CAPITOLO PRIMO

IL CONCETTO DI ALGORITMO[1]

1.1 Che cos'è un algoritmo

Gli *algoritmi* sono metodi per la soluzione di problemi. Possiamo caratterizzare un problema mediante i dati di cui si dispone all'inizio e dei risultati che si vogliono ottenere: risolvere un problema significa ottenere in uscita i risultati desiderati a partire da un certo insieme di dati presi in ingresso. I dati in ingresso vengono anche detti (valori in) *input* e i risultati in uscita (valori in) *output*. Possiamo assumere che ciascun problema consista di un insieme di casi particolari, o istanze. Ogni istanza di un problema è caratterizzata da un insieme specifico di dati in ingresso e da un determinato risultato. Supponiamo ad esempio che il problema generale consista nel calcolare la lunghezza dell'ipotenusa di un triangolo rettangolo date le lunghezze dei due cateti. In questo caso le istanze del problema corrispondono agli specifici triangoli di cui calcolare l'ipotenusa. Le informazioni in ingresso (i dati in *input*) sono le lunghezze dei cateti; il risultato che ci si attende in uscita (l'*output*) è la lunghezza dell'ipotenusa.

Un algoritmo per risolvere un problema è un metodo che consente di calcolare il risultato desiderato a partire dai dati di partenza. Cioè, a partire dai dati in input, consente di calcolare l'output corrispondente. Il comportamento di un algoritmo può essere schematizzato come nella fig. I-1.

Figura I-1

Affinché un metodo per la soluzione di un problema costituisca un algoritmo deve essere totalmente esplicito: vanno specificati in maniera precisa e particolareggiata tutti i passi del procedimento da eseguire per ottenere i risultati in uscita a partire dai dati in ingresso. Nel caso del triangolo rettangolo un possibile algoritmo è costituito dalla seguente lista di istruzioni:

[1] Da M. Frixione e D. Palladino, *Funzioni, Macchine, Algoritmi*, Carocci, Roma, 2004

- si prenda in input la lunghezza A del primo cateto
- si prenda in input la lunghezza B del secondo cateto
- si calcoli il quadrato di A
- si calcoli il quadrato di B
- si sommino i due quadrati
- si estragga la radice quadrata del valore così ottenuto
- si produca in output quest'ultimo valore

In generale, un algoritmo è un procedimento di calcolo costituito da un insieme di istruzioni. Tali istruzioni fanno uso di un insieme finito di operazioni elementari, le quali si possono assumere come note e primitive (nell'esempio precedente sono state assunte come note le operazioni di elevamento al quadrato, addizione ed estrazione di radice quadrata). Le istruzioni devono essere tali che, per poterle applicare, basti saper eseguire le operazioni elementari. Inoltre, affinché un procedimento sia un algoritmo, deve godere delle seguenti proprietà:

- L'insieme delle istruzioni di cui è composto deve essere finito.
- Se la soluzione esiste, deve poter essere ottenuta mediante un numero finito di applicazioni delle istruzioni.
- All'inizio del calcolo, e ogni qual volta sia stata eseguita un'istruzione, si deve sempre sapere in maniera precisa quale istruzione va eseguita al passo successivo, e quindi non devono esserci due istruzioni diverse che possono essere applicate nello stesso momento. In altri termini, in ogni fase del calcolo non deve mai accadere che, per sapere quale istruzione si deve eseguire, ci si debba basare sull'intuizione, o si debba tirare a indovinare. Un procedimento che goda di questa proprietà è detto *deterministico*.
- Infine, deve essere sempre chiaro se si è giunti o meno al termine del procedimento, e se sono stati ottenuti i risultati desiderati.

Quindi gli algoritmi (che sono detti anche *metodi effettivi*) sono procedimenti deterministici che consentono di risolvere determinati problemi senza far ricorso ad alcuna forma di creatività o di inventiva. Per eseguire un algoritmo è sufficiente applicare le istruzioni passo dopo passo, badando solo a non commettere sviste.

Dal fatto che un algoritmo è un processo deterministico consegue che, una volta fissati i dati, il risultato ottenuto è sempre lo stesso. Non può succedere che, eseguendo più volte lo stesso algoritmo con lo stesso input, vengano prodotti output diversi.

Si noti che, pur essendo un algoritmo caratterizzato da un insieme finito di istruzioni, le possibili istanze del problema che esso risolve sono, di

norma, infinite. Ad esempio, il precedente algoritmo calcola la lunghezza dell'ipotenusa per ogni coppia A, B di lunghezze dei cateti, ed esiste un numero infinito di tali coppie.

Esempi di algoritmi possono essere tratti dalle matematiche elementari. Sono algoritmi, ad esempio, i procedimenti che consentono di eseguire le quattro operazioni aritmetiche. In questo caso l'input è costituito dai due numeri su cui operare e l'output dal risultato dell'operazione. Come pure è un algoritmo il procedimento euclideo per la ricerca del massimo comun divisore di due numeri naturali non nulli (si veda la finestra "L'algoritmo euclideo per il calcolo del massimo comun divisore"). In logica, il metodo delle tavole di verità è un algoritmo che permette di stabilire se una formula del linguaggio proposizionale è o meno una tautologia. In questo caso l'input è costituito da una formula proposizionale F, l'output è una risposta del tipo *sì/no*: *sì* se F è una tautologia, *no* in caso contrario.

L'algoritmo euclideo per il calcolo del massimo comun divisore

Il cosiddetto *algoritmo euclideo delle divisioni successive* è un metodo per determinare il massimo comun divisore (MCD) di due numeri naturali non nulli a e b. L'input dell'algoritmo sono i due numeri a e b, e l'output è il loro massimo comun divisore. Ne illustriamo il funzionamento mediante due esempi.

Supponiamo che si voglia determinare il MCD di 2079 e 987.
Dividendo 2079 per 987 otteniamo come quoziente 2 e come resto 105:

	2079	=	987 · 2 + 105
Dividiamo ora 987 per 105. Si ha:	987	=	105 · 9 + 42
Procediamo dividendo 105 per 42:	105	=	42 · 2 + 21
Dividiamo ora 42 per 21:	42	=	21 · 2 + 0

Siamo pervenuti a un resto nullo. L'ultimo divisore, in questo caso 21, è il MCD cercato. Quindi MCD(2079, 987) = 21.

Calcoliamo ora MCD(2835, 1540).

Dividiamo 2835 per 1540:	2835	=	1540 · 1 + 1295
dividiamo 1540 per 1295:	1540	=	1295 · 1 + 245
dividiamo 1295 per 245:	1295	=	245 · 5 + 70
dividiamo 245 per 70:	245	=	70 · 3 + 35
dividiamo 70 per 35:	70	=	35 · 2 + 0

Il resto è 0, per cui MCD(2835, 1540) = 35.

Giustifichiamo la correttezza del procedimento di calcolo illustrato, ossia proviamo che esso conduce sempre al risultato richiesto dopo un numero

finito di passaggi. La proprietà che sta alla base del procedimento è la seguente:

se $a = bq + r$, *allora* MCD(a, b) = MCD(b, r).

Dimostrazione. Se d è un divisore comune ad a e b, allora esso è anche divisore di bq e quindi di $r = a - bq$ (se un numero è divisore di altri due è evidentemente divisore anche della loro differenza), e pertanto è divisore di b e r. Se d è un divisore comune a b e r, allora è divisore di bq e r e quindi di $a = bq + r$ (se un numero è divisore di altri due è evidentemente divisore della loro somma), e pertanto è divisore di a e b. Quindi, i divisori comuni ad a e b coincidono con i divisori comuni a b e r, per cui sono uguali anche i massimi comuni divisori MCD(a, b) e MCD(b, r).

L'idea alla base del procedimento è allora semplice: anziché calcolare il massimo comun divisore di a e b, si divide a per b, si trova il resto r e si calcola il massimo comun divisore di b e r, che sono numeri più piccoli (nel primo dei due esempi precedenti, anziché calcolare MCD(2079, 987), si calcola MCD(987, 105)).

Iterando il procedimento, i resti via via calcolati sono decrescenti e, dopo un numero finito di passi, si trova necessariamente un resto uguale a 0 (una sequenza decrescente di numeri naturali non può procedere all'infinito). A questo punto il procedimento termina in quanto ci si è ricondotti al calcolo del massimo comun divisore di due numeri di cui uno è multiplo dell'altro (nel primo dei due esempi 42 e 21), per cui il massimo comun divisore cercato è il più piccolo dei due.

Vi sono algoritmi che operano su dati di tipo numerico e algoritmi che elaborano altri tipi di dati. Vediamo un esempio di questo secondo tipo. Si dice che una parola è *palìndroma* (oppure che essa è un *palìndromo*) se può essere letta indifferentemente da sinistra a destra o viceversa. Ad esempio, *ossesso* è un palindromo. Il problema che vogliamo risolvere consiste nello stabilire se, data una certa parola, essa è palindroma o meno. L'input del problema è costituito dalla parola di cui vogliamo sapere se è palindroma. L'output è una risposta di tipo *sì* o *no*: *sì* se la parola presa in input è palindroma, *no* se non lo è.

