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.

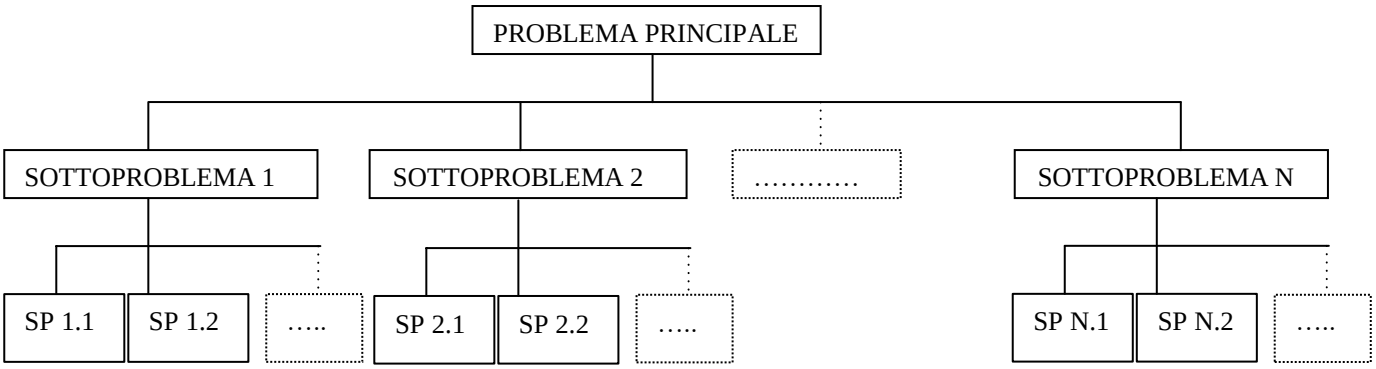
Il programmatore deciderà all'atto dell'implementazione del programma se implementare tutti i sottoproblemi individuati o, nonostante la loro individuazione, accorparne alcuni.
Tale metodologia è diventata una tra le principali tecniche di progettazione software.

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:



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 (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.

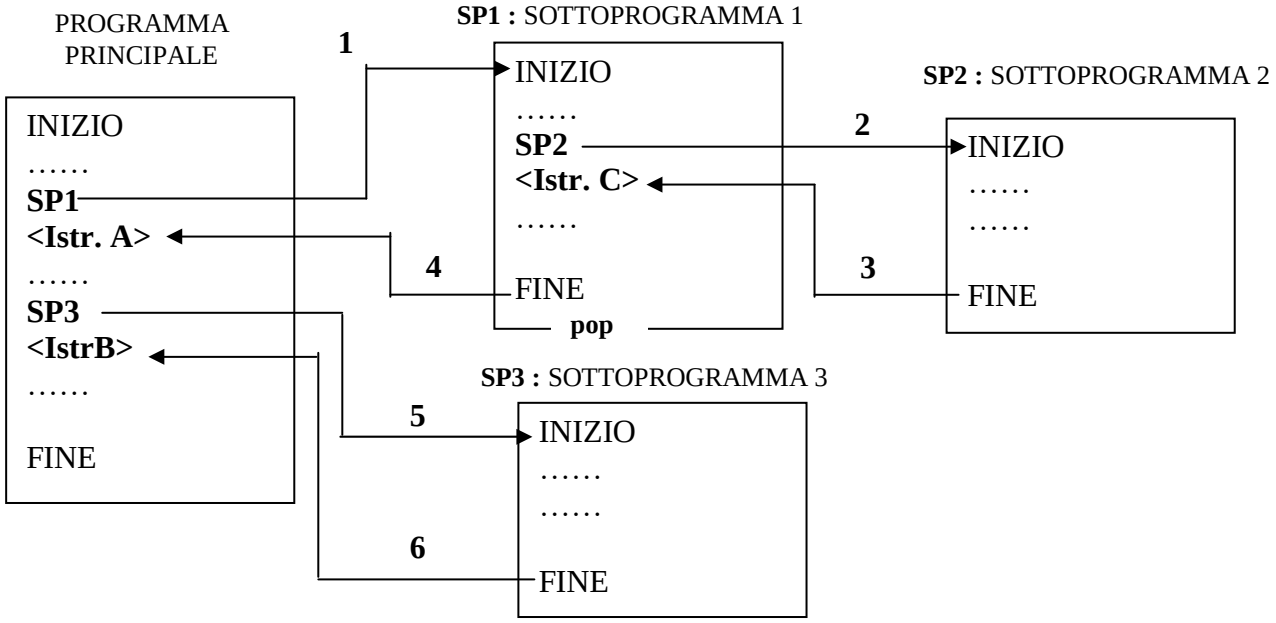
L'esecuzione di un sottoprogramma

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 la CPU incontra una istruzione di chiamata a sottoprogramma (nel nostro esempio successivo indicata con **SP1, SP2, SP3 ossia SP<n>)** sospende l'esecuzione del programma corrente e passa ad eseguire le istruzioni contenute nel sottoprogramma chiamato. Terminata l'esecuzione la CPU quando incontra la parola FINE riprende l'esecuzione del programma ripartendo dall'istruzione successiva a quella di chiamata.
Il sottoprogramma chiamato viene caricato in memoria centrale al momento della chiamata e terminata l'esecuzione viene rilasciato liberando la memoria precedentemente occupata **(allocazione dinamica del codice da parte del sistema operativo).**

