UNIVERSITÀ DEGLI STUDI DI TRIESTE

DEAMS

APPUNTI PER IL CORSO DI ELEMENTI DI INFORMATICA – modulo 2

ANNO ACCADEMICO 2014 – 2015

Autori: Liviana Picech e Renato Pelessoni

L'INFORMATICA

Campi di applicazione:
- scientifico–ingegneristico
- gestione aziendale
- intelligenza artificiale
- elaborazione di dati geografici ed ambientali
- …

Non soltanto scienza e tecnologia degli elaboratori elettronici.

Non coincide con nessuna delle molteplici applicazioni dei calcolatori.

INFORMATICA: *"scienza della rappresentazione e dell'elaborazione dell'informazione"*.

⇒ Il prodotto principale: l'informazione
⇒ I calcolatori come strumento
⇒ Il modo in cui l'informazione viene strutturata ed elaborata

Def. di *INFORMATICA* della *Association for Computing Machinery (ACM)*: *"L'informatica è lo studio sistematico degli algoritmi che descrivono e trasformano l'informazione: la loro teoria, analisi, progetto, efficienza, realizzazione e applicazione"*.

⇒ I calcolatori sono *esecutori di algoritmi*.

UNA PRIMA DEFINIZIONE DI ALGORITMO

<u>*ALGORITMO*</u>: *"una sequenza precisa di operazioni comprensibili e perciò eseguibili da uno strumento automatico"*.

⇒ Sequenza di passi, definiti con precisione, che portano alla risoluzione di un problema.

L'algoritmo deve essere:
- *comprensibile al suo esecutore*
- *corretto*
- *efficiente*

<u>Esempio di algoritmo</u>: somma di due numeri con un pallottoliere.

Il numero *n* viene rappresentato allineando *n* palline sulla sinistra.

Primo addendo nella prima riga, secondo addendo nella seconda riga, risultato nella terza.

1. Si sposta una pallina dalla sinistra alla destra nella prima riga e contestualmente se ne sposta una dalla destra alla sinistra della terza riga.
2. Si ripete l'operazione precedente finché non si è svuotata la parte sinistra della prima riga.
3. Si sposta una pallina dalla sinistra alla destra nella seconda riga e contestualmente se ne sposta una dalla destra alla sinistra nella terza riga.
4. Si ripete l'operazione precedente finché non si è svuotata la parte sinistra della seconda riga.

Il numero di palline che si viene a trovare alla sinistra nella terza riga, al termine delle operazioni, è il risultato cercato.

ARCHITETTURA DI UN CALCOLATORE

⇒ Chiamiamo *sistema informatico* un qualsiasi esecutore di algoritmi (PC, PC portatile, workstation, mainframe, ...)

Hardware: componenti fisici del sistema informatico.

- Unità di elaborazione o processore o CPU (Central Processing Unit)
- Memoria centrale o RAM (Random Access Memory)
- Memoria secondaria o memoria di massa
- Bus di sistema
- Unità periferiche

Software: programmi che vengono eseguiti dal sistema.