Un possibile algoritmo per questo compito si comporta nel modo seguente. Si inizia confrontando la prima e l'ultima lettera della parola:

ossesso

Se la prima lettera è diversa dall'ultima, il procedimento termina e la risposta è negativa: non si tratta di un palindromo. Altrimenti si cancellano la prima e l'ultima lettera, e si ripete il procedimento dall'inizio con le lettere rimaste:

ssess

Se, dopo un certo numero di iterazioni del procedimento, non è rimasta alcuna lettera, allora il procedimento termina e la risposta è positiva: la parola di partenza era effettivamente un palindromo (se la parola presa in input è palindroma, ed è composta da un numero dispari di lettere – come è appunto il caso di *ossesso* – allora nell'ultima iterazione la prima e l'ultima lettera coincidono e vengono cancellate).

Un altro esempio di algoritmo che elabora dati di tipo non necessariamente numerico riguarda la ricerca di un dato elemento in un elenco ordinato. Si supponga di voler controllare se un certo nome N figura o meno in un elenco di nomi L ordinato alfabeticamente. In questo caso l'input del problema è costituito dal nome N e dall'elenco L, mentre l'output consiste anche qui in una risposta del tipo *sì* o *no*: *sì* se il nome cercato è presente in L, *no* in caso contrario. Un algoritmo ovvio per risolvere questo problema consiste nello scorrere tutto l'elenco partendo dall'inizio fino a che non si trova N, oppure si arriva alla fine dell'elenco senza averlo trovato. Ovviamente questo metodo, che viene detto *ricerca di tipo sequenziale*, funziona, ma è estremamente inefficiente: se, ad esempio, N compare all'ultimo posto nell'elenco, per trovarlo è necessario controllare tutti i nomi presenti.

Dato però che, per ipotesi, l'elenco L è ordinato, si può progettare un algoritmo più efficiente basandosi su una tecnica che gli informatici chiamano *ricerca binaria*. Il principio è il seguente. Si confronta il nome N con il nome che si trova a metà dell'elenco. Ovviamente, se l'elenco è composto da un numero pari di nomi, bisogna precisare cosa si deve intendere per nome che si trova a metà dell'elenco. Assumiamo che, se L ha n elementi, l'elemento a metà di L sia quello in posizione $quoz(n, 2) + 1$, dove *quoz* è il quoziente della divisione intera tra numeri naturali. Così, se L ha 10 elementi, il nome a metà di L è quello nella sesta posizione. A questo punto si possono dare tre possibilità:

(*a*) il nome a metà dell'elenco coincide con N
(*b*) il nome a metà dell'elenco segue N in ordine alfabetico
(*c*) il nome a metà dell'elenco precede N in ordine alfabetico

Se si è verificato il caso (*a*) la ricerca ha avuto successo, e il procedimento può terminare.

Se invece si è verificato il caso (*b*), allora se *N* figura nell'elenco, deve trovarsi nella sua metà iniziale. Ripetiamo quindi il procedimento prendendo ora in considerazione la prima metà dell'elenco. Vale a dire, da questo punto in poi chiamiamo *L* la metà iniziale dell'elenco, e procediamo come prima: confrontiamo il nome cercato con il nome che si trova a metà di *L*, a quel punto si danno di nuovo tre possibilità, e così via.

Analogamente, se si è verificato il caso (*c*), il procedimento va eseguito sulla seconda metà dell'elenco.

È evidente che, se il nome *N* è nell'elenco, ripetendo questo procedimento un numero finito di volte, esso prima o poi verrà trovato. Altrimenti, se *N* non è nell'elenco, il procedimento avrà comunque termine: a un certo punto, dopo avere ripetuto il procedimento un numero finito di volte, il pezzo di elenco *L* che dovremmo prendere in considerazione sarà vuoto, e il calcolo avrà termine.

È anche evidente che questo secondo algoritmo è molto più "intelligente" di quello basato sulla ricerca sequenziale. Esso infatti, in media, per ottenere il risultato richiede un numero molto minore di passi di calcolo[2].

Gli algoritmi di questi due esempi forniscono entrambi una risposta di tipo *sì* o *no* (gli algoritmi con questa caratteristica vengono detti *algoritmi di decisione*). Non è necessario però che le cose stiano in questo modo. È facile ad esempio immaginare una variante del secondo problema, che consiste nel cercare il numero di telefono di una data persona nella guida telefonica. In questo caso l'input è costituito dalla guida (cioè, da un elenco di nomi ordinati alfabeticamente, a ciascuno dei quali è associato il rispettivo numero telefonico) e dal nome della persona di cui si vuole il numero; l'output è dato dal numero di telefono corrispondente (o da una risposta di fallimento se il nome non è sulla guida).

Sin dall'antichità sono stati sviluppati algoritmi per risolvere svariati tipi di problemi. Tuttavia, soltanto nel corso del ventesimo secolo la nozione stessa di algoritmo è diventata uno specifico oggetto di ricerca. Ciò è avvenuto con la nascita di una nuova disciplina, detta *teoria della computabilità*, o *teoria della computabilità effettiva*, o anche *teoria della ricorsività*. La nozione di algoritmo presentata sopra ha un carattere intuitivo, non è basata su una definizione rigorosa di tipo matematico. La teoria della computabilità è stata sviluppata a partire dagli anni intorno al

[2] Per la precisione, con il primo algoritmo, se *L* ha *n* elementi, nel caso peggiore saranno necessarie *n* operazioni di confronto. Con il secondo algoritmo, nel caso peggiore il numero delle operazioni di confronto necessarie risulterà dell'ordine di $log_2 n$.

1930, ed è stata motivata dell'esigenza di fornire un equivalente rigoroso del concetto intuitivo di algoritmo, al fine di indagare le possibilità ed i limiti dei metodi effettivi.

Per le caratteristiche di determinismo e finitezza che abbiamo enunciato, ogni algoritmo si presta, almeno in linea di principio, ad essere automatizzato, ad essere cioè eseguito da una macchina opportunamente progettata. Con lo sviluppo dei calcolatori digitali la teoria della computabilità effettiva ha dunque assunto lo statuto di teoria dei fondamenti per l'informatica, e svolge un ruolo importante nelle riflessioni teoriche relative a tutte le discipline che a qualche titolo sono collegate all'informatica.

A partire dal quarto capitolo tratteremo in maniera approfondita la nozione rigorosa di algoritmo sviluppata nell'ambito della teoria della computabilità, le conseguenze che ne derivano, e la portata di tali risultati per varie discipline. Nella parte restante di questo capitolo continueremo ad occuparci della nozione intuitiva di algoritmo, ed introdurremo alcune definizioni che ci saranno utili nel resto del testo. Il secondo capitolo tratta la nozione matematica di funzione in relazione allo studio della computabilità; nel terzo capitolo presenteremo alcuni aspetti più avanzati sempre inerenti la nozione intuitiva di algoritmo.

1.2 Diagrammi di flusso

Non appena si abbia a che fare con algoritmi un po' più complicati di quelli visti nel paragrafo precedente, una descrizione a parole come quelle impiegate fino ad ora diventa scomoda e di difficile comprensione. Un modo più perspicuo e sintetico per rappresentare algoritmi è costituito dai cosiddetti *diagrammi di flusso* (in inglese *flow chart*), talvolta chiamati anche *diagrammi a blocchi*. I diagrammi di flusso utilizzano una notazione di tipo grafico. La fig. I-2 mostra un possibile diagramma di flusso per l'algoritmo descritto nel paragrafo precedente, che calcola la lunghezza dell'ipotenusa di un triangolo rettangolo a partire da quelle dei due cateti.

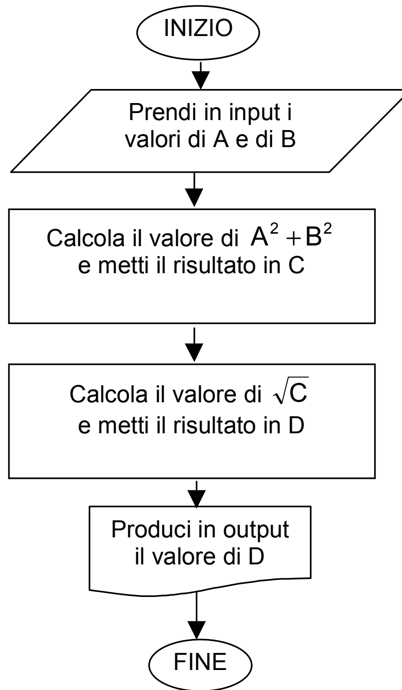


Figura I-2

La lettura di un diagramma come questo è intuitiva. In generale, i diagrammi di flusso sono dei grafi orientati i cui nodi rappresentano le istruzioni da eseguire. La forma di ciascun nodo indica il tipo di istruzione corrispondente. Gli archi che li collegano rappresentano l'ordine in cui tali istruzioni devono essere effettuate. Essi rappresentano appunto il *flusso* delle informazioni durante l'esecuzione dell'algoritmo.

Vediamo nei particolari i tipi di nodi che compongono il diagramma. In ogni diagramma di flusso vi è un nodo *inizio* e un nodo *fine*, raffigurati entrambi mediante delle ellissi (fig. I-3). Essi indicano rispettivamente da dove si deve partire per iniziare il calcolo e quando si è giunti al termine dell'esecuzione dell'algoritmo[3].

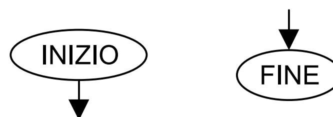


Figura I-3

I nodi la cui forma è raffigurata in fig. I-4 a) e b) rappresentano rispettivamente operazioni di input e di output.