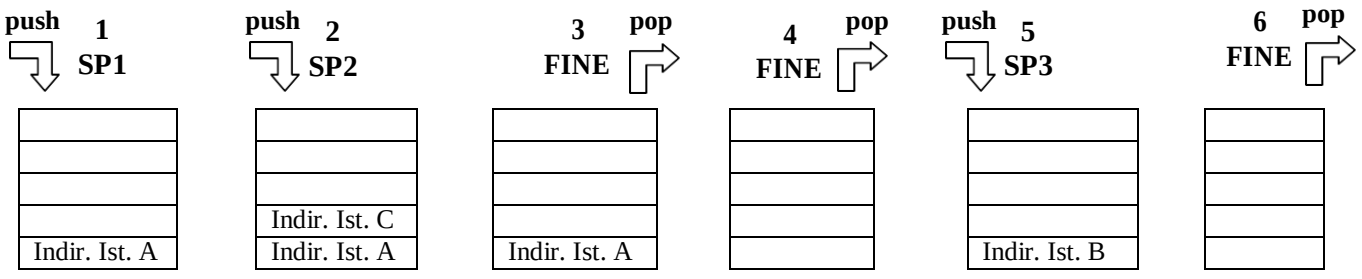
Per ricordare da quale istruzione va ripresa l'esecuzione dopo un sottoprogramma, la CPU si serve di una apposita struttura detta **PILA delle attivazioni o STACK** che è una particolare struttura dalla quale i dati possono essere inseriti o estratti solo da una estremità che viene detta **testa della pila**. Essa 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 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 **l'indirizzo di memoria in esso contenuto da dove riprendere l'esecuzione.**

Nel nostro caso l'utilizzo della pila delle attivazioni è la seguente:



LE PROCEDURE

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

In 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

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*.

LE FUNZIONI

DEF: La funzione è un sottoprogramma che, attivato dall'apposita istruzione di chiamata, svolge le azioni in esso specificate allo scopo di risolvere il problema per il quale è stato realizzato e manifesta il suo unico effetto restituendo un valore.
Questo valore è restituito nel nome della funzione e può essere usato come elemento di una istruzione o come output.

A differenza delle procedure le funzioni restituiscono un risultato, oltre a svolgere una serie di azioni.

In 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 <risultato>
FINE

Attualmente una funzione non è più intesa come il sottoprogramma che restituisce un solo risultato bensì come un sottoprogramma che restituisce un risultato primario avvicinandosi così sempre più alle procedure.

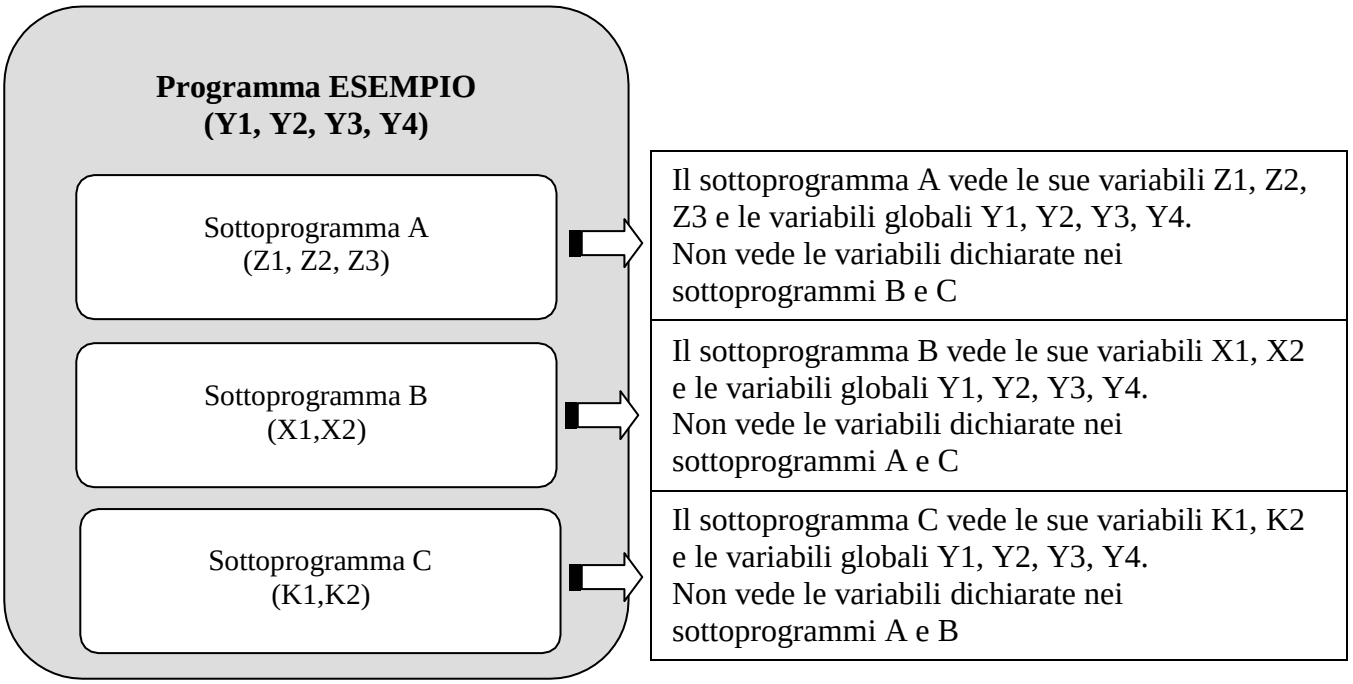
Una funzione è caratterizzata da:
- **un nome**, grazie al quale è possibile richiamarla ed identificarla univocamente;
- **una lista di parametri** che è *opzionale* e permette lo scambio in input e/o in output di informazioni tra il *programma chiamante* ed la funzione stessa ossia il *programma chiamato*;
- **un risultato** che indica il valore che deve essere restituito dalla funzione nel suo nome.

