



## **A.F. 2019 - 2020**

**Operazione Rif. PA 2018 – 11389/Rer**

**Approvata con DGR n.873/2019 del 31/05/2019 cofinanziato dal Fondo Sociale Europeo**

**Progetto 1, edizione 1 ANALISTA PROGRAMMATTORE PER LA MANIFATTURA  
4.0**

**Periodo di svolgimento:** ottobre 2019 – giugno 2020

**Durata:** 600 ore

# **DISPENSE DIDATTICHE**

**Modulo n°: 4**

**Titolo modulo: Fondamenti di programmazione**

**Docente: Claudio Rossi**

**Coordinatrice del corso: Ricci Erika**



Fondazione En.A.I.P. S. Zavatta Rimini  
Viale Valturio, 4 47923 Rimini  
Tel. 0541.367100 – fax. 0541.784001  
[www.enaiprimini.org](http://www.enaiprimini.org); e-mail: [info@enaiprimini.org](mailto:info@enaiprimini.org)



# Cos'è un problema

Esistono molti tipi di problemi, non tutti si prestano ad essere risolti mediante l'uso di computer. Vediamo alcuni tipi di problema:

## Problema aritmetico

$$\begin{array}{r} 3521+ \\ \underline{4768=} \\ ? \end{array}$$

Qual è il risultato dell'operazione?

## Problema crittoaritmetico

MARE +  
SOLE =  
FREDDO

Quali cifre aritmetiche devono essere sostituite ai caratteri per far sì che la somma abbia senso?

## Problema non matematico

Che cravatta abbinare ad una giacca blu e ad un pantalone grigio?

## Problema di programmazione lineare

Un commesso viaggiatore deve visitare un certo numero di città almeno una volta e conosce la distanza tra le varie città. Com'è possibile determinare il percorso più breve?

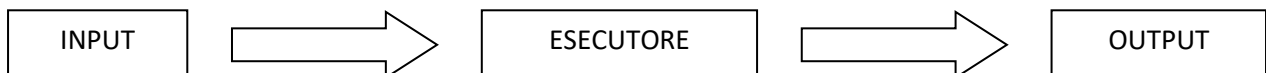
Il numero dei percorsi possibili dipende da  $n!$  per cui nel caso di 20 città i percorsi sono già più svariati miliardi di miliardi

## Risolvere un problema significa:

Individuare e rappresentare un elenco di istruzioni che, interpretate da un esecutore, conducono da un insieme di informazioni iniziali ad un insieme di informazioni finali soddisfacenti un criterio di verifica.

Il risultato è costituito dall'insieme dei dati finali.

La soluzione di un problema è la sequenza di istruzioni che, a partire dai dati di ingresso, permettono di ottenere i risultati.



## Come si risolve un problema

Per risolvere un problema è necessario seguire un procedimento del seguente tipo:

- Definire il problema
- Individuare un criterio di verifica
- Descrivere il problema mediante un modello matematico
- Approssimare il modello mediante un metodo numerico
- Sviluppare l'algoritmo risolutivo ed applicarlo ad uno specifico ambiente

## Un problema è ben formulato se:

- Il criterio di verifica è univoco ed applicabile
- l'insieme dei dati iniziali è completo.

Inoltre non deve essere evidente in partenza la mancanza di soluzioni.

La risoluzione di un problema è fortemente condizionata dall'esecutore disponibile, cioè dall'insieme delle istruzioni che esso è in grado di interpretare

Il **processo risolutivo** è un'azione composta da una sequenza di azioni elementari (**passi**)

## La soluzione di un problema è effettiva per un esecutore se:

- l'esecutore è in grado di interpretare la soluzione
- l'esecutore è in grado di compiere le azioni richieste in un tempo finito

**Un esecutore è caratterizzato da:**

- il linguaggio che è in grado di interpretare
- l'insieme di azioni che è in grado di compiere
- l'insieme delle regole che associano ad ogni costrutto linguistico le relative azioni

**concetto di algoritmo (definizione intuitiva)**

**L'algoritmo è una sequenza finita di istruzioni** univocamente interpretabili da un esecutore che individua una sequenza finita di operazioni elementari, atte a fornire i risultati di una classe di problemi, per qualsiasi valore dei dati in ingresso.

- **sequenza:** insieme in cui è stabilita una relazione d'ordine
- **istruzioni:** comandi interpretabili da un esecutore che li trasforma in azioni
- **non ambigue:** ciascuna istruzione è trasformata in un insieme di operazioni certamente definibile, noto il contesto in cui si opera
- **operazioni:** azioni elaborative ottenute dall'interpretazione delle istruzioni
- **problema:** quesito che attende una risposta
- **classe di problemi:** insieme di problemi della medesima tipologia

I concetti di **algoritmo** ed **esecutore** sono strettamente connessi.

Una sequenza di istruzioni costituisce o meno un algoritmo a seconda dell'esecutore su cui lavora.

La **progettazione** e la **descrizione** di un algoritmo dipendono dalle capacità dell'esecutore.

**L'esecutore** interpreta ed esegue nel medesimo modo una stessa istruzione a parità di condizioni al contorno.

Se l'algoritmo può anche **non terminare**, si parla di **metodi computazionali**: sono ammesse esecuzioni con un numero infinito di passi.

# La macchina di Turing

Il matematico inglese **Alan Turing** ha formalizzato la nozione di algoritmo descrivendo in maniera formale un esecutore, detto **macchina di Turing** (MdT).

La **MdT** è una Terna

$$Z=(A,Q,P)$$

dove:

**A** è l'alfabeto finito dei simboli leggibili/scrivibili

**Q** è l'insieme finito degli stati dell'organo di controllo

**P** è l'insieme finito delle quintuple che ne descrivono il funzionamento.

Cioè, dato

- uno stato iniziale
- un carattere letto

siamo in grado di stabilire

- lo stato successivo
- il carattere scritto
- la direzione della testina di lettura/scrittura

Avendo la nozione di Macchina di Turing possiamo rispondere alle seguenti domande:

- Quando un problema è risolvibile?
- Quando una funzione è effettivamente calcolabile?

## Tesi di CHURCH

Ogni funzione calcolabile è Turing-calcolabile cioè può essere eseguita su una macchina di Turing.

In definitiva i problemi possono essere

- **solvibili**: risolvibili da MdT o da procedure effettive
- **insolvibili**

## Caratteristiche che deve possedere un esecutore

- capacità di comunicare con l'esterno (operazioni di I/O)
- capacità di conservare informazioni (memorizzazione di dati ed istruzioni)
- capacità di effettuare operazioni logico-aritmetiche
- capacità di interpretare ed eseguire un algoritmo, decodificando ciascuna istruzione ed individuando quale sarà la prossima istruzione da eseguire.

Gli algoritmi applicati ad un calcolatore vengono detti **programmi**

### Esempio l'algoritmo per la determinazione del massimo comune divisore di due numeri

- scomporre i due numeri in fattori primi
- moltiplicare tra loro i fattori comuni presi una sola volta con il minimo esponente.

Ciò comporta che l'**esecutore sappia scomporre** i numeri in fattori primi, sappia individuare i fattori comuni con il minimo esponente, sappia effettuare la moltiplicazione.

## Algoritmo di Euclide

### Problema

Determinare il massimo comune divisore di due numeri naturali diversi da 0.

### Ipotesi

L'esecutore è in grado di:

- effettuare la divisione,
- individuare il resto della divisione,
- effettuare operazioni logiche,
- stabilire quale è la prossima istruzione da eseguire.

Chiamiamo A il primo numero, B il secondo ed r il resto.

L'algoritmo di Euclide è il seguente:

- 1) dividi il primo numero per il secondo;
- 2) chiama r il resto della divisione.
- 3) se  $r = 0$  il secondo numero è il MCD cercato
- 4) altrimenti
- 5) primo numero  $\longleftarrow$  secondo numero
- 6) secondo numero  $\longleftarrow$  r
- 7) ritorna alla prima istruzione

Scriviamo l'algoritmo di Euclide con un linguaggio detto di pseudoprogrammazione, ossia un linguaggio a metà strada fra quello umano e quelli di programmazione:

ripeti

$r = A \bmod B$

    se  $r = 0$  allora

        scrivi (" il MCD è", B)

    altrimenti

$A = B$

$B = r$

    fine se

finché  $r \neq 0$  ;

# Elementi di programmazione imperativa

## Caratteristiche generali di un programma

- Un programma permette di acquisire dati (Input), elaborarli, produrre risultati (Output).
- Un programma in esecuzione necessita di un processore, in grado di interpretare le istruzioni e di eseguirle, e di una memoria in grado di conservare dati ed istruzioni per il tempo necessario all'esecuzione.
- I programmi sono costituiti da istruzioni, ciascuna con una propria sintassi ed una propria semantica, cui corrisponde l'azione da eseguire.
- I dati possono essere di vari tipi.
- Ciascun tipo di dato è individuato da un nome, dall'insieme dei valori che può assumere e dalle operazioni possibili.
- Ad esempio *giallo*, *rosso*, *blu* sono dei valori del tipo *colore* e *Mescola* (a,b) è una delle operazioni possibili.