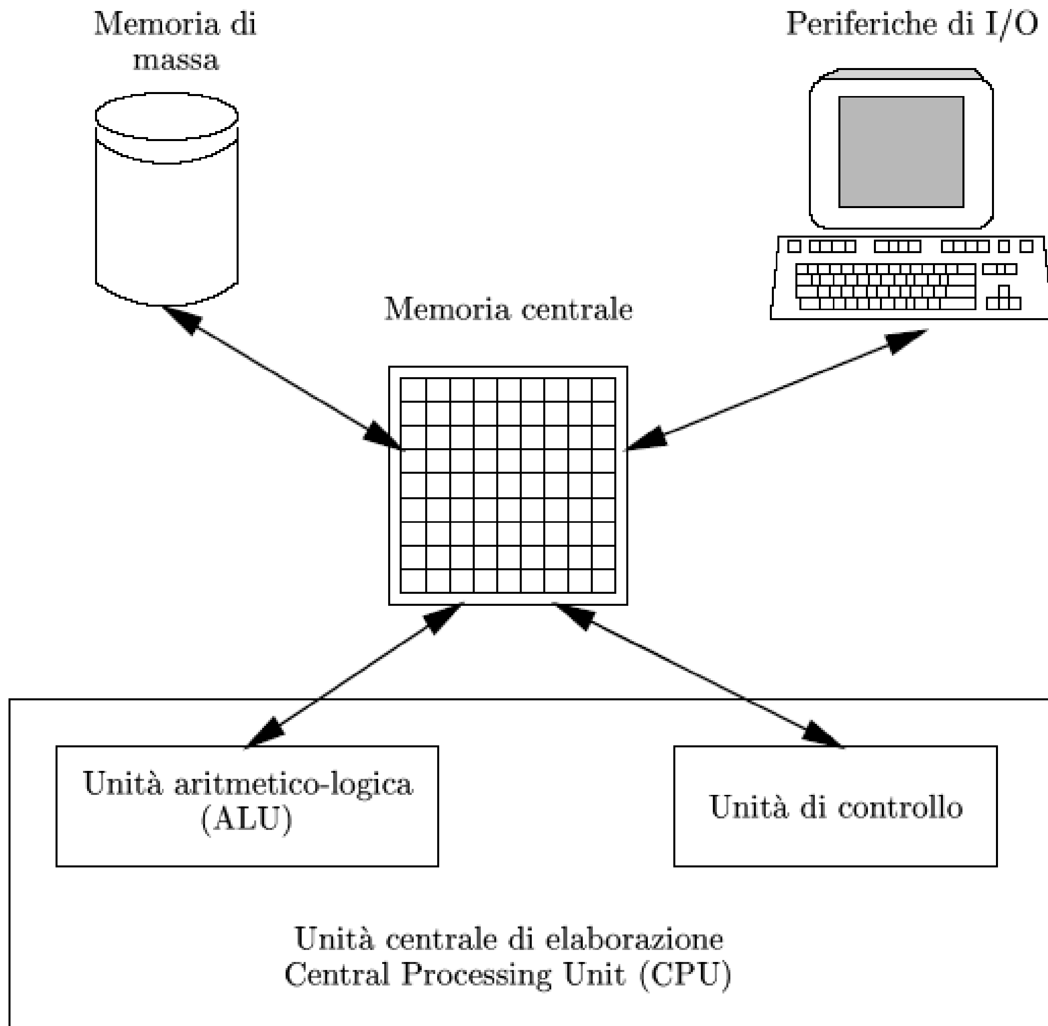
LA MACCHINA DI VON NEUMANN (1945)

È costituita da quattro elementi funzionali fondamentali:

- ➢ la *CPU*
- ➢ la *memoria centrale*
- ➢ le *periferiche*
- ➢ il *bus di sistema*

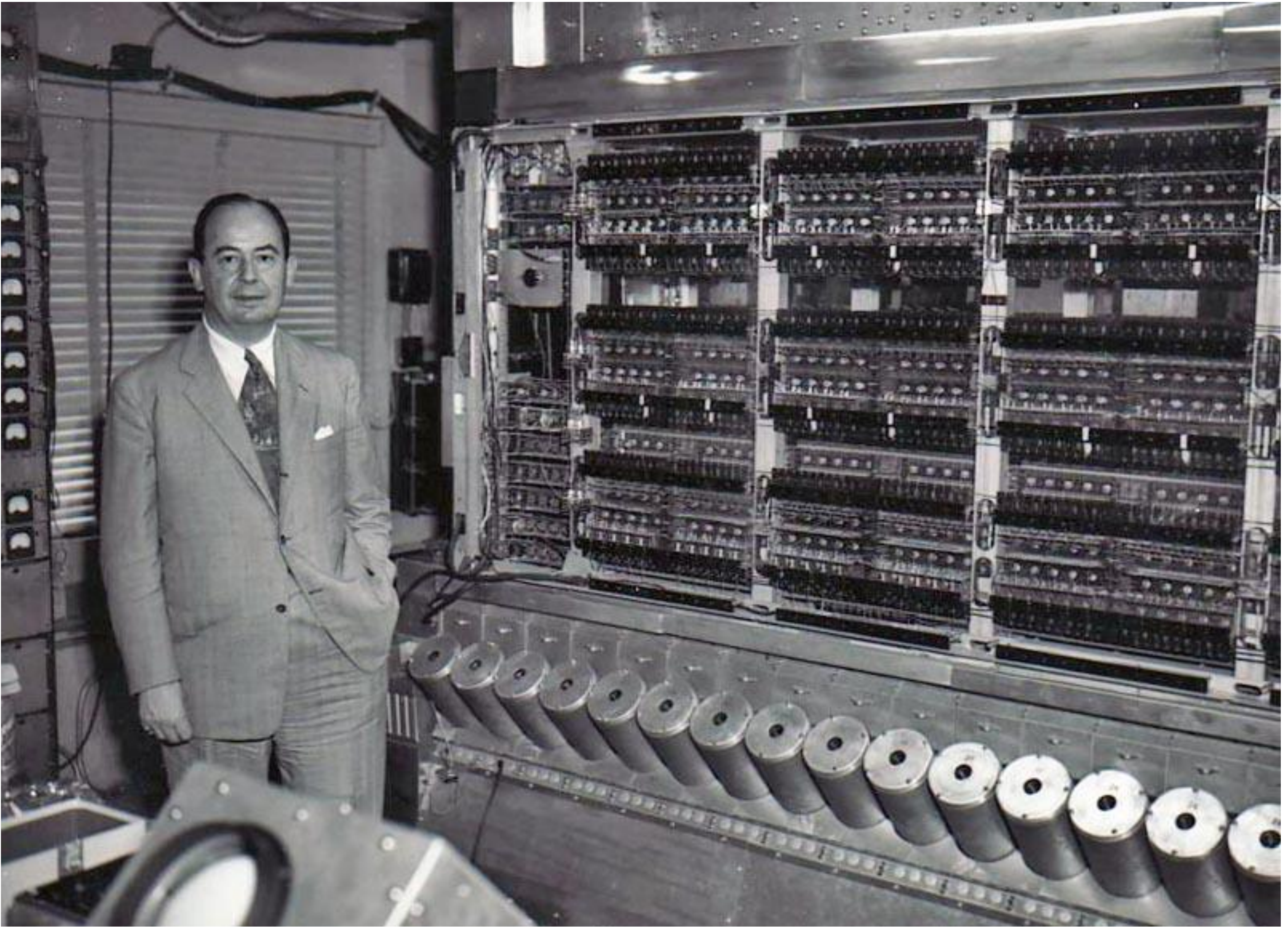
La CPU coordina le varie attività:
- ➢ **estrae** istruzioni dalla memoria centrale
- ➢ **decodifica** le istruzioni
- ➢ **esegue** le istruzioni