Ambiente e risorse locali e globali di un sottoprogramma

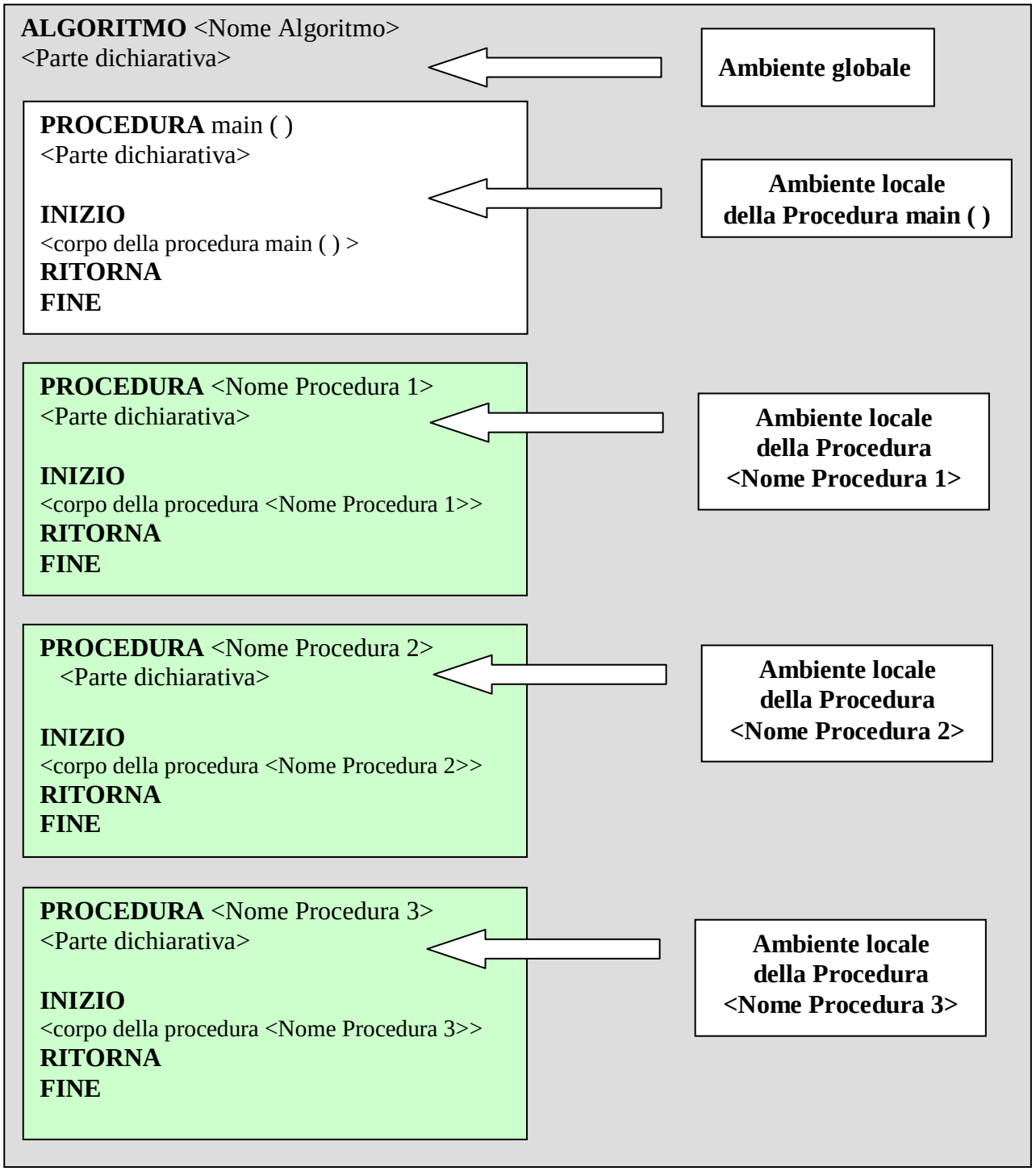
DEF: Con il termine "ambiente di un sottoprogramma" definiamo l'insieme delle risorse (variabili, costanti, sottoprogrammi, parametri) alle quali esso può accedere.

