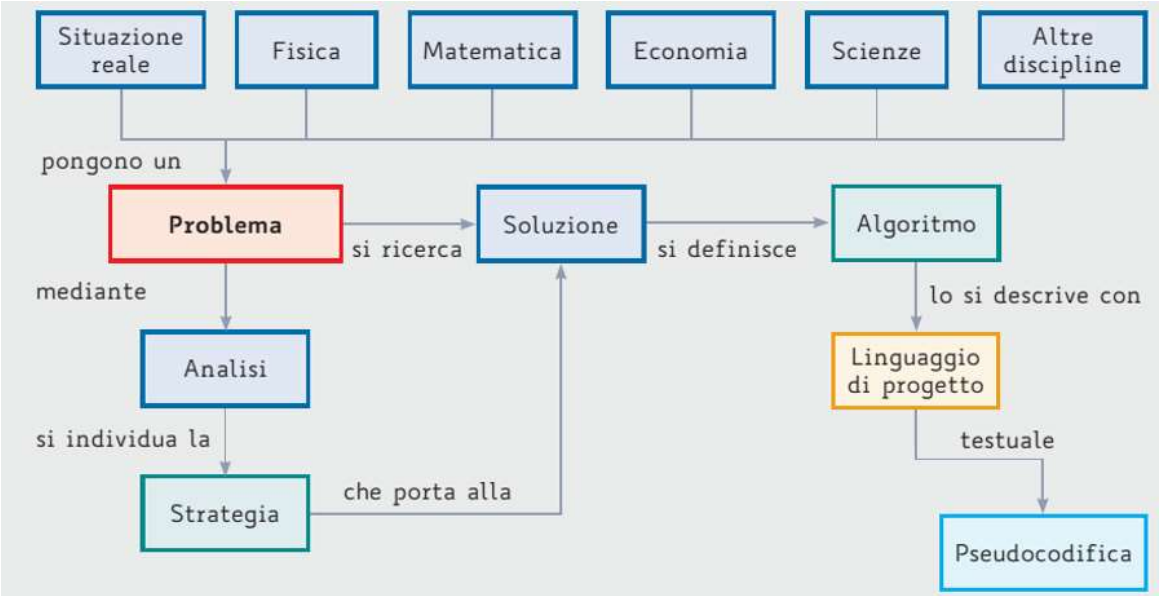
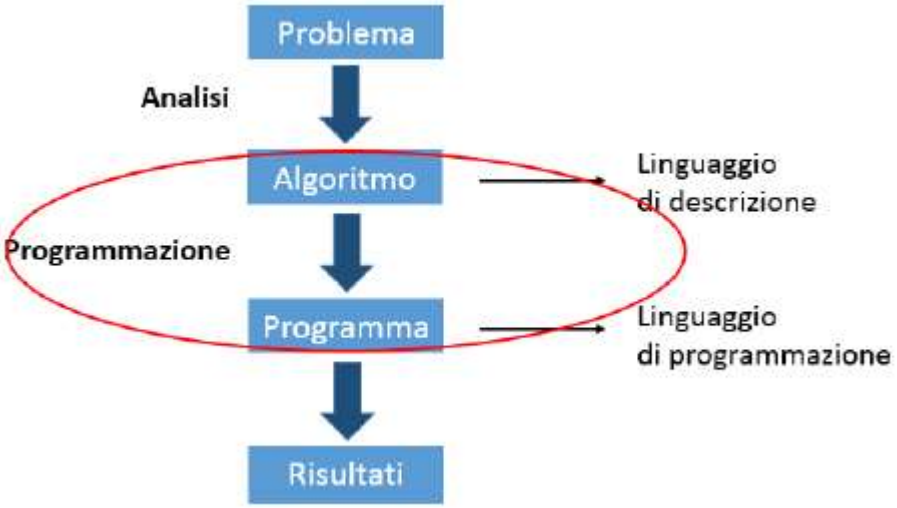
Problemi e possibili soluzioni informatizzate



In un contesto informatico, la soluzione di un **problema** può richiedere le **fasi** seguenti:



In fase di **ANALISI** si individueranno lo **scopo (OUTPUT *reale*)** che si vuole raggiungere, i **dati di INPUT** *(espliciti o impliciti* **UTILI per quello scopo***)*, e le eventuali regole (**MODELLO**) o vincoli.

Flusso di controllo: la descrizione a priori di tutte le possibili ***sequenze*** nell'esecuzione dei passi dell'***algoritmo***, in particolare di operazioni in ***alternativa*** tra loro e di operazioni da ***ripetere*** più volte ciclicamente (***strutture di controllo***)

Flusso di esecuzione: la sequenza di operazioni effettivamente seguita durante una particolare **esecuzione** del ***programma*** *(elenco di istruzioni)* che dipende dai particolari valori che i dati assumono durante quel *processo.*

Strutture di controllo del flusso: tecniche di rappresentazione

Per realizzare un **algoritmo strutturato** nell'approccio **tradizionale** (programmazione imperativa) si usano le seguenti **strutture** di controllo del flusso:

- la **sequenza**,
- la selezione o **alternativa**,
- l'iterazione o **ripetizione**.