## Costanti e variabili

- I dati utilizzati in un programma possono essere costanti o variabili.
- Una costante è un numero, una scritta, oppure un nome simbolico il cui valore non varia nel corso dell'esecuzione del programma.
- Una variabile è un nome simbolico cui è associato un tipo ed un valore variabile nel corso del programma.
- Variabili e costanti sono memorizzate in memoria centrale durante l'esecuzione del programma e possono occupare una o più locazioni di memoria a seconda della loro tipologia.

### Esempio di dichiarazione di variabili

Var i:integer; La variabile i è intera e può assumere i valori interi compresi tra -32768 e 32767. Occupa due locazioni di memoria da 8 bit.

Var Cognome:string[20]; La variabile Cognome è una stringa di 20 caratteri che possono assumere qualsiasi valore tra le 256 configurazioni del codice ASCII. Occupa 20 locazioni di memoria.

Var x,y:real; Le due variabili x e y sono di tipo reale, cioè possono assumere valori interi o decimali. Occupano una quantità di byte variabile a seconda del linguaggio di programmazione utilizzato, memorizzano mantissa ed esponente con una notazione detta floating point.

## Assegnazione

L'operazione di assegnazione permette di memorizzare nella variabile presente nel primo membro il valore dell'espressione presente nel secondo membro, ottenuto sostituendo a ciascuna variabile eventualmente presente il valore corrispondente ed effettuando le operazioni necessarie.

### Esempio

I =1; I vale 1, cioè in memoria centrale le locazioni corrispondenti alla variabile I contengono il valore 1.

I =I+1; I valeva 1; dopo questa assegnazione I vale 2;

J =5; J vale 5;

K =(I+J)\*(I+1); K vale (2+5)\*(2+1), cioè 21.

## Scomposizione di un problema

Per risolvere un problema è opportuno scomporlo in sottoproblemi secondo la metodologia top-down, le procedure e le funzioni sono utili a questo scopo.

Procedure e funzioni sono sottoprogrammi che possono essere richiamati ricevendo parametri di input, elaborando i dati e restituendo, al termine dell'esecuzione, parametri di output al programma chiamante.

La scomposizione può avvenire secondo tre modalità di base:

- 1) sequenziale
- 2) condizionale
- 3) iterativa.

### Scomposizione sequenziale

Il problema viene scomposto in una sequenza di sottoproblemi, le istruzioni corrispondenti vengono eseguite in ordine sequenziale, una dopo l'altra.

#### Esempio

```
...  
Disegna_testa  
Disegna_corpo  
Disegna_arti  
....
```

La realtà esterna, l'evolversi dei dati non modifica l'ordine di esecuzione delle istruzioni.

#### Esempio

Calcolare la somma di due numeri interi.

- 1) Leggi a
- 2) Leggi b
- 3) Somma a + b
- 4) Scrivi Somma

### Scomposizione condizionale

Il problema viene scomposto in due sottoproblemi alternativi tra loro.

Per stabilire quale dei due percorsi risolutivi bisogna seguire ci si avvale di proposizioni, cioè di espressioni linguistiche delle quali si può valutare la verità, e più specificamente di proposizioni variabili (o predicati) cioè di affermazioni la cui verità dipende dai valori delle variabili contenute. Sinteticamente si parla di condizioni.

#### Esempio

```
leggi(a);  
if a >= 0 then scrivi("a è positivo")  
else scrivi("a è negativo");
```

### Scomposizione iterativa (ripetitiva)

Il problema è scomposto in un insieme di operazioni che possono essere ripetute più volte (ciclo), sino al verificarsi di una certa condizione (predicato di controllo).

In quasi tutti i linguaggi di programmazione sono disponibili tre costrutti sintattici per eseguire ciclicamente un blocco strutturato di istruzioni:

```
While...,  
For..,  
Repeat.. Until.
```



## Il costrutto while

Prevede l'esecuzione ciclica di un insieme (blocco strutturato) di istruzioni se la condizione (guardia) è vera.

La condizione può anche essere composta.

Se la condizione è falsa il programma salta il blocco di istruzioni contenuto nel ciclo while e continua con la prima istruzione successiva.

Il while necessita di alcuni elementi minimi per poter funzionare correttamente:

- 1) la presenza di una condizione che contenga almeno una variabile di controllo (senza variabili il ciclo continua all'infinito o non inizia mai).
- 2) la memorizzazione di un valore nella variabile di controllo prima di testare la condizione
- 3) la modifica del valore della variabile di controllo all'interno del ciclo, per evitare che il ciclo continui all'infinito.

Il costrutto while è molto flessibile in quanto non pone limiti al numero di cicli eseguibili, al tipo di condizione, al numero delle variabili di controllo ed alle relative modalità di modifica.

È particolarmente adatto a quelle tipologie di problemi in cui non si conosce a priori il numero di cicli che bisogna eseguire.

### Esempio

Somma di N numeri positivi immessi da tastiera.

(Zero per terminare)

Somma = 0;

leggi(Numero);

while Numero <> 0 do

    Somma = Somma + Numero;

    leggi(Numero);

end while;

scrivi(Somma);

## Il costrutto repeat-until

È molto simile al costrutto sintattico while ma la condizione viene valutata alla fine del ciclo, per cui le istruzioni cicliche vanno eseguite almeno una volta.

Al contrario del costrutto while, se il predicato di controllo è vero si esce dal ciclo, altrimenti si continua l'iterazione.

Esempio

Somma = 0;

repeat

    leggi(Numero);

    Somma = Somma + Numero;

until Numero <> 0;

scrivi(Somma);

## Il costrutto for

Questo costrutto consente di effettuare un determinato numero di volte un blocco di istruzioni (ciclo).

È adatto a situazioni in cui è noto a priori il numero di cicli da eseguire.

Esempio

for i = 1 to n do

    leggi(Num);

    Somma = Somma + Num;

end;

## **Istruzioni di input e di output**

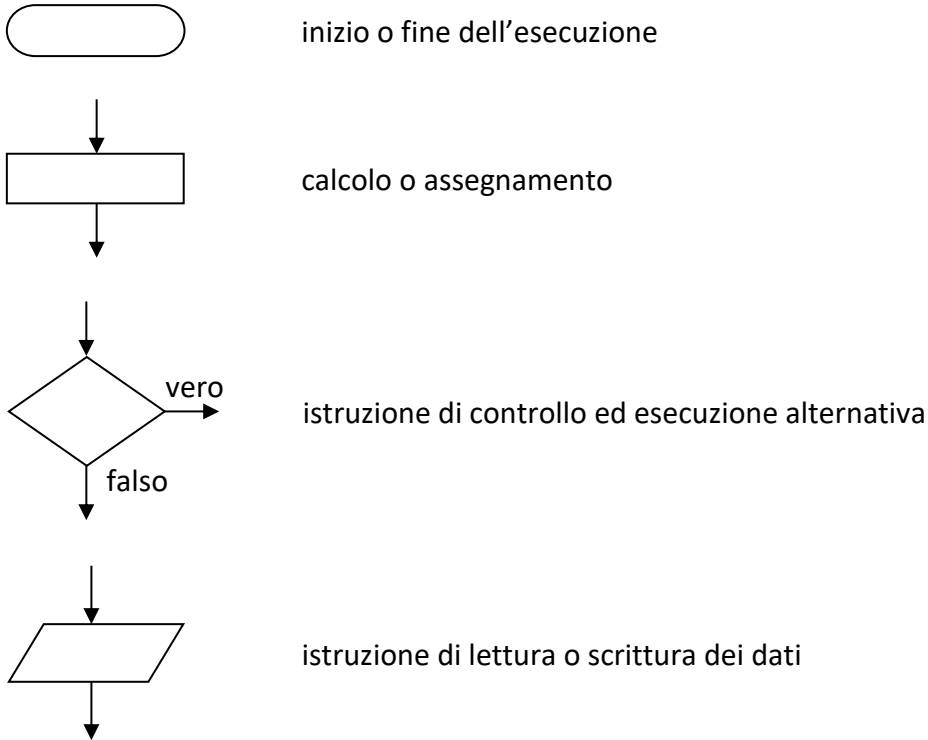
Permettono di acquisire dati o di trasferirli all'esterno.

La sintassi dipende dalle periferiche che intendiamo utilizzare (tastiera, monitor, stampante).

# Diagrammi a blocchi

Per rappresentare graficamente un algoritmo è stato standardizzato un metodo chiamato **diagramma a blocchi**.

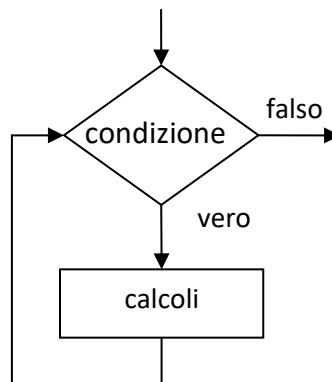
In un diagramma a blocchi si rappresenta ogni istruzione con un elemento geometrico e si rappresenta il flusso delle operazioni (ossia l'ordine in queste vengono eseguite) collegando gli elementi con degli archi orientati (linee che terminano con una freccia).



