6. METODOLOGIA TOP-DOWN E SOTTOPROGRAMMI

Nella realtà per risolvere problemi è possibile individuare, analizzare i **sottoproblemi** più semplici che lo compongono e le loro **interrelazioni** (ossi come concorrono a comporre il problema complessivo).
In questo modo è possibile vedere la progettazione dell'algoritmo che descrive il processo risolutivo come la progettazione di una serie di **(sotto-)algoritmi** più semplici che verranno poi assemblati per ottenere la risoluzione del problema complessivo.

METODOLOGIE DI PROGETTAZIONE

Una delle metodologie di progettazione ossia di analisi dei problemi più note è quella cosiddetta **top-down** (*TOP* = ALTO e *DOWN* = BASSO). Gli aggettivi **alto** e **basso** si riferiscono al livello di *dettaglio o astrazione* al quale ci si pone.
Il livello più alto o TOP è quello di descrizione del processo risolutivo del **problema principale** mediante descrizione fondamentale dei suoi passi fondamentali chiamati **sottoproblemi.**
Ciascun sottoproblema viene dettagliato a parte e, se complesso, può essere a sua volta scomposto in ulteriori sottoproblemi più semplici.
In pratica si scende *dal generale al particolare* mediante affinamenti successivi.

La tecnica top-down nasce come tecnica di analisi dei problemi.

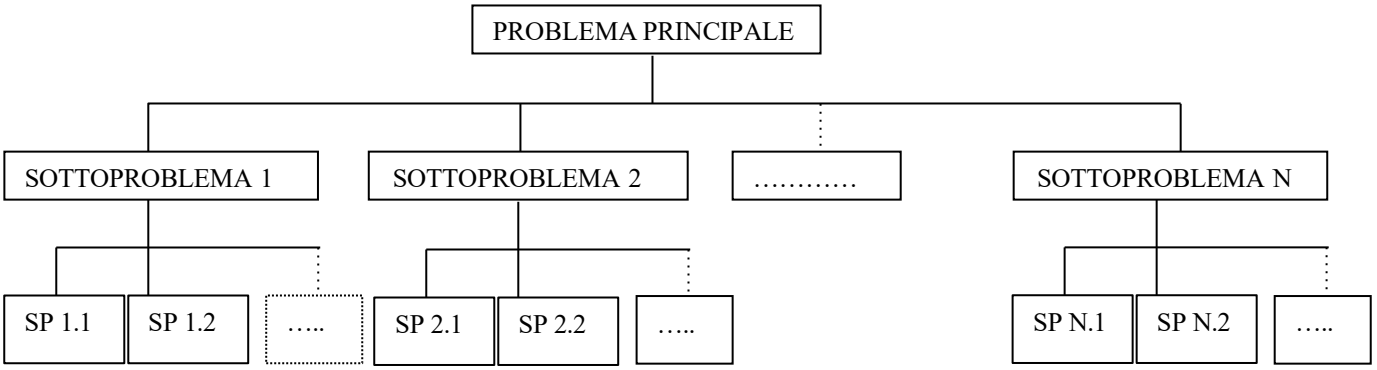
Il programmatore deciderà all'atto dell'implementazione del programma se implementare tutti i sottoproblemi individuati o, nonostante la loro individuazione, accorparne alcuni.
Tale metodologia è diventata una tra le principali tecniche di progettazione software.

N.B. Tale metodologia di progettazione utilizza una strategia di tipo deduttivo

Un altro tipo di metodologia di progettazione è quella **bottom-up** (*BOTTOM* = FONDO, BASSO e *UP* = CIMA, ALTO). Anche in questo caso gli aggettivi **fondo** e **cima** si riferiscono al livello di *dettaglio o astrazione* al quale ci si pone.
In pratica si sale *dal particolare al generale* mediante accorpamenti successivi.
Tale metodologia consente di concentrarsi e di occuparsi subito sui punti cardine del problema che però sono molto difficili da individuare immediatamente.
Per questo motivo è meno adatta alla progettazione di software.

N.B. Tale metodologia di progettazione utilizza una strategia di tipo induttivo

Schematizzando la tecnica di progettazione **top-down** abbiamo:



Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

In generale nel campo della progettazione di software si preferisce l'uso **della metodologia di sviluppo top-down** in quanto:

1. è uno strumento concettuale *molto potente* per costruire algoritmi;
2. rende il lavoro di analisi e di progettazione *più snello* (è meglio se divido un problema complesso in sottoproblemi più semplici ed è altresì evidente che in caso debba apportare qualche modifica potrò limitarmi a modificare soltanto il sottoproblema interessato senza toccare il resto);
3. permette lo svolgimento dell'intero lavoro da parte di *persone diverse* (perché ogni sottoproblema èp visto in modo autonomo rispetto al problema principale);
4. facilita la lettura e la comprensione dell'algoritmo risolutivo anche a distanza di molto tempo.

I SOTTOPROGRAMMI

Appare evidente che è possibile realizzare un **sottoprogramma per ogni sottoproblema** non più ulteriormente scomponibile. Unendo ala fine tutti i sottoprogrammi si ottiene il programma che risolve il problema principale.

DEF: Il sottoprogramma è una parte del programma in cui viene dettagliata una particolare attività descritta da un sottoalgoritmo

Non esiste una formula in grado di stabilire quanti programmi occorrono per risolvere un problema e quando essi vanno utilizzati.

E' possibile fornire soltanto delle linee guida basate sull'esperienza che vanno considerate come indicazioni operative:

a) CONVIENE descrivere un sottoproblema per mezzo di un sottoprogramma se
- è di interesse generale;
- pur non essendo di interesse generale si presenta più volte all'interno del programma;
- pur essendo di scarso interesse generale, permette una maggiore leggibilità del programma.

b) NON CONVIENE descrivere un sottoproblema per mezzo di un sottoprogramma se
- è di scarso interesse generale;
- non migliora la leggibilità del programma, anzi la complica;
- non garantisce un risparmio di tempo, soprattutto se si tratta di programma breve.

Riassumendo i **vantaggi** derivanti dall'uso dei sottoprogrammi sono:

- migliorano la leggibilità del programma;
- permettono l'astrazione (ossia il sottoprogramma permette al programmatore di interessarsi di "cosa" fare e non di "come" farlo);
- consentono di scrivere meno codice (e quindi al programma eseguibile di occupare meno memoria);
- sono riutilizzabili anche in altri contesti.

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

SPECIALE: L'esecuzione di un sottoprogramma

Per eseguire un sottoprogramma è necessario utilizzare una apposita **istruzione di chiamata di sottoprogramma** che è prevista da tutti i linguaggi di programmazione.

Meccanismo di funzionamento

Quando un programma non è in esecuzione risiede su una **MEMORIA DI MASSA** e subito dopo la compilazione ed il linkaggio, sarà costituito esclusivamente dal codice o istruzioni e dai dati ed occuperà un'area di memoria (la cui dimensione in BYTE dipende esclusivamente dalle istruzioni e dai dati utilizzati) che è possibile essere pensata come suddivisa in **due segmenti**:

- **il Segmento "CODICE"** o "ISTRUZIONI"**:** area contenente le istruzioni del programma (codice) scritte in linguaggio macchina
- **il Segmento "DATI":** area contenente variabili e costanti allocate staticamente

Quando un programma è in esecuzione (chiamato anche **TASK** o **processo)** viene allocato in memoria centrale (**RAM**) e gli viene assegnato, dal sistema operativo (un particolate programma chiamato **loader**), nella memoria di lavoro (**RAM**) anche una zona di memoria aggiuntiva rispetto a quella posseduta quando è "in quiete".

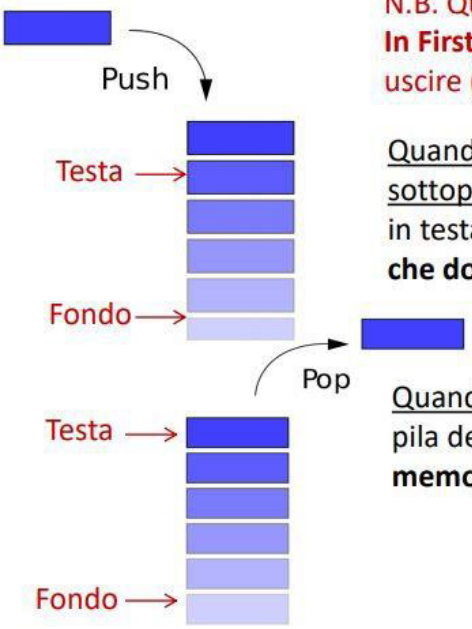
Un programma in esecuzione quindi occuperà una zona della memoria di lavoro che è possibile pensare ora suddivisa in **quattro segmenti**:

- **il Segmento "CODICE"** o "ISTRUZIONI"**:** *vedi descrizione già data in precedenza*
- **il Segmento "DATI":** *vedi descrizione già data in precedenza*
- **il Segmento "STACK":** area destinata a gestire la "**PILA DELLE ATTIVAZIONI**"
- **il Segmento HEAP** (lett. mucchio) **di sistema:** area destinata a raccogliere i dati gestiti dinamicamente che come tali verranno allocati e deallocati dinamicamente (che verranno illustrati IN SEGUITO)

Una volta terminata l'esecuzione del programma questa zona di memoria allocata nella RAM verrà rilasciata liberando la memoria precedentemente occupata **(allocazione dinamica del codice da parte del sistema operativo)**

L'intero meccanismo della gestione delle chiamate a sottoprogrammi avviene, come già accennato in precedenza, tramite una particolare struttura dati detta "PILA DELLE ATTIVAZIONI".

Per ricordare da quale istruzione va ripresa l'esecuzione dopo un sottoprogramma, la CPU si serve di una apposita STRUTTURA DATI detta **PILA delle attivazioni o STACK** dalla quale i dati possono essere inseriti o estratti solo da una estremità che viene detta **testa della pila**.



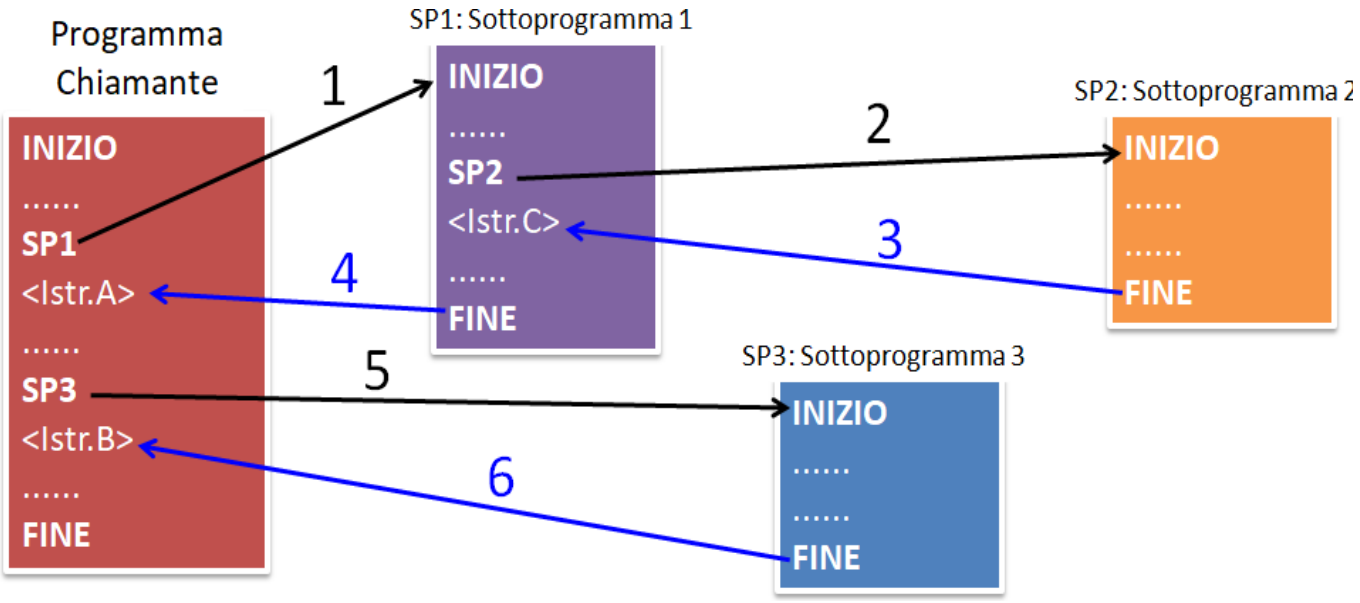
N.B. Questa struttura dati ha una struttura di tipo **LIFO o Last In First Out** nel senso che l'ultimo ad entrare è il primo ad uscire (esempio pila di piatti o di cd).