Tale ambiente è costituito da:
- un **ambiente locale** ossia costituito dalle risorse dichiarate al suo interno (**risorse locali**);
- un **ambiente globale** ossia costituito dalle risorse utilizzabili da tutti i sottoprogrammi (**risorse globali**).

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



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:



Le regole di visibilità o "*scope"*

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

Si parte dai seguenti principi:

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

Nella descrizione di un algoritmo può accedere 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. Nel caso che essa venga usata in una o più istruzioni all'interno del sottoprogramma essa (la variabile locale) oscurerà (**shadowing**) l'omonima variabile più esterna (in questo caso la globale), impedendone la visibilità.

I PARAMETRI ED IL LORO PASSAGGIO

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 principale.

Sono caratterizzati da:

- **un identificatore o nome;**
- **un tipo;**
- **un valore;**
- **un numero;**
- **una posizione;**
- **una direzione.**

Grazie ad essi si stabilisce attraverso quali oggetti debba avvenire l'input dei dati (al sottoprogramma) e l'output dei risultati (al programma chiamante).

L'identificatore ed il tipo dei parametri devono essere noti sin dal momento della dichiarazione dei parametri; il valore è noto solo all'atto della chiamata del sottoprogramma; la posizione occupata è fondamentale; la direzione ha a che fare con l'essere parametro solo di input, solo di output o di input ed output.

I **parametri** sono dunque degli oggetti che il programma chiamante trasmette al sottoprogramma chiamato e su cui esso deve operare. Essi permettono dunque di gestire **la comunicazione del sottoprogramma con l'esterno**.

Al momento della dichiarazione del sottoprogramma si specifica la tipologia dei parametri da utilizzare (**nome, tipo ed ordine**): essi prendono il nome di **parametri formali.**
Al momento della chiamata del sottoprogramma occorrerà specificare la tipologia dei parametri da trasmettere(**nome, tipo ed ordine**): essi prendono il nome di **parametri attuali.**

N.B. Ci deve essere un costante accordo di numero, tipo e posizione tra parametri formali e parametri attuali.

I parametri formali ed i parametri attuali possono anche avere lo stesso nome casualmente: in tal caso l'eventuale ambiguità viene risolta dallo **shadowing.**
Comunque è buona norma di programmazione utilizzare nomi differenti tra parametri formali e parametri attuali 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 viene associato (trasmesso) a quello dei parametri formali.

Tale passaggio può avvenire secondo due modalità distinte:

Con questa tipologia di passaggio si ha solo una **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 dei parametri attuali.

Con questa tipologia di passaggio i parametri formali vanno a coincidere con i parametri attuali.
Di conseguenza una qualsiasi modifica durante l'esecuzione del sottoprogramma chiamato sui parametri formali **provoca una modifica** dei corrispondenti parametri attuali..
N.B. Non viene allocata nessuna area di memoria per i parametri formali perché quest'ultimi utilizzano la stessa area di memoria usata per i parametri attuali.

N.B.
NON TUTTI I LINGUAGGI DI PROGRAMMAZIONE PREVEDONO AMBEDUE I TIPI DI PASSAGGIO DEI PARAMETRI

(ad esempio il PASCAL si, mentre il LINGUAGGIO C no: infatti per quest'ultimo linguaggio di programmazione del quale ci occuperemo, vengono utilizzate particolari tipi di variabili dette PUNTATORI per simulare, con questo artificio, il meccanismo di passaggio di parametri per riferimento

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.

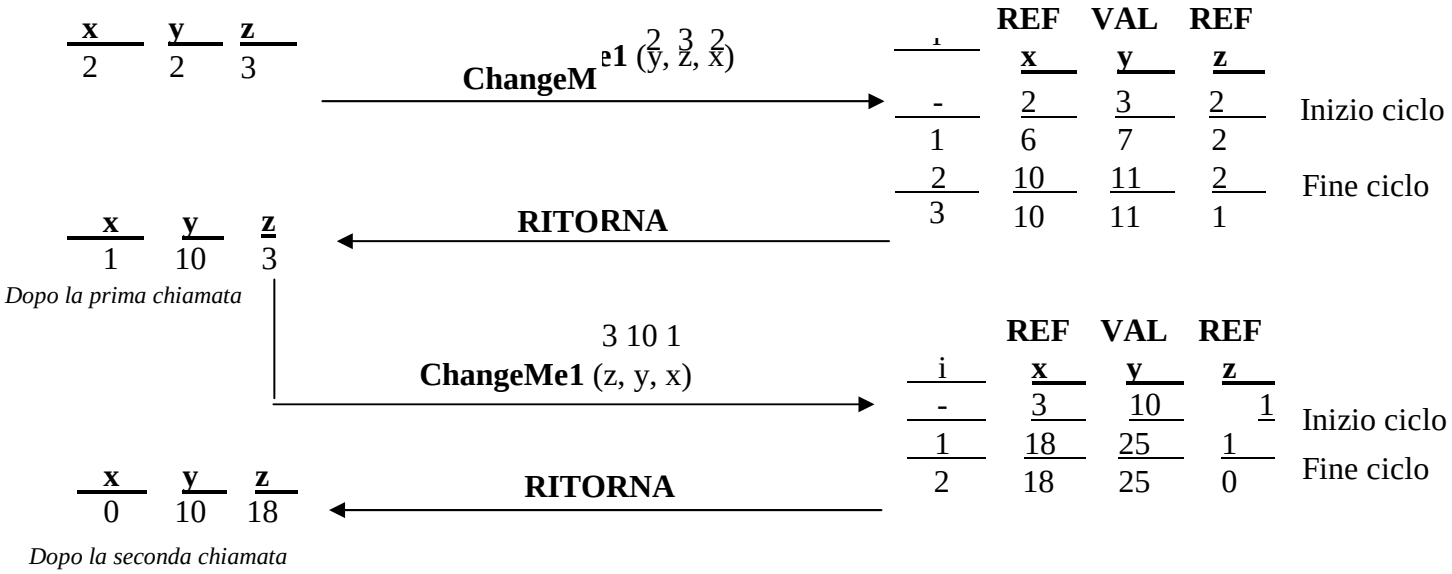
```
ALGORITMO Passaggio1
PROCEDURA main ( )
x, y, z : INT
INIZIO
Leggi (x)
Leggi (y)
Leggi (z)
/* Prima chiamata */
ChangeMe1 (y, z, x)
Scrivi (x)
Scrivi (y)
Scrivi (z)
/* Seconda chiamata */
ChangeMe1 (z, y, x)
Scrivi (x)
Scrivi (y)
Scrivi (z)
RITORNA
FINE

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
```