Sequenza	**Selezione o alternativa**	**Iterazione o ripetizione**
Le *direttive* verranno eseguite a seconda dell'ordine in cui sono scritte	In base al valore di una **condizione**, verrà eseguito un diverso *flusso (**alternativa binaria:** scelta tra due possibili* blocchi di *direttive*)	In base al valore di una **condizione**, verrà ripetuta l'esecuzione di una o più direttive

Tali costrutti logici si possono illustrare con **due tecniche di rappresentazione**: è preferibile usare un **flow-chart** (diagramma di flusso) per **algoritmi semplici**, **facilmente traducibili in linguaggio macchina** (*linguaggio assemblativo*: a *basso livello*).

Si usa, invece, lo **pseudocodice** per **algoritmi complessi**, **facilmente traducibili in linguaggi strutturati** (*linguaggio di programmazione*: a *medio-alto livello*).

In quest'ultimo caso, utilizzando il simbolo **{** col significato di *inizio* ed il simbolo **}** col significato di *fine*, si possono illustrare le strutture di controllo del flusso con il seguente **pseudocodice**:

Sequenza

```
{
  direttiva 1
  direttiva 2
   ……
}
```

Alternativa binaria:

```
{
  se (condizione)
      direttiva/e_seVera
  altrimenti
      direttiva/e_seFalsa
}
```

Alternativa ternaria:

```
{
  se (condizione 1)
      direttiva/e_seVeraCondizione1
  altrimenti se (condizione 2)
                direttiva/e_seVeraCondizione2
            altrimenti
                direttiva/e_seFalsa
}
```

Ripetizione con ***controllo in testa***:

```
{
  mentre (condizione)
     direttiva/e
}
```

Ripetizione con ***controllo in coda***:

```
{
  fai
     direttiva/e
  mentre (condizione)
 }
```

Nel proporre programmi ***efficienti***, si utilizzerà il costrutto **ripetizione** con uso di **contatore**

Ripetizione con ***controllo in testa con uso contatore***:

```
{
 inizializzo contatore
 mentre (condizione)
    direttiva/e
    aggiorno contatore
}
```

NB: si noti l'***indentazione*** (margine rientrato) in particolare per evidenziare la *direttiva* o il blocco di *direttive* che l'***esecutore discreto*** dovrà eseguire in base al verificarsi o meno di una condizione.

Nella **scomposizione** dei problemi in sotto-problemi, **risolvendo per approfondimenti successivi**, possiamo riferirci mediante un ***nome*** ad blocco di comandi (*direttive*) e proporre una soluzione "**modulare**" più chiara. Ad esempio si usa lo pseudocodice per indicare in **sequenza** i tre *moduli* essenziali nella soluzione che analizzeremo dopo, individuandone i singoli passi: un modulo di inizializzazione dei dati utili (col nome **inizio**), un'elaborazione col nome **elabora** ed un modulo per visualizzare a monitor (CLI) il risultato dell'elaborazione col nome **visualizza**:

	Potendo passare dati da elaborare
{ **inizio** **elabora** **visualizza** }	{ ***k*** ← valore **elabora (*k*)** **visualizza** }

Progettare soluzioni **modulari**, serve per *ridurre la complessità* di un problema consentendo facile implementazione con approccio *top*-down, lavorando in *equipe*.

Progettare bene un **modulo** significa pensarlo con le seguenti caratteristiche:

- efficienza,
- generalizzazione
- portabilità

Per approfondire

Approcci nella soluzione di problemi

Top-down

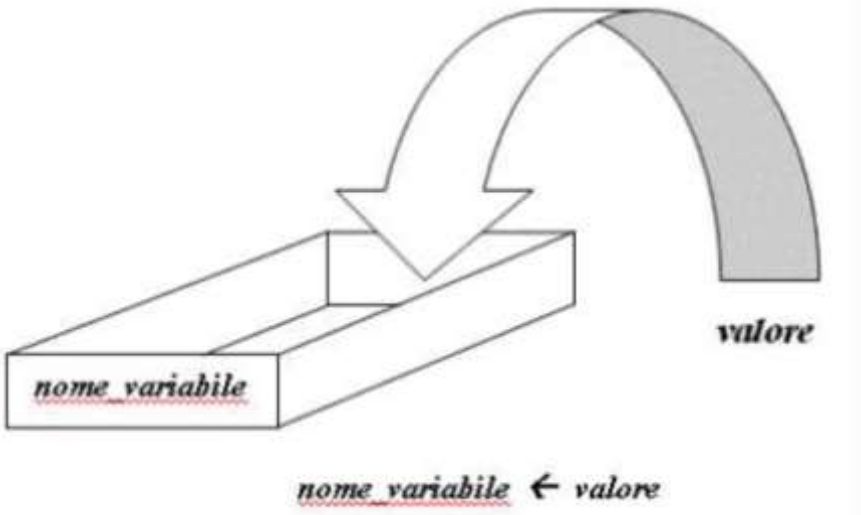
Una metodologia di trattamento dei problemi per *scomposizione* → *sviluppo degli algoritmi per* ***raffinamenti successivi.***