Quando la CPU esegue una istruzione di chiamata a sottoprogramma allora inserisce (**Push**) nella pila delle attivazioni in testa **l'indirizzo della cella di memoria contenente l'istruzione che dovrà essere eseguita al rientro dal sottoprogramma.**

Quando la CPU esegue una istruzione di FINE allora utilizza la pila delle attivazioni per estrarre dalla testa (**Pop**) **l'indirizzo di memoria in esso contenuto da dove riprendere l'esecuzione.**

Quando la **CPU** incontra una istruzione di chiamata a sottoprogramma (nel nostro esempio successivo indicate sinteticamente con **SP1**, **SP2** e **SP3**), sospende l'esecuzione del programma corrente e passa ad eseguire le istruzioni contenute nel sottoprogramma chiamato.

Terminata l'esecuzione del sottoprogramma (si arriva alla relativa istruzione FINE) , la **CPU** riprende l'esecuzione del programma ripartendo dall'istruzione immediatamente successiva a quella di chiamata.



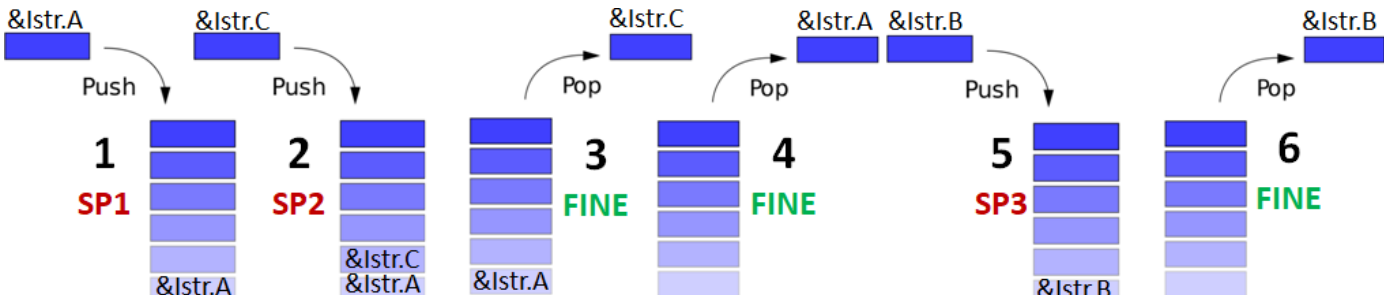
Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

Tutto questo avviene utilizzando la "**PILA DELLE ATTIVAZIONI**" nel seguente modo:

- Quando la CPU esegue una istruzione di chiamata a sottoprogramma allora inserisce nela pila delle attivazioni, IN TESTA, **l'indirizzo della cella di memoria contenente l'istruzione che dovrà essere eseguita al rientro dal sottoprogramma.**

- Quando la CPU esegue una istruzione di FINE allora utilizza la pila delle attivazioni per estrarre, DALLA TESTA, **l'indirizzo della cella di memoria in esso contenuto da dove riprendere l'esecuzione.**

Nell'esempio specificato in precedenza l'utilizzo della pila delle attivazioni da parte del nostro programma in esecuzione (TASK) sarà il seguente:



AMBIENTE E RISORSE LOCALI E GLOBALI DI UN SOTTOPROGRAMMA

DEF: Con il termine "Ambiente (di visibilità) **di un sottoprogramma" definiamo l'insieme delle risorse (variabili, costanti, tipoi di dato, sottoprogrammi, parametri) alle quali esso può accedere.**

Tale ambiente è costituito da:

- un **Ambiente (di visibilità) Locale** ossia costituito dalle risorse dichiarate al suo interno (**risorse locali**);
- un **Ambiente (di visibilità) Globale** ossia costituito dalle risorse utilizzabili da tutti i sottoprogrammi (**risorse globali**).

N.B. Un corretto stile di programmazione impone di non utilizzare l'ambiente globale di un sottoprogramma ma di privilegiare quello locale.

LE REGOLE DI VISIBILITÀ O "SCOPE"

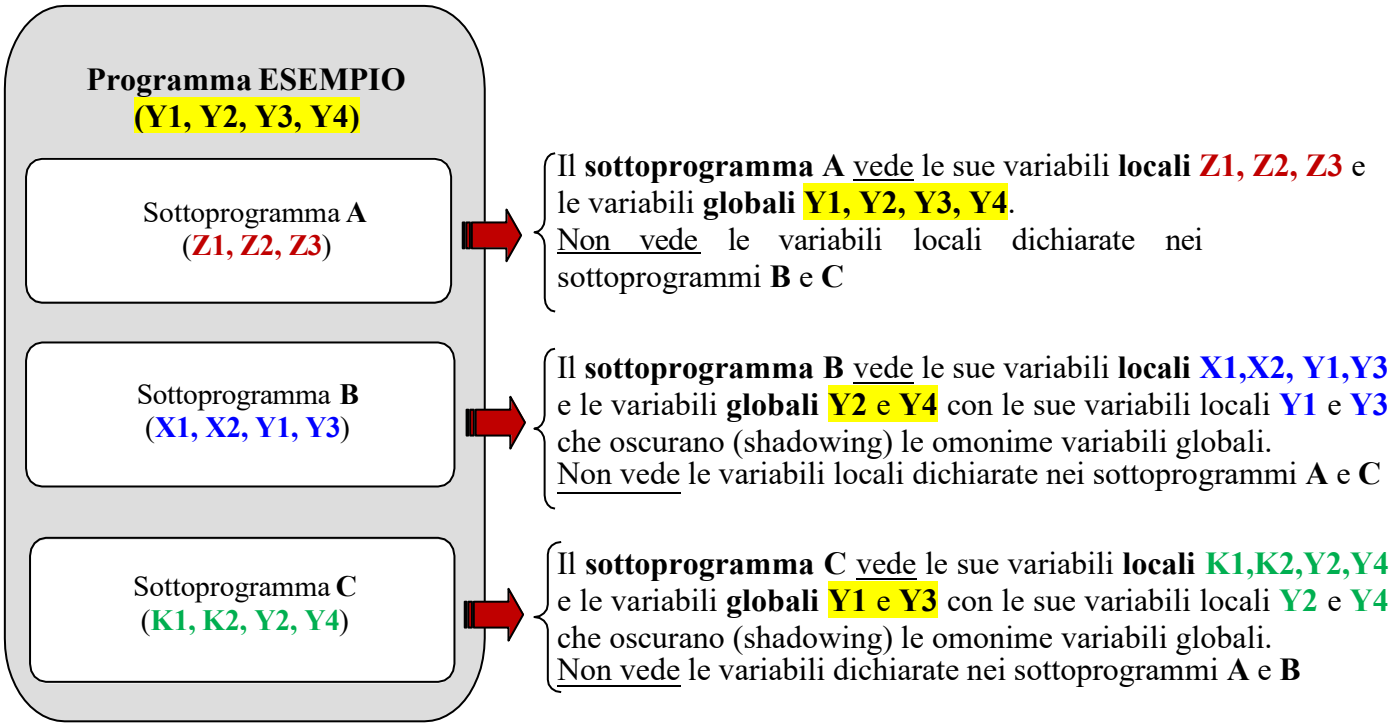
Esistono delle regole per determinare il campo di visibilità degli oggetti ***globali*** e ***locali*** di un programma (algoritmo).

Si parte dai seguenti principi:

1. **Un oggetto (globale e/o locale) non può essere usato se prima non viene dichiarato**
2. **Gli oggetti globali sono accessibili a tutto il programma (algoritmo) ed a tutti i sottoprogrammi che lo compongono;**
3. **Un oggetto dichiarato in un sottoprogramma (locale) ha significato solo in quel sottoprogramma ed in tutti quelli in esso dichiarati.**

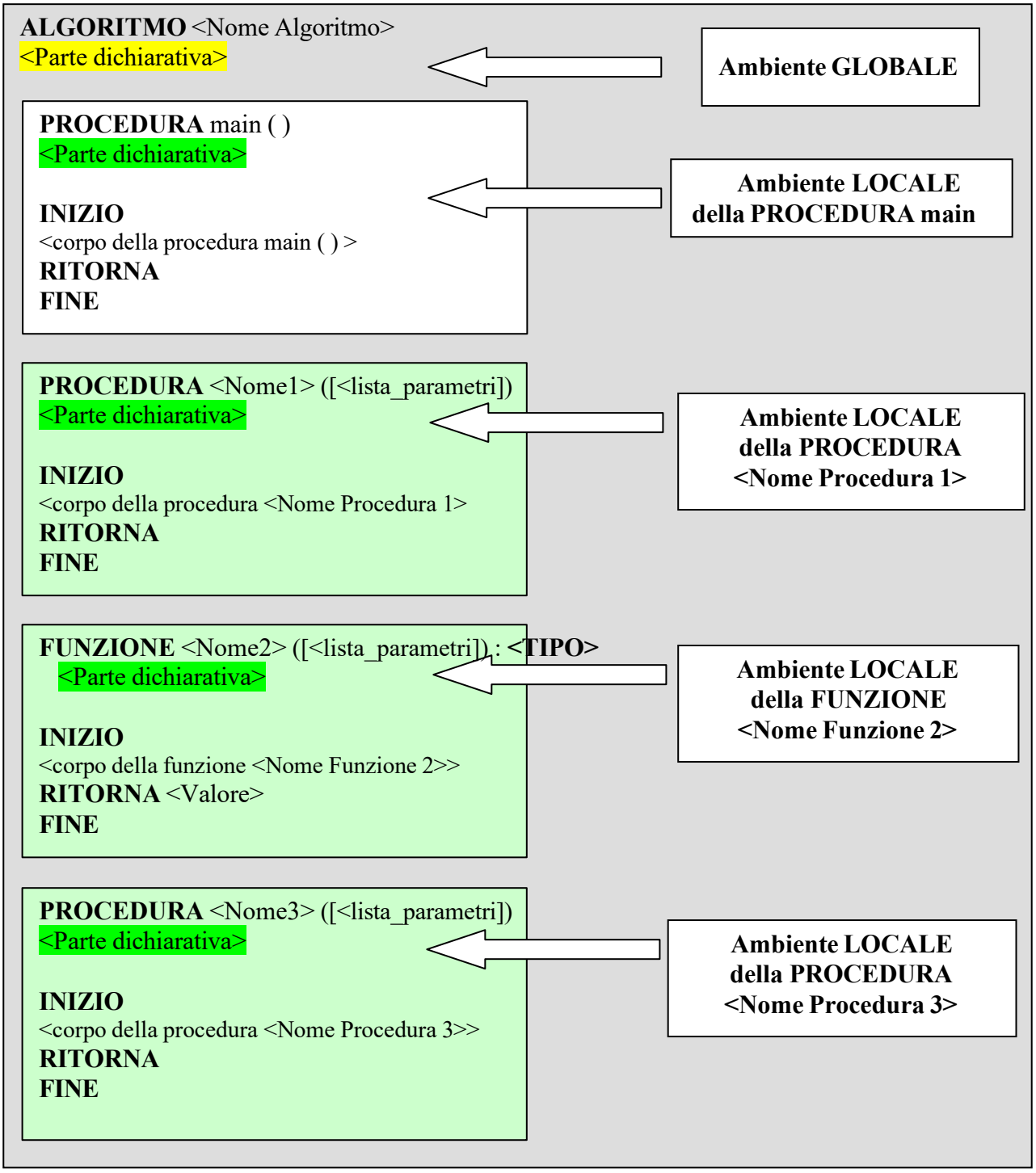
PROBLEMA DELLA OMONIMIA: Nella descrizione di un programma (algoritmo) può accadere che una variabile sia dichiarata con lo stesso nome (il tipo potrebbe essere anche non uguale) tanto a livello globale che a livello locale all'interno di un sottoprogramma