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 chiamante) 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:

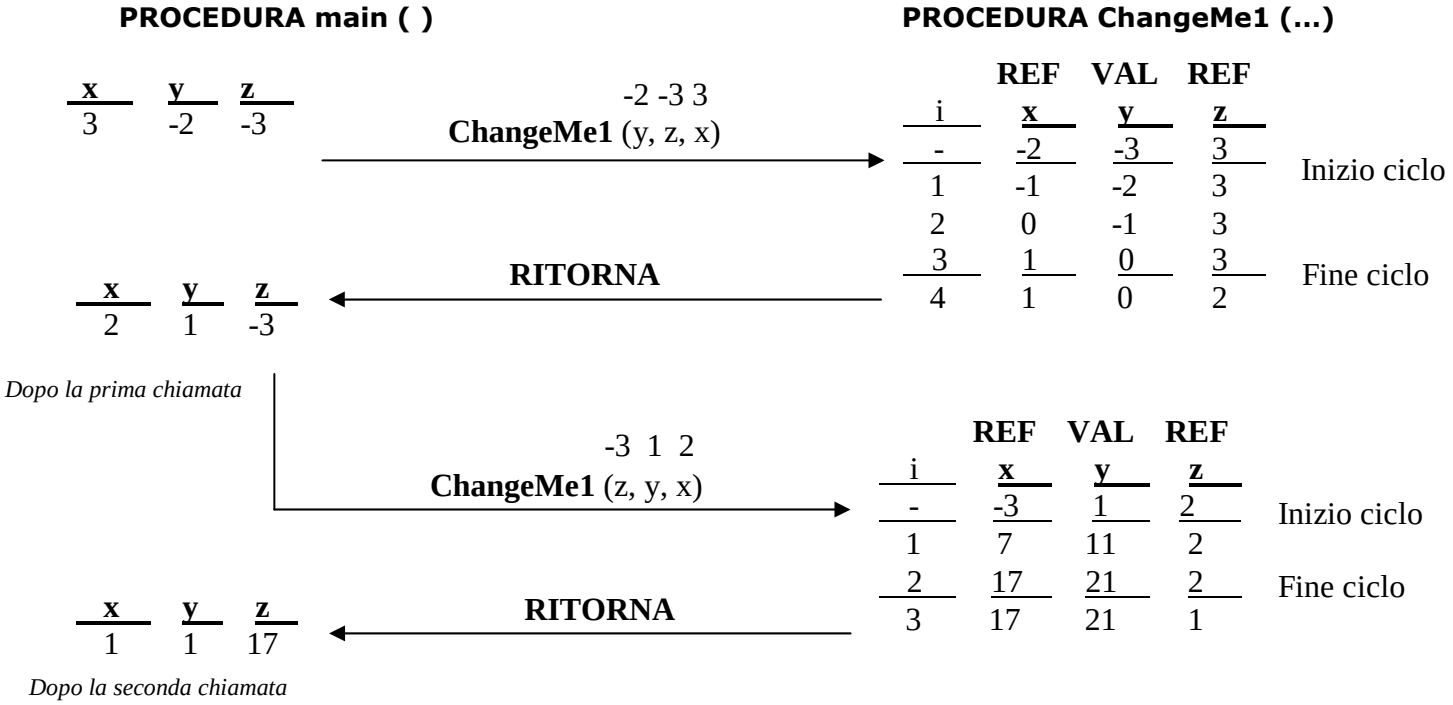
PROCEDURA ChangeMe1 (**REF** x: **INT**, **VAL** y: **INT, REF** z: **INT**)



Calcoli: 1° Chiamata (nella procedura **ChangeMe1**)
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)
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)…exit ciclo per
Fuori ciclo z ← z – 1 (z = 2 – 1 = 1)