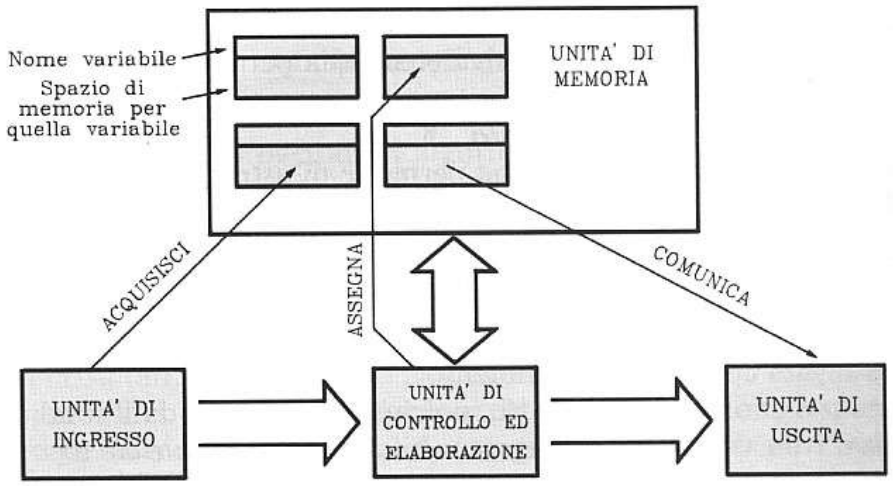
Dal problema complesso ...

... scomponendo in sottoproblemi più semplici ...

si risolve un problema elementare spesso di soluzione già nota

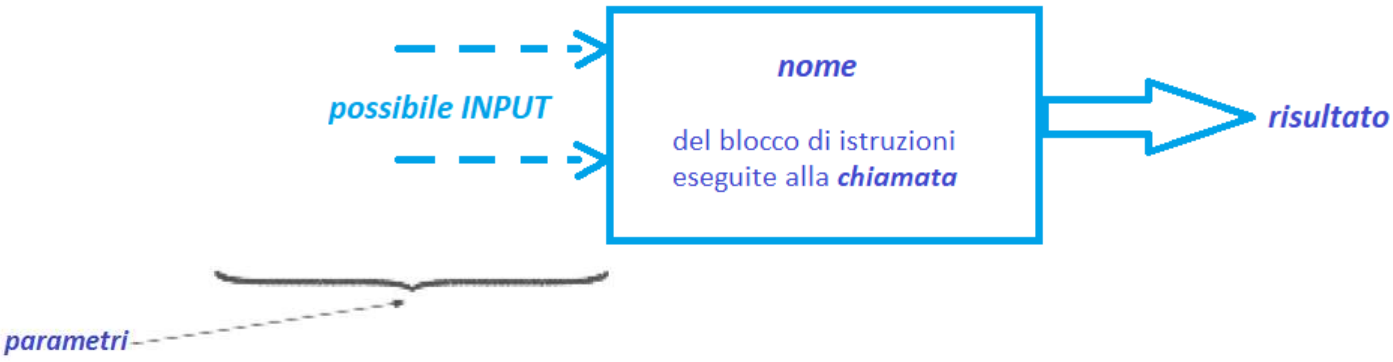
Il concetto di variabile e tipi





Scomposizione in moduli: concetto di funzione

Una **funzione** è un *modulo di programma* (sotto-insieme di istruzioni detto *corpo*) che può essere inteso come una "scatola nera" capace di realizzare una specifica elaborazione fornendo sempre un **risultato** (se la funzione non restituisce un valore è detta *void*) potendo utilizzare dati che le vengono passati, ad esempio come valori, al momento della *chiamata* da programma



Tale "scatola" non può essere inserita in un'altra "scatola" ma può essere usata trattando il suo ***nome*** come una ***variabile***.

Motivazione all'uso

Una funzione è un modo di *arricchire il linguaggio*.

Le funzioni possono essere raggruppate in **librerie** e ***riutilizzate*** da altri programmi;

rendono un programma più immune da errori imprevisti grazie alla possibilità di *nascondere* i dati

prevedendo meccanismi di passaggio controllato

delle informazioni;

consentono una scrittura più semplice e aumentano la *leggibilità* del programma.

Una **funzione**, infatti, serve per *ridurre la complessità* di un problema consentendo facile implementazione di soluzioni con approccio *top-down* che prevedono un lavoro parallelo in *equipe*.

Progettare bene una **funzione** significa pensarla con le seguenti caratteristiche: *efficienza*, *generalizzazione* e *portabilità*.

L'**efficienza** si valuta in base all'*utilizzo* che la funzione fa *delle risorse* del calcolatore: CPU (tempo), memoria RAM (spazio).

Con **portabilità** si intende *usabilità in ambienti di esecuzione diversi* e più generale l'indipendenza - in fase di esecuzione - dalla piattaforma hardware e/o software di esecuzione.

Con **generalizzazione** si intende *applicabile*, spesso arbitrariamente, alla maggior parte o alla totalità dei casi.