Nei diagrammi a blocchi non si ha la fase di dichiarazione delle variabili, infatti manca l'equivalente della istruzione di dichiarazione, questo non è importante dato che i diagrammi servono a visualizzare il **flusso** dell'esecuzione (infatti si chiamano anche diagrammi di flusso) e non a rappresentare esattamente un programma.

Non esiste nessun blocco particolare per indicare le strutture di iterazione; si usano semplicemente dei blocchi di selezione in testa o in coda al gruppo di istruzioni facenti parte del ciclo.

Esempio: ciclo while



# Metodologia di scomposizione top-down

Per risolvere un problema complesso è utile individuarne i diversi aspetti in maniera generale prima di passare allo sviluppo di dettaglio.

È particolarmente vantaggioso riuscire a scomporre un problema in sottoproblemi, eventualmente a loro volta scomponibili, in modo da poter affidare le varie fasi della soluzione anche a gruppi di lavoro diversi, questo tipo di scomposizione prende il nome di top-down.

La metodologia top-down richiede particolare attenzione nell'individuazione dei sottoproblemi, nella progettazione dei sottoprogrammi che li risolvono e nei rapporti tra sottoprogrammi.

Le procedure e le funzioni della maggior parte dei linguaggi di programmazione permettono di realizzare tali sottoprogrammi.

Ogni sottoprogramma è caratterizzato da una fase elaborativa che permette di ottenere gli output desiderati a partire dagli input inseriti.

## Esempio

Vogliamo calcolare l'area di figure geometriche, costituite da un tipo (cerchio, quadrato,...) e dalle misure dei lati.

La figura geometrica va vista come un insieme di parti, un rettangolo può essere visto, ad esempio, come un insieme di elementi costituito da "tipo=rettangolo; base=20; altezza=12".

Program Calcola\_Area

...

for i =1 to n do

Leggi\_Dati(tipo, lati);

Calcola\_Area(tipo, lati, area);

Stampa\_Area(area)

...

end.

## Osservazioni

Alcune parole riservate (come while, end, for) assumono un significato fisso definito nel linguaggio.

Altre parole (Leggi\_Dati, Calcola\_Area, tipo, lati, area) assumono il significato che l'utente definisce nel corso del programma specifico.

tipo, lati, area sono variabili, contengono un valore che può variare nel corso del programma.

Alcune istruzioni vanno eseguite in sequenza, una dopo l'altra, altre istruzioni vanno eseguite ripetutamente fino a quando non si verifica una determinata condizione

## Programmi e sottoprogrammi

I programmi possono essere suddivisi in sottoprogrammi.

Ciascun sottoprogramma può essere richiamato dal programma principale con il proprio nome.

Ogni programma può chiamare al suo interno uno o più sottoprogrammi cui sono delegate particolari funzioni.

...

acquisisci(a,b);

elabora(a,b,c);

visualizza(c);

....

Ciò aumenta notevolmente la flessibilità e la leggibilità del programma.

L'utilizzo ripetuto di un sottoprogramma evita la ripetizione delle singole istruzioni che lo compongono semplificando la struttura del programma chiamante e la manutenzione del codice (eventuali errori nella sequenza di istruzioni si devono correggere una sola volta).

Il sottoprogramma si comporta come una scatola nera che si relaziona con il programma chiamante tramite i parametri, riceve in input dal programma chiamante un insieme di valori costanti o variabili, i

parametri di input.

Il sottoprogramma esegue le sue fasi elaborative e può assegnare valori opportuni ad un insieme di variabili, i parametri di output, che vengono rese disponibili al programma chiamante.

La scomposizione di un programma in sottoprogrammi è molto utile nella progettazione perché permette di razionalizzare il lavoro sviluppandolo in tempi diversi ed assegnandolo anche a team distinti, a condizione che siano ben definiti i significati dei parametri di input e di output.

## Parametri

I parametri sono la maniera di comunicare del sottoprogramma con il mondo esterno.

Alla fine della sua esecuzione il sottoprogramma trasferisce il controllo al programma chiamante.

### Esempio

Programma principale

...

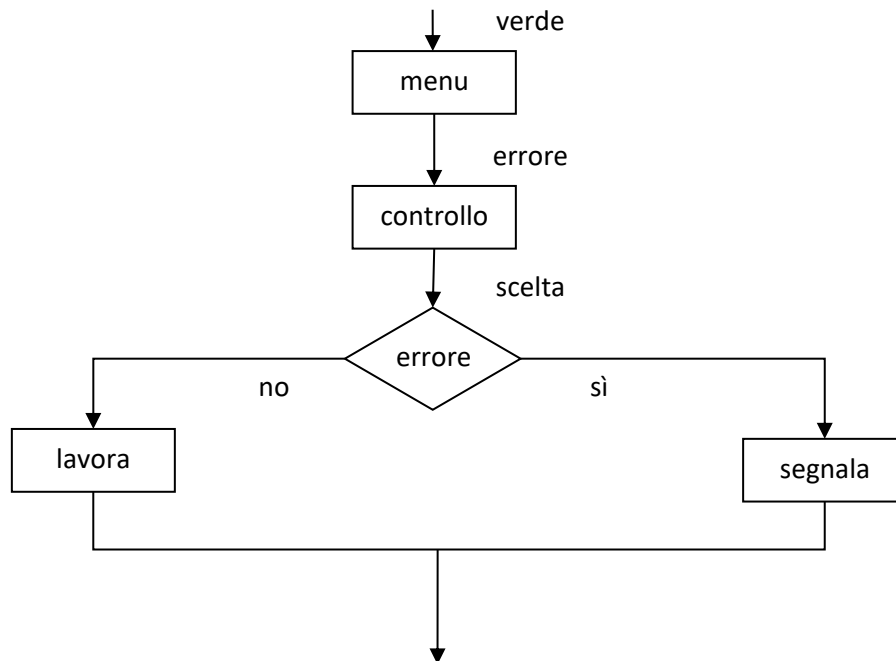
Menu (Verde,Scelta);

Controllo(Errore,Scelta);

If Errore=true then Segnala(Errore)

else Lavora (Scelta);

...



## Procedure e funzioni

In quasi tutti i linguaggi di programmazione sono presenti due tipi di sottoprogrammi: procedure e funzioni.

Le procedure comunicano con il programma chiamante esclusivamente tramite i parametri.

### Esempio di utilizzo di procedure

Visualizza(a,b);

Anche le funzioni comunicano tramite parametri ma possono anche associare al proprio nome un valore che può essere direttamente utilizzato nel programma chiamante.

### Esempio di utilizzo di funzione

a =random(n);

scrivi(random(n));

# Ricorsione

Si chiamano *ricorsive* le procedure o funzioni che, direttamente o indirettamente, richiamano se stesse; si dice anche che si sta usando la tecnica della ricorsione.

Alcuni problemi di natura matematica si prestano all'implementazione ricorsiva perché sono per loro natura ricorsivi, un ottimo esempio è la funzione fattoriale.

Ricordiamo che il fattoriale di un numero  $n$  si indica con  $n!$  e significa  $n*(n-1)*(n-2)*...*1$ ; per convenzione si pone  $0! = 1$

## Esempio: algoritmo per il calcolo del fattoriale

È facile scrivere l'algoritmo per calcolare  $\text{fatt}(n)$ , cioè  $n!$ , usando un "for":

```
Leggi(n);  
fattoriale = 1;  
for i = 1 to n  
    fattoriale = fattoriale * i  
end
```

Vediamo ora come calcolare  $n!$  usando la ricorsione

Per risolvere il problema consideriamo la seguente eguaglianza:  $(n+1)! = (n+1) * n!$

Questa uguaglianza può essere utilizzata per dare una definizione ricorsiva della funzione fattoriale:

|         |   |
|---------|---|
| [base]  | $\text{fatt}(0) = 1$                        |
| [passo] | $\text{fatt}(n+1) = (n+1) * \text{fatt}(n)$ |

Che possiamo tradurre nella seguente funzione scritta in pseudocodice

Funzione  $\text{fatt}(n : \text{integer}) : \text{integer}$

```
    if n=0 then  
        return 1  
    else  
        return (n * fatt (n-1))  
    end
```

end

Per capire come procede l'esecuzione di una chiamata, supponiamo che in un programma ci sia l'istruzione `scrivi( fatt (3))`.

L'esecuzione procede così:

$\text{fatt}(3) \rightarrow 3 * \text{fatt}(2) \rightarrow 3 * 2 * \text{fatt}(1) \rightarrow 3*2*1* \text{fatt}(0) \rightarrow 3*2*1*1 \rightarrow \text{scrivi } 6$

Il fattoriale è un esempio di ricorsione diretta: nel corpo della funzione `fatt` compare una chiamata a `fatt`.

Il punto chiave è che il nome "fatt" è noto all'interno del corpo della funzione `fatt` e si riferisce alla funzione stessa.

## Ricorsione indiretta e incrociata

Finora abbiamo parlato di ricorsione diretta: una funzione o procedura contiene, nel corpo, una o più chiamate a se stessa.

Si può anche avere una ricorsione indiretta:

la procedura  $P$  chiama direttamente la procedura  $Q$ , che chiama direttamente la procedura  $R$ ,....., che chiama la procedura  $P$ .