Istruzioni:
- ➢ operazioni di **elaborazione** dell'informazione
- ➢ operazioni di **trasferimento** dell'informazione



Rappresentazione schematica della macchina di von Neumann

- *I dati e le istruzioni che operano su di essi sono codificati in forma* ***binaria****.*
  - *bit* (binary digit): 1, 0;
  - *byte* (8 bit): 2^8 sequenze diverse di 1 e di 0.
- *Le operazioni vengono eseguite in stretta sequenza.*
- *Durante l'esecuzione del programma, dati ed istruzioni risiedono in memoria centrale.*



John von Neumann dinanzi all'EDVAC (Electronic Discrete Variables Automatic Computer)

I sistemi basati sull'architettura di Von Neumann sono particolarmente versatili poiché sono *programmabili* in senso stretto.

Il programma non è parte della componente hardware del sistema ma è integralmente software (*programma memorizzato*).

La macchina di von Neumann si può considerare composta da alcuni sottosistemi:

SOTTOSISTEMA DI MEMORIZZAZIONE

È costituito dalla *memoria centrale* e può contenere sia istruzioni sia dati.

SOTTOSISTEMA DI ELABORAZIONE

È costituito dalla *C.P.U.* ed opera in sequenza leggendo dalla memoria un'istruzione e gli eventuali dati su cui l'istruzione deve operare, eseguendo tale istruzione e scrivendo eventualmente in memoria il risultato di tale operazione. Continua così fino a quando tutte le istruzioni del programma siano state eseguite

SOTTOSISTEMA DI INTERFACCIA

È l'interfaccia con il mondo esterno e fornisce all'elaboratore i dati da trattare e le istruzioni da eseguire e presenta all'esterno i risultati.

RAPPRESENTAZIONE DEI DATI E DELLE ISTRUZIONI

Esempi di rappresentazione binaria:

- con 1 byte sono rappresentabili i naturali da 0 a 255 ($255=2^8-1$);
- con 1 byte si possono rappresentare i numeri interi compresi tra –127 e 127: 1 bit indica il segno (0 = + e 1 = -); con 7 bit si possono rappresentare i numeri naturali da 0 a 127 ($127 = 2^7-1$);
- con 2 byte si può rappresentare un numero reale mediante la notazione *in virgola fissa* (1 byte per la parte intera ed 1 byte per la parte decimale);
- il codice *ASCII* (American Standard Code for Information Interchange) utilizza 7 bit per codificare 128 caratteri; alcune varianti del codice ASCII utilizzano anche l'ottavo bit per codificare altri 128 caratteri.

Le *istruzioni* si compongono di:
- codice operativo
- uno o più operandi

La *memoria centrale* è formata da una sequenza di celle di memoria; ciascuna cella contiene una *parola* (*word*).
Esempi: parole a 8, 16, 32, 64 bit.