[3] In un diagramma di flusso può esserci più di un nodo *fine* (anche se, in questi casi, è sempre possibile scrivere un altro diagramma di flusso equivalente in cui il nodo *fine* compare una sola volta). Non può comparire invece più di un nodo *inizio*, poiché altrimenti non sarebbe più stabilito univocamente da dove si deve iniziare il calcolo, e si perderebbe così la caratteristica del determinismo.

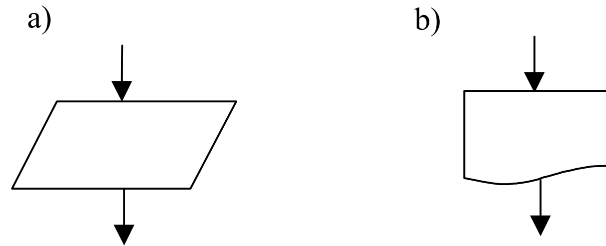


Figura I-4

I valori presi in input e i risultati delle elaborazioni compiute durante il calcolo vengono immagazzinati in *variabili*. Ad esempio, l'algoritmo della fig. I-2 utilizza le variabili A, B, C e D. Le variabili impiegate nei diagrammi di flusso vanno intese come locazioni di memoria, come celle in cui sono depositati dei dati, e il cui contenuto può essere modificato nel corso del calcolo. Le istruzioni che portano a modificare il valore di una variabile vengono rappresentate mediante nodi di forma rettangolare, come quello della fig. I-5.

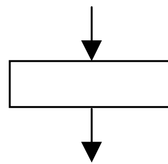


Figura I-5

Un ulteriore tipo di nodi impiegato nei diagrammi di flusso, che non compare nell'algoritmo della fig. I-2, è costituito dai *test*. Graficamente i test sono rappresentati per mezzo di rombi, come nella fig. I-6. All'interno del rombo è scritta un'espressione che viene detta la *condizione* del test. Affinché un'espressione possa fungere da condizione deve poter assumere un valore di verità, vero o falso. Se la condizione di un test è vera, allora nell'esecuzione dell'algoritmo si segue la freccia contrassegnata con 'SI'. Se la condizione è falsa, allora si segue la freccia contrassegnata con 'NO'.

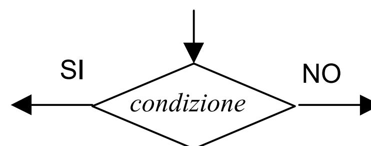


Figura I-6

Come esempio di impiego del test, presentiamo un semplice algoritmo che, preso in input un numero, produce in output il suo valore assoluto (fig. I-7). L'algoritmo memorizza nella variabile A il numero preso in input; dopo di che controlla se il valore di A è maggiore o uguale a 0. In caso affermativo assegna il valore di A alla variabile B. Altrimenti assegna a B il valore di A cambiato di segno. Dopo di che produce in output il valore di B, e il calcolo termina.

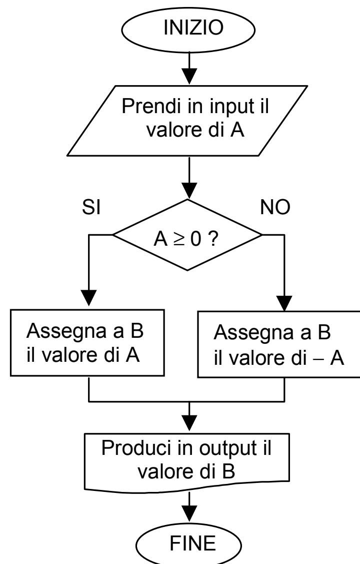


Figura I-7

Questi sono gli elementi di base che impiegheremo per la costruzione dei diagrammi di flusso. Per ora non ci interessa stabilire con precisione quali operazioni si possano utilizzare all'interno dei blocchi. Basta che si tratti di operazioni che, intuitivamente, siano effettuabili in modo algoritmico (come è il caso delle usuali operazioni aritmetiche). In tal caso è evidente che l'intero procedimento descritto da un diagramma di flusso è a sua volta un algoritmo.

Nei diagrammi di flusso i test possono essere impiegati per la definizione di *cicli*, mediante i quali una stessa istruzione, o un gruppo di istruzioni, possono essere ripetuti più volte. La fig. I-8 mostra due tipici esempi di strutture cicliche. Nel ciclo di fig. I-8 a) per prima cosa viene controllato il valore della condizione; se essa è vera, allora vengono eseguite le istruzioni $istruzione_1$,…, $istruzione_n$. Dopo di che, si torna a controllare la condizione, e fino a che essa resta vera le istruzioni vengono ripetute. Il ciclo termina la prima volta che la condizione diventa falsa.

Nel ciclo di fig. I-8 b) per prima cosa vengono eseguite le istruzioni $istruzione_1$,…, $istruzione_n$; dopo di che viene controllata la condizione del test; se essa risulta falsa, allora $istruzione_1$,…, $istruzione_n$ vengono eseguite di nuovo, e ciò si ripete fino a quando la condizione diventa vera. Il ciclo termina la prima volta che la condizione diventa vera.

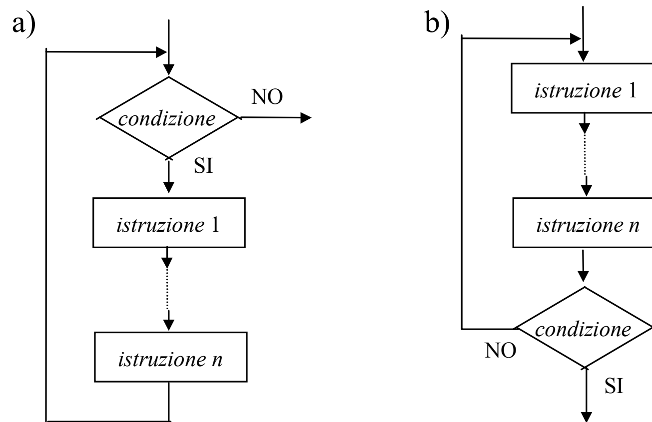


Figura I-8

Vediamo ora un esempio di diagramma di flusso che rappresenta un algoritmo basato su di un ciclo come quello della fig. I-8 a). Assumendo come nota l’operazione di addizione, l’algoritmo della fig. I-9 prende in input due numeri naturali e ne calcola il prodotto.

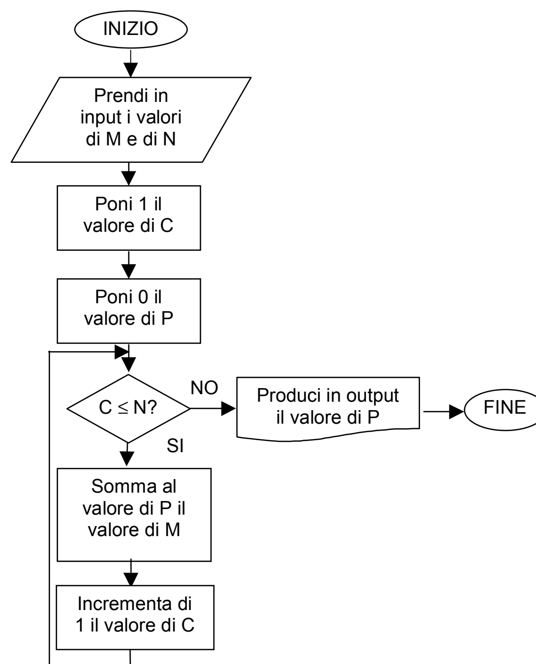


Figura I-9

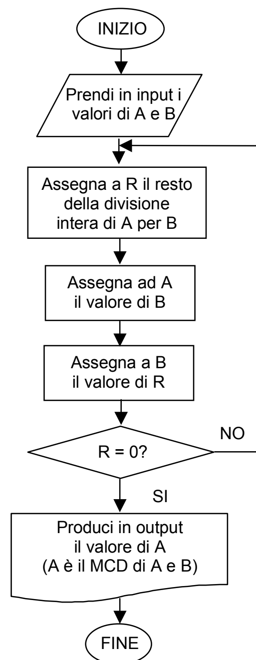
Questo algoritmo prende in input i due fattori da moltiplicare, e li memorizza nelle variabili M e N. Dopo di che calcola il prodotto di M per N sommando M a se stesso per N volte. Ciò si ottiene mediante un ciclo e, per fare sì che esso venga ripetuto N volte, viene impiegata un’altra variabile, che abbiamo chiamato C. Prima del ciclo il valore di C è posto uguale a 1. A ogni iterazione del ciclo C viene incrementata di 1; quando il valore di C supera quello di N il ciclo viene fatto terminare, ed è prodotto il valore in output. In informatica una variabile usata come C viene detta

contatore. Per calcolare il risultato si è usata la variabile P. All'inizio il valore di P viene posto uguale a 0. Ad ogni iterazione, al valore di P è sommato il valore di M, di modo che alla fine in P si ottiene il valore di M sommato a se stesso N volte.

Un esempio di diagramma di flusso che impiega un ciclo come quello della fig. I-8 b è illustrato nella finestra "Un diagramma di flusso per l'algoritmo euclideo per il MCD", e rappresenta l'algoritmo euclideo per il calcolo del MCD descritto nella precedente finestra "L'algoritmo euclideo per il calcolo del massimo comun divisore".