È anche possibile la ricorsione incrociata, nel caso più semplice:  $P$  chiama  $Q$  e  $Q$  chiama  $P$ .

# Dati e codifica

I dati vengono rappresentati da sequenze di simboli, scelti da un insieme finito, detto alfabeto.

Ad ogni alfabeto va associato un insieme di regole di composizione

Esempio: rappresentazione di un numero nel sistema di numerazione decimale

153 significa

$$1 \times 100 + 5 \times 10 + 3 \times 1$$

in quanto il sistema di numerazione decimale è posizionale.

## Alfabeto binario

I computer utilizzano segnali elettrici per rappresentare le informazioni, questo porta come logica conseguenza l'utilizzo della base 2 come base di numerazione.

L'elemento base è detto **bit** e può assumere due soli valori: 0 oppure 1

0 è un messaggio di un bit

1 è un messaggio di un bit

0100 è un messaggio di quattro bit

01001101 è un messaggio di 8 bit (byte)

Con  $k$  bit si possono ottenere  $2^k$  configurazioni diverse, ad esempio, con 4 bit si possono ottenere 16 configurazioni diverse.

Il numero di simboli (bit) necessario per codificare un messaggio in alfabeto binario è maggiore rispetto ad un alfabeto con più simboli.

Ad esempio

$14_{10}$  richiede due cifre decimali

il corrispondente binario 1110 richiede quattro cifre binarie (bit).

In generale il numero di bit necessario a codificare un insieme costituito da  $n$  diversi messaggi è dato da

$$b = \log_2 n$$

Se il valore ottenuto non è un numero intero si arrotonda per eccesso.

## Elaboratori ed alfabeto binario

I computer trattano l'informazione in maniera discreta, cioè considerano significativi solo alcuni valori delle grandezze fisiche trattate, quindi quando si digitalizza un'informazione analogica (suono, immagine,...) si ottiene sempre una approssimazione.

Il byte è significativo perché ad esso è comunemente associato un carattere nei codici di uso più comune, come il codice ASCII ed il codice EBCDIC. Ad esempio, nel codice ASCII al carattere *a* corrisponde 0110 0001

## Codifica dei dati non numerici

Un singolo bit consente di distinguere i dati in due sottoinsiemi.

Più bit consentono di distinguere i dati in più sottoinsiemi.

Esempio

$S$  = cuori, quadri, fiori, picche

La cardinalità (numero di elementi) di  $S$  è 4

Per rappresentare  $S$  sono necessari almeno 2 bit ( $2^2=4$ )

Ad esempio possiamo codificare le informazioni nella seguente maniera:

|        | primo bit | secondo bit |
|--------|-----------|-------------|
| cuori  | 0         | 0           |
| quadri | 0         | 1           |

|        |   |   |
|--------|---|---|
| fiori  | 1 | 0 |
| picche | 1 | 1 |

Codice ASCII (American Standard Code For Information Interchange)

Utilizza 8 bit, quindi consente di rappresentare  $2^8=256$  caratteri

In particolare rappresenta:

- 26 lettere dell'alfabeto maiuscole
- 26 lettere dell'alfabeto minuscole
- 10 cifre decimali
- 30 simboli di interpunzione e di uso corrente
- caratteri speciali.

Codice EBCDIC

Anch'esso utilizza 8 bit

È stato sviluppato da IBM

Codice UNICODE

Utilizza 16 bit, quindi è in grado di rappresentare  $2^{16}=65536$  diversi caratteri.

È utile per rappresentare diversi alfabeti (arabo, greco, matematico, grafico).

I primi 128 caratteri coincidono con quelli del codice ASCII (i 7 bit meno significativi).

## La codifica delle istruzioni

Le istruzioni sono interpretate ed eseguite da un elaboratore.

Il linguaggio macchina è direttamente interpretabile dall'elaboratore ed è costituito da istruzioni basate su un alfabeto binario.

### Esempio

01100000 significa ADD (somma)

Ad ogni istruzione è associato un codice univoco, detto codice operativo.

Spesso le istruzioni sono costituite da un codice operativo e da operandi.

### Esempio

Se voglio sommare il contenuto due locazioni di memoria l'istruzione sarà costituita da :

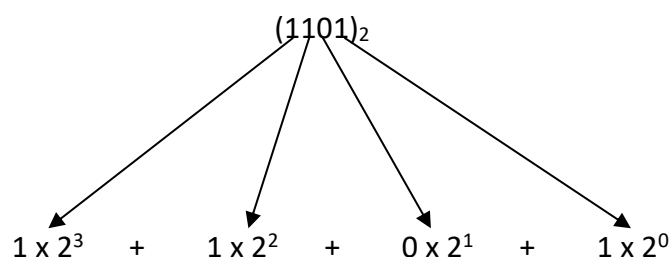
- codice operativo;
- indirizzo del primo operando;
- indirizzo del secondo operando.

Ogni microprocessore ha il proprio set di istruzioni e proprie regole di codifica, questo significa che un programma scritto per un particolare tipo di processore non può funzionare su processori che hanno set di istruzioni diversi.

## La codifica di dati numerici

Per codificare i dati numerici servono tre cose:

- una base di riferimento (decimale, binario, esadecimale, ottale);
- un insieme di regole di codifica;
- uno o più sistemi di numerazione posizionali.





Per gli esseri umani la base di riferimento è decimale, per gli elaboratori è quella binaria. I codici esadecimale ed ottale si usano perché rendono sinteticamente la notazione utilizzata dagli elaboratori (ogni cifra rappresenta rispettivamente 2B e 1B).

### Conversione da binario a decimale

Per convertire un numero binario in uno decimale, è sufficiente moltiplicare ogni cifra per la base elevata alla posizione occupata dalla cifra e sommare i valori così ottenuti.

#### Esempio

Cerchiamo il valore decimale del numero binario 1101

$$\begin{array}{cccc}
 1 & 1 & 0 & 1 \\
 2^3 & 2^2 & 2^1 & 2^0 \\
 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 = 13 \\
 1101_2 = 13_{10}
 \end{array}$$

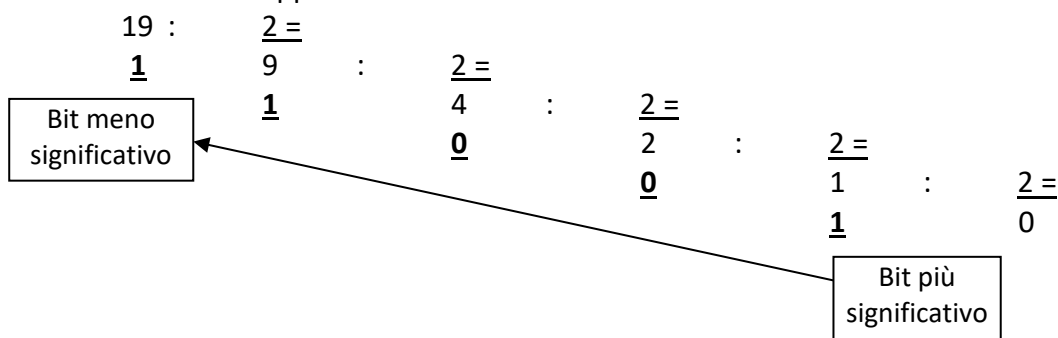
### Conversione da decimale a binario

Per convertire un numero decimale in uno binario, bisogna dividere il numero per 2, prendere il risultato ottenuto e dividerlo per 2, continuare con il procedimento fino a quando il risultato della divisione è zero.

Il numero binario cercato si ottiene leggendo i resti delle divisioni.

#### Esempio

Cerchiamo la rappresentazione binaria del numero decimale 19



$$19_{10} = 10011_2$$

Infatti

$$1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 19$$

### Codifica esadecimale

È una notazione sintetica comoda per rappresentare i byte, ciascuna cifra esadecimale corrisponde ad un gruppo di quattro bit, quindi due cifre rappresentano esattamente un byte.

L'alfabeto esadecimale è rappresentato nella seguente tabella

| cifra<br>esadecimale | valore decimale |
|----------------------|-----------------|
| 0                    | 0               |
| 1                    | 1               |
| 2                    | 2               |
| 3                    | 3               |
| 4                    | 4               |
| 5                    | 5               |
| 6                    | 6               |
| 7                    | 7               |
| 8                    | 8               |
| 9                    | 9               |
| A                    | 10              |

|   |    |
|---|----|
| B | 11 |
| C | 12 |
| D | 13 |
| E | 14 |
| F | 15 |

### Esempio

$$237_{10} = 11111100_2 = FC_{16}$$

$$125_{10} = 01111101_2 = 7D_{16}$$

### Codifica ottale

Anche questa è una notazione che viene utilizzata perché è sintetica, ciascuna cifra ottale corrisponde ad un gruppo di 3 cifre binarie

### Codifica binaria di numeri interi con segno

Finora abbiamo visto solamente conversione di numeri positivi, ovviamente nell'utilizzo pratico è necessario codificare anche numeri negativi.

In binario è possibile codificare segno e modulo, il bit più significativo è riservato al segno

- 0 segno positivo
- 1 segno negativo

gli altri bit sono riservati al modulo (valore del numero senza segno).