Byte	Cod.	Char	Byte	Cod.	Char	Byte	Cod.	Char	Byte	Cod.	Char
00000000	0	Null	00100000	32	Spc	01000000	64	@	01100000	96	`
00000001	1	Start of heading	00100001	33	!	01000001	65	A	01100001	97	a
00000010	2	Start of text	00100010	34	"	01000010	66	B	01100010	98	b
00000011	3	End of text	00100011	35	#	01000011	67	C	01100011	99	c
00000100	4	End of transmit	00100100	36	$	01000100	68	D	01100100	100	d
00000101	5	Enquiry	00100101	37	%	01000101	69	E	01100101	101	e
00000110	6	Acknowledge	00100110	38	&	01000110	70	F	01100110	102	f
00000111	7	Audible bell	00100111	39	'	01000111	71	G	01100111	103	g
00001000	8	Backspace	00101000	40	(	01001000	72	H	01101000	104	h
00001001	9	Horizontal tab	00101001	41	)	01001001	73	I	01101001	105	i
00001010	10	Line feed	00101010	42	*	01001010	74	J	01101010	106	j
00001011	11	Vertical tab	00101011	43	+	01001011	75	K	01101011	107	k
00001100	12	Form Feed	00101100	44	,	01001100	76	L	01101100	108	l
00001101	13	Carriage return	00101101	45	-	01001101	77	M	01101101	109	m
00001110	14	Shift out	00101110	46	.	01001110	78	N	01101110	110	n
00001111	15	Shift in	00101111	47	/	01001111	79	O	01101111	111	o
00010000	16	Data link escape	00110000	48	0	01010000	80	P	01110000	112	p
00010001	17	Device control 1	00110001	49	1	01010001	81	Q	01110001	113	q
00010010	18	Device control 2	00110010	50	2	01010010	82	R	01110010	114	r
00010011	19	Device control 3	00110011	51	3	01010011	83	S	01110011	115	s
00010100	20	Device control 4	00110100	52	4	01010100	84	T	01110100	116	t
00010101	21	Neg. acknowledge	00110101	53	5	01010101	85	U	01110101	117	u
00010110	22	Synchronous idle	00110110	54	6	01010110	86	V	01110110	118	v
00010111	23	End trans. block	00110111	55	7	01010111	87	W	01110111	119	w
00011000	24	Cancel	00111000	56	8	01011000	88	X	01111000	120	x
00011001	25	End of medium	00111001	57	9	01011001	89	Y	01111001	121	y
00011010	26	Substitution	00111010	58	:	01011010	90	Z	01111010	122	z
00011011	27	Escape	00111011	59	;	01011011	91	[	01111011	123	{
00011100	28	File separator	00111100	60	<	01011100	92	\	01111100	124	|
00011101	29	Group separator	00111101	61	=	01011101	93	]	01111101	125	}
00011110	30	Record Separator	00111110	62	>	01011110	94	^	01111110	126	~
00011111	31	Unit separator	00111111	63	?	01011111	95	_	01111111	127	Del

Tabella ASCII (128 elementi)

MACCHINA DI VON NEUMANN: ESEMPIO DI FUNZIONAMENTO

Vogliamo calcolare il valore dell'espressione $(a+b)\cdot(c+d)$, leggendo i valori delle variabili da un dispositivo di ingresso e scrivendo i risultati sul dispositivo di uscita.

⇒ Algoritmo generale per risolvere questo problema:

1. leggi i valori *a*, *b*, *c* e *d* dal dispositivo di input;
2. somma i valori di *a* e *b*;
3. salva il risultato parziale in memoria;
4. somma i valori di *c* e *d*;
5. moltiplica il risultato parziale appena ottenuto con quello precedentemente salvato;
6. scrivi il risultato finale del calcolo sul dispositivo di output;
7. arresta l'esecuzione del programma.

⇒ Traduzione dell'algoritmo in un "programma eseguibile" dalla macchina di von Neumann:

Si riservano alcune celle della memoria centrale per contenere i dati ed i risultati:

variabili: *a* *b* *c* *d* *x* *y* *z*

Programma per il calcolo di $(a+b)\cdot(c+d)$:

1. INPUT dei dati: preleva i dati dal dispositivo di input ed inseriscili in memoria centrale nelle celle con i nomi *a*, *b*, *c* e *d*.
   1.1. leggi *a*;
   1.2. leggi *b*;
   1.3. leggi *c*;
   1.4. leggi *d*;
2. Calcola $x=a+b$.
   2.1. preleva il dato contenuto nella cella *a*;
   2.2. preleva il dato contenuto nella cella *b*;
   2.3. calcola $a+b$;

3. salva il risultato parziale in memoria.
    3.1. inserisci il risultato ottenuto nella cella x;
4. Calcola $y = c + d$.
    4.1. preleva il dato contenuto nella cella c;
    4.2. preleva il dato contenuto nella cella d;
    4.3. calcola $c + d$;
    4.4. inserisci il risultato ottenuto nella cella y;
5. Calcola $z = x \cdot y$.
    5.1. preleva il dato contenuto nella cella x;
    5.2. preleva il dato contenuto nella cella y;
    5.3. calcola $x \cdot y$;
    5.4. inserisci il risultato ottenuto nella cella z;
6. OUTPUT dei dati: preleva il dato contenuto nella cella z e scrivilo (visualizzalo) sul dispositivo di output;
7. arresta l'esecuzione del programma.