Un diagramma di flusso per l'algoritmo euclideo per il MCD

Presentiamo un diagramma di flusso che raffigura l'algoritmo euclideo per il calcolo del MCD descritto nella finestra "L'algoritmo euclideo per il calcolo del massimo comun divisore". Vengono presi in input i valori dei due numeri A e B. Dopo di che si assegna a R il valore del resto della divisione di A per B, si assegna ad A il valore di B e a B il valore di R; poi si controlla se R vale 0. In tal caso il calcolo è terminato e si produce in output il risultato. Altrimenti si torna a calcolare il valore di R a partire dai nuovi valori di A e di B, e così via.



Anche l'algoritmo che verifica se una parola è palindroma, illustrato nel paragrafo precedente, può essere espresso mediante un diagramma di flusso basato su un ciclo (fig. I-10).

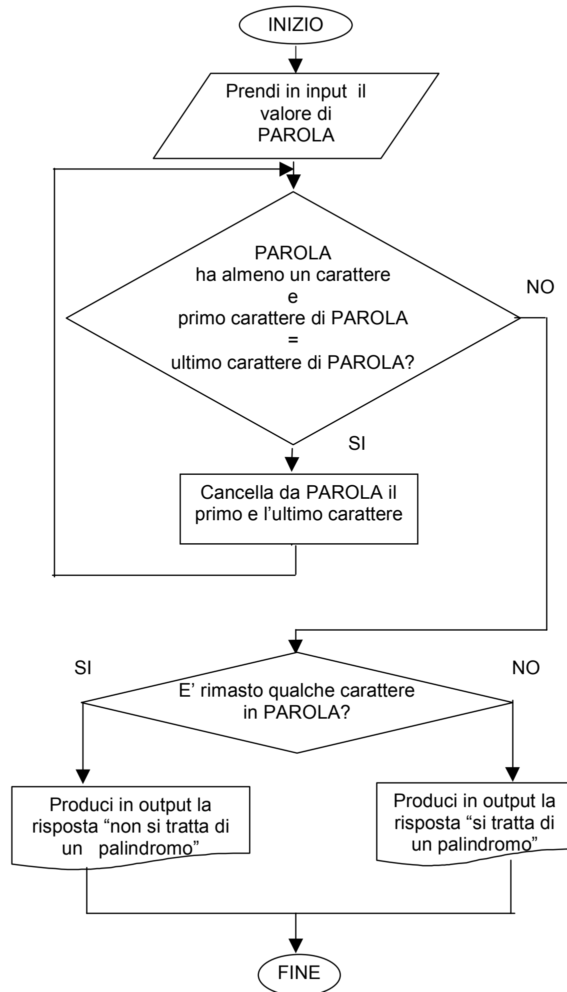


Figura I-10

Per specificare in tutti i particolari gli algoritmi come questo che operano su dati non numerici (al fine, ad esempio, di implementarli mediante un linguaggio di programmazione), bisognerebbe precisare come devono essere rappresentati i dati da elaborare (cosa che invece può essere data per scontata nel caso di algoritmi che elaborano dati numerici). In questo caso, ad esempio, andrebbe specificato come va rappresentata la parola di cui si vuole controllare se è palindroma; nel caso dell'algoritmo per la ricerca di un nome in un elenco (si veda il paragrafo precedente) andrebbe specificato come vanno rappresentati i nomi e l'elenco ordinato. Si dovrebbe cioè (secondo la terminologia informatica) precisare su quali *strutture dati* l'algoritmo opera. Questi aspetti, che sono centrali dal punto di vista informatico, non sono rilevanti per i nostri scopi, per cui nel seguito li tralasceremo.

1.3 Algoritmi che non sempre producono risultati

Ci sono algoritmi che, per alcuni dei possibili input, non producono in output alcun risultato. Un semplice esempio è costituito dall'algoritmo di fig. I-11, il quale esegue la sottrazione tra due numeri naturali. Essa è definita soltanto se il minuendo è maggiore o uguale al sottraendo. Pertanto l'algoritmo di fig. I-11 si comporta come segue: prende in input due numeri naturali A e B; dopo di che, se A è maggiore o uguale a B, produce come output la differenza tra A e B; altrimenti termina senza produrre risultato.

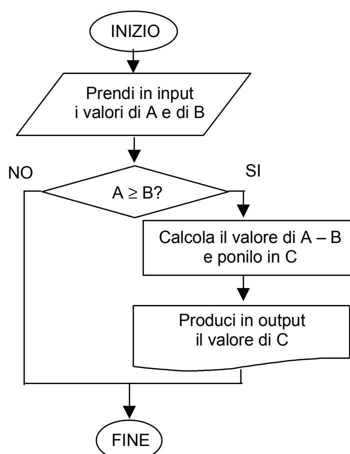


Figura I-11

L'algoritmo di fig. I-11 in alcuni casi non produce risultati. Tuttavia, per ogni coppia di numeri presi in input, dà sempre origine a un calcolo che termina. Vi sono algoritmi che, per alcuni input, non producono alcun risultato in quanto danno origine a un calcolo che non termina. Si considerino ad esempio i diagrammi della fig. I-12.

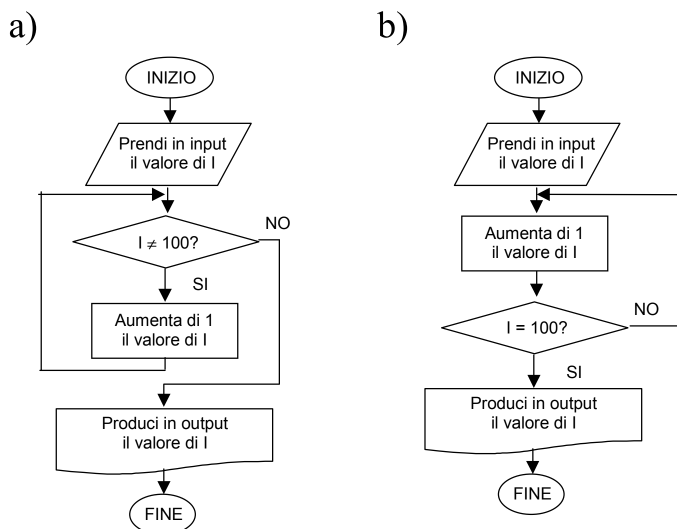


Figura I-12

Nel caso dell'algoritmo di fig. I-12 a), supponiamo che venga dato in input alla variabile I un qualsiasi numero maggiore di 100. La condizione risulterà vera, quindi si inizierà ad eseguire il ciclo. Verrà incrementato di 1 il valore di I; si tornerà quindi a verificare la condizione, che risulterà ancora vera. Poiché ad ogni iterazione il valore di I è destinato a crescere, la condizione del ciclo non diventerà mai falsa, e il ciclo in linea di principio è destinato a continuare all'infinito. Considerazioni analoghe valgono nel caso dell'algoritmo di fig. I-12 b): se il valore preso in input è maggiore di 100, la condizione resterà sempre falsa, e il ciclo non terminerà mai.

Quindi, per alcuni valori in input, cicli come questi possono dare luogo a un calcolo che non termina. In gergo informatico, si dice che in questi casi un algoritmo va in *loop* (in inglese "loop" significa "cappio", "occhiello").

Un altro esempio di algoritmo che in alcuni casi va in loop è dato dal diagramma di flusso di fig. I-13. Esso prende in input un numero naturale x e, se x è un quadrato perfetto, ne calcola la radice quadrata[4]. Dopo aver preso in input il valore di x, l'algoritmo pone a zero il valore di una variabile y, e controlla se x è uguale a y^2. Nel caso il risultato di questo test sia positivo, il calcolo è terminato: y è la radice quadrata di x, e il suo valore viene prodotto in output. Altrimenti il valore di y viene incrementato di uno, e si torna a controllare se x è uguale a y^2. È facile constatare che, se x è un quadrato perfetto, allora prima o poi il calcolo termina, e viene prodotta in output la radice quadrata di x. Altrimenti, se x non è un quadrato perfetto, il test $y^2 = x$ sarà sempre falso, e il calcolo andrà avanti all'infinito senza produrre alcun risultato.

[4] Il calcolo viene effettuato in modo molto inefficiente, ma ciò, in questa sede, non è rilevante.

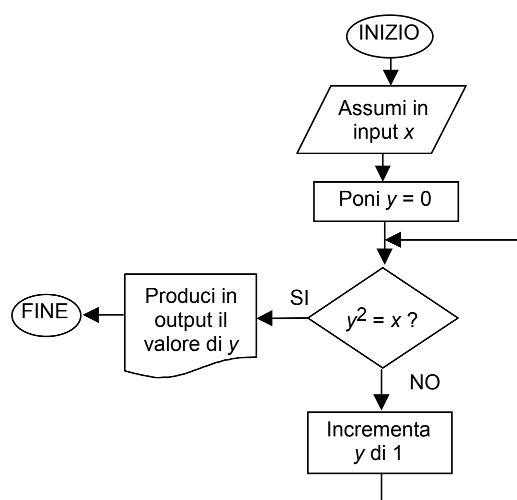


Figura I-13

1.4 Numeri naturali e codifiche dei dati

Nei prossimi capitoli prenderemo in considerazione quasi esclusivamente algoritmi che elaborano numeri naturali. Ciò potrebbe sembrare riduttivo: abbiamo visto che esistono algoritmi definiti su enti assai diversi dai numeri naturali. Vi sono algoritmi che stabiliscono se un certo oggetto appartiene o meno a un dato insieme, o se gode o non gode di una certa proprietà. Vi sono algoritmi che eseguono manipolazioni di vario genere sulle espressioni di un linguaggio o di un sistema formale. In informatica poi vengono impiegati algoritmi per elaborare dati della natura più diversa, dai testi alle immagini, dai suoni ai filmati.