### Esempio

0100 significa +4

1100 significa -4

Utilizzando questa rappresentazione, con n bit si possono rappresentare i numeri da

$$-(2^{n-1} - 1) \text{ a } (2^{n-1} - 1)$$

NB: zero è rappresentato due volte (+0 e -0).

### Codifica binaria di numeri razionali

In questo caso siamo di fronte ad un problema molto grande: il numero di cifre decimali può essere superiore al numero di cifre rappresentabili o addirittura infinito!

Per cercare di ovviare al problema vengono memorizzate solo le cifre più significative.

Un altro problema è la necessità di rappresentare numeri con valore assoluto molto grande o molto piccolo, per risolvere questo problema si usa la rappresentazione a virgola mobile (notazione scientifica).

$$\pm \text{mantissa} * \text{base}^{\text{esponente}}$$

### Esempio

750 si scrive come  $0,75 * 10^3$

Esiste uno standard di codifica (altrimenti sarebbe delirante cercare di far comunicare fra loro sistemi diversi)

Esempio: rappresentazione con 64 bit

|                |                     |                    |
|----------------|---------------------|--------------------|
| 1 bit<br>segno | 11 bit<br>esponente | 52 bit<br>mantissa |
|----------------|---------------------|--------------------|

Se si vuole rappresentare un numero maggiore del massimo rappresentabile si verifica un errore detto di overflow.

Se si vuole rappresentare un numero minore del minimo rappresentabile si verifica una situazione detta di underflow.

In tutti gli altri casi la rappresentazione è possibile ma può verificarsi la necessità di arrotondamenti, ogni arrotondamento introduce un errore di cui va valutata la tollerabilità.

# I dati

Quando si lavora con i linguaggi di programmazione, le variabili sono caratterizzate da nome, valore e tipo.

## Esempio

La dichiarazione `var Contatore: integer;`

e l'istruzione `Contatore = 1;`

indicano una variabile di nome `Contatore`, di tipo intero, contenente il valore 1.

Alcuni tipi di dato sono predefiniti, altri possono essere definiti dall'utente.

Un tipo di dato è caratterizzato da:

- un nome (ad esempio `integer`)
- un insieme dei valori ammissibili (  $-32768 < x < 32767$  )
- un insieme delle operazioni possibili ( `-`, `+`, `/`, `div`, `*` ).

Vediamo nel seguito i tipi di dato più utilizzati.

## Il tipo boolean

Può assumere due valori: `true` oppure `false`

I principali operatori booleani sono:

| AND   | true  | false |
|-------|-------|-------|
| true  | true  | false |
| false | false | false |

| OR    | true | false |
|-------|------|-------|
| true  | true | true  |
| false | true | false |

| NOT   |       |
|-------|-------|
| true  | false |
| false | true  |

## Esempio

`if (x > 0) and (x < 20) then....`

è vera per i valori di `x` interni all'intervallo di estremi 0 e 20 (estremi esclusi).

`if ( x < 10 ) or (x > 50 ) then..`

è vera per i valori di `x` esterni all'intervallo di estremi 10 e 50 (estremi esclusi)..

`if (i <= n) and (ok = false) then...`

è vera se si verificano entrambe le condizioni, cioè se `i <= n` e `ok = false`.

## Tipi strutturati

Sono tipi di dato composti da più elementi anche di diverso tipo.

Un array individua un insieme di elementi **dello stesso tipo** individuabili tramite la loro posizione.

Un record individua un insieme di elementi **di qualsiasi tipo** individuabili tramite il loro nome.

Un array unidimensionale (vettore) è un tipo di dato strutturato caratterizzato da un nome e da un numero di componenti, tutti dello stesso tipo, individuabili tramite un solo indice.

## Esempio

`type Vettore: array [1..5] of integer;`

dichiara un vettore di 5 elementi di tipo intero.

Esistono anche gli array multidimensionali, gli elementi di un array multidimensionale si identificano tramite più indici.

# Complessità

Un algoritmo deve essere eseguito rapidamente, il tempo di esecuzione dipende dalla velocità dell'esecutore, dal numero e dal tipo di istruzioni eseguite, il numero di istruzioni da eseguire dipende dai dati da elaborare.

A parità di esecutore, di tipologia di istruzioni e di dati di input per migliorare le prestazioni di un algoritmo bisogna minimizzare le istruzioni da eseguire.

Per esempio, la ricerca completa ha una complessità di tempo che dipende da  $N/2$ , mentre la ricerca dicotomica ha una complessità di tempo che dipende da  $\log_2 N$ , la ricerca dicotomica è più efficiente della ricerca completa, questo significa che a parità di esecutore sarà sempre più veloce.

L'efficienza di un algoritmo dipende da:

- Numero di operazioni logico/aritmetiche effettuate
- Spazio di memoria richiesto dai dati

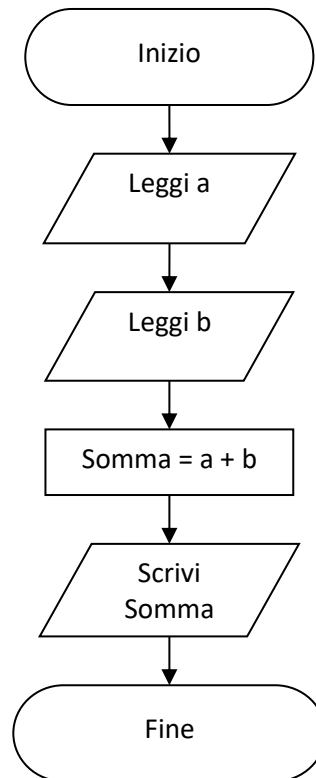
La riduzione del numero di operazioni non permette solo di diminuire il tempo di esecuzione ma anche di ridurre le approssimazioni proprie di alcune operazioni.

# Esempi di scomposizione

## Scomposizione sequenziale

### Somma di due numeri interi.

Leggi a  
Leggi b  
 $Somma = a + b$   
Scrivi somma



### Somma di n numeri.

#### Prima versione

Leggi  $a_1$   
Leggi  $a_2$   
...  
Leggi  $a_n$   
 $Somma = a_1 + a_2 + \dots + a_n$   
scrivi Somma

#### Seconda versione

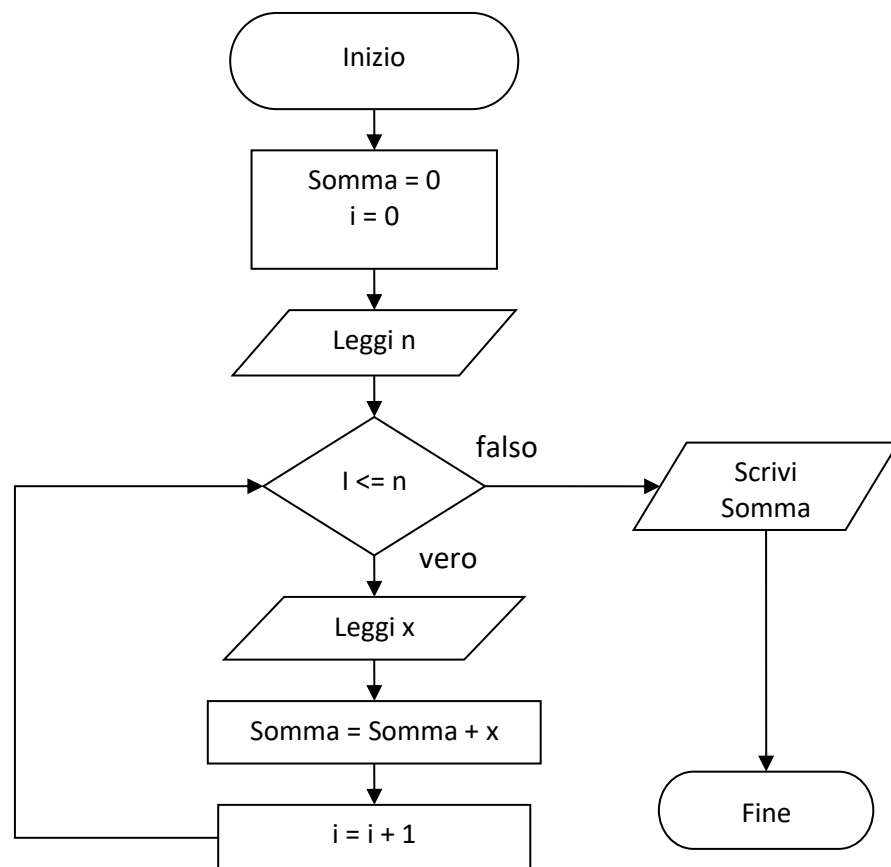
$Somma = 0$   
Leggi  $a_1$   
Leggi  $a_2$   
...  
Leggi  $a_n$   
 $Somma = Somma + a_1$   
 $Somma = Somma + a_2$   
...