Osservazioni:

a) durante l'esecuzione le istruzioni del programma sono memorizzate in memoria centrale, da dove vengono prelevate, decodificate dall'unità di controllo ed eseguite ad una ad una;

b) nella fase di esecuzione avvengono operazioni di elaborazione e trasferimento dei dati;

c) le elaborazioni (passi 2.3, 4.3 e 5.3 del programma) vengono eseguite dalla ALU;

d) i passi 1.1, 1.2, 1.3 ed 1.4 sono trasferimento di dati dall'interfaccia di una periferica alla memoria centrale;

e) i passi 2.1, 2.2, 4.1, 4.2, 5.1 e 5.2 sono trasferimenti di dati dalla memoria centrale all'unità di elaborazione;

f) i passi 3.1, 4.4 e 5.4 sono trasferimenti di dati dall'unità di elaborazione alla memoria centrale;

g) il passo 6 è un trasferimento di dati dalla memoria centrale all'interfaccia di una periferica (es. stampante, video, …).

RAPPRESENTAZIONE IN BASE

Base 10: $3124 = 3\times10^3+1\times10^2+2\times10^1+4\times10^0$

Base 2: $1011_2 = 1\times2^3+0\times2^2+1\times2^1+1\times2^0$

Base b ($b\in N^+$, $b\geq2$):
$x_n\, x_{n-1} \ldots x_1\, x_0 = x'_n \times b^n + x'_{n-1} \times b^{n-1} + \ldots + x'_1 \times b^1 + x'_0 \times b^0$
x'_i intero associato ad x_i
Basi più comuni: 2, 10, 16
Base *b*: *b* simboli per la rappresentazione

Esempio:
$b = 10$ Simboli: 0, 1, 2, 3, …, 9
$b = 2$ Simboli: 0, 1
$b = 16$ Simboli: 0, 1, 2, 3, …, 9, A, B, C, D, E, F

Esempio: trasformare 624_{10} in base 2

624:2 = 312 con resto r = 0
312:2 = 156 con resto r = 0
156:2 = 78 con resto r = 0
78:2 = 39 con resto r = 0
39:2 = 19 con resto r = 1
19:2 = 9 con resto r = 1
9:2 = 4 con resto r = 1
4:2 = 2 con resto r = 0
2:2 = 1 con resto r = 0
1:2 = 0 con resto r = 1

$$624_{10} = 1001110000_2$$

Trasformazione tra le basi 2 e 16

TRASFORMAZIONE DA BASE 2 A BASE 16

Si raggruppano le cifre binarie in gruppi di 4 a partire da destra e si converte ogni gruppo in base 16

$11101101111110_2 = ?_{16}$

$0011|1011|0111|1110 = 3B7E_{16}$

3 B 7 E

TRASFORMAZIONE DA BASE 16 A BASE 2

Si trasforma ogni singola cifra esadecimale in binario

$52A_{16} = ?_2$

5 2 A = 10100101010_2

0101 0010 1010

CORRISPONDENZA DECIMALE – BINARIO - ESADECIMALE											
0	0	0	4	100	4	8	1000	8	12	1100	C
1	1	1	5	101	5	9	1001	9	13	1101	D
2	10	2	6	110	6	10	1010	A	14	1110	E
3	11	3	7	111	7	11	1011	B	15	1111	F

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Tabella ASCII (128 elementi)

DEFINIZIONE DI PROGRAMMA

PROGRAMMA: *"descrizione di un algoritmo mediante sequenze di istruzioni scritte in un opportuno linguaggio, comprensibile al calcolatore"*.

I LINGUAGGI DI PROGRAMMAZIONE

Prima generazione: *il linguaggio macchina*.

- Sequenze di bit eseguibili direttamente.
- Non portabile, difficile per l'uomo.

Seconda generazione: *il linguaggio assembler*.

- Introduce nomi simbolici per istruzioni e celle di memoria.
- Ogni istruzione corrisponde ad un'istruzione in linguaggio macchina.
- Più comprensibile per l'uomo.
- Poco portabile, necessita di compilazione tramite *assemblatore*.

