5. Algoritmi, Linguaggi e Programmi

5.1 Algoritmi

5.1.1 Definizione di algoritmo e caratteristiche di un algoritmo

Nel linguaggio comune, da qualche decina d'anni, è entrata a far parte la parola *"programma"*; il significato di questo termine è ormai noto: essendo il computer una macchina non intelligente ma un mero esecutore di ordini impartiti dall'esterno, necessita di una serie di istruzioni, scritte in un opportuno codice detto **linguaggio di programmazione**, che costituiscono appunto il **programma**. Prima però di poter scrivere un programma il programmatore deve analizzare il problema da risolvere, definire gli obiettivi da raggiungere, individuare i dati iniziali e finali e descrivere tutti i passi necessari per ottenere il risultato voluto.
Affinché il computer sia in grado di eseguire con successo i compiti ad esso assegnati, è necessario che la descrizione del procedimento sia accurata; è opportuno che alla macchina non venga indicato come risolvere un *singolo* problema, ma tutta una classe di problemi che differiscono per i dati iniziali. Tutto ciò sottende al concetto di **algoritmo** che è generalmente usato come sinonimo di:

- procedura effettiva
- procedimento di calcolo
- metodo di risoluzione di un problema
- insieme di regole per eseguire una data operazione

in effetti, la definizione corretta e completa è la seguente:

Un **algoritmo**[1] è una procedura *generale, finita, completa, non ambigua* ed *eseguibile* che lavora su dati d'ingresso fornendo alcuni dati d'uscita. Analizziamo nel dettaglio le proprietà:

- *generale*: il metodo deve risolvere una classe di problemi e non un singolo problema (ad esempio deve essere in grado di calcolare l'area di tutti i triangoli e non solo quella di un particolare triangolo)
- *finita*: le istruzioni che la compongono ed il numero di volte che ogni azione deve essere eseguita devono essere finiti
- *completa*: deve contemplare tutti i casi possibili del problema da risolvere
- *non ambigua*: ogni istruzione deve essere definita in modo preciso ed univoco, senza alcuna ambiguità sul significato dell'operazione
- *eseguibile*: deve esistere un agente di calcolo in grado di eseguire ogni istruzione in un tempo finito

Gli algoritmi possono venire classificati secondo la seguente suddivisione:

- algoritmi deterministici
- algoritmi non deterministici

[1] Il termine **algoritmo** deriva dal nome del matematico arabo Al Khwarismi, vissuto nel IX secolo, che pubblicò l'opera Kitab Al-jabrwal Muquabala (L'arte di numerare ed ordinare le parti in tutto) da cui deriva il nome algebra.

un algoritmo si dirà **deterministico** se per ogni istruzione esiste, a parità di dati d'ingresso, un solo passo successivo; in pratica esiste uno e un solo possibile percorso dell'algoritmo e quindi a fronte degli stessi dati di partenza produrrà gli stessi risultati.
Si dirà **non deterministico** se contiene almeno una istruzione che ammette più passi successivi che hanno la possibilità di essere scelti: l'algoritmo potrà produrre risultati diversi a partire da uno stesso insieme di dati compiendo percorsi diversi; tra gli **algoritmi** non deterministici troviamo quelli **probabilistici** nei quali almeno una istruzione ammette più passi successivi, ognuno dei quali ha una certa probabilità di essere scelto.

5.1.2 Algoritmi e Formalismi di Codifica

Dato un problema, quindi, si cercherà di trovare un algoritmo che lo risolva; se tale algoritmo perviene alla soluzione del problema si dirà *corretto*, se inoltre la soluzione è raggiunta anche nel modo più breve possibile si dirà *efficiente*.
Per chiarire quanto finora detto si prenda in considerazione un esempio :
Si supponga di dovere confrontare le aree di due triangoli e comunicare il rapporto tra le aree (se la prima è maggiore minore o uguale alla seconda); la sequenza di operazioni necessarie alla risoluzione del problema, essendo $A = \frac{b*h}{2}$ l'area del triangolo, è la seguente:

1) acquisire l'altezza del primo triangolo
2) acquisire la base del primo triangolo
3) calcolare l' area del primo triangolo
4) acquisire l'altezza del secondo triangolo
5) acquisire la base del secondo triangolo
6) calcolare l'area del secondo triangolo
7) se l'area del primo triangolo è maggiore dell'area l'area del secondo triangolo comunicare che l'area del primo triangolo è maggiore dell'area del secondo triangolo
8) se l'area del primo triangolo è minore dell'area del secondo triangolo comunicare che l'area del secondo triangolo è maggiore dell'area del primo triangolo
9) se l'area del primo triangolo non è minore né maggiore dell'area del secondo triangolo comunicare che le aree dei due triangoli sono uguali

Per risolvere questo problema non è necessario compiere le operazioni descritte; sarà sufficiente rappresentare in modo fittizio i passi necessari che, partendo dai dati posseduti, facciano pervenire al risultato che si otterrebbe agendo realmente come descritto.
Si indichi con:

- $\mathbf{h_1}$ l'altezza del primo triangolo
- $\mathbf{h_2}$ l'altezza del secondo triangolo
- $\mathbf{b_1}$ la base del primo triangolo
- $\mathbf{b_2}$ la base del secondo triangolo
- $\mathbf{A_1}$ l'area del primo triangolo
- $\mathbf{A_2}$ l'area del secondo triangolo

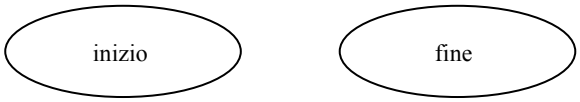
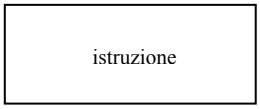
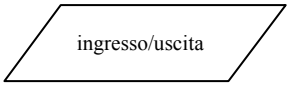
Ecco dunque le operazioni aritmetico-logiche che devono essere eseguite:

Fasi 1-2-3	$A_{1=} b_{1*} h_1/2$	; acquisizione dati, calcolo prima area
Fasi 4-5-6	$A_{2=} b_{2*} h_2/2$	; acquisizione dati, calcolo seconda area
Fase 7-8-9	se $A_1 > A_2 \rightarrow$ output: "l'area 1 è maggiore dell'area 2" se $A_1 < A_2 \rightarrow$ output: "l'area 1 è minore dell'area 2" se $A_1 = A_2 \rightarrow$ output: "l'area 1 è uguale all'area 2"	; confronto logico, comunicazione ; confronto logico, comunicazione ; confronto logico, comunicazione

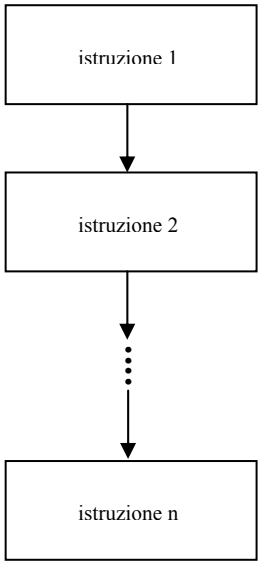
Quello costruito è un modello rappresentativo dell'algoritmo risolutivo; è chiaro che in tal modo rimane risolta una classe di problemi che differiscono solo per il valore dei dati.
I metodi da usare per rappresentare in modo efficiente ed esaustivo gli algoritmi vengono detti *formalismi di codifica* poiché rappresentano l'algoritmo mediando tra il semplice linguaggio comune e il formale linguaggio matematico; ne esistono di diversi tipi con la caratteristica comune di essere ben definiti e non ambigui.
La rappresentazione esaminata precedentemente illustra a parole le operazioni da svolgere; anche se è semplice non risulta essere un modo universale, comprensibile da chiunque voglia esaminare l'algoritmo, poiché ciascuno illustrerebbe i passi in modo personale dando appunto luogo ad ambiguità. Negli anni si sono sviluppate varie tecniche di rappresentazione, ne descriveremo due: i **diagrammi di flusso** o *flow-chart* e la **pseudocodifica**.

5.1.3 Diagrammi di Flusso

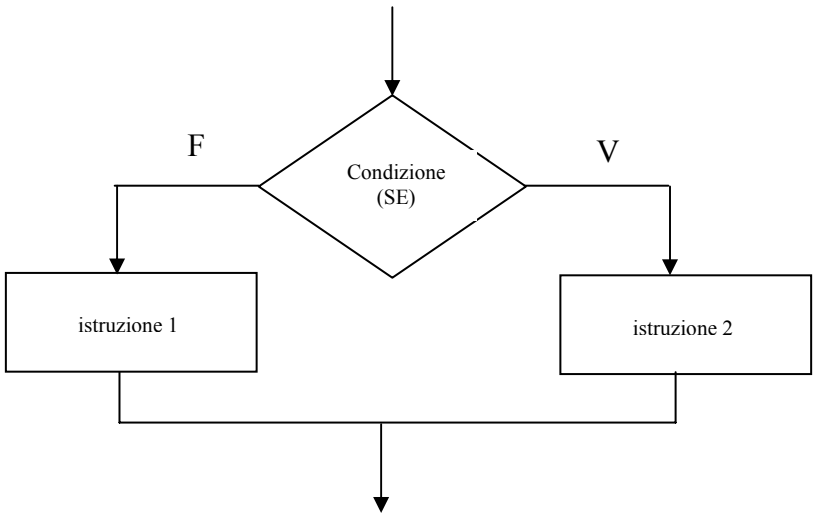
I **diagrammi di flusso** sono una *forma grafica* di formalismo di codifica degli algoritmi; si collocano a metà strada tra i formalismi più "matematici", tra i quali citiamo la *macchina di Turing*, e quelli derivanti dalla formalizzazione del linguaggio normale (di cui è un esempio la rappresentazione vista precedentemente oppure la *pseudocodifica* che verrà esaminata più avanti).
In tale rappresentazione ad ogni simbolo corrisponde un preciso tipo di operazione; è un metodo semplice da imparare ed offre una immediata percezione del flusso di esecuzione delle istruzioni.
Nella tabella seguente si riportano i simboli base con la relativa specifica:

Rappresentazione	**Funzionalità**
	Blocco di inizio e fine dell'algoritmo
	Blocco di azione contenente una singola istruzione dell'algoritmo
	Blocco ingresso/uscita contenente le operazioni dall'esterno verso il computer e dal computer verso l'esterno.
	Frecce che connettono i blocchi ed indicano la sequenza di esecuzione delle istruzioni

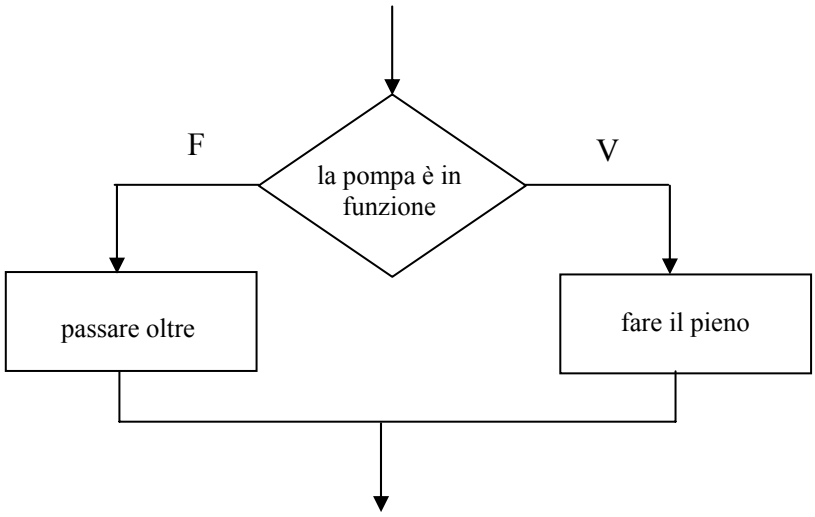
I simboli elencati nella tabella precedente sono adatti a rappresentare algoritmi elementari, che prevedono solo operazioni strettamente *sequenziali*; per poter rappresentare qualsiasi algoritmo è necessario introdurre dei simboli che rappresentino la *selezione* di due percorsi possibili e *l'iterazione* (cioè la ripetizione) di istruzioni o gruppi di istruzioni.
Per rappresentare una **sequenza** di istruzioni, ovvero una serie di operazioni che devono essere eseguite una di seguito all'altra in base all'ordine con il quale sono state definite, lo schema è quello riportato in figura:



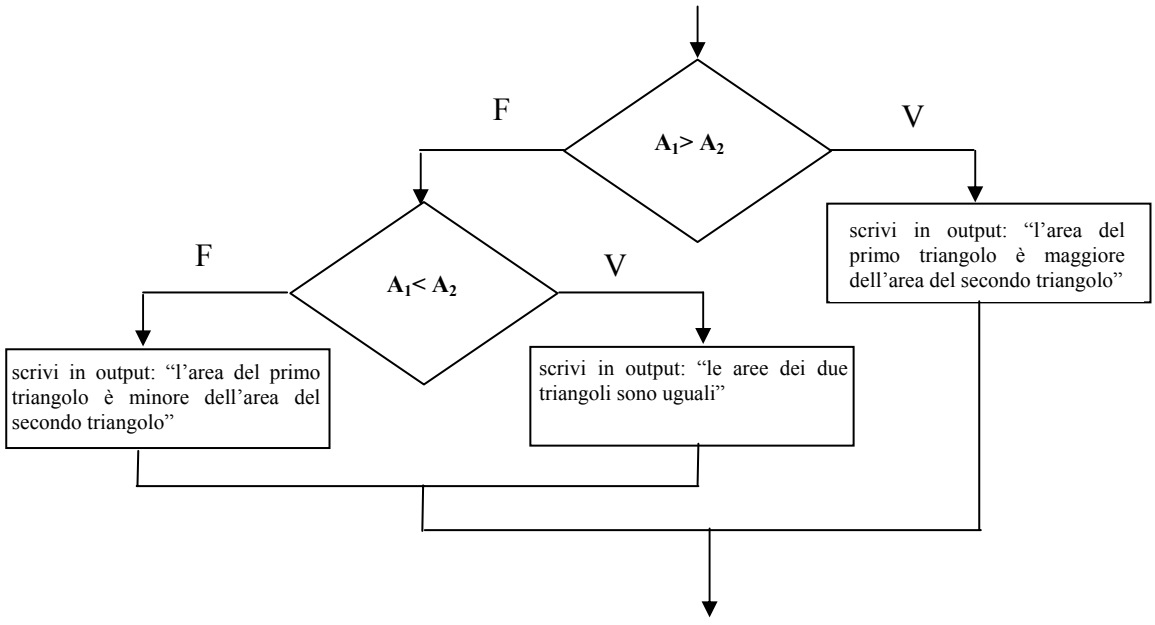
Consideriamo ora un esempio molto semplice che ci aiuta ad introdurre una nuova rappresentazione; supponiamo di doverci rifornire ad una pompa di carburante, se la pompa è in funzione facciamo il pieno altrimenti passiamo oltre; si individuano due operazioni diverse (fare il pieno, passare oltre) che sono eseguite in alternativa a seconda che **una condizione** sia **vera** (la pompa è in funzione) oppure **falsa** (la pompa non è in funzione).
Per poter rappresentare la **selezione** di due percorsi diversi al verificarsi o meno di una data condizione si usa il seguente *flow-chart*:



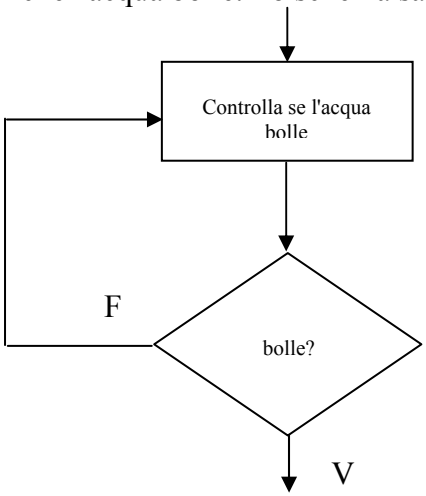
La rappresentazione relativa alla situazione precedentemente considerata sarà quindi:



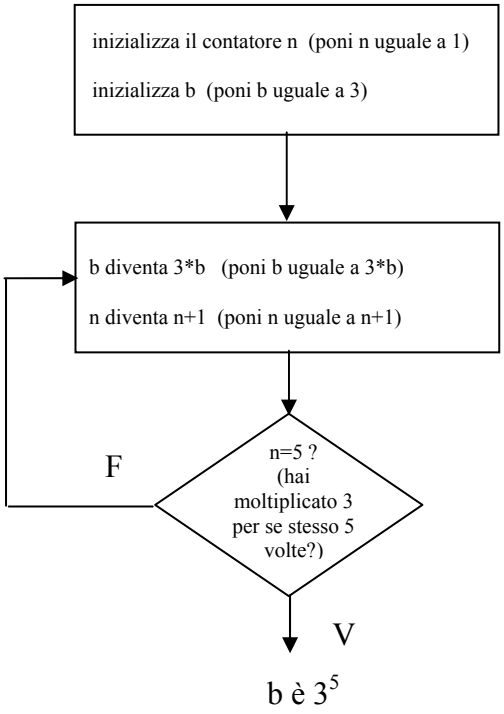
Con la rappresentazione sopra descritta si possono quindi schematizzare algoritmi che prevedono "strade alternative" da percorrere in dipendenza dal verificarsi o meno di una data condizione; specifichiamo che ognuna delle due "strade" possibili può prevedere, a sua volta, non una istruzione sola (come nell'esempio visto) ma una sequenza di istruzioni oppure altre selezioni (*selezione in cascata*), iterazioni o combinazioni di tutte queste.
In riferimento a ciò, riprendendo l'esempio delle aree precedentemente fatto, la condizione da valutare è A1>A2 (A_1 "maggiore" di A_2) se tale condizione risulta vera (A_1 è maggiore di A_2) in output sarà scritto "La prima area è maggiore della seconda", se è falsa viene valutata una ulteriore condizione A1 < A2 (A_1 "minore" di A_2). Se tale condizione è vera (A_1 è minore di A_2) sul video verrà fatta apparire la scritta "La prima area è minore della seconda", se è falsa significa che le aree sono uguali (A1 non è ne maggiore ne minore ad A2, sarà quindi uguale) e sul video verrà fatta apparire la scritta "Le due aree sono uguali".
La rappresentazione in questo caso sarà:



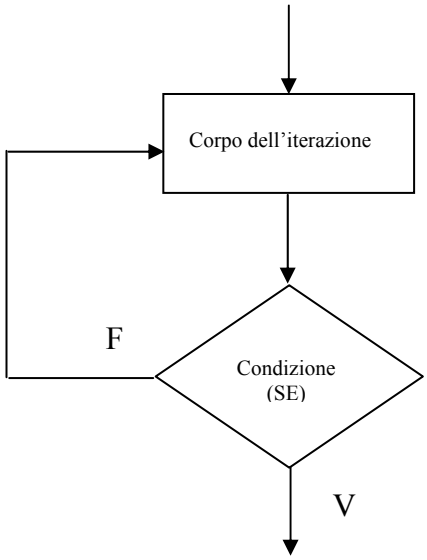
Ci sono degli algoritmi che non possono essere rappresentati con i precedenti *diagrammi di flusso*; un esempio di questi, il processo (algoritmo) di "cottura della pasta", composto dai seguenti passi: prendi la pentola, riempila d'acqua, mettila sul fuoco finché l'acqua bolle, ecc., prevede un "ciclo d'attesa", nel senso che si deve rimanere in attesa controllando periodicamente se l'acqua sta bollendo finché l'acqua bolle. Lo schema sarà:



Il "ciclo d'attesa" consiste quindi nella ripetizione (*iterazione*) di un'azione (il controllo dello stato dell' acqua) per un numero di volte che, in questo caso, non è noto a priori. Per chiarire meglio il concetto consideriamo un esempio numerico: supponiamo di dover calcolare il numero b^n, con b=3 e n=5; la procedura risolutiva consiste nel moltiplicare il numero 3 per sé stesso 5 volte, bisogna quindi ripetere, *iterare* una moltiplicazione. Il *flow-chart* corrispondente è:

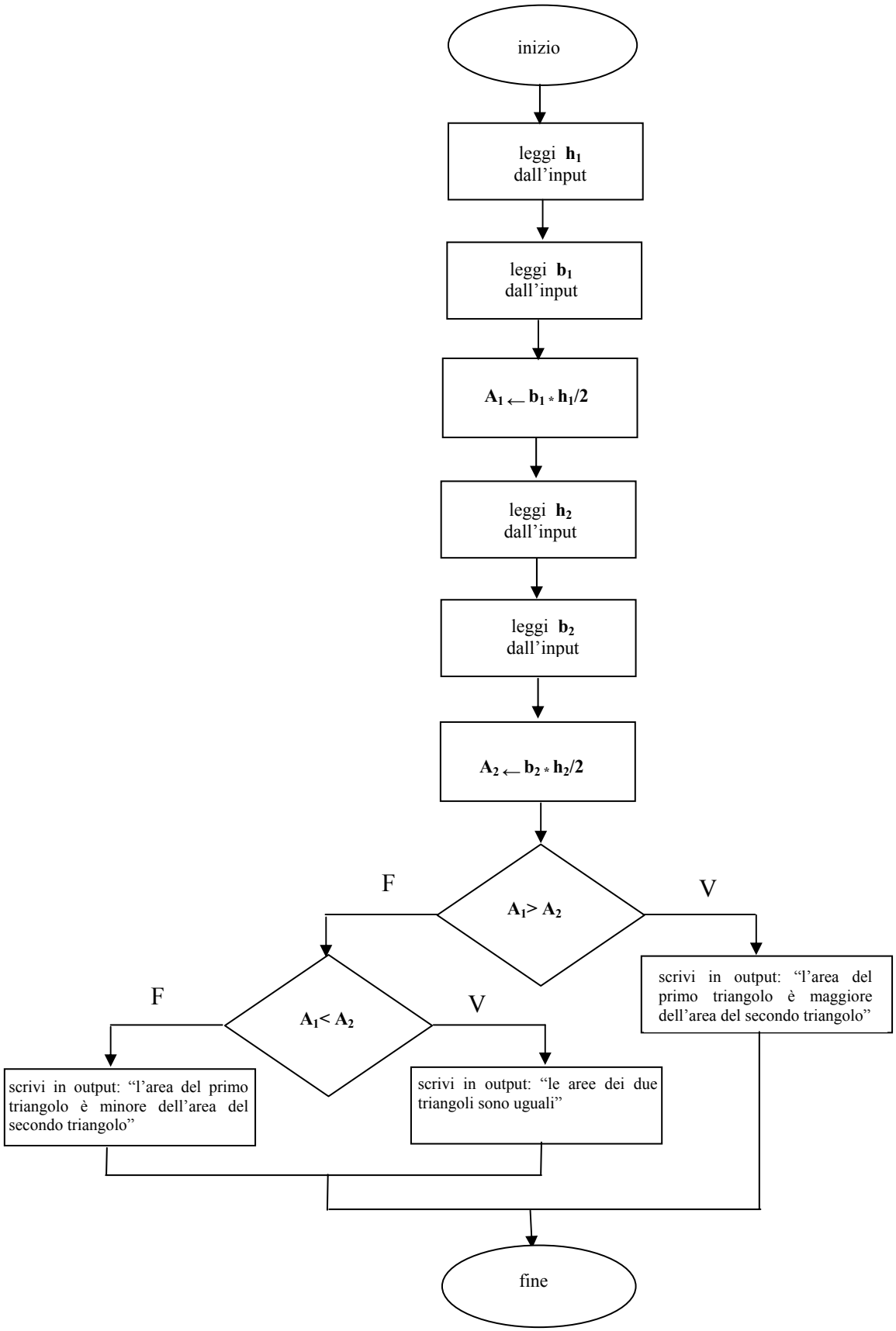


In generale lo schema di riferimento è il seguente:



Prima sono eseguite le *istruzioni* che costituiscono il corpo dell'iterazione (cioè le operazioni che devono essere ripetute) e *dopo* è eseguito il *controllo* per stabilire se ripetere il corpo dell'iterazione (se **"ciclare"** ancora); se la condizione di uscita dal ciclo non è verificata (è falsa) allora il ciclo viene ripetuto, altrimenti l'esecuzione prosegue con la prima istruzione che si trova sul ramo del vero; il corpo dell'iterazione viene eseguito almeno una volta. Bisogna però fare attenzione poiché, se l'esecuzione dei cicli non modifica la condizione il ciclo dura all'infinito (*loop*)! Facendo riferimento all'esempio precedente: se non si incrementa n questi rimane sempre a 1, la condizione è sempre falsa ed il ciclo dura all' infinito (si verifica un *loop*).