Tuttavia il fatto di concentrarsi su algoritmi che elaborano numeri naturali non comporta una perdita di generalità. Infatti, esistono varie tecniche mediante le quali dati di tipo diverso possono essere codificati mediante numeri naturali, per cui algoritmi come quelli sopra citati vengono ricondotti ad algoritmi che elaborano dati di tipo numerico. Nei capitoli seguenti vedremo nei particolari alcune di queste tecniche. In questo capitolo ci limitiamo ad alcune considerazioni molto elementari, che si basano su esempi tratti dall'informatica.

È noto che nella memoria di un calcolatore tutti i dati sono rappresentati sotto forma di sequenze di *bit*, ossia di cifre 0 e 1. Tali sequenze possono essere interpretate come la rappresentazione di numeri naturali espressi in notazione binaria (*bit* è infatti la contrazione di *binary digit*, ossia, appunto, *cifra binaria*). In questo modo tutte le manipolazioni che un calcolatore esegue sui dati possono essere lette in termini aritmetici, come calcoli di tipo algoritmico che operano su (insiemi di) numeri naturali. Esistono

molteplici tecniche per codificare i dati sotto forma di sequenze di bit. Vediamo sinteticamente un paio di esempi tra i più semplici.

Consideriamo un tipo di dati molto diversi da quelli visti sino ad ora, ossia le *immagini*. In informatica sono state sviluppate svariate tecniche per codificare immagini in modo che possano essere elaborate con un calcolatore. Vediamo a grandi linee come funziona il metodo più semplice per ottenere la codifica binaria di un'immagine. Si supponga di avere a che fare con un disegno in bianco e nero. Si immagini di sovrapporre al disegno una griglia abbastanza fitta da riuscire a rappresentare la figura con il grado di dettaglio desiderato. A questo punto si assegni ad ogni cella della griglia il valore nero se la porzione corrispondente del disegno è prevalentemente nera, il valore bianco se la porzione corrispondente del disegno è prevalentemente bianca. Si otterrà così un'immagine come quella di fig. I-14.

Figura I-14

Se ne ingrandiamo un dettaglio, ad esempio la punta di uno dei baffi di Miomao, otterremo un particolare come quello di fig. I-15, in cui sono evidenti le celle bianche e nere che compongono la griglia (che in questo caso è di dimensioni 266×398).

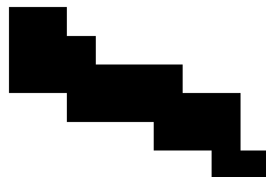


Figura I-15

Ovviamente, quanto più fitta è la griglia, tanto migliore sarà la qualità dell'immagine. A titolo di esempio, riportiamo nella fig. I-16 cinque

versioni della stessa immagine, ottenute rispettivamente con griglie di dimensioni 20×30, 40×60, 50×75, 80×120 e 120×180.

Figura I-16

A questo punto è facile passare a una codifica mediante cifre binarie. Basta rappresentare, ad esempio, ogni cella bianca con 0, e ogni cella nera con 1. In questo modo il disegno di partenza viene codificato con una sequenza di bit. Con buona approssimazione, questo è il tipo di procedimento che uno *scanner* esegue quando acquisisce un'immagine.

In informatica un'immagine codificata mediante questa tecnica viene chiamata *bitmap* (ossia, "mappa di bit"), e ognuna delle celle che la compongono è detta *pixel* (*pixel* sta per *picture element*). Anche immagini più ricche, ad esempio a colori o con vari toni di grigio, possono essere codificate sotto forma di bitmap. In questi casi ogni cella dovrà ammettere più di due valori (in particolare, sarà necessario un valore diverso per ogni possibile colore o tono di grigio). Sarà quindi necessaria più di una cifra binaria per rappresentare lo stato di ciascuna cella, ma, nella sostanza, la tecnica rimarrà immutata.

Grazie a codifiche di questo tipo le elaborazioni eseguite sulle immagini (come aumentarne il contrasto, passare da un'immagine al suo negativo, passare da un'immagine a colori ad una con toni di grigio, eccetera) possono essere viste come procedure che alla codifica dell'immagine iniziale presa come input associano come output la codifica dell'immagine elaborata. Ad esempio, sostituendo nella codifica di fig. I-14 ciascuno 0 con 1 e ciascun 1 con 0 si ottiene il negativo dell'immagine di partenza (fig. I-17).

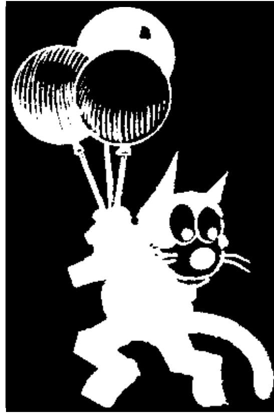


Figura I-17

Tecniche analoghe si possono impiegare per rappresentare altri tipi di dati. Consideriamo ad esempio i *suoni*. Supponiamo di voler rappresentare un'onda sonora, come quella mostrata nella fig. I-18. L'asse delle ascisse rappresenta la dimensione del tempo. In estrema sintesi, si può impiegare una tecnica di questo tipo. Si suddivide l'asse delle ascisse x in intervalli che possono essere scelti anche molto piccoli. Dopo di che, per ciascuno di questi intervalli i, si calcola il valore medio delle ordinate dei punti che hanno ascissa in i, e lo si codifica con un numero naturale. Tale operazione viene detta *campionamento*. L'onda sonora di partenza è ora rappresentata sotto forma di un insieme ordinato finito di numeri naturali. La rappresentazione sarà tanto più fedele quanto più piccoli sono gli intervalli della suddivisione.

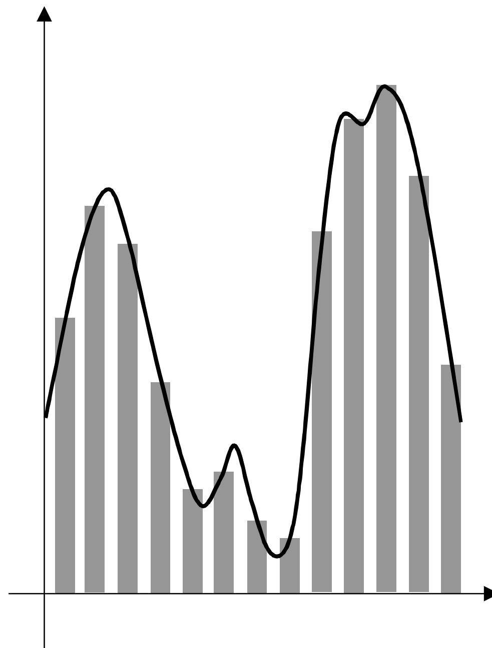


Figura I-18

I numeri naturali possono dunque essere visti come un mezzo per rappresentare dati generici di tipo discreto, e in questo modo saranno intesi nel seguito di questo testo. Si noti tra l'altro che nell'ultimo esempio abbiamo codificato con numeri naturali i numeri decimali che sono le ordinate della curva di fig. I-18. Come noto i numeri reali sono rappresentati da numeri decimali finiti o periodici (se sono razionali), o infiniti e non periodici (se sono irrazionali). In un calcolatore digitale in ogni caso essi vengono approssimati con numeri decimali finiti, i quali possono essere codificati mediante numeri naturali.

Analogico e digitale: il regolo calcolatore

In un *calcolo di tipo analogico* i dati vengono rappresentati per mezzo di grandezze fisiche che variano in modo continuo (ad esempio grandezze elettriche come la corrente o il voltaggio, oppure grandezze geometrico-meccaniche, come la rotazione o lo spostamento reciproco di determinate componenti). I calcoli vengono effettuati agendo fisicamente su tali grandezze. I calcoli analogici si contrappongono ai *calcoli di tipo digitale*, in cui i dati vengono codificati mediante un insieme discreto di simboli, e le computazioni consistono di manipolazioni definite su tali codifiche simboliche. Le codifiche descritte nel testo sono tutte di tipo digitale.

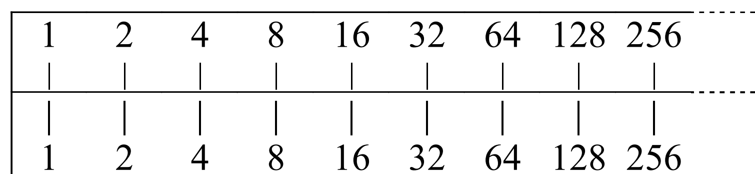
La distinzione analogico/digitale riguarda il modo in cui vengono rappresentati ed elaborati i dati, e non è una distinzione tra tipi diversi di *hardware*. Così vi sono calcolatori elettronici analogici (in cui ad esempio i dati sono rappresentati in termini di voltaggio), come pure calcolatori meccanici sia analogici, sia digitali.

In questo libro ci occuperemo esclusivamente di calcoli digitali. Tuttavia, per meglio chiarire la distinzione, esaminiamo qui un semplice dispositivo di calcolo analogico. Si tratta del *regolo calcolatore*, inventato nel XVII secolo dal matematico inglese Edmund Gunter. Probabilmente si tratta del calcolatore analogico che storicamente ha avuto la maggiore diffusione. Vediamo in sintesi come funziona.