Terza generazione: *i linguaggi orientati ai problemi*.

- Istruzioni complesse: un'istruzione corrisponde a più istruzioni in linguaggio macchina.
- Tipi di dati complessi e strutturati.
- Ricordano il linguaggio umano.
- Portabili, previa compilazione con un *compilatore*
- Esempi:
  - FORTRAN (FORmula TRANslator)
  - COBOL (Common Business Orientated Language)
  - BASIC (Beginners All-purpose Symbolic Instruction Code)
  - PASCAL

Quarta e Quinta generazione:

- Ulteriore grado di astrazione
- Impiego di norma specifico (SQL, linguaggi per l'intelligenza artificiale,…)

⇒ Ci sono diversi linguaggi, adatti a codificare specifici algoritmi. L'esecuzione è in ogni caso in linguaggio macchina.

I linguaggi si possono distinguere anche in:

- *Imperativi*: il programmatore definisce una sequenza di cambiamenti di stato della memoria fino a produrre lo stato finale voluto. Si possono classificare in:
    - *Procedurali*: il programma viene organizzato in procedure (funzioni)
    - *Orientati agli oggetti*: l'elemento di base è l'oggetto, che rappresenta dal punto di vista software un oggetto o un concetto della realtà.
    - *Paralleli*: è importante la sincronizzazione di processi
- *Dichiarativi*: il programmatore definisce delle affermazioni che consentono alla macchina di risolvere il problema
    - Utilizzati in particolare in intelligenza artificiale o per database

Esempi (ipotetici) di linguaggi:

ALTO LIVELLO

```
Guadagno = Ricavo - Costo
```

ASSEMBLER

```
LD R2, Ricavo
LD R5, Costo
SUB R0, R2, R5
ST R0, Guadagno
HLT
```

LINGUAGGIO MACCHINA

```
1 4 6 C
1 2 6 D
4 2 5 6
2 0 6 E
B 1 0 1
```

ALGORITMI

Un algoritmo è un metodo che in modo esplicito e non ambiguo realizza una trasformazione dei dati di un problema in risultati, e costituisce uno dei passi per la soluzione di un problema.

Un algoritmo deve soddisfare alle seguenti caratteristiche (Knuth, 1973):

- **Definitezza**: ogni passo deve essere privo di ambiguità
- **Effettività**: ogni passo deve essere effettivamente eseguibile
- **Generalità**: deve poter essere applicabile a qualunque insieme di dati di un certo dominio
- **Finitezza**: la sua esecuzione deve terminare in un numero finito di passi

Per essere eseguito da un calcolatore elettronico un algoritmo deve essere codificato nel *linguaggio macchina*.

Il linguaggio macchina si presenta come una *sequenza di codice binario* di difficile comprensione.

Per rendere più agevole la programmazione si utilizzano *linguaggi di programmazione di alto livello* che sono:

- *precisi*: non danno adito a dubbi interpretativi sul significato delle operazioni da eseguire;
- *comprensibili*: sono "abbastanza" vicini al linguaggio naturale.

Per descrivere gli algoritmi spesso si prescinderà dall'uso di un particolare linguaggio e si adotterà uno *pseudolinguaggio* "vicino" al linguaggio naturale.

Problema 1: determinare il maggiore di due numeri reali *X* ed *Y*

Si supponga che l'esecutore non conosca la relazione d'ordine nell'insieme dei numeri reali, ma sia in grado di calcolare la differenza tra due numeri e di valutare il segno di un numero.

Il problema può dunque essere formulato matematicamente:

$$\max(X,Y)=\begin{cases} X & \text{se } X>Y \\ Y & \text{se } X\leq Y \end{cases} = \begin{cases} X & \text{se } X-Y>0 \\ Y & \text{se } X-Y\leq 0 \end{cases}$$

L'algoritmo sarà dunque :

Passo	**Istruzione**
P1:	Leggi X
P2:	Leggi Y
P3:	diff ⟵ X-Y
P4:	Se diff > 0 allora
	max ⟵ X
	altrimenti
	max ⟵ Y
P5:	Visualizza max

Se invece l’esecutore conosce la relazione d’ordine tra numeri reali, l’algoritmo chiaramente diventa:

Passo Istruzione

P1: Leggi X

P2: Leggi Y

P3: Se X>Y allora

max ⟵ X

altrimenti

max ⟵ Y

P4: Visualizza max

Un algoritmo consiste nella descrizione della soluzione di un problema espressa come *sequenza di regole* che, operando sui dati iniziali, consentono di ottenere dei risultati che costituiscono la soluzione del problema.