Somma = Somma +  $a_n$   
scrivi Somma

## Scomposizione iterativa

### Somma di n numeri

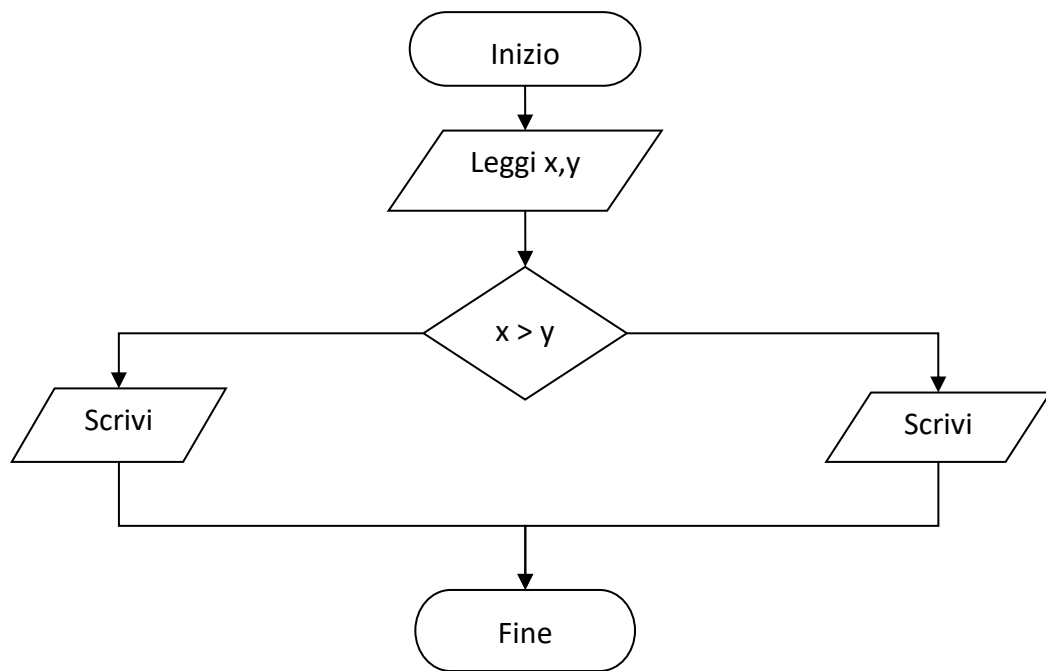
```
i = 1
Somma = 0
Leggi n
finché i <= n ripeti
    Leggi x
    Somma = Somma + x
    i = i+1
fine ciclo
scrivi somma
```



## Scomposizione condizionale

### Determinare il valore massimo tra due numeri x e y

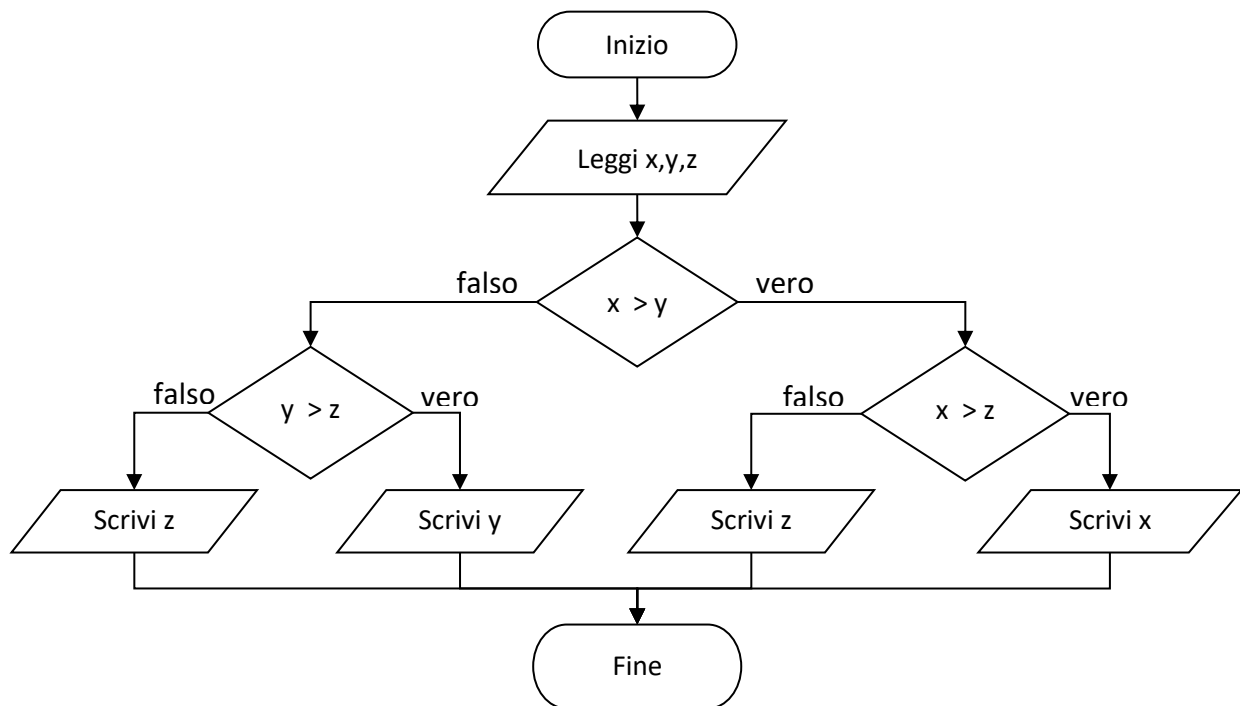
```
Leggi x
Leggi y
se x > y allora
    scrivi x
altrimenti
    scrivi y
fine se
```



**Determinare il valore massimo tra 3 numeri x,y,z.**

```
Leggi x,y,z
se x > y allora
    se x > z allora
        scrivi x
    altrimenti
        scrivi z
    fine se
altrimenti
    se y > z allora
        scrivi y
    altrimenti
        scrivi z
    fine se
fine se
```







# Ciclo di vita del software

Il ciclo di vita del software è la scomposizione delle attività legate alla produzione di un software. Le fasi principali che caratterizzano il ciclo di vita di un software sono:

- analisi
- progettazione
- implementazione
- collaudo
- rilascio
- manutenzione

## Analisi del software

L'analisi del software è la prima attività dello sviluppo e consiste nello studio del contesto applicativo nel quale opererà il software e delle caratteristiche che questo deve possedere.

Una parte fondamentale dell'attività di analisi è l'interazione con il cliente per definire le caratteristiche che il software dovrà possedere, a tal fine è necessario realizzare una o più interviste con il personale del cliente al fine di definire ogni possibile aspetto rilevante (dall'aspetto estetico delle interfacce agli aspetti di usabilità richiesti).

Il fine del processo di analisi è quello di creare un documento (detto *documento di specifiche funzionali*) che contiene tutte le caratteristiche individuate durante l'analisi, normalmente questo documento viene sottoposto al committente e fatto firmare per l'accettazione delle richieste software, al fine di evitare contestazioni successive.

## Progettazione del software

La progettazione serve a definire la struttura del software.

La struttura del software è costituita da un elenco o da uno schema grafico dei moduli che lo comporranno.

Per ogni modulo devono essere indicate le specifiche funzionali e le interfacce.

Le specifiche funzionali indicano quali funzionalità deve implementare il modulo, le interfacce indicano quali proprietà e metodi il modulo deve esporre all'esterno e quali firme devono avere le proprietà e i metodi.

### Esempio

L'analisi produce un documento nel quale indica di dover produrre due moduli:

Un modulo che si chiama DBInterface che si occupa di leggere e scrivere sul database, il modulo deve esporre il metodo FindEmailAdresses che accetta due parametri di tipo stringa e restituisce un vettore di stringhe, i due parametri dovranno contenere rispettivamente il nome ed il cognome del destinatario, il metodo dovrà cercare nella tabella Contatti del database tutti coloro che hanno nome e cognome corrispondenti, costruire un vettore di stringhe contenente i relativi indirizzi email e restituirlo all'esterno.

Un modulo che si chiama MailServices che si occupa di fornire servizi email, il modulo deve esporre il metodo SendEmail che accetta tre parametri di tipo stringa contenenti rispettivamente il nome, il cognome del destinatario ed il corpo della email, il metodo dovrà utilizzare il metodo FindEmailAdresses del modulo DBInterface passando nome e cognome, se il metodo ritorna un vettore vuoto dovrà mostrare un messaggio per avvisare l'utente dell'impossibilità di trovare i dati, se restituisce un vettore non vuoto dovrà inviare una email contenente i dati scritti nel terzo parametro per ogni indirizzo indicato nel vettore.

## Implementazione del software

L'implementazione è la fase di programmazione, ossia della stesura del codice in uno o più linguaggi di programmazione.

È compito del team di sviluppo definire i metodi privati del modulo, gli unici vincoli ai quali devono sottostare sono quelli delle interfacce e delle specifiche funzionali.

Eventuali metodi privati possono essere aggiunti secondo le necessità che si presenteranno durante la stesura del codice.

### Esempio

Durante l'implementazione del metodo FindEmailAdresses nel modulo DBInterface gli sviluppatori decidono di creare un metodo privato CheckAddress che accetta un parametro di tipo stringa e restituisce un parametro di tipo boolean: se il valore passato nel parametro è un indirizzo valido restituirà Vero, altrimenti restituirà Falso.

## Controllo di versione

Il controllo di versione è un sistema che consente di gestire in maniera organica più versioni del software che si sta sviluppando.