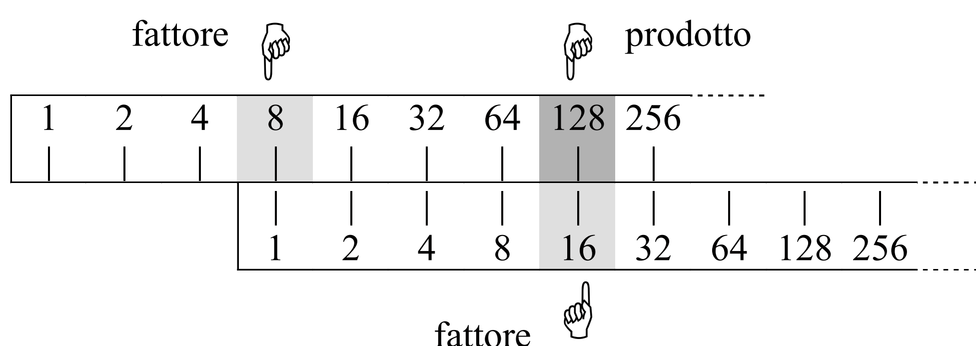
Affianchiamo due righelli, in maniera che siano liberi di scorrere l'uno rispetto all'altro verso destra e verso sinistra. Segniamo su ciascuno di essi delle tacche a distanze uguali, e associamo a ogni tacca una potenza di 2:

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	......
2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	......

In maniera equivalente, possiamo indicare sui righelli i valori corrispondenti:



A questo punto i righelli possono essere usati per moltiplicare tra loro le potenze di 2. Supponiamo di voler moltiplicare 8 per 16. Si fa scorrere verso destra il righello inferiore in modo che la tacca del numero 1 si venga a trovare in corrispondenza della tacca del numero 8 sul righello superiore:

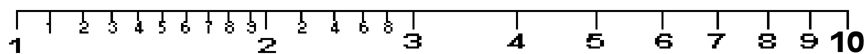


Ora, per ottenere il prodotto di 8 e di 16, basta andare a leggere sul righello superiore il numero che corrisponde a 16 sul righello in basso; si ottiene così che $8 \cdot 16 = 128$. Infatti, la distanza $d = 3$ tra la tacca 1 e la tacca 8 sommata alla distanza $d' = 4$ tra la tacca 1 e la tacca 16 è uguale alla distanza $d'' = 7$ tra la tacca 1 e la tacca 128. Questo accade perché, per come abbiamo segnato le tacche, d è tale che $2^d = 8$, ossia d è il *logaritmo in base* 2 di 8 (in simboli, $d = log_2\ 8$), d' è il logaritmo in base 2 di 16, e, di conseguenza, $d'' = d + d' = 7 = log_2 128$ in quanto $128 = 8 \cdot 16 = 2^3 \cdot 2^4 = 2^{3+4}$.

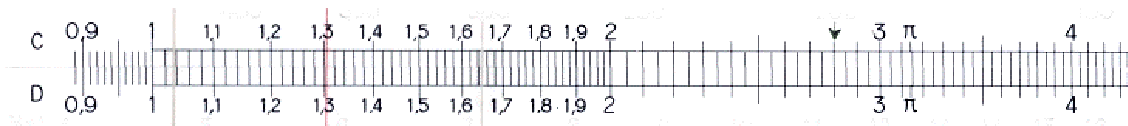
In generale, il logaritmo in base 2 di un numero n ($log_2\ n$) è quel numero d tale che $2^d = n$. Dati due numeri n e n', se $d = log_2\ n$ e $d' = log_2\ n'$, allora $d + d' = log_2(n \cdot n')$. Infatti $n \cdot n' = 2^d \cdot 2^{d'} = 2^{d+d'}$. Su questo si basa il funzionamento del regolo.

Si può infatti generalizzare il procedimento dei righelli scorrevoli in modo da moltiplicare numeri qualsiasi. A tal fine bisogna completare i righelli aggiungendo le tacche dei numeri che non sono potenze di due. Ad esempio si dovrà aggiungere la tacca del 3 tra il 2 e il 4, le tacche del 5, del 6 e del 7 tra il 4 e l'8, e così via. Tutto questo facendo in modo che la distanza della tacca di ciascun numero n da 1 sia uguale al logaritmo in base 2 di n. Il logaritmo di un numero che non sia una potenza di due è un

numero reale non razionale. Ad esempio, log_2 3 = 1,58496250072…, per cui la tacca del 3 dovrà essere segnata a una distanza d = 1,58496250072… dalla tacca dell'1. O ancora, log_2 7 = 2,80735492205…, per cui la tacca del 7 dovrà essere segnata a una distanza d' = 2,80735492205… dalla tacca dell'1. E così via. In questo modo, sommando le distanze d e d', si ottiene d'' = 4,39231742277…, che è il logaritmo in base 2 di 21 (infatti 21 è il prodotto di 3 e di 7). La scala delle tacche sui righelli del regolo avrà l'aspetto seguente (in questo caso tra le tacche 1 e 2 e tra le tacche 2 e 3 sono indicati anche alcuni valori decimali):



Riportiamo qui di seguito il particolare di un regolo reale:



Il regolo è un calcolatore analogico in cui la grandezza fisica che viene utilizzata per il calcolo è lo spostamento reciproco dei due righelli. Essa viene usata per rappresentare i numeri da moltiplicare. Si tratta di una grandezza continua poiché il procedimento che abbiamo descritto non funziona soltanto per le tacche che sono segnate esplicitamente sui righelli (che saranno comunque un insieme discreto). Facendo scorrere in modo continuo i righelli l'uno rispetto all'altro, ogni posizione è "significativa", nel senso che, scelte tre distanze qualunque d, d' e d'', se $d'' = d + d'$, allora $2^{d} \cdot 2^{d'} = 2^{d''}$, a prescindere dal fatto che nei punti corrispondenti sia segnata sui righelli una tacca o meno. In altri termini, in linea di principio le tacche sui righelli potrebbero essere fitte quanto si vuole. Ovviamente dal punto di vista pratico tale possibilità di principio è limitata dalla precisione con cui è possibile effettuare le misurazioni. Questo è un problema generale di tutti i calcolatori di tipo analogico, che non trova corrispettivo nel calcolo digitale. Le limitazioni nella precisione delle misurazioni comportano che, in generale, i calcoli di tipo digitale consentano di ottenere una precisione maggiore di quelli analogici.

ESERCIZI RELATIVI AL CAPITOLO PRIMO

Esercizio 1.1. Una sequenza di parentesi si dice *bilanciata* se, per ogni parentesi aperta, esiste una parentesi chiusa corrispondente. Ad esempio, le due sequenze di parentesi seguenti sono bilanciate:

((( ) (( ))))   ( ) ((( ) ( )) ( ))

mentre le seguenti non lo sono:

)) (( ))   (( ) ( )))

Descrivere a parole un algoritmo che, presa in input una sequenza di parentesi, controlli se essa è bilanciata o meno.

Esercizio 1.2. Rappresentare mediante un diagramma di flusso un algoritmo per la ricerca sequenziale di un nome in un elenco ordinato.

Esercizio 1.3. Rappresentare mediante un diagramma di flusso l'algoritmo per la ricerca binaria di un nome in un elenco ordinato, che abbiamo presentato nel paragrafo 1.1.

Esercizio 1.4. Rappresentare mediante un diagramma di flusso l'algoritmo dell'esercizio 1.1 che stabilisce se una sequenza di parentesi è bilanciata.

Esercizio 1.5. L'algoritmo della fig. I-9 impiega un ciclo come quello della fig. I-8 a). Formulare un algoritmo che produca gli stessi risultati impiegando una struttura come quella della fig. I-8 b).

Esercizio 1.6. Formulare un algoritmo che, assunto come noto il prodotto, prenda in input due numeri naturali m ed n e produca in output m^n.

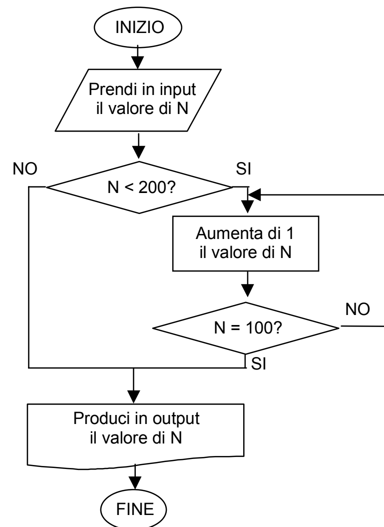
Esercizio 1.7. Si assuma come nota l'operazione *mod*, tale che X *mod* Y sia il resto della divisione di X per Y. Si rappresenti quindi mediante un diagramma di flusso un algoritmo che, preso in input un numero naturale N, produca in output 1 se N è primo, 0 se N non è primo.

Esercizio 1.8. Modificare l'algoritmo della fig. I-11 in modo che produca sempre un risultato: se A < B, produca come output 0.

Esercizio 1.9. Determinare due algoritmi che vadano in *loop* per qualsiasi input, uno basato su un ciclo come quello della fig. I-8 a), e l'altro basato su un ciclo come quello della fig. I-8 b).

Esercizio 1.10. È possibile fare in modo che l'algoritmo della fig. I-13 dia luogo a un calcolo che termina per qualunque input?