RISOLUZIONE: Nel caso che essa venga usata in una o più istruzioni all'interno del sottoprogramma, tale variabile locale più interna oscurerà l'omonima variabile più esterna (la globale), **impedendone la visibilità** (**SHADOWING**)



Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

Impiegando un metodo di rappresentazione che vede un sottoprogramma come un programma composto ***da una intestazione****,* ***da una parte dichiarativa*** *e* ***da una parte esecutiva*** *possiamo schematizzare quanto detto come segue*:



Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

I PARAMETRI

DEF: I parametri sono oggetti messi a disposizione da tutti i linguaggi di programmazione per rendere i sottoprogrammi autonomi ed indipendenti (funzionalmente indipendenti) dai dati del programma chiamante.

Come le comuni variabili, i parameri sono caratterizzati da:

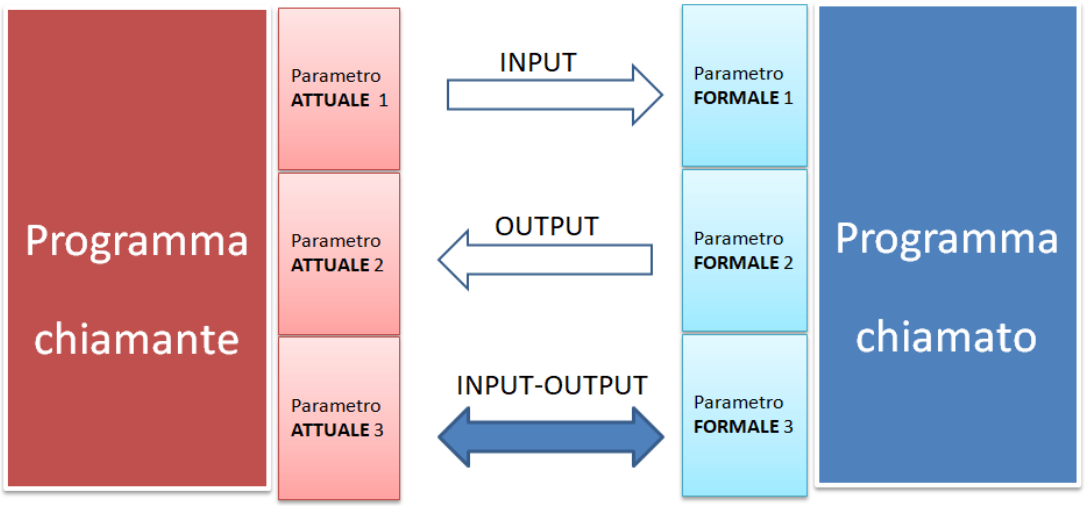
- **un identificatore o nome;**
- **un tipo;**
- **un valore.**

Inoltre per svolgere il ruolo di INTERFACCIAMENTO tra programma chiamante e programma chiamato, i paramentri sono caratterizzati da:

- **un numero;**
- **una posizione;**
- **una direzione.**

Sono dunque oggetti attraverso i quali averrà l'input dei dati (dal programma chiamante al sottoprogramma) ed il conseguente output dei risultati (dal sottoprogramma al programma chiamante).

Essi dunque permettono lo scambio di informazioni tra programma chiamante e programma chiamato.



Al momento della dichiarazione del sottoprogramma si deve specificare la tipologia dei parametri da utilizzare (**identificatore o nome, tipo e posizione**): essi prendono il nome di **parametri formali.**

Al momento della chiamata del sottoprogramma occorrerà specificare la tipologia dei parametri da trasmettere (**identificatore o nome, tipo e posizione**): essi prendono il nome di **parametri attuali.**

Inoltre:

- il **numero** dei parametri attuali deve coincidere con il numero dei parametri formali;

- i parametri attuali ed i parametri formali che si trovano in **posizione** omologa, devono avere lo stesso **tipo**;

- la **direzione** ha a che fare con l'essere parametro solo di input, solo di output o di input ed output.

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

LA PROCEDURA

DEF: La procedura è un sottoprogramma che, attivato dall'apposita istruzione di chiamata, svolge le azioni in esso specificate allo scopo di risolvere il (sotto)problema per il quale è stato realizzato

Con la **PSEUDOCODIFICA** la procedura viene indicata come segue:

```
PROCEDURA <Nome Procedura> ( [ REF | VAL <Nome param 1>: <Tipo param 1> ,
                               REF | VAL <Nome param 2>: <Tipo param 2> ,
                               ……,
                               REF | VAL <Nome param N>: <Tipo param N> ] )
< Sezione dichiarativa Procedura>
INIZIO
< Corpo della Procedura>
RITORNA
FINE
```

N.B. Gli specificatori di passaggio REF e VAL verranno illustrati nelle pagine seguenti

Una **procedura** è caratterizzata da:

- **un nome**, grazie al quale è possibile richiamarla ed identificarla univocamente;
- **una lista di parametri** che è *opzionale* e permette lo scambio in input e/o in output di informazioni tra il *programma chiamante* ed la procedura stessa ossia il *programma chiamato*.

LA FUNZIONE

DEF: La funzione è un sottoprogramma che, attivato dall'apposita istruzione di chiamata, oltre a svolgere le azioni in esso specificate allo scopo di risolvere il (sotto)problema per il quale è stato realizzato, può restituire un valore.
Questo valore è restituito nel nome della funzione e può essere usato direttamente come elemento di una istruzione di assegnazione o in una espressione oppure come output.

Quindi, a differenza delle procedure, le funzioni sono in grado di restituire un risultato nel proprio nome, oltre a svolgere una serie di azioni al pari delle procedure.

Con la **PSEUDOCODIFICA** la funzione viene indicata come segue:

```
FUNZIONE <Nome Funzione> ( [ REF | VAL <Nome param 1>: <Tipo param 1> ,
                           REF | VAL <Nome param 2>: <Tipo param 2> ,
                           ……,
                           REF | VAL <Nome param n>: <Tipo param N> ] ) : < Tipo Risultato>
< Sezione dichiarativa Funzione>
INIZIO
< Corpo della Funzione>
RITORNA <Valore>
FINE
```

N.B. Gli specificatori di passaggio REF e VAL verranno illustrati nelle pagine seguenti

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

Una **funzione** è caratterizzata da:

- **un nome**, grazie al quale è possibile richiamarla ed identificarla univocamente;
- **una lista di parametri** che è *opzionale* e permette lo scambio in input e/o in output di informazioni tra il *programma chiamante* ed la funzione stessa ossia il *programma chiamato*;
- **un valore** ritornato direttamente nel nome della funzione.

IL PASSAGGIO DEI PARAMETRI

PREMESSA: Nonostante i **parametri formali** ed i **parametri attuali** possano anche avere lo stesso nome poiché in tal caso l'eventuale ambiguità verrebbe risolta grazie allo **shadowing,** è buona norma di programmazione utilizzare nomi differenti al fine di evitare inutili confusioni.

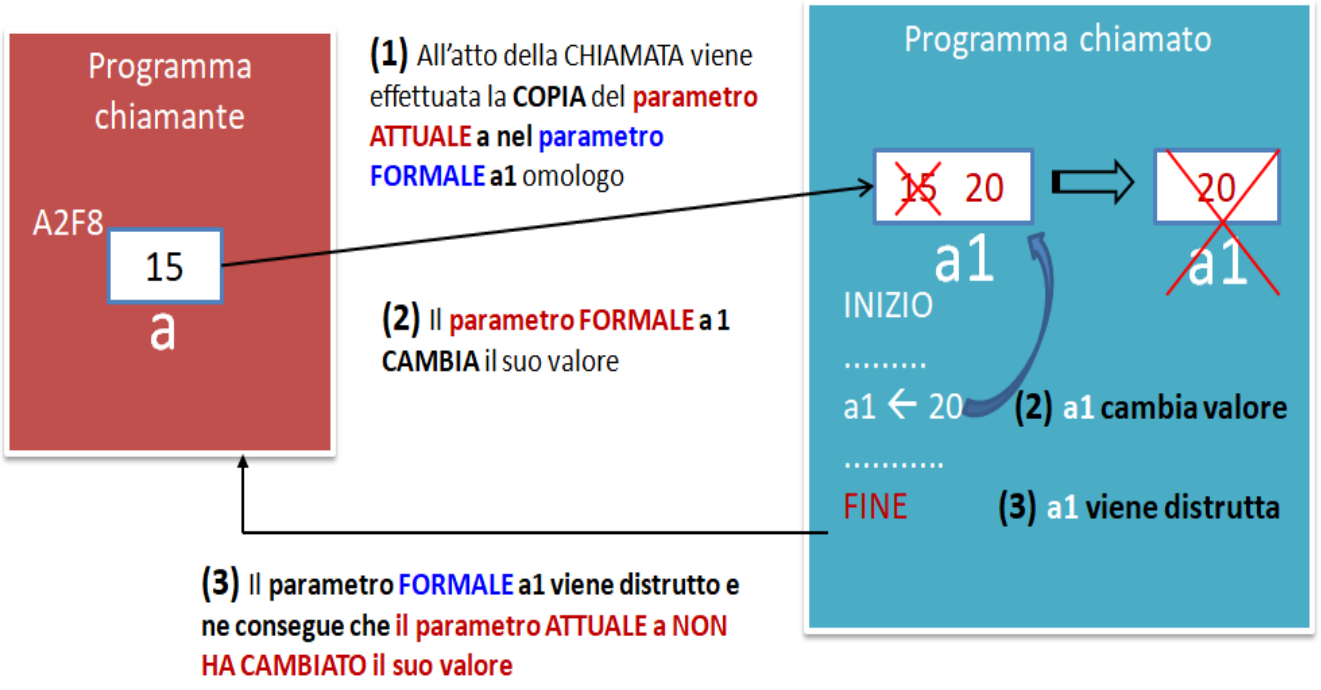
Con il termine **passaggio o trasmissione dei parametri** intendiamo l'operazione con la quale il valore dei **parametri attuali** (appartenenti al programma chiamante) viene associato (trasmesso) a quello dei **parametri formali** (appartenenti al programma chiamato).