Il sistema si basa su un'architettura client-server che può essere locale, distribuita in LAN o in WAN.

Il server archivia i dati in un *repository*, questo può essere un file, un insieme di cartelle o un database, a seconda del prodotto che si utilizza.

Ogni modifica al progetto viene archiviata nel repository e le viene assegnato un numero progressivo univoco (detto numero di versione) che consentirà di identificarla in maniera semplice da quel momento in poi.

In qualunque momento del lavoro è possibile:

- sincronizzare la propria copia locale del progetto con l'ultima versione presente nel repository
- tornare ad una versione precedente
- confrontare due diverse versioni
- verificare quale elemento del gruppo di lavoro ha prodotto una particolare riga di codice o modifica di progetto
- generare branch di sviluppo
- integrare branch di sviluppo

## Continuous testing

Il continuous testing è un sistema di verifica automatica che consente di verificare il corretto funzionamento del codice scritto o in fase di scrittura.

Si basa su un software che analizza il codice sorgente, esegue dei test predeterminati (detti classi di test) e fornisce i risultati dell'esecuzione del codice (presenza di errori, tempo di esecuzione ed altre informazioni), il tutto mentre lo sviluppatore continua a lavorare.

## Collaudo

Il collaudo consiste nella verifica di quanto il software realizzato rispetti (totalmente o parzialmente) i requisiti stabiliti in fase di analisi.

Qualora le specifiche stilate e sottoscritte in fase di analisi non siano state rispettate, viene prodotto un documento contenente l'elenco delle mancanze e degli errori riscontrati ed il software torna alla fase di implementazione.

## Bug Tracking

il Bug Tracking System è un sistema con architettura client-server che consente al team di sviluppo di tenere traccia di segnalazioni bug, richieste di nuove funzionalità e altre informazioni.

In alcuni casi si lascia agli utenti del software la possibilità di inserire direttamente le segnalazioni sul server di bugtracking, in modo da consentire la concentrazione delle segnalazioni in un unico punto e da rendere agevole la gestione della correzione dei bug.

Gli elementi che costituiscono un Bug Tracking System sono

- un server contenente il database con tutte le segnalazioni
- un client per l'inserimento delle segnalazioni e l'aggiornamento delle stesse

Spesso come client viene utilizzato un comune Web Browser in modo da semplificare l'accesso alle informazioni.

Le informazioni principali che vengono registrate sono:

- data della segnalazione
- tipo di segnalazione (bug, richiesta nuova funzionalità, miglioramento di funzionalità esistente e altre)
- sviluppatore al quale assegnare la gestione della segnalazione
- ore di lavoro stimate per la chiusura della segnalazione
- ore di lavoro effettivamente svolte
- stato di chiusura (bug corretto, situazione non riproducibile, funzionalità aggiunta e altre)

## Rilascio

La fase di rilascio consiste nell'installazione del software nell'infrastruttura di utilizzo (detta *ambiente di produzione*).

Il rilascio può essere costituito dalla semplice copia di un file così come da una procedura estremamente complessa che richiede l'intervento di tecnici appositamente formati.

## Manutenzione

La manutenzione consiste nel supporto al software già distribuito tramite la creazione e la distribuzione di patch che correggono errori sfuggiti alle fasi di implementazione e collaudo, aggiunge supporto per nuovi ambienti hardware o software, aggiunge nuove funzionalità al prodotto.

# Architettura del software

Lo standard ANSI/IEEE Std 1471-2000 definisce l'architettura del software come:

*l'organizzazione basilare di un sistema, rappresentato dalle sue componenti, dalle relazioni che esistono tra di loro e con l'ambiente circostante, e dai principi che governano la sua progettazione ed evoluzione.*

Detto in termini semplici, l'architettura di un software è l'organizzazione strutturale del sistema: definisce i componenti software<sup>1</sup>, le proprietà visibili di questi componenti e le relazioni fra gli elementi. L'architettura di un software non comprende solamente la sua struttura ma anche le modalità con cui le diverse parti si integrano e interagiscono, gli aspetti legati all'interoperabilità con altri sistemi, il livello con cui l'applicazione soddisfa i requisiti funzionali, le caratteristiche orientate a favorire l'evoluzione nel tempo del sistema a fronte dei suoi cambiamenti strutturali e in relazione all'ambiente in cui esso è inserito (scalabilità, performance, manutenibilità, sicurezza, affidabilità, ecc.).

Ciascun componente entra a far parte dell'architettura in funzione del ruolo che esso ricopre. Ogni componente presenta caratteristiche peculiari (nella definizione denominate "proprietà visibili esternamente") che influenzano il modo con cui ciascuna parte del sistema comunica e interagisce con le altre. L'architettura considera gli aspetti che sono inerenti la comunicazione tra le parti, si focalizza sulle modalità di interazione, tralasciando i dettagli di funzionamento interni.

L'architettura è una rappresentazione che permette all'architetto di analizzare l'efficacia del progetto per rispondere ai requisiti stabiliti, di considerare e valutare le alternative strutturali in una fase in cui i cambiamenti abbiano ancora un impatto relativo sull'andamento del progetto e sul risultato finale e di gestire in modo appropriato i rischi che sono collegati alla progettazione e alla realizzazione del software.

Nei moderni sistemi software le parti interagiscono tra loro per mezzo di interfacce che suddividono in modo netto ciò che non è direttamente accessibile dall'esterno da ciò che è pubblico. L'architettura si concentra unicamente sul secondo aspetto, tralasciando i dettagli interni che in generale non influenzano (o quasi) il modo con cui i componenti si relazionano tra loro. Le interazioni fra i componenti possono essere semplici (come una chiamata a funzione) oppure complesse (come un protocollo di comunicazione o un meccanismo di serializzazione).

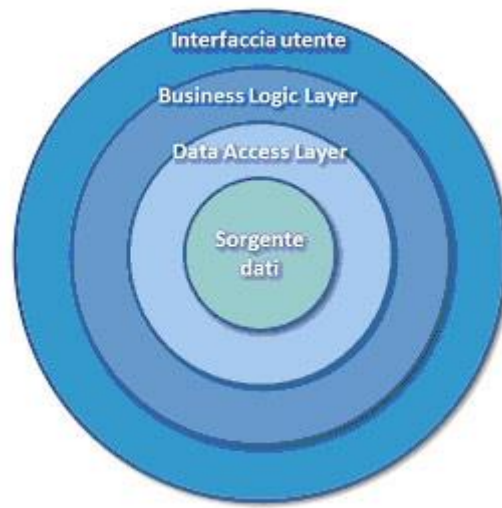
## Stratificazione del software

Questo stile di architettura prevede la strutturazione logica di un sistema software in strati sovrapposti (layer), tra loro comunicanti, ciascuno caratterizzato da una forte omogeneità funzionale.

La forma più nota e usata riguarda l'architettura a tre livelli composta da: *User Interface* (UI) o strato di presentazione, dove vengono gestite le interazioni dell'utente col sistema, *Business Logic Layer* (BLL) o strato di business, dove sono presenti i servizi applicativi, e *Data Access Layer* (DAL) o strato di accesso ai dati, dove sono gestite le interazioni con il sistema di persistenza delle informazioni.

---

<sup>1</sup> Per componente software si intende qualsiasi entità facente parte di un sistema: dal semplice modulo applicativo (es: una classe in un'applicazione basata sul paradigma ad oggetti) al sottosistema complesso (per esempio, un DBMS).



# Programmazione object oriented

I sistemi informatici hanno una struttura interna complessa e sono composti da molti moduli ciascuno dei quali può contenere migliaia di istruzioni, pertanto diventa essenziale la corretta comunicazione tra moduli e la garanzia di funzionamento di ognuno di questi.

La complessità di un sistema aumenta in maniera esponenziale con l'aumentare dei componenti ed il comportamento del sistema può essere inatteso, considerato il numero enorme di possibili stati in cui può trovarsi, la scomposizione di un sistema in funzioni risolve parzialmente il problema, con una tendenza al degrado delle prestazioni conseguentemente ad una modifica dei requisiti.

Per ovviare a tutti questi problemi è nata la Programmazione Object Oriented (OOP).

La programmazione orientata agli oggetti<sup>2</sup> (o programmazione ad oggetti) è un paradigma di programmazione che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi, è particolarmente adatta nei contesti nei quali si possono definire delle relazioni fra i moduli che costituiscono il software.

La programmazione ad oggetti prevede di raggruppare in una classe<sup>3</sup> la dichiarazione delle strutture dati e delle procedure che operano su di esse.

La OOP presenta molti vantaggi, i principali sono:

- fornisce un supporto naturale alla modellazione software degli oggetti del mondo reale o del modello astratto
- aiuta la suddivisione del lavoro all'interno di un team di sviluppo
- permette una facile gestione e manutenzione di progetti di grandi dimensioni
- l'organizzazione del codice sotto forma di classi favorisce la modularità e il riuso di codice
- il codice prodotto è compatto perché riutilizza al meglio componenti di piccole dimensioni

Il progetto finale si ottiene tramite la realizzazione degli oggetti nelle loro componenti strutturali (classi ed oggetti) e comportamentali (proprietà e metodi).