Calcoli: 2° Chiamata (nella procedura **ChangeMe1**)
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) exit ciclo per
Fuori ciclo z ← z – 1 (z = 1 – 1 = 0)

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   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)
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)
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)…exit ciclo per
Fuori ciclo  z ← z – 1     (z = 3 – 1 = 2)
```

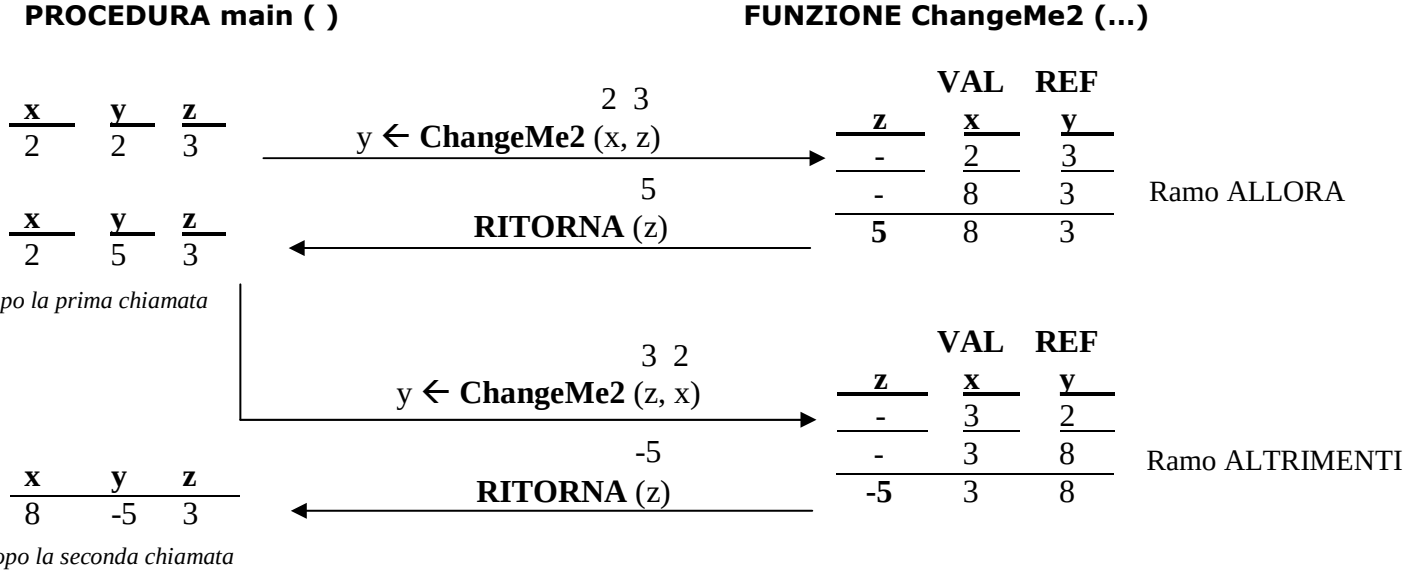
Calcoli: 2° Chiamata (nella procedura **ChangeMe1**)

```
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)
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)  exit ciclo per
Fuori ciclo  z ← z – 1  (z = 2 – 1 = 1)
```

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 */
y ← ChangeMe2 (z, x)
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
   ALTRIMENTI
     y ← y + 2*x
FINE SE
z ← x - y
RITORNA (z)
FINE
```

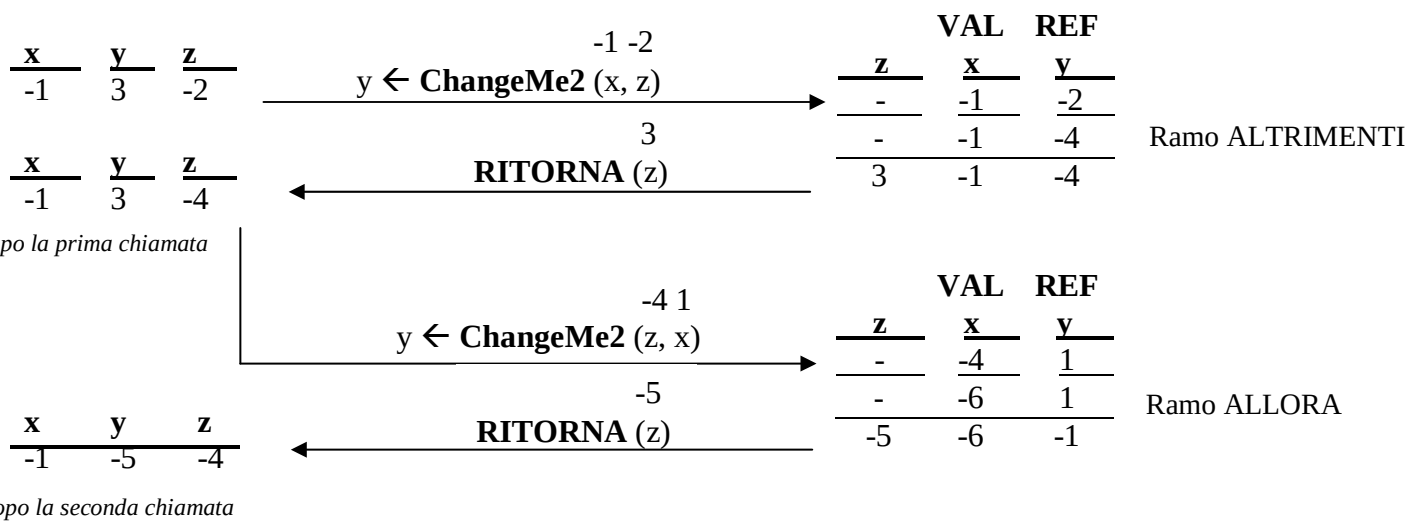
2.a) Si supponga che inizialmente le variabili siano così valorizzate: **x = 2, y = 2, z = 3**



