = \sum_{r=1}^{n} a_{hr}\, b_{rk}$$

per *h,k=1,...,n*.

L'algoritmo è descritto in Matlab ed è simile a quellafornita nella sezione 10.4

```
for h=1:n
     for k=1:n
          s=0;
          for r=1:n
               s=s+a(h,r)*b(r,k);
          endfor
          c(h,k)=s;
     endfor
endfor
```

Calcoleremo il costo temporale in funzione di *n*. Notiamo subito che il costo non dipende dalle matrici A e B, ma solo da *n*, pertanto la nostra analisi sarà abbastanza semplice.

Le operazioni elementari che si riscontrano sono

1. addizione tra numeri reali
2. moltiplicazione tra numeri reali
3. assegnamenti tra numeri reali
4. esecuzione di cicli for

E' facile vedere che le operazioni di tipo 1 e 2 si trovano nell'istruzione che si trova nel ciclo più interno e che quindi sono eseguite n^3 volte. Esistono tre assegnamenti tra numeri reali, il primo ed il

terzo sono eseguiti n^2 volte, mentre il secondo è eseguito n^3, complessivamente abbiamo n^3+2n^2.

Complessivamente il tempo di esecuzione, relativamente ai numeri reali, è

$$t_p\ n^3+t_m\ n^3+t_a(n^3+2n^2)=(t_p+t_m+t_a)n^3+2t_a\ n^2=O(n^3)$$

ove t_p è il tempo di esecuzione di un'addizione, t_m di una moltiplicazione e t_a di un assegnamento.

L'aggiunta del numero di operazioni relativi alla gestione dei cicli for non influisce, e nel prosieguo verrà proprio omessa, per due motivi. Innanzitutto le operazioni sui numeri interi sono sempre più veloci di quelle sui numeri reali. Inoltre il conteggio complessivo di tali operazioni

1. n assegnamenti, addizioni e confronti su *h*
2. n^2 assegnamenti, addizioni e confronti su *k*
3. n^3 assegnamenti, addizioni e confronti su *r*

è dell'ordine di $O(n^3)$ ed è già presente nel computo delle operazioni sui numeri reali. Infatti il numero delle operazioni per gestire i cicli for sarà, come ordine di infinito, sempre uguale a quello delle operazioni complessive.

Nel secondo esempio analizziamo il costo di ricerca di un elemento *x* in un vettore di *n* elementi, in funzione di *n*.

Il codice Matlab è la funzione *appartiene*, già presentata nel capitolo 10.

```
function a=appartiene(x,v)
     a=false;
     n=length(v);
     j=1;
     while (j<1 && a==false)
          if(v(j)==x)
               a=true;
          endif
          j=j+1;
     endwhile
endfunction
```

Le operazioni che si trovano in questa funzione sono

- assegnamenti e confronti su numeri interi (incluso anche le variabili logiche);
- addizione di numeri interi;
- confronti tra x e elementi del vettore v.

L'ultimo tipo di operazione potrebbe essere più pesante delle altre se il vettore contiene numeri reali o oggetti più grandi.

E' chiaro che in questo algoritmo, il numero di operazioni complessive dipende dal fatto se *x* è presente in *v* e, in caso affermativo, in quale posizione si trova.

Il caso peggiore è quando l'elemento si trova in ultima posizione, oppure, se si conteggiano solo i confronti tra *v(j)* e *x*, e non gli incrementi su *j* (variabile intera) e gli assegnamenti su *a*, allora anche il caso in cui *x* non è presente in *v*.

In entrambe le situazioni il numero di iterazioni del ciclo while è *n*. Ad ogni iterazione viene effettuato un confronto sugli elementi del vettore e due confronti su numeri interi, un incremento ed eventualmente un assegnamento su variabile intera. Quindi il numero di operazioni nel caso peggiore è *n* confronti, ovvero O(*n*) operazioni complessive.

Cosa succede se una funzione ha due o più cicli e il ciclo interno dipende da quello esterno ? Analizziamo la seguente funzione che calcola la somma degli elementi al di sotto della diagonale principale:

```
function s=somma_inferiore(a)
     n=rows(a);
     s=0;
     for h=1:n
          for k=1:h-1
               s=s+a(h,k);
          endfor
     endfor
endfunction
```

Il numero di addizioni di numeri reali è pari a *0+1+2+3+...+(n-1)*, infatti per *h*=1 non sono svolte addizioni, per *h*=2 una sola, per *h*=3 due, ecc.

Tale numero è pari a $\frac{n(n-1)}{2}$, ovvero a O(n^2). Quindi anche se il ciclo sull'indice *k* non va da 1 a *n*, il costo totale ha lo stesso ordine di infinito come se i due cicli operassero sull'intervallo *1:n*.

Analizziamo ora una funzione ricorsiva che, dati un numero reale *x* e un numero naturale *n*, calcola x^n. Per il momento calcoliamo il tempo in funzione di *n*.

```
function p=potenza_ricorsiva_veloce(x,n)
     if (n==0)
          p=1;
     elseif (rem(n,2)==0)
          p=potenza_ricorsiva_veloce(x*x,n/2);
     else
          p=x*potenza_ricorsiva_veloce(x*x,(n-1)/2);
     endif
endfunction
```

Le operazioni che si trovano in questa funzione sono

- moltiplicazioni con numeri reali
- assegnamenti tra numeri reali
- divisioni tra numeri interi
- confronti tra numeri interi

Si può notare che ad ogni chiamata ricorsiva vengono effettuate al più 2 confronti tra interi, 2 divisioni tra interi, 2 moltiplicazioni tra reali e 1 assegnamento tra reali.