La codifica è basata sulla cooperazione delle classi che sono state progettate per svolgere un compito assegnato.

Le classi costituiscono dei modelli astratti, che durante l'esecuzione del codice vengono invocate per istanziare oggetti software relativi alla classe invocata, questi oggetti sono dotati di proprietà e metodi secondo quanto definito dalle rispettive classi.

Le classi sono caratterizzate da:

- proprietà
- metodi
- costruttore
- distruttore

Una classe si utilizza in maniera simile ad un tipo di dato: si dichiara una variabile del tipo definito dalla classe e, se necessario, si istanzia la variabile (si invoca il costruttore della classe) dopodiché si utilizzano i metodi e le proprietà che la classe espone.

Ad esempio: disponendo di una classe Automobile, è possibile dichiarare una variabile miaAuto di tipo Automobile, istanziarla, verificare la proprietà LivelloCarburante e chiamare i metodi Accendi e Accelera.

Un linguaggio di programmazione è definito ad oggetti quando la sua sintassi nativa permette di implementare tre meccanismi:

- incapsulamento
- ereditarietà

---

<sup>2</sup> La programmazione orientata agli oggetti viene spesso indicata dalla sigla OOP, Object Oriented Programming

<sup>3</sup> Nella programmazione orientata agli oggetti una classe è una zona isolata del codice sorgente



## Incapsulamento

Si definisce incapsulamento la possibilità di nascondere il funzionamento interno di una parte di programma (tipicamente di una classe), questo consente di isolare le modifiche apportate al codice in modo da ottenere moduli che lavorano assieme ma che sono modificabili in maniera completamente indipendente.

Un'altra conseguenza positiva dell'incapsulamento è che questo rende possibile la modifica dell'implementazione di uno o più oggetti senza dover modificare il codice che li utilizza, in pratica le modifiche effettuate al codice di una classe non richiedono modifiche al codice delle altre classi.

Il permesso di accedere ai dati interni all'oggetto viene gestito interamente dall'oggetto stesso, in questo modo i dati sono protetti da possibili modifiche effettuate dall'esterno e si ha sempre la garanzia che gli stati interni di una classe siano al sicuro.

## Ereditarietà

L'ereditarietà è un meccanismo che consente di derivare nuove classi (dette sottoclassi o classi figlie) a partire da classi esistenti (dette superclassi o classi padre), creando una gerarchia di classi.

Quando una sottoclasse eredita da una superclasse mantiene i metodi e gli attributi della classe da cui deriva, inoltre può definire i propri metodi o attributi e ridefinire il codice di alcuni dei metodi ereditati. Ad esempio ipotizziamo di avere una classe *automobile* con le sue proprietà (colore, cilindrata, ...) e i suoi metodi (accendi, spegni, ...); se creiamo una classe *utilitaria* derivata dalla classe *automobile*, la classe *utilitaria* erediterà automaticamente proprietà e metodi di *automobile*.

## Polimorfismo

Il polimorfismo è una caratteristica che garantisce che le istanze di una sottoclasse possono essere utilizzate al posto di istanze della superclasse, detto in altri termini è la possibilità di utilizzare lo stesso nome per funzioni diverse applicate a classi diverse.

Ad esempio consideriamo una automobile, questa ha dei comportamenti diversi a seconda del modello.

In particolare per accenderla (se si trattasse di un programma invocando il metodo `Accendi()` della classe `Automobile`) il comportamento effettivo del motore è diverso a seconda che si tratta di una Ferrari monoposto di formula uno, o una utilitaria per andare alla spesa ma in ogni caso entrambe si accendono.

In pratica ad una richiesta che è abbastanza standard (accendere la macchina), la specifica automobile sa come comportarsi.

## Proprietà

Una proprietà identifica lo stato di un oggetto, può far riferimento ad una o più variabili appartenente alla classe.

Le variabili di una classe possono essere viste dall'esterno solamente tramite le proprietà che la classe stessa mette a disposizione del programma chiamante, questo garantisce che nessuno possa modificare in maniera impropria lo stato di un oggetto.

Tornando all'esempio dell'automobile: la proprietà `colore` indica di che colore è l'automobile, leggendolo da una o più variabili interne della classe.

## Metodo

Un metodo è una funzione appartenente ad una classe, può essere esposto all'esterno o essere privato della classe.

I metodi (sia pubblici che privati) hanno pieno accesso alle variabili interne della classe, pertanto possono leggerle o modificarle liberamente.

Ancora dall'esempio dell'automobile: la proprietà colore potrà essere modificata solamente utilizzando un apposito metodo implementato dalla classe automobile (ad es. Dipingi) e non direttamente, questo consente a chi scrive la classe automobile di decidere quando e come sia possibile cambiarne il colore.

## Costruttore

In una classe possono essere presenti uno o più metodi (tra loro si differenziano per tipo o numero di argomenti) che vengono invocati automaticamente quando un oggetto viene istanziato e solitamente servono per inizializzare le proprietà dell'oggetto.

Se una classe ha uno o più padri (perché eredita da altre classi) anche i costruttori dei padri vengono automaticamente invocati.

## Distruttore

In una classe può essere presente un solo metodo distruttore che viene invocato quando un oggetto viene distrutto e che consente di gestire la pulizia delle variabili interne.

Solitamente viene utilizzato per deallocare la memoria occupata dall'oggetto, altri utilizzi tipici sono la chiusura delle connessioni aperte (verso database, file o entità di rete).

Molti linguaggi orientati agli oggetti eseguono automaticamente l'operazione di deallocazione della memoria.

## Overloading

Con il termine overloading si indica la definizione, in una stessa classe, di più funzioni con stesso nome ma argomenti differenti.

Il termine overloading si utilizza solamente per indicare una definizione di un metodo con firma (numero e tipo dei parametri) diversa.

## Overriding

Una classe figlio può ridefinire (quindi usando stesso nome metodo, numero e tipo argomenti) un metodo definito nella classe padre, se questo metodo viene invocato verrà utilizzato quello della classe figlio, come se nella classe padre non esistesse.

# Sommario

|   |    |
|---|----|
| Cos'è un problema                               | 3  |
| La macchina di Turing                           | 5  |
| Caratteristiche che deve possedere un esecutore | 5  |
| Algoritmo di Euclide                            | 5  |
| Elementi di programmazione imperativa           | 7  |
| Caratteristiche generali di un programma        | 7  |
| Costanti e variabili                            | 7  |
| Assegnazione                                    | 7  |
| Scomposizione di un problema                    | 8  |
| Scomposizione sequenziale                       | 8  |
| Scomposizione condizionale                      | 8  |
| Scomposizione iterativa (ripetitiva)            | 8  |
| Il costrutto while                              | 9  |
| Il costrutto repeat-until                       | 9  |
| Il costrutto for                                | 9  |
| Istruzioni di input e di output                 | 10 |
| Diagrammi a blocchi                             | 11 |
| Metodologia di scomposizione top-down           | 12 |
| Programmi e sottoprogrammi                      | 12 |
| Parametri                                       | 13 |
| Procedure e funzioni                            | 13 |
| Ricorsione                                      | 14 |
| Ricorsione indiretta e incrociata               | 14 |
| Dati e codifica                                 | 15 |
| Alfabeto binario                                | 15 |
| Elaboratori ed alfabeto binario                 | 15 |
| Codifica dei dati non numerici                  | 15 |
| La codifica delle istruzioni                    | 16 |
| La codifica di dati numerici                    | 16 |
| Codifica binaria di numeri interi con segno     | 18 |
| Codifica binaria di numeri razionali            | 18 |
| I dati  | 19 |
| Il tipo boolean                                 | 19 |
| Tipi strutturati                                | 19 |
| Complessità                                     | 21 |
| Esempi di scomposizione                         | 22 |
| Scomposizione sequenziale                       | 22 |

|  |    |
|--|----|
| Somma di due numeri interi. _____                        | 22 |
| Somma di n numeri. _____                                 | 22 |
| Scomposizione iterativa _____                            | 23 |
| Somma di n numeri _____                                  | 23 |
| Scomposizione condizionale _____                         | 23 |
| Determinare il valore massimo tra due numeri x e y _____ | 23 |
| Determinare il valore massimo tra 3 numeri x,y,z. _____  | 24 |
| Ciclo di vita del software _____                         | 27 |
| Analisi del software _____                               | 27 |
| Progettazione del software _____                         | 27 |
| Implementazione del software _____                       | 28 |
| Controllo di versione _____                              | 28 |
| Continuous testing _____                                 | 28 |
| Collaudo _____   | 28 |
| Bug Tracking _____                                       | 29 |
| Rilascio _____   | 29 |
| Manutenzione _____                                       | 29 |
| Architettura del software _____                          | 30 |
| Stratificazione del software _____                       | 30 |
| Programmazione object oriented _____                     | 32 |
| Incapsulamento _____                                     | 33 |
| Ereditarietà _____                                       | 33 |
| Polimorfismo _____                                       | 33 |
| Proprietà _____  | 33 |
| Metodo _____   | 34 |
| Costruttore _____  | 34 |
| Distruttore _____  | 34 |
| Overloading _____  | 34 |
| Overriding _____   | 34 |