Per finire diamo la rappresentazione dell'algoritmo risolutivo riferito al primo esempio considerato:



5.1.4 Pseudocodifica

Un altro formalismo di codifica, che risulta essere più legato al linguaggio naturale scritto rispetto ai diagrammi di flusso, è la **pseudocodifica**.
In tale rappresentazione si utilizza un linguaggio speciale che descrive le istruzioni con frasi rigorose anziché con i simboli grafici; si usano parole chiave, scritte in maiuscolo ed operatori (←, +, *, -, /); si usa inoltre *l'identazione*, una tecnica che prevede il rientro dei gruppi di istruzioni riferite a cicli o a strutture di scelta.
In seguito si illustra la pseudocodifica corrispondente ai diagrammi di flusso esaminati nelle pagine precedenti:
La **sequenza** si rappresenta in questo modo :

INIZIO
Istruzione 1
Istruzione 2
.
.
Istruzione n
FINE

quindi l'algoritmo per fare la pasta avrà la seguente rappresentazione:

INIZIO
prendi la pentola
riempila d'acqua.
mettila sul fuoco finché l'acqua bolle
assaggia l'acqua
se l'acqua non è salata sala l'acqua, altrimenti metti la pasta
.
.
scola la pasta
FINE

La pseudocodifica della **selezione**, che rappresenta l'istruzione **"** se l'acqua non è salata sala l'acqua, altrimenti metti la pasta ", è:

SE l'acqua non è salata
ALLORA sala l'acqua
ALTRIMENTI metti la pasta

In riferimento all'esempio del rifornimento di benzina, si avrà la seguente pseudocodifica:

SE la pompa è in funzione
ALLORA fare il pieno
ALTRIMENTI passare oltre

In generale si avrà:

SE condizione
ALLORA <istruzione 1>
ALTRIMENTI <istruzione 2>

Esiste, chiaramente, la possibilità di rappresentare le selezioni in cascata; in riferimento all'esempio delle aree si ha:

SE $A_1 > A_2$
ALLORA Scrivi("l'area del primo triangolo è maggiore dell'area del secondo triangolo")
ALTRIMENTI
SE $A_1 < A_2$
ALLORA Scrivi("l'area del primo triangolo è minore dell'area del secondo triangolo")
ALTRIMENTI Scrivi("le due aree sono uguali")

L'**iterazione** si rappresenta in questo modo:

RIPETI
<corpo dell'iterazione>
FINCHE' <condizione>

L'azione schematizzata è chiaramente la stessa che con i *flow-chart,* il ciclo viene ripetuto finché la condizione diventa vera (mentre rimane falsa); la pseudocodifica per l'istruzione "mettila sul fuoco finché l'acqua bolle" è la seguente**:**

RIPETI
controlla se l'acqua bolle
FINCHE' *l'acqua bolle*

Per concludere diamo la pseudocodifica del programma di confronto delle aree:

Programma CONFRONTO AREE

INIZIO
Leggi($\mathbf{h_1}$)
Leggi($\mathbf{b_1}$)
$\mathbf{A_1 \leftarrow b_1 * h_1/2}$
Leggi($\mathbf{h_2}$)
Leggi($\mathbf{b_2}$)
$\mathbf{A_2 \leftarrow b_2 * h_2/2}$
SE $A_1 > A_2$
ALLORA Scrivi("l'area del primo triangolo è maggiore dell'area del secondo triangolo")
ALTRIMENTI
SE $A_1 < A_2$
ALLORA Scrivi("l'area del secondo triangolo è maggiore dell'area del primo triangolo")
ALTRIMENTI Scrivi("le due aree sono uguali")
FINE

Sopra abbiamo parlato impropriamente di programma in luogo di algoritmo. Questo perché il passaggio dalla *pseudocodifica* di un algoritmo al programma vero e proprio risulta molto semplice; notiamo come a questo vantaggio si contrapponga il fatto che nella *pseudocodifica* il percorso dei dati risulta meno chiaro che nei diagrammi di flusso.