Calcoli: 1° Chiamata (nella funzione **ChangeMe2**)
Test SE (x < y) ossia 2 < 3 VERO --→ ALLORA
x ← x + 2*y (x = 2 + 2*3 = 8)
Fuori SE z ← x – y (z = 8 – 3 = 5)

Calcoli: 2° Chiamata (nella funzione **ChangeMe2**)
Test SE (x < y) ….3 < 2 FALSO --→ ALTRIMENTI
y ← y + 2*x (y = 2 + 2*3 = 8)
Fuori SE z ← x – y (z = 3 – 8 = -5)

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



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

Calcoli: 2° Chiamata (nella funzione **ChangeMe2**)
Test (x < y) ossia -4 < -1 VERO --→ ALLORA
x ← x + 2*y (x = -4 + 2*(-1) = -6)
Fuori SE z ← x – y (z = -6 – (-1) = -5)

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?**

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**)*

3 [2, 0, 3]
y ← **ChangeArray1** (n, 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 v[i] ← (3*i + 4) % 5 (v[1] = (3*1 + 4) % 5 = 7 % 5 = 2)
i ← i + 1 (i = 1+ 1 = 2)

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

i = 3 v[i] ← (3*i + 4) % 5 (v[1] = (3*3 + 4) % 5 = 13 % 5 = 3)
i ← i + 1 (i = 1+ 1 = 4) exit ciclo per di caricamento vettore

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

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)

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) exit ciclo all'interno della funzione

Fuori ciclo v[n] = (y+1) DIV 2 (v[3] = (14 + 1) DIV 2 = 15 DIV 2 = 7

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)

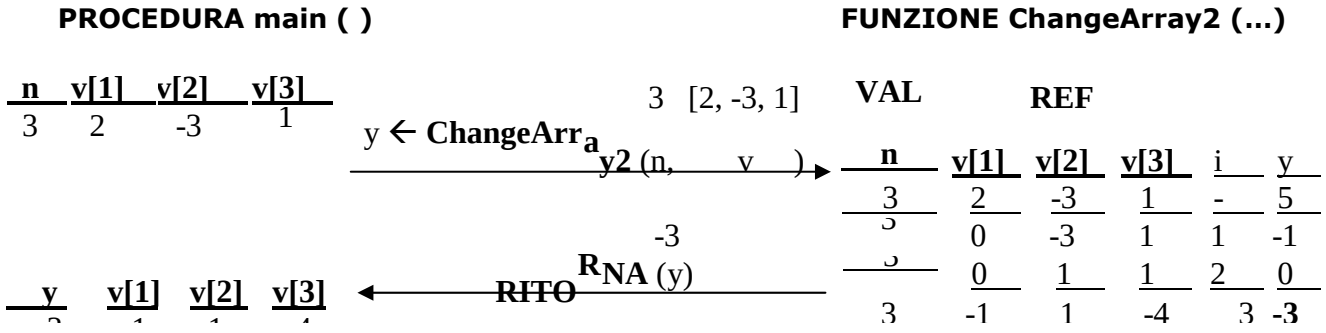
Vedi DOMANDA 2)

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?**



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

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

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)
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) 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

LA RICORSIONE

Abbiamo visto che nel corpo di un sottoprogramma è possibile richiamare (attivare) altri sottoprogrammi dichiarati esternamente o localmente ad esso, ma è anche possibile, in determinate condizioni, fornire l'istruzione per richiamare (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: Parleremo di **ricorsività** quando è possibile definire qualcosa riferendoci (ossia ricorrendo) alla sua stessa definizione.
DEF: Un sottoprogramma implementa la **ricorsione DIRETTA** quando nella sua definizione compare **UNA SOLA CHIAMATA** al sottoprogramma stesso.

Per potere attivare un **processo di tipo ricorsivo**:

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

Quindi per poter attivare un processo ricorsivo occorre avere tre elementi caratteristici

a. una **condizione** che consenta di capire se si è di fronte ad un caso particolare risolvibile banalmente o se è necessario procedere per più passate;
b. la soluzione del **caso particolare**;
c. la soluzione del **caso generale** che contiene una o più chiamate ricorsive.