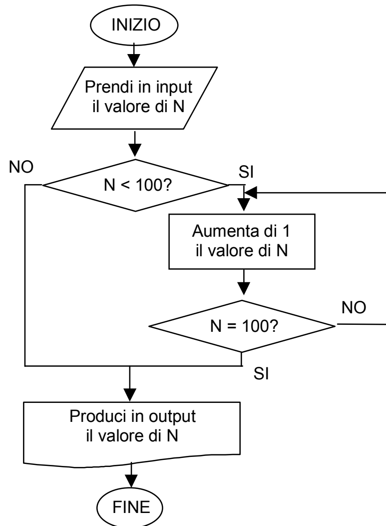
Esercizio 1.11. Stabilire se il seguente algoritmo:



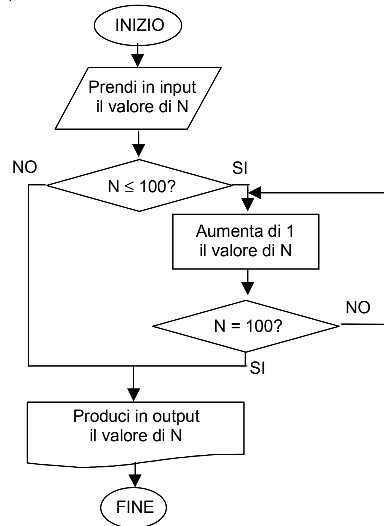
(a) termina per qualunque valore di N
(b) non termina per alcun valore di N
(c) termina se e solo se $N > 200$ oppure $N < 100$
(d) termina se e solo se $N \geq 200$ oppure $N \leq 100$
(e) termina se e solo se $N \geq 200$ oppure $N < 100$
(f) termina se $N > 200$
(g) termina solo se $N > 200$

Esercizio 1.12. Determinare per quali valori di input i seguenti algoritmi terminano.

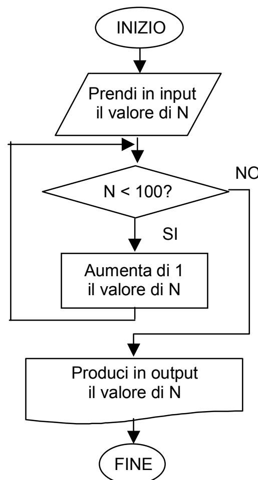
(a)



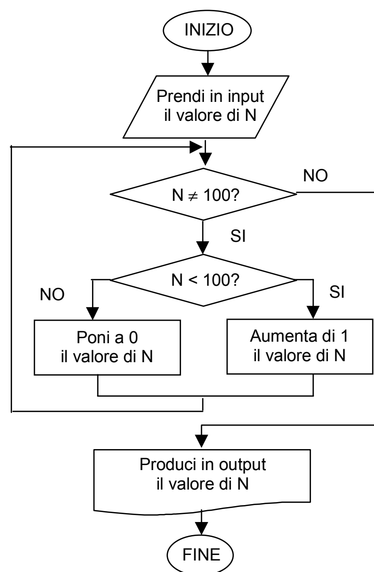
(b)



(c)

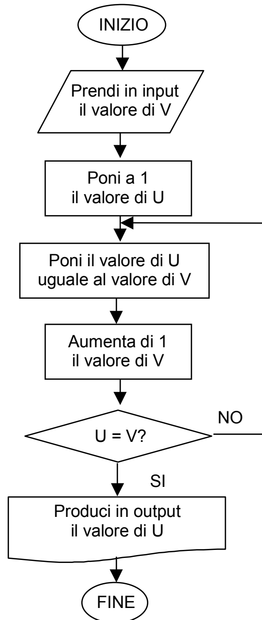


(d)

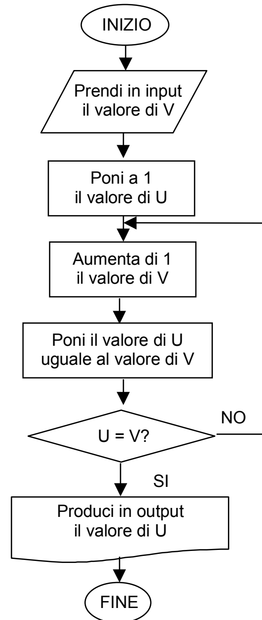


Esercizio 1.13. Stabilire quali dei seguenti algoritmi generano un calcolo che termina sempre.

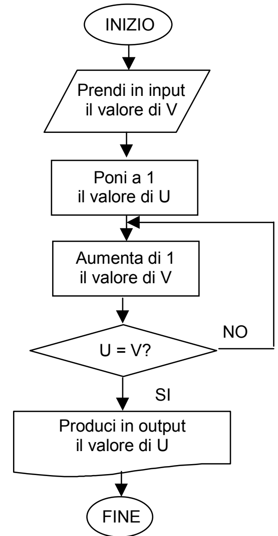
(a)



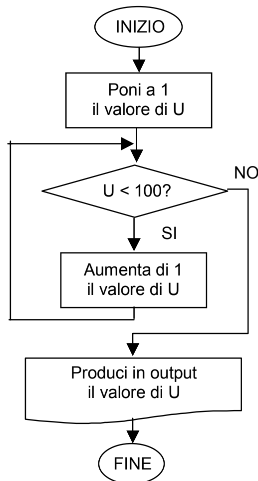
(b)



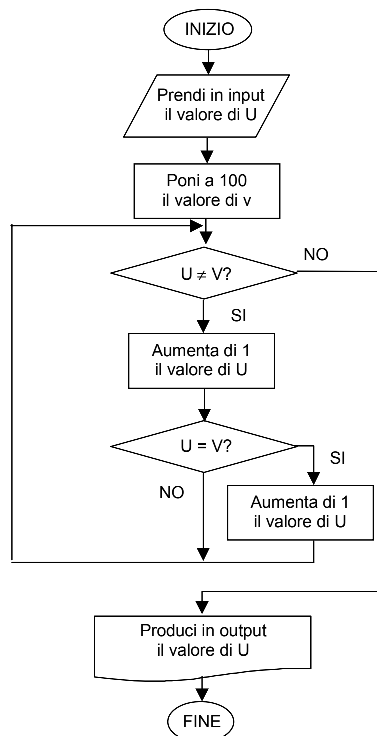
(c)



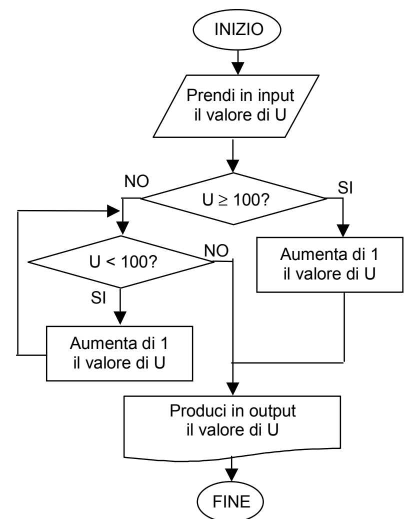
(d)



(e)



(f)



SOLUZIONI DEGLI ESERCIZI RELATIVI AL CAPITOLO PRIMO

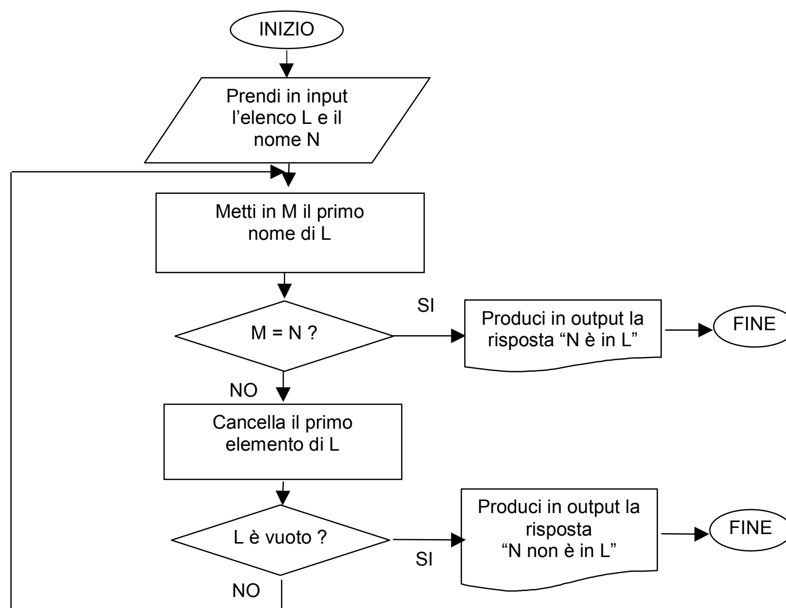
Esercizio 1.1. Un possibile algoritmo per questo compito è il seguente:

- Prendi in input una sequenza S di parentesi.