Tale passaggio può avvenire secondo due modalità distinte:

a) **Passaggio dei parametri per VALORE o BY VALUE**

Con questa tipologia di passaggio si ha la **copia dei valori dei parametri attuali nei rispettivi parametri formali**. Durante l'esecuzione del sottoprogramma chiamato qualsiasi modifica apportata ai parametri formali sarà visibile solo all'interno del sottoprogramma stesso e **non** verrà riportata su i parametri attuali che continueranno a conservare il valore inizialmente trasmesso.
N.B. Viene allocata un'apposita area di memoria per i parametri formali che di fatto costituiscono una copia esatta dei parametri attuali.

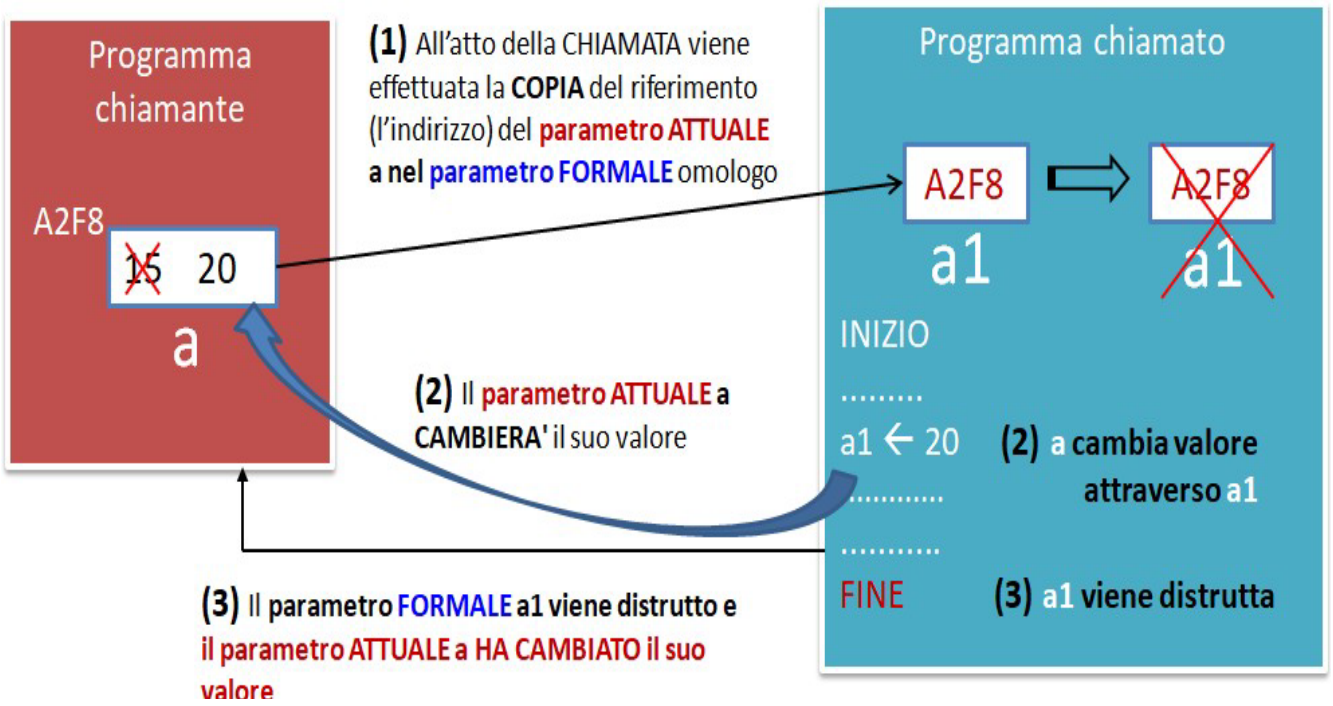
SCHEMATIZZAZIONE DI MASSIMA



b) **Passaggio dei parametri per RIFERIMENTO o BY REFERENCE (o per INDIRIZZO)**

Con questa tipologia di passaggio si ha la **copia dell'indirizzo dei parametri attuali nei rispettivi parametri formali**.
Quindi i parametri formali devono essere particolari tipi di variabili che sono in grado di contenenere indirizzi di memoria (**tipo REFERENCE**) che permettono, durante l'esecuzione del sottoprogramma chiamato, di **provocare una modifica** dei corrispondenti parametri attuali.
N.B. Come nel passaggio BY VALUE viene allocata un'apposita area di memoria per i parametri formali (PUNTATORI) che di fatto costituiscono una copia dell'indirizzo dei parametri attuali.

SCHEMATIZZAZIONE DI MASSIMA



NOTA BENE

Una variabile (parametro) **REFERENCE** contiene l'indirizzo di un'altra variabile (parametro) e permette di accedere al dato "riferito" o "puntato" **SENZA DOVER UTILIZZARE nella (PSEUDO)codifica l'OPERATORE * (**detto operatore di "DEFERENZIAZIONE")

invece

Una variabile (parametro) **PUNTATORE** contiene anch'essa l'indirizzo di un'altra variabile (parametro) ma permette di accedere al dato "riferito" o "puntato" **ESCLUSIVAMENE UTILIZZANDO nella (PSEUDO)codifica l'OPERATORE *** sopra citato.

NON TUTTI I LINGUAGGI DI PROGRAMMAZIONE PREVEDONO AMBEDUE I TIPI DI DATO (REFERENCE e PUNTATORE)
Ad esempio il **LINGUAGGIO C** ha solo variabili PUNTATORI per svolgere di passaggio di parametri per RIFERIMENTO, mentre il **LINGUAGGIO C++** le possiede entrambi.

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

1) **ESEMPIO svolto** (PROCEDURA): sia dato il seguente algoritmo contenente la procedura di nome *Passaggio1* e supponiamo di voler conoscere i valori delle variabili **x**, **y** e **z** mostrati a video nel *main* dopo la prima chiamata e dopo la seconda chiamata.

```
ALGORITMO Passaggio1
PROCEDURA main ( )
x, y, z : INT
INIZIO
Leggi (x)
Leggi (y)
Leggi (z)
/* Prima chiamata */
ChangeMe1 (y, z, x)
Scrivi (x)
Scrivi (y)
Scrivi (z)
/* Seconda chiamata */
ChangeMe1 (z, y, x)
Scrivi (x)
Scrivi (y)
Scrivi (z)
RITORNA
FINE

PROCEDURA ChangeMe1 (REF x: INT, VAL y: INT, REF z: INT)
i: INT
INIZIO
PER i ← 1 A z ESEGUI
  x ← 2*y – x + z
  y ← 2*x – y - z
  i ← i + 1
FINE PER
z ← z - 1
RITORNA
FINE
```


Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

1.a) Si supponga che inizialmente le variabili siano così valorizzate: **x = 2, y = 2, z = 3**

N.B. La scelta dei nomi e dell'ordine nel quale inserire i parametri attuali (programma chiamante) nella tabella di traccia è **ARBITRARIO**. Però una volta fissato, non va modificato

N.B. La scelta dei nomi e dell'ordine nel quale inserire i **parametri formali** (programma chiamato) nella tabella di traccia è **OBBLIGATO** e corrisponde all'ordine scelto dal progettista. Quindi occorre seguire il prototipo della procedura che, in questo caso, è:

PROCEDURA ChangeMe1 (**REF** x: **INT**, **VAL** y: **INT, REF** z: **INT**)

PROCEDURA main ( ) **PROCEDURA ChangeMe1 (...)**

x	y	z
2	2	3

2 3 2
ChangeMe1 (y, z, x)

i	REF x	VAL y	REF z	
-	2	3	2	Inizio ciclo
1	6	7	2	
2	10	11	2	Fine ciclo
3	10	11	1	

RITORNA

x	y	z
1	10	3

Dopo la prima chiamata

3 10 1
ChangeMe1 (z, y, x)

i	REF x	VAL y	REF z	
-	3	10	1	Inizio ciclo
1	18	25	1	Fine ciclo
2	18	25	0	

RITORNA

x	y	z
0	10	18

Dopo la seconda chiamata

Calcoli: 1° Chiamata (nella procedura **ChangeMe1**)

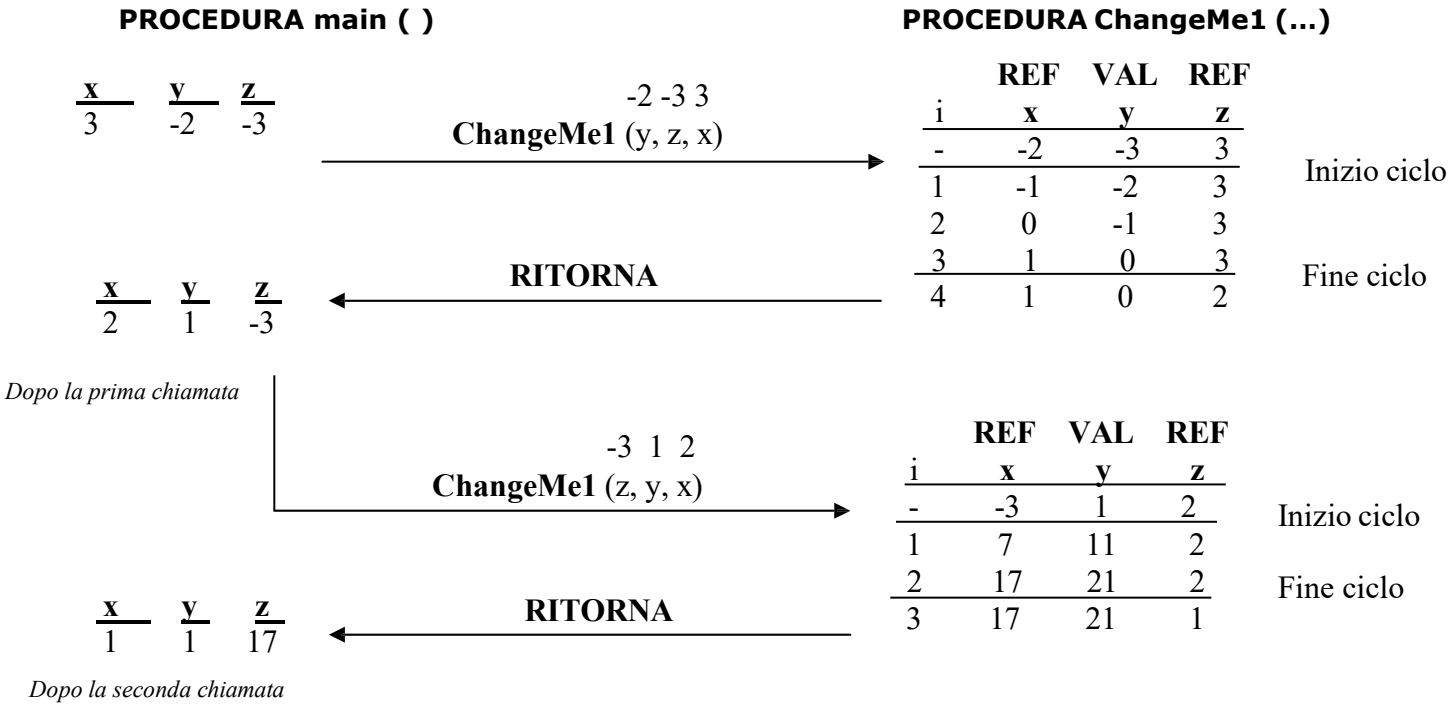
i ← 1 (i=1) prima che inizi ciclo PER
TEST PER (i ≤ z) ossia (1 ≤ 2) VERO
i=1 x ← 2*y – x + z (x = 2*3 – 2 + 2 = 6)
y ← 2*x – y – z (y = 2*6 – 3 – 2 = 7)
i ← i + 1 (i = 1 + 1 = 2)
TEST PER (i ≤ z) ossia (2 ≤ 2) VERO
i=2 x ← 2*y – x + z (x = 2*7 – 6 + 2 = 10)
y ← 2*x – y – z (y = 2*10 – 7 – 2 = 11)
i ← i + 1 (i = 2 + 1 = 3)
TEST PER (i ≤ z) ossia (3 ≤ 2) FALSO exit ciclo PER
Fuori ciclo PER z ← z – 1 (z = 2 – 1 = 1)