Le chiamate ricorsive sono, in funzione di *n*, pari alla parte intera di $log_2 n$, che è il numero di volte che è necessario dividere successivamente *n* per ottenere 1.

Complessivamente avremo, nel caso peggiore, O(*log n*), dato che tra *log n* e $log_2 n$ differiscono per un fattore moltiplicativo costante.

Il prossimo esempio illustrerà cosa succede se *n* non è una misura congrua della dimensione dell'input: analizziamo la semplice funzione *primo* che verifica che se un numero naturale *n* applicando direttamente la definizione. Il codice, visto nel capitolo 9, è

```
function p=primo(n)
     p=true;
     r=ceil(sqrt(n);
     d=2;
     while(d<=r && p==true)
          if(rem(n,d)==0)
               p=false;
          endif
          d=d+1;
     endwhile
endfunction
```

Non è difficile vedere che ad ogni iterazione sono svolti al più 3 confronti, 1 divisione, 2 assegnamenti e 1 addizione, tutti svolti su numeri interi. Il massimo numero di iterazioni effettuate è pari alla parte intera di $\sqrt{n}$, che accade quando *n è* primo oppure è il quadrato di un numero primo.

Il numero di operazioni complessive è quindi $O(\sqrt{n})$.

Ragioniamo un attimo sul fatto che *n* è l'input del problema, non la vera dimensione dell'input. Se si esprime *n* in base 2, servono $b=\lceil log_2 n \rceil$ cifre binarie. Quindi è *b* la dimensione dell'input e il numero di operazioni in funzione di *b* diventa $O(2^{b/2})$, cioè esponenziale in *b*. Ciò indica che tale algoritmo ha un tempo che cresce "troppo velocemente" e non è utilizzabile per controllare la primalità di numeri interi molto grandi.

Nel caso della funzione ricorsiva *potenza_ricorsiva_veloce*, il numero di chiamate ricorsive è invece pari a *b*, il numero di bit che servono per rappresentare *n*, e quindi il costo è O(*b*).

15.4 Tempo polinomiale e P=NP

In informatica un algoritmo è ritenuto accettabile se il suo costo cresce al più come un polinomio in n, cioè se è pari a $O(n^k)$, per un certo k costante. Andrebbero bene anche andamenti del tipo $O(n \log n)$, dato $n \log n < n^2$. Un problema appartiene alla classe **P** se esiste un algoritmo con costo al più polinomiale che lo risolve.

Il motivo che spiega la preferenza per algoritmi con costo al più polinomiale è che si tratta di un tasso di crescita controllato. Mettendo a confronto n^2 e 2^n per alcuni valori di n

n	n^2	2^n
10	100	1024
20	400	$\sim 10^6$
30	900	$\sim 10^9$
50	2500	$\sim 10^{15}$
100	10000	$\sim 10^{30}$
1000	10^6	$\sim 10^{301}$

ci rendiamo conto che 2^n "esplode" e passa a valori molto alti all'aumentare di n anche solo in maniera sensibile: nel passaggio da 20 a 30 ci sono ben 3 ordini di grandezza di differenza.

Molte operazioni e molti problemi importanti nella matematica e nelle scienze sono in **P**. Ad esempio le operazioni aritmetiche, il calcolo di funzioni trigonometriche e trascendenti, le principali operazioni su matrici e vettori, compresi il calcolo di determinante, inversa e determinazione di autovalori e autovettori. Recentemente è stato trovato un algoritmo in **P** che controlla se un numero intero (grande) è primo oppure no.

Esistono però molti problemi che resistono a tutti i tentativi e che per ora non si sa se appartengono a **P**. Uno di questi è la fattorizzazione dei numeri interi (grandi): trovare un fattore non banale di un numero naturale. Questo problema è importante, anche dal punto di vista pratico, dato che molti protocolli crittografici basano la loro sicurezza sulla difficoltà computazionale della fattorizzazione, ovvero sull'impossibilità "pratica" di fattorizzare numeri molto grandi.

Un altro problema che non si sa se appartiene a **P** è il problema della programmazione intera, ovvero quello di risolvere un sistema di disequazioni lineari con variabili intere.

Molti problemi della matematica e dell'informatica appartengono ad una classe apparentemente più grande di problemi, chiamata **NP.**

Un problema appartiene a questa classe se è risolubile da un algoritmo con costo al più polinomiale in un computer ideale che può eseguire un numero illimitato di operazioni contemporaneamente, conteggiando non le operazioni complessive, ma solo il numero di passi paralleli compiuti.

Ad esempio per trovare un fattore non banale di n, si può controllare in parallelo per ogni numero naturale d, con $1<d<n$, se n è divisibile per d.

Uno dei problemi più importanti nell'informatica è dimostrare che **NP** è distinto da **P** o, cosa meno plausibile, che coincidono: in mancanza di dimostrazioni o controesempi per ora la questione è ancora aperta.

Se fosse dimostrata la prima ipotesi, allora molti problemi utili, tra cui quello citato della programmazione intera, non sarebbero risolubili in tempo polinomiale, ossia in un tempo ritenuto "accettabile" e quindi di fatto non sarebbero risolubili in modo pratico.