5.2 Linguaggi e programmi

5.2.1 Linguaggi di programmazione

Come già detto, affinché un dato problema possa venire risolto da un elaboratore è necessario che "qualcuno" definisca il relativo algoritmo risolutivo; tale algoritmo è scritto in linguaggio naturale, anche se codificato con i formalismi precedentemente trattati.
Gli algoritmi, per poter essere compresi e quindi eseguiti dai circuiti del computer, devono essere scritti in linguaggio macchina; tale linguaggio è molto lontano dal modo di pensare dell'uomo e molto vicino alla struttura fisica del computer. I dati su cui operare e le istruzioni da eseguire sono stringhe binarie (sequenze di 0 e 1), si capisce quindi come la "traduzione" degli algoritmi in tale linguaggio sia prerogativa di pochi esperti che conoscono a fondo i circuiti della macchina che deve eseguire le operazioni.
Proprio per poter dare alla macchina, con una serie di comandi "vicini" al linguaggio naturale, degli ordini ad essa "comprensibili" sono nati i **linguaggi di programmazione**. Questi sono *linguaggi formali* costituiti da parole in genere non difficili da ricordare (*grammatica* del linguaggio), combinate secondo rigide regole grammaticali (*sintassi* del linguaggio).
Per utilizzare un linguaggio di programmazione bisogna conoscere quindi la sua *sintassi*, la sua *grammatica* e la sua *semantica*.
Da quanto detto risulta chiaro che un **programma** è la **traduzione** di un algoritmo in un **linguaggio di programmazione.**

5.2.2 Evoluzione e tipologia dei linguaggi

Agli inizi degli anni '50 per ovviare alle difficoltà che i programmatori incontravano, dovendo scrivere i loro programmi tenendo conto esattamente dell'hardware della macchina che avrebbe poi dovuto eseguirli, sono nati i *linguaggi simbolici*; tali linguaggi consentono l'uso di un gruppo di parole, dette parole *chiave*, che possono essere combinate secondo una determinata *sintassi*.
Negli ultimi cinquant'anni sono stati fatti passi da gigante, oggi esistono infatti dei linguaggi molto vicini alla sintassi del linguaggio naturale, ma è chiaro che quanto più questi linguaggi risultano comprensibili, tanto più essi si allontanano dal linguaggio macchina.
Per interfacciare i linguaggi simbolici con il linguaggio macchina si sono sviluppati nel tempo dei *programmi* che *traducono* il programma simbolico (*programma sorgente*) in linguaggio macchina (*programma oggetto*); i *programmi traduttori* si suddividono in:

- compilatori
- interpreti

nella parte relativa agli ambienti di sviluppo saranno specificate le loro funzionalità.

I linguaggi di programmazione risultano così catalogati:

1) linguaggio macchina
2) linguaggi assemblativi
3) linguaggi procedurali
4) linguaggi non procedurali

Si è già parlato del linguaggio macchina, al primo livello e quindi molto vicino al computer ma scomodo per il programmatore; i linguaggi assemblativi, situati al secondo livello, presentano una struttura elementare ma possiedono delle sigle che identificano le istruzioni. In questo tipo di linguaggi, per i quali in media una istruzione corrisponde ad una istruzione in linguaggio macchina, si usano riferimenti a componenti fisiche dell'elaboratore; è chiaro che pur essendo in presenza di un linguaggio simbolico si è molto vicini al linguaggio macchina, in questo caso si parla di *linguaggi a basso livello.* Un esempio di linguaggio a basso livello è l'Assembler che usa istruzioni del tipo: MOV A, 7 (scrive 7 nella variabile A), ADD A, 4 (somma 4 alla variabile A); INC A (incrementa A di 1) ecc. I linguaggi a livello 3 , che assieme a quelli di cui al punto 4 sono detti anche *linguaggi evoluti* (o ad *alto livello*) e sono frutto di un progressivo tentativo di avvicinamento del linguaggio macchina al linguaggio naturale, sono nati verso la fine degli anni '50, molto prima di quelli non procedurali, e sfruttano le tecniche della *programmazione strutturata*.
Precisiamo che la *programmazione* si dice *strutturata* quando nella costruzione degli algoritmi, e quindi dei programmi, si fa uso sistematico della *sequenza* della *selezione* e dell' *iterazione*.

I principali programmi procedurali evoluti, tutt'oggi in uso, sono:

- FORTRAN (FORmula TRANslator): nato nel 1956 ed usato per scopi scientifici ed ingegneristici è stato il primo linguaggio ad alto livello

- COBOL (COmmon Businnes Oriented Language): nato nel 1960, ancora in uso per la sua semplicità ed elasticità, per applicazioni commerciali e gestionali

- Pascal: nato nel 1971 in ambiente prettamente scientifico, molto usato anche dal punto di vista didattico per l'uso coerente dei principi della programmazione strutturata

- C: nato nel 1974 sempre in ambito scientifico, con struttura aderente ai principi della programmazione strutturata, è molto diffuso e permette di scrivere programmi per risolvere problemi molto complessi.

Altri linguaggi molto usati furono il LISP, il Prolog, e l'ADA, anche questi sviluppati dal '60 alla fine degli anni '70.
I linguaggi evoluti non procedurali, nati a partire dagli anni '70, non sono linguaggi strutturati nel senso che il flusso dei dati non può essere rappresentato per mezzo delle codifiche sopraesposte; la programmazione ad oggetti ed ad eventi sono sviluppi dei linguaggi non procedurali, in questo tipo di programmi il flusso delle informazioni non è "rigido", "prevedibile" ma varia a seconda degli eventi generati dall'azione dell'utente sugli oggetti.
Esistono versioni ad oggetti del C e del Pascal (rispettivamente C++ e Turbo Pascal versione 5.5); esempio di linguaggio ad eventi è il Visual Basic di linguaggio ad oggetti Power Builder.