Calcoli: 2° Chiamata (nella procedura **ChangeMe1**)

i ← 1(i=1) prima che inizi ciclo PER
TEST PER (i ≤ z) ossia (1 ≤ 1) VERO
i=1 x ← 2*y – x + z (x = 2*10 – 3 + 1 =18)
y ← 2*x – y – z (y = 2*18 – 10 – 1 = 25)
i ← i + 1 (i = 1 + 1 = 2)
TEST PER (i ≤ z) ossia (2 ≤ 1) FALSO exit ciclo PER
Fuori ciclo PER z ← z – 1 (z = 1 – 1 = 0)

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

1.b) Si supponga che inizialmente le variabili siano così valorizzate: **x = 3, y = -2, z = -3**



Calcoli: 1° Chiamata (nella procedura **ChangeMe1**)

i ← 1 (i=1) prima che inizi ciclo PER

TEST PER (i ≤ z) ossia (1 ≤ 3) VERO

i=1 x ← 2*y – x + z (x = 2*(-3) – (-2) + 3 = -1)
y ← 2*x – y – z (y = 2*(-1) – (-3) – 3 = -2)
i ← i + 1 (i = 1 + 1 = 2)

TEST PER (i ≤ z) ossia (≤ 3) VERO

i=2 x ← 2*y – x + z (x = 2*(-2) – (-1) + 3 = 0)
y ← 2*x – y – z (y = 2*0 – (-2) – 3 = -1)
i ← i + 1 (i = 2 + 1 = 3)

TEST PER (i ≤ z) ossia (≤ 3) VERO

i=3 x ← 2*y – x + z (x = 2*(-1) – 0 + 3 = 1)
y ← 2*x – y – z (y = 2*1 – (-1) – 3 = 0)
i ← i + 1 (i = 3 + 1 = 4)

TEST PER (i ≤ z) ossia (≤ 3) FALSO exit ciclo PER

Fuori ciclo PER z ← z – 1 (z = 3 – 1 = 2)

Calcoli: 2° Chiamata (nella procedura **ChangeMe1**)

i ← 1(i=1) prima che inizi ciclo PER

TEST PER (i ≤ z) ossia (1 ≤ 2) VERO

i=1 x ← 2*y – x + z (x = 2*1 – (-3) + 2 = 7)
y ← 2*x – y – z (y = 2*7 – 1 – 2 = 11)
i ← i + 1 (i = 1 + 1 = 2)

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

TEST PER (i ≤ z) ossia (2 ≤ 2) VERO
i=2 x ← 2*y – x + z (x = 2*11 – 7 + 2 = 17)
y ← 2*x – y – z (y = 2*17 – 11 – 2 = 21)
i ← i + 1 (i = 2 + 1 = 3)
TEST PER (i ≤ z) ossia (3 ≤ 2) FALSO exit ciclo PER

Fuori ciclo PER z ← z – 1 (z = 2 – 1 = 1)

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

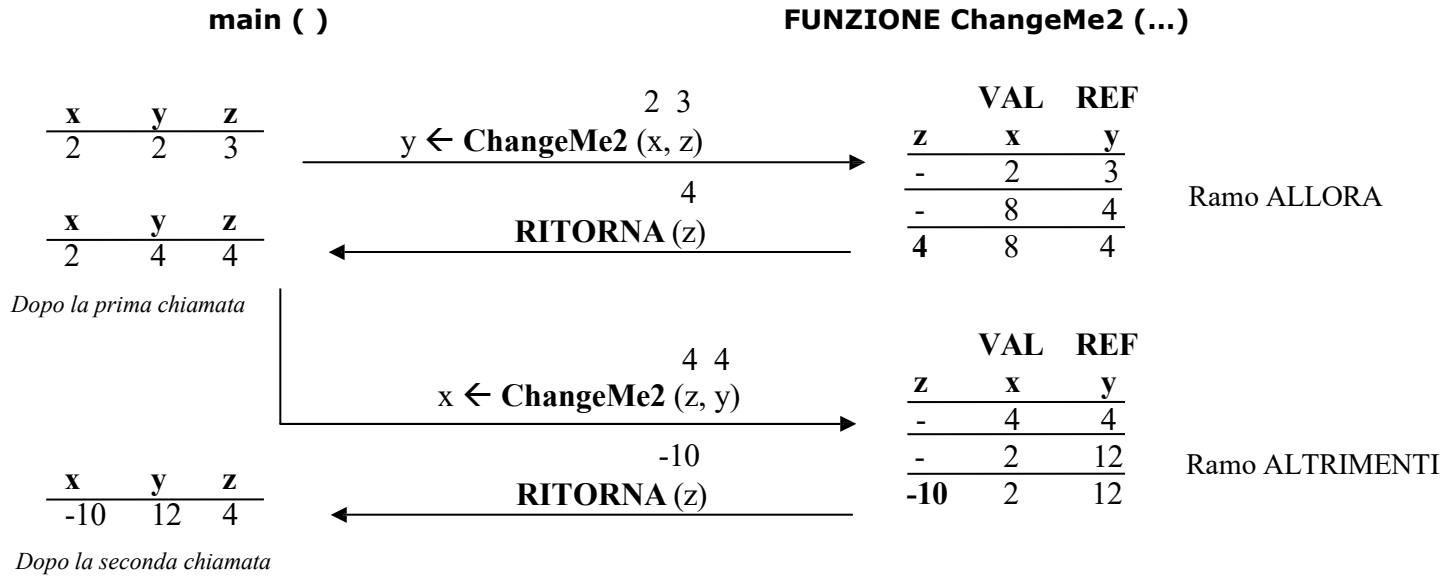
2) ESEMPIO svolto (FUNZIONE): sia dato il seguente algoritmo contenente la funzione di nome *Passaggio2* e supponiamo di voler conoscere i valori delle variabili **x**, **y** e **z** mostrati a video nel *main* dopo la prima chiamata e dopo la seconda chiamata.

```
ALGORITMO Passaggio2
PROCEDURA main ( )
x, y, z : INT
INIZIO
Leggi (x)
Leggi (y)
Leggi (z)
/* Prima chiamata */
y ← ChangeMe2 (x, z)
Scrivi (x)
Scrivi (y)
Scrivi (z)
/* Seconda chiamata */
x ← ChangeMe2 (z, y)
Scrivi (x)
Scrivi (y)
Scrivi (z)
RITORNA
FINE

FUNZIONE ChangeMe2 (VAL x: INT, REF y: INT) : INT
z: INT
INIZIO
SE (x < y)
  ALLORA
     x ← x + 2*y
     y ← y + 1
   ALTRIMENTI
     y ← y + 2*x
     x ← x - 2
FINE SE
z ← x - y
RITORNA (z)
FINE
```


Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

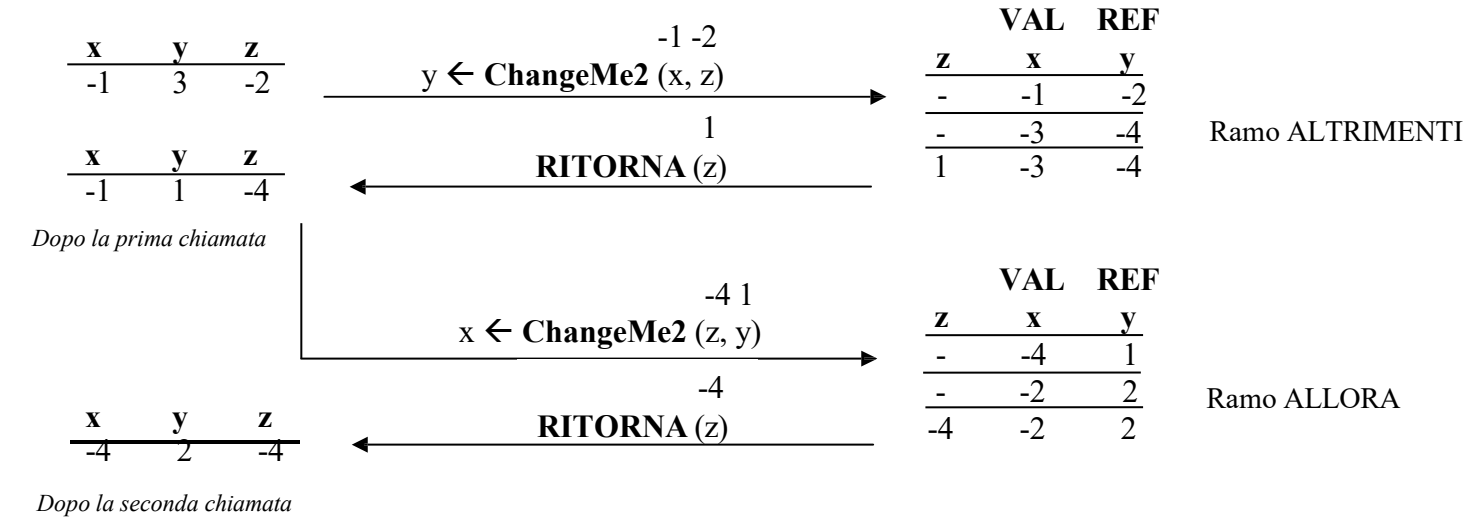
2.a) Si supponga che inizialmente le variabili siano così valorizzate: **x = 2, y = 2, z = 3**



Calcoli: 1° Chiamata (nella funzione **ChangeMe2**)
Test SE (x < y) ossia (2 < 3) VERO - RAMO ALLORA
x ← x + 2*y (x = 2 + 2*3 = 8)
y ← y + 1 (y = 3 + 1 = 4)
z ← x – y (z = 8 – 4 = 4)

Calcoli: 2° Chiamata (nella funzione **ChangeMe2**)
Test SE (x < y) ossia (4 < 4) FALSO - RAMO ALTRIMENTI
y ← y + 2*x (y = 4 + 2*4 = 12)
x ← x – 2 (x = 4 – 2 = 2)
z ← x – y (z = 2 – 12 = -10)

2.b) Si supponga che inizialmente le variabili siano così valorizzate: **x = -1, y = 3, z = -2**



Calcoli: 1° Chiamata (nella funzione **ChangeMe2**)
Test SE (x < y) ossia -1 < -2 FALSO -RAMO ALTRIMENTI
y ← y + 2*x (y = -2 + 2*(-1) = -4)
x ← x – 2 (x = -1 -2 = -3)
z ← x – y (z = (-3) – (-4) = 1)