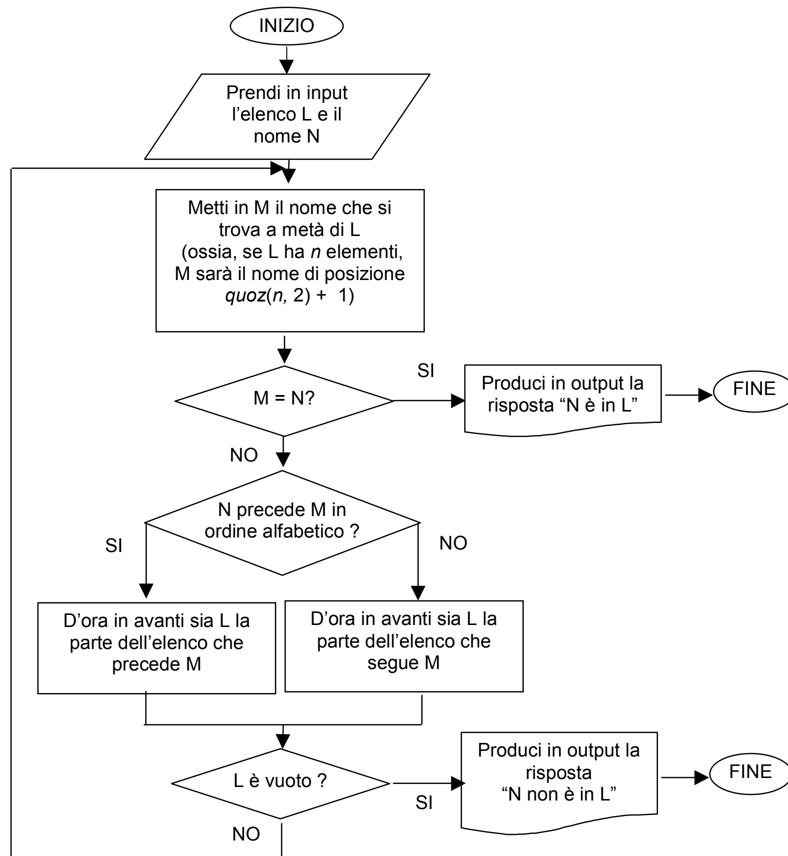
(*) - Se S è vuota fermati: S è bilanciata.

- Altrimenti vai avanti fino a che non trovi una “)” oppure fino a che arrivi in fondo a S.
- Se sei arrivato in fondo a S senza trovare “)”, allora fermati: S non è bilanciata.
- Altrimenti cancella “)” e torna indietro di uno.
- Se trovi una “(“ cancellala e torna a (*). Altrimenti fermati: S non è bilanciata.

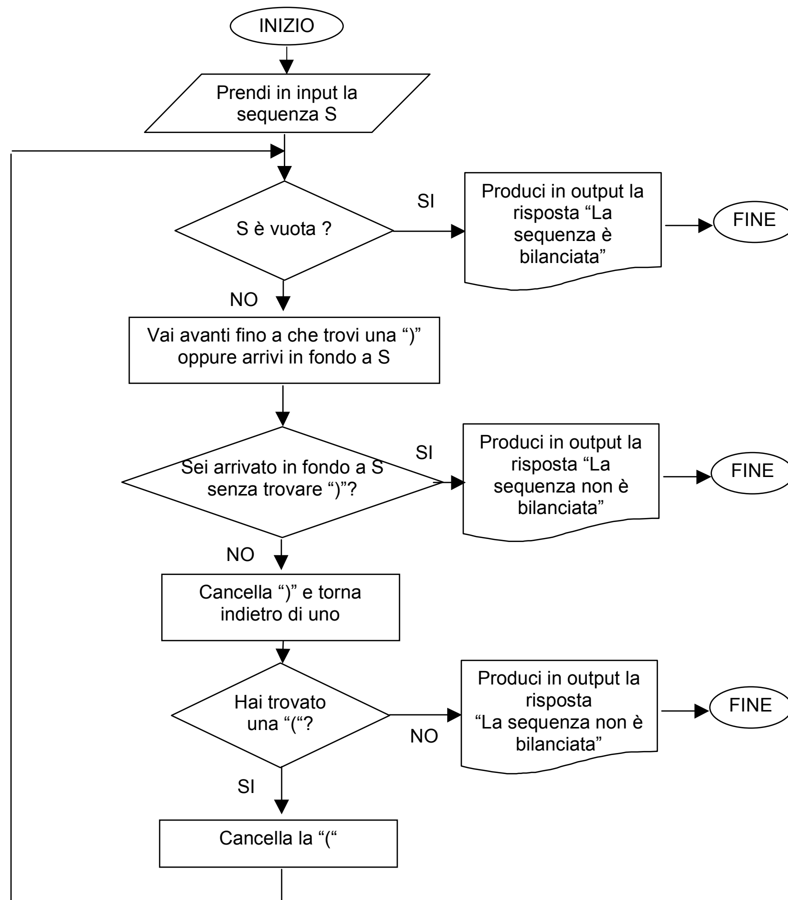
Esercizio 1.2. Un possibile diagramma per il compito specificato è il seguente:



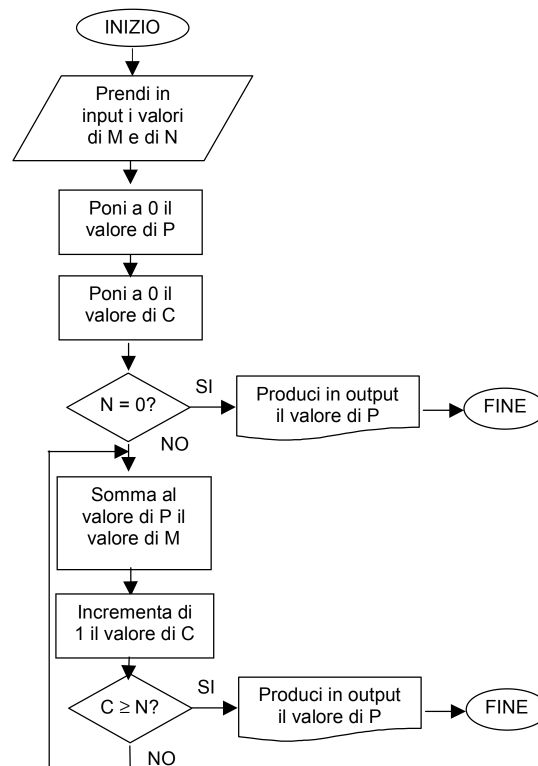
Esercizio 1.3. Un possibile diagramma per il compito specificato è il seguente:



Esercizio 1.4. Un possibile diagramma per il compito specificato è il seguente:

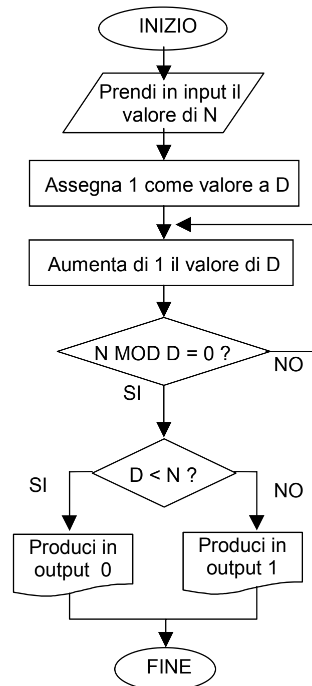


Esercizio 1.5. Un possibile algoritmo con le caratteristiche richieste è il seguente:



Esercizio 1.6. Una maniera per ottenere l'algoritmo voluto consiste nel modificare l'algoritmo di fig. I-9 del testo, in modo che, all'inizio del calcolo, il valore di P venga posto uguale a 1 anziché a 0, e che l'istruzione *Somma al valore di* P *il valore di* M all'interno del ciclo diventi *Moltiplica il valore di* P *per il valore di* M.

Esercizio 1.7. Un possibile algoritmo per il compito specificato è il seguente:



Esercizio 1.8. È sufficiente inserire un'opportuna istruzione di output nell'arco etichettato NO in uscita dal test.

Esercizio 1.9. Ci sono ovviamente infiniti algoritmi che hanno questa caratteristica. Nel caso di un ciclo come quello della fig. I-8 a) basta scrivere nel test una qualsiasi condizione che risulti sempre vera (ad esempio "2 = 2"); nel caso di un ciclo come quello della fig. I-8 b) basta scrivere nel test una qualsiasi condizione che risulti sempre falsa (ad esempio "2 = 3").

Esercizio 1.10. Un modo consiste nel sostituire la condizione del test di fig. I-13 con la seguente: $y^2 = x$ *oppure* $y > x$, e modificare poi di conseguenza la parte successiva del diagramma, facendo in maniera che, se y è maggiore di x, venga prodotta una risposta opportuna, del tipo "non si tratta di un quadrato perfetto".

Esercizio 1.11. Sono vere e) e f). I casi in cui l'algoritmo termina sono tutti e soli quelli in cui N ≥ 200 (in tal caso il test del condizionale è falso, e il ciclo non viene mai eseguito) oppure quelli per cui N < 100 (in tal caso il ciclo termina); e) esprime quindi la condizione necessaria e sufficiente per la terminazione dell'algoritmo, e f) esprime una condizione sufficiente.

Esercizio 1.12. (a) termina sempre; (b) non termina per N = 100; (c) termina sempre; (d) termina sempre.

Esercizio 1.13. (b), (d) e (f). (a) non termina mai, perché al momento del test si ha sempre $U \neq V$; (b) termina subito: qualunque sia il valore di V, alla prima iterazione del ciclo si ha $U = V$; (c) non termina se $V > 0$; (d) termina sempre per ovvie ragioni; (e) non termina se $U > 100$ (se $U = 100$ il ciclo non viene mai eseguito; f) se $U \geq 100$ il calcolo termina subito; altrimenti il ciclo termina quando U arriva a 100).