Tali regole vengono determinate tramite la scomposizione iterativa del problema di partenza in *sottoproblemi elementari* (che sono direttamente eseguibili dall'esecutore), la soluzione di ognuno dei quali (durante l'esecuzione) viene detta *passo* o *step* dell'algoritmo.

È fondamentale che l'algoritmo termini in un numero *finito* di passi.

Alcuni tipi di istruzioni utilizzate nel precedente algoritmo:

Istruzione di input: ad esempio *leggi X*, con la quale il primo dei due numeri viene fornito al sistema di elaborazione (l'esecutore) da una periferica (cioè dall'esterno del sistema) ed inserito in una cella di memoria centrale denominata X

Istruzione di assegnazione: ad esempio *diff⟵X-Y*, con la quale la differenza tra i contenuti della celle X ed Y viene inserita nella (assegnata alla) cella denominata diff, senza che X ed Y siano modificati.

Istruzione condizionale: ad esempio *se…altrimenti…*, che contiene il *test* diff>0, il quale dà come risultato i valori logici *vero (true)* o *falso (false)*.

Istruzione di output: ad esempio *visualizza max*, con la quale il contenuto della cella di memoria di nome max viene inviato ad una periferica per venir "visualizzato" all'esterno.

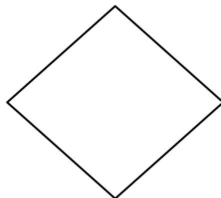
I nomi delle celle di memoria utilizzati, come ad esempio X, Y, diff, max, sono detti *variabili di memoria*.