Calcoli: 2° Chiamata (nella funzione **ChangeMe2**)
Test SE (x < y) ossia -4 < -1 VERO -RAMO ALLORA
x ← x + 2*y (x = -4 + 2*1 = -2)
y ← y + 1 (y = 1 + 1 = 2)
z ← x – y (z = -2 – 2 = -4)

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

3) ESEMPIO svolto (FUNZIONE con vettore): sia dato il seguente algoritmo

```
ALGORITMO Vettore1
PROCEDURA main( )
v: ARRAY[MAXDIM] DI INT
i, n, y: INT

INIZIO

/* Controllo della dimensione del vettore */
RIPETI
   Leggi (n)
FINCHE' (n>= 1) AND (n<= MAXDIM)

/* Caricamento del vettore */
PER i ← 1 A n ESEGUI
   v[i] ← (3*i + 4) % 5
   i ← i + 1
FINE PER

/* Invocazione funzione ChangeArray1 */
y ← ChangeArray1 (n, v)

/* Visualizzazione del vettore */
PER i ← 1 A n ESEGUI
   Scrivi (v[i])
   i ← i + 1
FINE PER

/* Visualizzazione variabile y */
Scrivi (y)

RITORNA
FINE

FUNZIONE ChangeArray1 (VAL n: INT, REF v: ARRAY[MAXDIM] DI INT) : INT
i, y : INT
INIZIO

y ← 2

PER i ← n INDIETRO A 2 ESEGUI
   y ← 3*y – 2*v[i-1]
   v[i-1]← (v[i] – 2) * i
   i ← i - 1
FINE PER
v[n] = (y+1) DIV 2

RITORNA (y)
FINE
```

Vedi DOMANDA 1)

Vedi DOMANDA 2)

Vedi DOMANDA 3)

Ipotizzando che l'utente immetta per la dimensione n il valore 3 (**ossia n = 3**) ed utilizzando apposite tabelle di traccia rispondi alle seguenti domande:

1) **Quale sarà il valore iniziale del vettore v subito dopo il caricamento?**
2) **Quale sarà il valore finale del vettore v subito dopo la visualizzazione?**
3) **Quale sarà il valore della variabile y?**

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

PROCEDURA main ( ) **FUNZIONE ChangeArray1 (...)**

n	v[1]	v[2]	v[3]	i
3	2	-	-	1
3	2	0	-	2
3	2	0	3	3
3	2	0	3	4

*Dopo il caricamento di v (**Risposta 1**)*

y ← **ChangeArray1** (n, 3 [2, 0, 3] v)

VAL	REF				
n	v[1]	v[2]	v[3]	i	y
3	2	0	3	-	2
3	2	3	3	3	6
3	2	3	3	2	14
3	2	3	7	1	**14**

14
RITORNA (y)

y	v[1]	v[2]	v[3]
14	2	3	7

*Dopo la visualizzazione di v e la chiamata alla funzione ChangeArray1 (**Risposte 2 e 3**)*

Calcoli: Caricamento vettore (nella procedura **main ( )**)

i ← 1 (i = 1) prima che inizi ciclo PER

TEST PER (i ≤ 3) ossia (1 ≤ 3) VERO

i = 1 v[i] ← (3*i + 4) % 5 (v[1] = (3*1 + 4) % 5 = 7 % 5 = 2)
i ← i + 1 (i = 1+ 1 = 2)

TEST P R (i ≤ 3) ossia (2 ≤ 3) VERO

i = 2 v[i] ← (3*i + 4) % 5 (v[1] = (3*2 + 4) % 5 = 10 % 5 = 0)
i ← i + 1 (i = 2+ 1 = 3)

TEST P R (i ≤ 3) ossia (3 ≤ 3) VERO

i = 3 v[i] ← (3*i + 4) % 5 (v[1] = (3*3 + 4) % 5 = 13 % 5 = 3)
i ← i + 1 (i = 1+ 1 = 4)

TEST PER (i ≤ 3) ossia (4 ≤ 3) FALSO exit ciclo PER di caricamento vettore

Calcoli: Modifica vettore (nella funzione **ChangeArray1 ( )**)

i ← 3 (i = 3) prima che inizi ciclo PER

TEST PER (i ≥ 2) ossia (3 ≥ 2) VERO

i = 3 y ← 3*y – 2*v[i-1] (y = 3*2 – 2*v[3-1] = 6 – 2*v[2] = 6 – 2*0 = 6)
v[i-1]← (v[i] – 2) * i (v[3-1] = (v[3] – 2) * 3 ossia v[2] = (3 – 2) * 3 = 3
i ← i – 1 (i = 3 - 1 = 2)

TEST PER (i ≥ 2) ossia (2 ≥ 2) VERO

i = 2 y ← 3*y – 2*v[i-1] (y = 3*6 – 2*v[2-1] = 18 – 2*v[1] = 18 – 2*2 = 14)
v[i-1]← (v[i] – 2) * i (v[2-1] = (v[2] – 2) * 2 ossia v[2] = (3 – 2) * 2 = 2
i ← i - 1 (i = 2 - 1 = 1)

TEST PER (i ≥ 2) ossia (1 ≥ 2) FALSO exit ciclo PER all'interno della funzione
v[n] = (y+1) DIV 2 (v[3] = (14 + 1) DIV 2 = 15 DIV 2 = 7 FUORI ciclo PER

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

4) ESEMPIO svolto (FUNZIONE con vettore): sia dato il seguente algoritmo:

```
ALGORITMO Vettore2
PROCEDURA main( )
v: ARRAY[MAXDIM] DI INT
i, n, y: INT
INIZIO
/* Controllo della dimensione del vettore */
……..
/* Caricamento del vettore */
……..
/* Invocazione funzione ChangeArray2 */
y ← ChangeArray2 (n, v)
/* Visualizzazione del vettore */
PER i ← 1 A n ESEGUI
   Scrivi (v[i])
   i ← i + 1
FINE PER
/* Visualizzazione variabile x */
Scrivi (y)

RITORNA
FINE

FUNZIONE ChangeArray2 (VAL n: INT, REF v: ARRAY[MAXDIM] DI INT) : INT
i, y : INT
INIZIO
y ← 5
PER i ← 1 A n-1 ESEGUI
   y ← v[i+1] DIV 2*i
   v[i]← v[i+1] % i
   i ← i + 1
FINE PER
v[n] ← (y-1) * 4
v[1] ← v[n] DIV 3
y ← v[n] – v[1]
RITORNA (y)
FINE
```

Vedi DOMANDA 1)

Vedi DOMANDA 2)

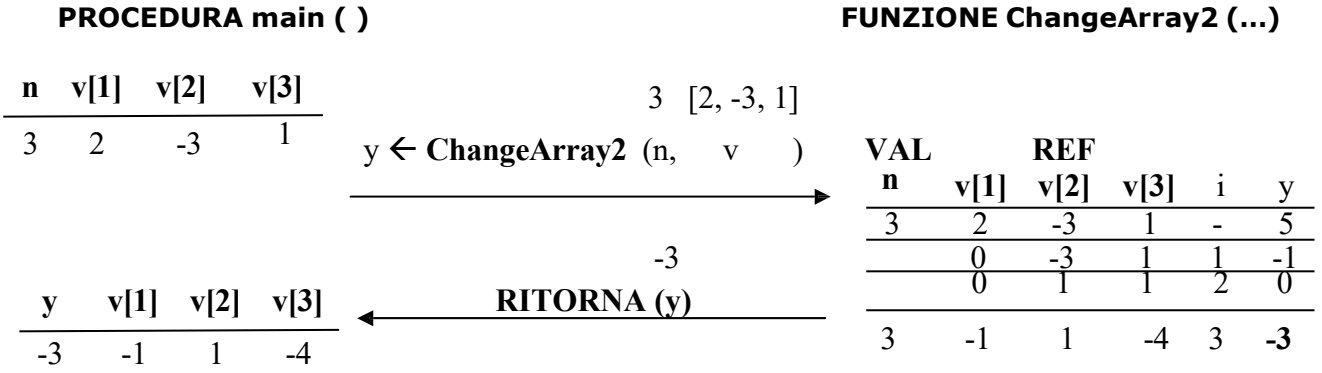
Ipotizzando che l'utente immetta il seguente vettore:

v = [2, -3, 1]

ed utilizzando apposite tabelle di traccia rispondi alle seguenti domande:

1) **Quale sarà il valore finale del vettore v subito dopo la visualizzazione?**
2) **Quale sarà il valore della variabile y?**

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)



*Dopo la visualizzazione di v e la chiamata alla funzione ChangeArray2 (**Risposte 1 e 2**)*

Calcoli: Modifica vettore (nella funzione **ChangeArray2 ( )**)

i ← 1 (i = 1) prima che inizi ciclo PER
TEST PER (i ≤ 2) ossia (1 ≤ 2) VERO
i = 1 y ← v[i+1] DIV 2*i (y = v[1+1] DIV (2*1) = v[2] DIV 2 = (-3) DIV 2 = -1)
v[i]← v[i+1] % i (v[1] = v[1+1] % 1 = v[2] % 1 = (-3) % 1 = 0)
i ← i - 1 (i = 1 + 1 = 2)

TEST PER (i ≤ 2) ossia (2 ≤ 2) VERO
i = 2 y ← v[i+1] DIV 2*i (y = v[2+1] DIV (2*2) = v[3] DIV 4 = 1 DIV 4 = 0)
v[i]← v[i+1] % i (v[2] = v[2+1] % 2 = v[3] % 2 = 1 % 2 = 1)
i ← i - 1 (i = 2 + 1 = 3)

TEST PER (i ≤ 2) ossia (3 ≤ 2) FALSO exit ciclo all'interno della funzione
Fuori ciclo v[n] ← (y-1) * 4 (v[3] = (0-1) * 4 = -4)
v[1] ← v[n] DIV 3 (v[1] = (-4) DIV 3 = -1)
y ← v[n] – v[1] (y = (-4) – (-1) = -4 + 1 = -3)

5) ESEMPIO (FUNZIONE con vettore): sia dato lo pseudocodice della seguente funzione:

```
FUNZIONE ChangeArray2 (VAL n: INT, REF v: ARRAY[MAXDIM] DI INT) : INT
i, y : INT
INIZIO

y ← 5

PER i ← 1 A n-1 ESEGUI
   y ← v[i+1] DIV 2*i
   v[i]← v[i+1] % i
   i ← i + 1
FINE PER

v[n] ← (y-1) * 4
v[1] ← v[n] DIV 3
y ← v[n] – v[1]

RITORNA (y)
FINE
```

Ipotizzando che l'utente immetta il seguente vettore:

v = [2, -3, 1]

ed utilizzando apposite tabelle di traccia rispondi alle seguenti domande:

1. **Quale sarà il valore finale del vettore v dopo l'esecuzione della funzione?**
2. **Quale sarà il valore restituito dalla funzione?**

N.B Si tratta di una versione semplificata dell'esercizio precedente per il quale vale lo stesso svolgimento

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

LA RICORSIONE

Abbiamo visto che nel corpo di un sottoprogramma è possibile chiamare (attivare) altri sottoprogrammi dichiarati esternamente o localmente ad esso, ma è anche possibile, in determinate condizioni, fornire l'istruzione per chiamare (riattivare) se stesso.