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

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

- 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 al sottoproblema

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 non negativo

- 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 al sottoproblema

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

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

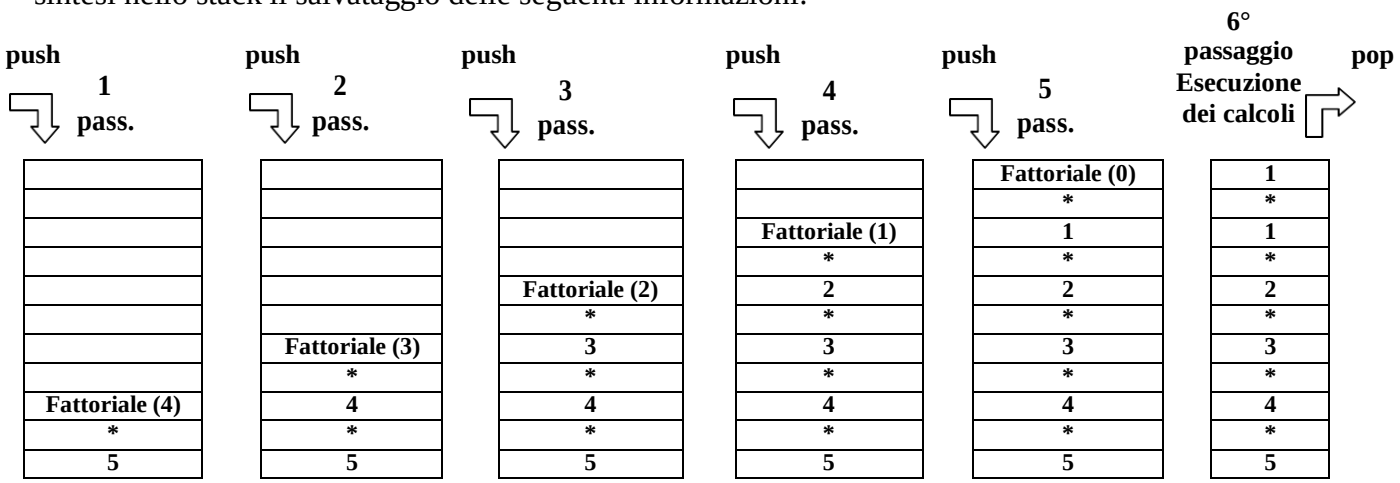
….
f ← Fattoriale (5)
….

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)*

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
```

ALTRI TIPI DI RICORSIONE

DEF: Un sottoprogramma implementa la **ricorsione MULTIPLA** quando nella sua definizione compare la chiamata al sottoprogramma stesso **ALMENO DUE VOLTE**

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:

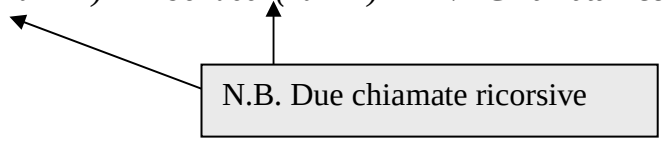
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…

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
```



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)*

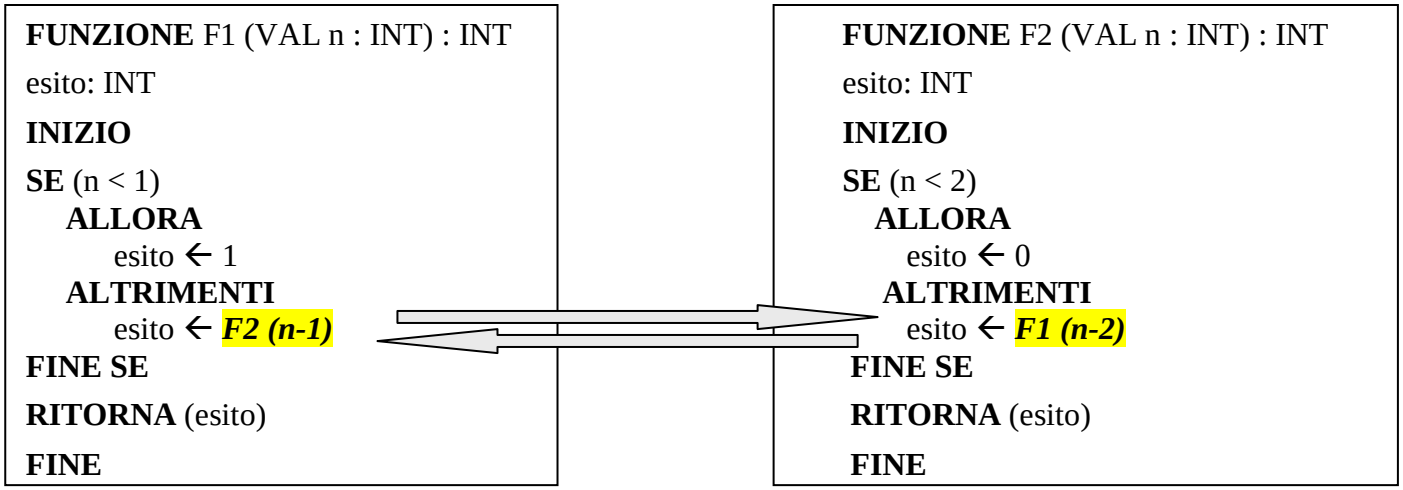
13° CALL *fib* ← **1**+ **1** + **1** + **1** + **1** + **1**+ *(Fibonacci (0)) + Fibonacci (1)*

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

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

DEF: 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



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

c) **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

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

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
```

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))

……