5.2.3 Ciclo di vita del software

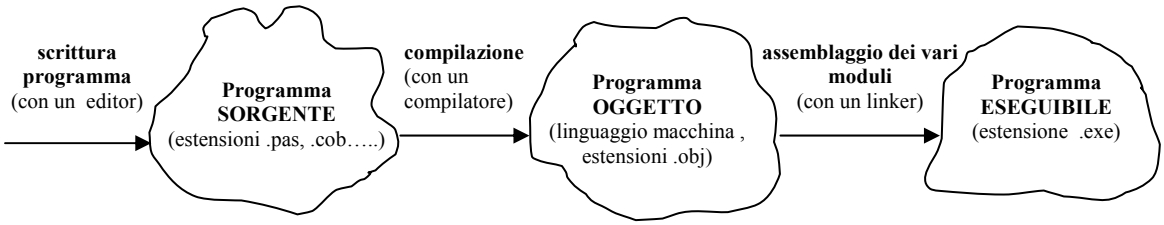
La costruzione dei prodotti informatici non può essere un procedimento casuale, ma deve prevedere delle fasi che costituiscono quello che viene definito il *ciclo di vita* del software; tale ciclo si sviluppa dalla nascita del software per arrivare, come passo estremo, alla sua definitiva eliminazione dal sistema.
Si individuano sette fasi fondamentali:

- lo studio di fattibilità
- l'analisi e la specifica dei requisiti
- il progetto dell'architettura del sistema
- la realizzazione dei singoli moduli
- l'integrazione del sistema
- l'installazione del prodotto
- la manutenzione del prodotto

E' chiaro che a monte della realizzazione di un progetto, anche non specificatamente di un programma, devono essere stimati i costi di sviluppo derivanti dalla realizzazione del prodotto stesso; tali costi devono essere poi confrontati con i benefici ricavabili. Si sottolinea come la fase dello *studio di fattibilità* di un prodotto informatico, presenti delle particolarità che la rendono particolarmente delicata: errori dei programmatori che potrebbero rallentarne lo sviluppo, reale stima della complessità del problema, non chiarezza del committente e conseguente difficoltà di capirne le reali esigenze, sono alcuni tra i molteplici fattori di difficile gestione.
Nella fase di *analisi e specifica dei requisiti* vengono esplicitate le proprietà richieste al programma e specificati gli obiettivi che si vogliono conseguire, fino a portare alla formalizzazione delle funzioni e delle strutture dati che risolvono il problema.
Il *progetto dell'architettura del sistema* prevede la scomposizione del problema generale in problemi più semplici detti *moduli* (metodo *top-down*); vengono quindi *realizzati* separatamente, per mezzo di un qualsiasi linguaggio, i vari *moduli* ed ogni singola componente è controllata e verificata per garantire le specifiche richieste.
Nella fase di *integrazione del sistema* le singole sottoparti vengono assemblate ottenendo il programma completo; la fase di verifica dell'intero sistema si esplicita tramite alcuni controlli effettuati dal produttore (*alfa-test*) e, in seguito, da terze parti (*beta-test*).
L'*installazione* è la fase in cui il sistema viene messo in opera; il programma viene trasportato dall'ambiente di sviluppo del produttore all'ambiente di utilizzazione del cliente; dalla consegna del programma fino alla sua definitiva disinstallazione, vi è una lunga fase che prende il nome di *manutenzione*. Offrire la manutenzione del prodotto significa fornire al cliente assistenza sia per la correzione degli eventuali errori, sia per modifiche che tengano conto delle mutate esigenze dello stesso.

5.2.4 Strumenti di sviluppo

Tutti i **linguaggi di programmazione** sono caratterizzati da un *ambiente di sviluppo*; l'ambiente di sviluppo è costituito da un insieme di programmi che agevolano la scrittura di programmi applicativi e la verifica della loro correttezza (sintattica e strutturale, non logica).
Ogni ambiente di sviluppo conterrà sicuramente un **editor** per scrivere il *programma sorgente*, ossia un programma scritto con istruzioni comprensibile all'utente ma non alla macchina; un **compilatore** si occuperà poi della traduzione del programma sorgente in *programma oggetto*, formato da istruzioni direttamente eseguibili dall'elaboratore.



Per i linguaggi più elementari, il compilatore può essere sostituito da un **interprete** che, traduce il *programma oggetto* direttamente in istruzioni eseguibili dal processore ogni volta che il programma viene eseguito.

Il programma oggetto non può ancora essere eseguito, necessita di essere ancora trattato, "*linkato*", per ottenere il programma eseguibile. Il programma che consente di assemblare i vari moduli si chiama, appunto, *linker*.

Il **debugger** serve ad analizzare il programma nel corso della sua esecuzione verificando passo-passo il risultato delle singole istruzioni.