In questo caso si parla di **sottoprogrammi ricorsivi**, una caratteristica prevista da alcuni linguaggi di programmazione (ad esempio per il C è prevista).

DEF: In generale si definisce **ricorsività o ricorsione** la possibilità di definire un problema riferendosi (ossia ricorrendo) alla sua stessa definizione.

Per potere attivare un **processo risolutivo di tipo ricorsivo** occorre rispettare i seguenti tre requisiti:

1. il problema principale (**caso generale**) può essere scomposto in sottoproblemi dello stesso tipo, ognuno dipendente dall'altro in base ad una **scala gerarchica** (ossia con ordine di complessità inferiore);
2. è necessario conoscere la soluzione di un **caso particolare** del problema principale; ciò è indispensabile per poter arrestare la ricorsione (**condizione di terminazione** della ricorsione);
3. devono essere note le relazioni funzionali che legano il problema principale (**caso generale**) ai sottoproblemi simili

DEF 1: Un sottoprogramma implementa la **ricorsione DIRETTA** quando nella sua definizione compare **UNA SOLA CHIAMATA** al sottoprogramma stesso.

Primo esempio: POTENZA N-ESIMA DI UN NUMERO

In matematica sono possibili due definizioni generali di potenza n-sima di un numero **a** intero non nullo elevato ad un esponente **n** intero non negativo.

- definizione **non ricorsiva**

$$\begin{cases} a^n = 1 & \text{se } n = 0 \text{ e } a \neq 0 \\ a^n = \underbrace{a * a * \ldots\ldots\ldots * a}_{\text{n volte}} & \text{se } n > 0 \end{cases}$$

- definizione **ricorsiva**

$$\begin{cases} a^n = 1 & \text{se } n = 0 \text{ e } a \neq 0 \\ a^n = a * a^{(n-1)} & \text{se } n > 0 \end{cases}$$

Questo problema è stato espresso in termini ricorsivi, in quanto risponde ai tre requisiti enunciati poco fa. Infatti:

1. sappiamo che calcolare a^n dipende esclusivamente dal calcolo di $a^{(n-1)}$ **(scala gerarchica);**
2. conosciamo la soluzione del **caso particolare** che $a^0 = 1$ **(condizione di terminazione);**
3. abbiamo una **relazione funzionale** $a^n = a * a^{(n-1)}$ che lega il problema principale (a^n) al sottoproblema simile ma di complessità minore ($a^{(n-1)}$).

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

Ed ipotizziamo di implementare la pseudocodifica della seguente funzione RICORSIVA DIRETTA di nome *Potenza*

FUNZIONE Potenza (VAL base : INT, VAL esp: INT) : INT

pot : INT

INIZIO

SE (esp = 0)
ALLORA
pot ← 1
ALTRIMENTI
pot ← base * *Potenza (base, (esp -1))* /* Chiamata ricorsiva DIRETTA*/
FINE SE

RITORNA (pot)

FINE

Secondo esempio: FATTORIALE DI UN NUMERO

In matematica sono possibili due definizioni generali di fattoriale di un numero **n** intero positivo.

- definizione **non ricorsiva**

$$\begin{cases} \mathbf{0! = 1} & \mathbf{se\ n = 0} \\ \mathbf{n! = n * (n-1) * (n-2)* \ldots *2*1} & \mathbf{se\ n > 0} \end{cases}$$

- definizione **ricorsiva**

$$\begin{cases} \mathbf{0! = 1} & \mathbf{se\ n = 0} \\ \mathbf{n! = n * (n-1)!} & \mathbf{se\ n > 0} \end{cases}$$

Anche questo problema è stato espresso in termini ricorsivi, in quanto risponde ai tre requisiti enunciati poco fa.
Infatti:

1. sappiamo che calcolare **n!** dipende esclusivamente dal calcolo di **(n-1)! (scala gerarchica);**
2. conosciamo la soluzione del **caso particolare** che **0! = 1 (condizione di terminazione);**
3. abbiamo una **relazione funzionale n! = n * (n-1)!** che lega il problema principale (**n!**) ad un sottoproblema simile ma di complessità minore (**(n-1)!**).

Ed ipotizziamo di implementare la pseudocodifica della seguente funzione RICORSIVA DIRETTA di nome *Fattoriale*

FUNZIONE Fattoriale (VAL num : INT) : INT

fatt : INT

INIZIO

SE (num = 0)
ALLORA
fatt ← 1
ALTRIMENTI
fatt ← num * *Fattoriale (num -1)* /* Chiamata ricorsiva DIRETTA */
FINE SE

RITORNA (fatt)

FINE

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

Per maggiore chiarezza ipotizziamo di volere calcolare il valore della chiamata ricorsiva **Fattoriale (5)** (che corrisponde in matematica a **5!**) e schematizziamo la sequenza dei passaggi:

Il programma chiamante invoca la funzione *Fattoriale* per calcolare il valore di **5!** ossia con un'istruzione di chiamata del tipo
f ← *Fattoriale(5)*
(il valore 5 è il valore del parametro attuale)

Il risultato ottenuto alla fine del procedimento ricorsivo è che la variabile f del programma chiamante vale 120 ossia il valore di 5!

1° passaggio ricorsivo:	fatt ← 5 * Fattoriale (4)	**5° calcolo:**	5 * 24 = 120
2° passaggio ricorsivo:	fatt ← 4 * Fattoriale (3)	**4° calcolo:**	4 * 6 = 24
3° passaggio ricorsivo:	fatt ← 3 * Fattoriale (2)	**3° calcolo:**	3 * 2 = 6
4° passaggio ricorsivo:	fatt ← 2 * Fattoriale (1)	**2° calcolo:**	2 * 1 = 2
5° passaggio ricorsivo:	fatt ← 1 * **Fattoriale (0)**	**1° calcolo:**	1 * 1 = 1
6° passaggio ricorsivo:	fatt ← 1		

Poiché n=0 allora fattoriale **vale 1**
(**condizione di terminazione**)

Questo processo ricorsivo così schematizzato viene gestito per mezzo della **pila delle attivazioni o stack** di cui abbiamo già parlato in precedenza, ricordando che ad ogni invocazione del sottoprogramma ricorsivo viene salvato nello stack l'indirizzo dell'istruzione cui ritornare che nel nostro caso coincide con l'istruzione di moltiplicazione **(in grassetto)**

fatt ← ***num * Fattoriale (num -1)***

Vediamo in dettaglio cosa succede quando il programma chiamante invocherà la funzione *Fattoriale* attraverso una chiamata del tipo

….
f ← Fattoriale (5)
….

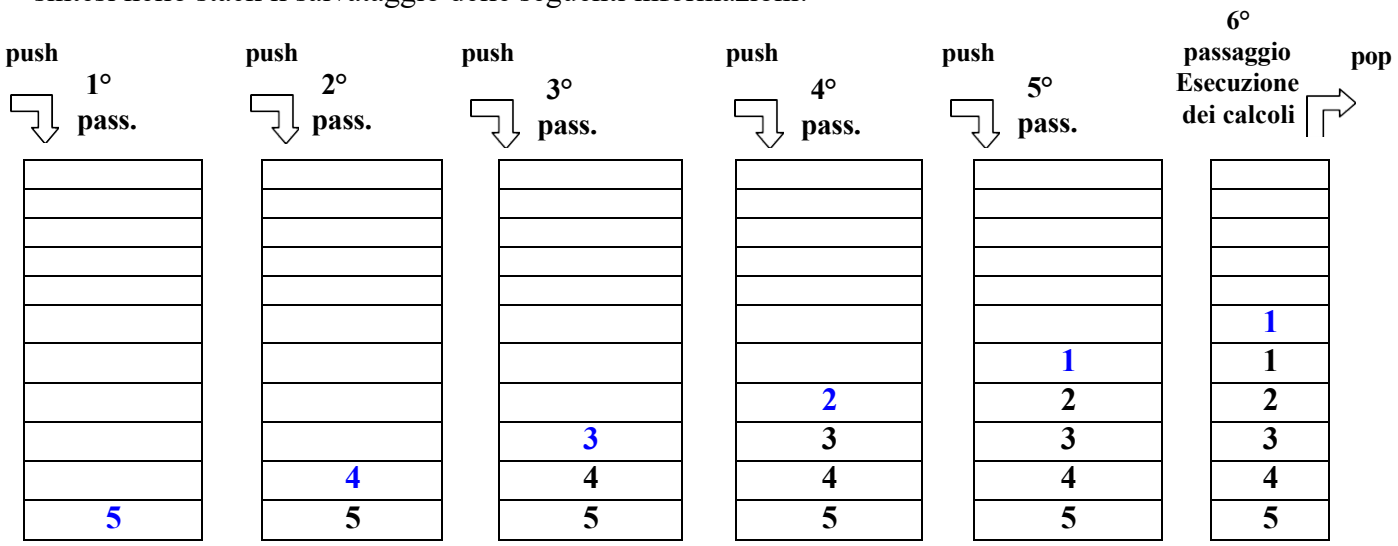
Il valore del parametro attuale (costante intera) trasferito al parametro formale *num* è 5.

Attivata la funzione viene eseguita l'istruzione
fatt ← *num * Fattoriale (num – 1)*
che non può essere eseguita e risolta direttamente in quanto contiene una nuova chiamata allo stesso sottoprogramma (procedura ricorsiva).

Viene memorizzato nella pila delle attivazioni l'indirizzo dell'istruzione di ritorno che nel nostro caso è l'istruzione di moltiplicazione contenuta nell'istruzione.
fatt ← *num * Fattoriale (num – 1)*

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

Ora si deve eseguire nuovamente la funzione per il valore *num -1* ossia 4 e così via: avremo in sintesi nello stack il salvataggio delle seguenti informazioni:



N.B. In realtà la sequenza delle chiamate alla funzione ricorsiva attivate dalla chiamata del programma chiamante è la seguente:

f ← *Fattoriale (5)*

1° call ricorsiva f ← 5 * *Fattoriale (4)*

2° call ricorsiva f ← 5 * (4 * *Fattoriale (3)*)

3° call ricorsiva f ← 5 * (4 * (3 * *Fattoriale (2)*))

4° call ricorsiva f ← 5 * (4 * (3 * (2 * *Fattoriale (1)*)))

5° call ricorsiva f ← 5 * (4 * (3 * (2 * (1 * *Fattoriale (0)*))))

6° call ricorsiva f ← 5 * (4 * (3 * (2 * (1 * 1))))

Alla fine delle chiamate ricorsive viene seguito il calcolo al contrario ossia eseguendo i prodotti a partire da quello più interno risalendo verso quello più esterno ossia

1 * 1 = 1

1 * 2 = 2

2 * 3 = 6

6 * 4 = 24

24 * 5 = 120

N.B. Si potevano tranquillamente decidere di implementare ricorsivamente sia il calcolo della potenza ennesima di un numero, sia il calcolo del fattoriale di un numero utilizzando come sottoprogramma una procedura al posto di una funzione.
Bastava infatti aggiungere tra i parametri formali delle nuove procedure implementate il parametro relativo al valore di interesse passato per riferimento.
Infatti:

PROCEDURA Potenza (VAL base : INT, VAL esp: INT, **REF pot : INT**)

INIZIO

SE (esp = 0)
ALLORA
pot ← 1
ALTRIMENTI
Potenza (base, (esp -1), pot) /* Chiamata ricorsiva DIRETTA */
*pot ← base * pot*
FINE SE

RITORNA

FINE

PROCEDURA Fattoriale (VAL num : INT, **REF fatt: INT**)

INIZIO

SE (num = 0)
ALLORA
fatt ← 1
ALTRIMENTI
Fattoriale ((num -1), fatt) /* Chiamata ricorsiva DIRETTA */
*fatt ← num * fatt*
FINE SE

RITORNA

FINE

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

ALTRI TIPI DI RICORSIONE

DEF 2: Un sottoprogramma implementa la **ricorsione MULTIPLA** quando nella sua definizione compaiono **ALMENO DUE CHIAMATE** al sottoprogramma stesso.

Primo esempio: la successione di FIBONACCI

L'intento di Fibonacci era quello di trovare una legge matematica che potesse descrivere la crescita di una popolazione di conigli individuando e registrandone la ragione quantitativa di incremento.

I primi numeri di Fibonacci sono:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…

In matematica la definizione ricorsiva della serie di Fibonacci relativa ad un numero **n** intero non negativo

$$\begin{cases} \mathbf{Fib(n) = 1} & \mathbf{se\ n = 0} \\ \mathbf{Fib(n) = 1} & \mathbf{se\ n = 1} \\ \mathbf{Fib(n) = Fib(n-1) + Fib(n-2)} & \mathbf{se\ n >= 2} \end{cases}$$

Una possibile pseudocodifica di una funzione RICORSIVA MULTIPLA per calcolare i numeri di Fibonacci di nome *Fibonacci* è la seguente:

```
FUNZIONE Fibonacci (VAL num : INT) : INT
fib : INT
INIZIO
SE ((num = 0) OR (num = 1))
   ALLORA
      fib ← 1
   ALTRIMENTI
      fib ← Fibonacci (num -1) + Fibonacci (num-2)      /* Chiamata ricorsiva MULTIPLA*/
FINE SE
RITORNA ( fib )
FINE
```

N.B. Due chiamate ricorsive

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

Vediamo in dettaglio cosa succede quando il programma chiamante invocherà la funzione *Fibonacci* attraverso una chiamata del tipo

….
fib ← Fibonacci (5)
….
Il valore del parametro attuale (costante intera) trasferito al parametro formale *num* è 5.

Attivata la funzione viene eseguita l'istruzione

1° CALL *fib* ← *Fibonacci (4) + Fibonacci (3)*

che non può essere eseguita e risolta direttamente in quanto contiene due chiamate allo stesso sottoprogramma (procedura ricorsiva multipla) che conduce alla seguente successione

2° CALL *fib* ← *(Fibonacci (3) + Fibonacci (2)) + Fibonacci (3)*

3° CALL *fib* ← *((Fibonacci (2) + Fibonacci (1)) + Fibonacci (2) + Fibonacci (3)*

4° CALL *fib* ← *((Fibonacci (1) + Fibonacci (0)) +Fibonacci (1) + Fibonacci (2) + Fibonacci (3)*

5° CALL *fib* ← **1**+ *(Fibonacci (0)) +Fibonacci (1) + Fibonacci (2) + Fibonacci (3)*

6° CALL *fib* ← **1**+ **1** + *(Fibonacci (1)) + Fibonacci (2) + Fibonacci (3)*

7° CALL *fib* ← **1**+ **1** + **1** + *Fibonacci (2) +Fibonacci (3)*

8° CALL *fib* ← **1**+ **1** + **1** + *((Fibonacci (1) + Fibonacci (0)) + Fibonacci (3)*

9° CALL *fib* ← **1**+ **1** + **1** + **1** + *(Fibonacci (0)) + Fibonacci (3)*

10° CALL *fib* ← **1**+ **1** + **1** + **1** + **1** + *Fibonacci (3)*

11° CALL *fib* ← **1**+ **1** + **1** + **1** + **1** + *((Fibonacci (2) + Fibonacci (1))*

12° CALL *fib* ← **1**+ **1** + **1** + **1** + **1** + *((Fibonacci (1) + Fibonacci (0)) + Fibonacci (1)*

13° CALL *fib* ← **1**+ **1** + **1** + **1** + **1** + **1**+ *(Fibonacci (0)) + Fibonacci (1)*

14° CALL *fib* ← **1**+ **1** + **1** + **1** + **1** + **1**+ **1**+ *Fibonacci (1)*

15° CALL *fib* ← **1**+ **1** + **1** + **1** + **1** + **1** + **1**+ **1 = 8**

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

DEF 3: Si parla invece di ricorsione **INDIRETTA** quando nella definizione di un sottoprogramma compare la chiamata ad un altro sottoprogramma il quale, **direttamente o indirettamente**, chiama il sottoprogramma iniziale

Esempio: funzioni cooperanti

```
FUNZIONE F1 (VAL n : INT) : INT
esito: INT
INIZIO
SE (n < 1)
   ALLORA
      esito ← 1
   ALTRIMENTI
      esito ← F2 (n-1)
FINE SE
RITORNA (esito)
FINE
```

```
FUNZIONE F2 (VAL n : INT) : INT
esito: INT
INIZIO
SE (n < 2)
   ALLORA
      esito ← 0
   ALTRIMENTI
      esito ← F1 (n-2)
FINE SE
RITORNA (esito)
FINE
```

VANTAGGI DEGLI ALGORITMI RICORSIVI RISPETTO A QUELLI ITERATIVI

a) Un algoritmo ricorsivo **è più facile da realizzare** perché risolve un problema complesso riducendolo a problemi dello stesso tipo ma su dati di ingresso più semplici

b) Un algoritmo ricorsivo è **più facile da capire da parte di altri programmatori** che non siano autori di quel software

SVANTAGGI DEGLI ALGORITMI RICORSIVI RISPETTO A QUELLI ITERATIVI

a) Per poter utilizzare un algoritmo ricorsivo occorre **un grosso sforzo iniziale** da parte del progettista/programmatore per poter acquisire una *visione ricorsiva* del problema posto

b) E' facilmente possibile avere ricorsioni che non terminano mai (**ricorsione infinita**) ossia un sottoprogramma che richiama se stesso infinite volte o perché è stata omessa la clausola di terminazione o perché i valori del parametro ricorsivo non si semplificano mai. Dopo un certo numero di chiamate la memoria disponibile per quel sottoprogramma si esaurisce ed esso viene terminato automaticamente con segnalazione **di errore di overflow nello stack** conseguente **crash** del programma in esecuzione

c) **Eccessiva occupazione dello spazio di memoria** attraverso l'utilizzo di variabili locali non necessarie

d) **Aumento dei tempi di esecuzione (soluzione meno efficiente)** soprattutto degli algoritmi ricorsivi che presentano ricorsioni multiple dovute al proliferare delle chiamate ricorsive con conseguente aggravio dei costi di chiamata di un sottoprogramma

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

Esempi di problemi con la ricorsione

(A) Secondo la matematica la definizione ricorsiva della somma dei primi **n** numeri interi strettamente positivi può essere formalizzata nel seguente modo:

$$\begin{cases} \textbf{Somma (n) = 1} & \textbf{se n = 1} \\ \textbf{Somma (n) = n + Somma(n-1)} & \textbf{se n > 1} \end{cases}$$

Possiamo quindi scrivere il seguente pseudocodice della funzione ricorsiva Somma

FUNZIONE Somma (VAL n : INT) : INT
s: INT
INIZIO
SE (n =1) ← **Clausola di terminazione**
  ALLORA
    s ← 1
  ALTRIMENTI
    s ← n + *Somma(n-1)* ← **Il valore di n si semplifica**
FINE SE
RITORNA (s)
FINE

Dettagliamo la seguente chiamata ricorsiva
….
s ← Somma(5)
…..
Il valore del parametro attuale (costante intera) trasferito al parametro formale *n* è 5.

Attivata la funzione viene eseguita l'istruzione
s ← 5 + Somma (4)
che non può essere eseguita e risolta direttamente in quanto contiene una chiamata allo stesso sottoprogramma (procedura ricorsiva diretta) che conduce alla seguente successione

s ← 5 + (4 +Somma (3))

s ← 5 + (4 +(3 + Somma (2)))

s ← 5 + (4 +(3 + (2 + Somma (1))))

s ← 5 + (4 +(3 + (2 + (1)))) che è uguale a **15**

Ricorsione infinita: i valori del parametro non si semplificano

FUNZIONE Somma (VAL n : INT) : INT
s: INT
INIZIO
SE (n =1) ← **Clausola di terminazione**
  ALLORA
    s ← 1
  ALTRIMENTI
    s ← n + *Somma(n)* ← **Il valore di n NON si semplifica**
FINE SE
RITORNA (s)
FINE

Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

La chiamata ricorsiva precedente

….
s ← Somma(5)
…..

non verrà mai risolta perché conduce alla seguente successione di chiamate ricorsive infinite

s ← 5 + (5 +Somma (5))

s ← 5 + (5 +(5 + Somma (5)))

s ← 5 + (5 +(5 + (5 + Somma (5))))

s ← 5 + (5 +(5 + (5 + (5)))) ……..

Ricorsione infinita: manca la clausola di terminazione

FUNZIONE Somma (VAL n : INT) : INT
s: INT
INIZIO
    s ← n + *Somma(n-1)*
RITORNA (s)
FINE

La chiamata ricorsiva precedente

….
s ← Somma(5)
…..

non verrà mai risolta perché conduce alla seguente successione di chiamate ricorsive infinite

s ← 5 + (4 +Somma (3))

s ← 5 + (4 +(3 + Somma (2)))

s ← 5 + (4 +(3 + (2 + Somma (1))))

s ← 5 + (4 +(3 + (2 + (1 + Somma (-1)))))……

(B) Consideriamo ad esempio le seguenti funzioni cooperanti che permettono di restituire 0 se il numero è pari, 1 se invece è dispari

```
FUNZIONE Ping (VAL n : INT) : INT
esito: INT
INIZIO
SE (n < 0)
   ALLORA
      esito ← 1
   ALTRIMENTI
      esito ← Pong (n-1)
FINE SE
RITORNA (esito)
FINE
```

```
FUNZIONE Pong(VAL n : INT) : INT
esito: INT
INIZIO
SE (n < 0)
   ALLORA
      esito ← 0
   ALTRIMENTI
      esito ← Ping (n-1)
FINE SE
RITORNA (esito)
FINE
```


Autore: Rio Chierego (email: riochierego@libero.it - sito web: www.riochierego.it)

Dettagliamo la seguente chiamata ricorsiva

....

esito ← Ping (2)

.....

Il valore del parametro attuale (costante intera) trasferito al parametro formale *n* è 2.

Attivata la funzione viene eseguita l'istruzione

esito ← (Pong (1))

esito ← (Ping (0))

esito ← (Pong (-1))

esito ← **0**

Se per errore scrivessimo

```
FUNZIONE Ping (VAL n : INT) : INT
esito: INT
INIZIO
SE (n < 0)
   ALLORA
      esito ← 1
   ALTRIMENTI
      esito ← Pong (n)
FINE SE
RITORNA (esito)
FINE
```

```
FUNZIONE Pong(VAL n : INT) : INT
esito: INT
INIZIO
SE (n < 0)
   ALLORA
      esito ← 0
   ALTRIMENTI
      esito ← Ping (n)
FINE SE
RITORNA (esito)
FINE
```

avremmo avuto per la chiamata ricorsiva precedente

....

esito ← Ping (2)

.....

la seguente successione infinita di chiamate ricorsive

esito ← (Pong (2))

esito ← (Ping (2))

esito ← (Pong (2))

……