RAPPRESENTAZIONE DI ALGORITMI MEDIANTE DIAGRAMMI DI FLUSSO (FLOW-CHART) O DIAGRAMMI A BLOCCHI

Inizio e fine esecuzione

Operazione di input/output

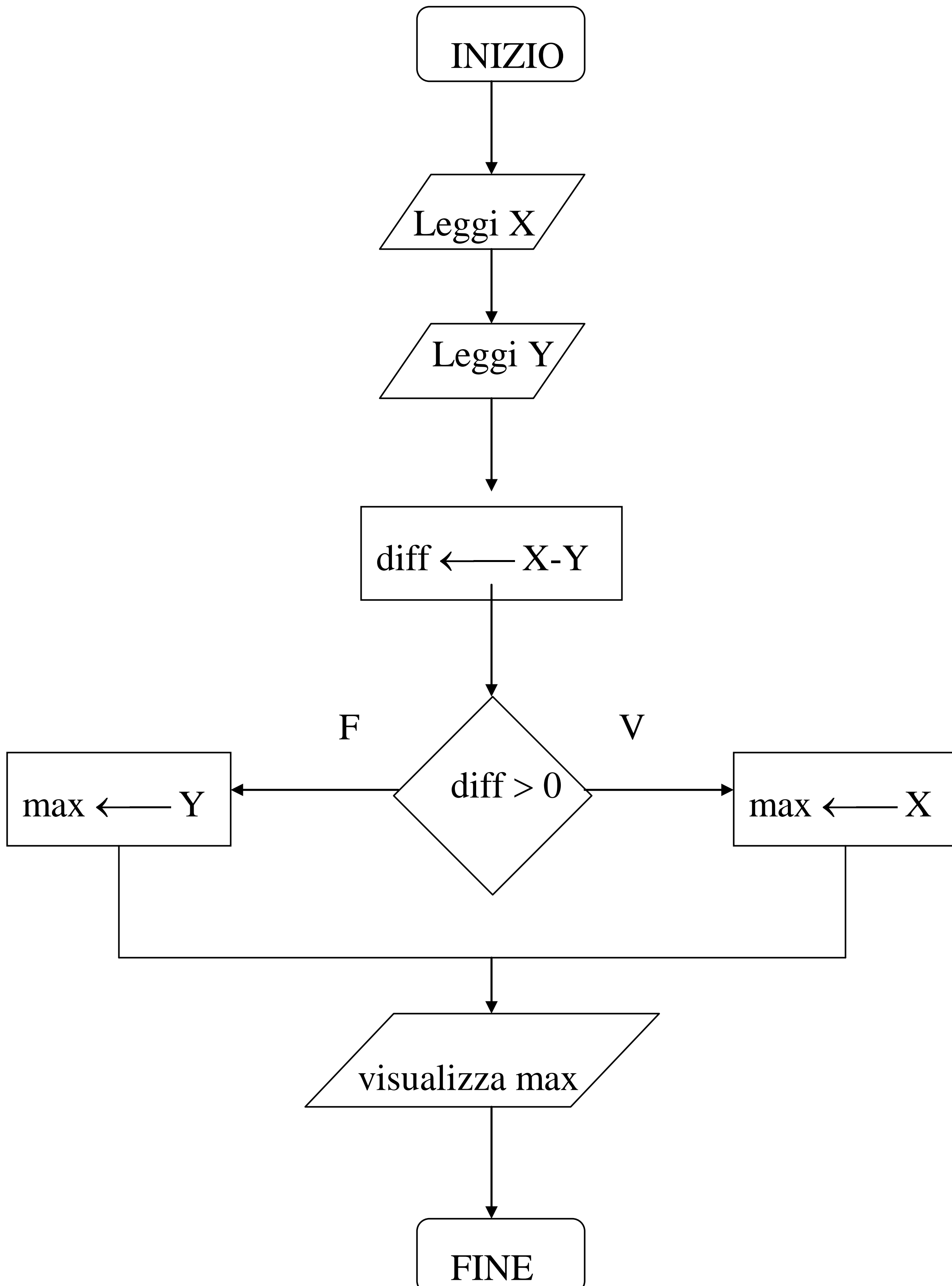
Operazioni da compiere in sequenza



Operazione di test

Le frecce indicano l'ordine di esecuzione delle istruzioni

FLOW CHART DEL PROBLEMA 1



Esercizio 1: calcolare il valore assoluto di un numero assegnato

Il problema può dunque essere formulato matematicamente:

$$|X| = \begin{cases} X & \text{se } X \geq 0 \\ -X & \text{se } X < 0 \end{cases}$$

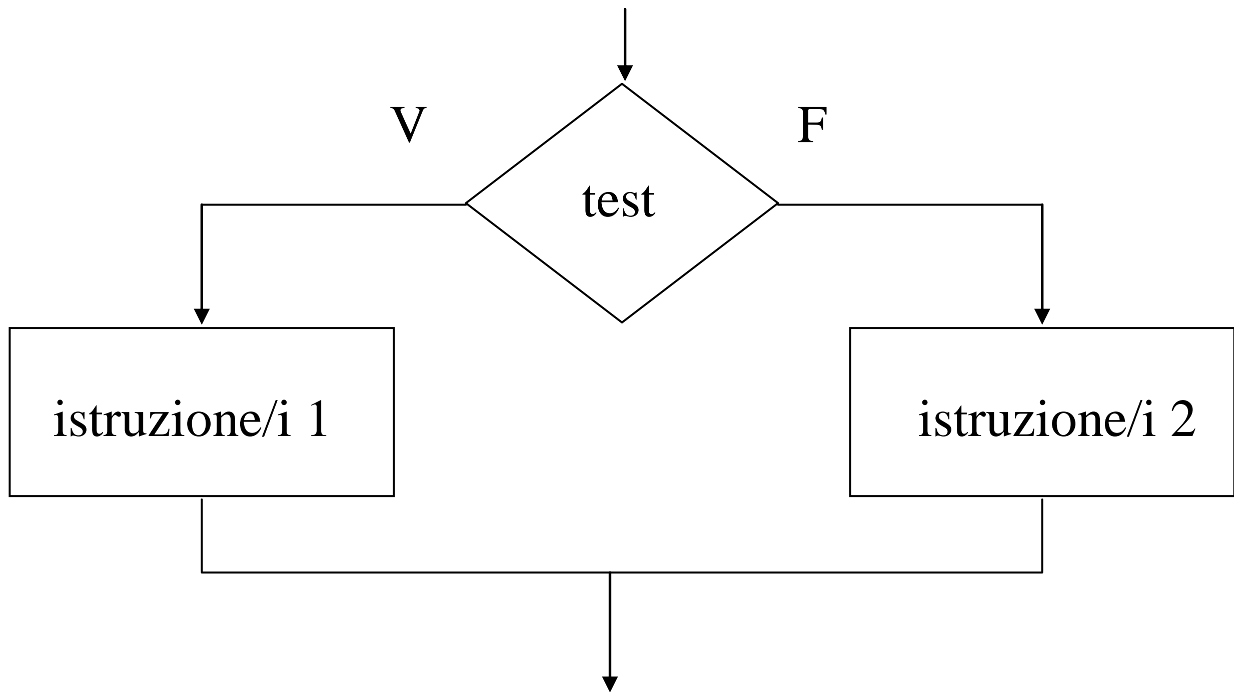
Supponiamo che l'esecutore sia in grado di verificare il segno di un numero e sia in grado di calcolare i prodotti.

Passo	**Istruzione**
P1:	Leggi X
P2:	Se X > 0 allora
	val_assoluto ⟵ X
	altrimenti
	val_assoluto ⟵ X*(–1)
P3:	Visualizza val_assoluto

ISTRUZIONI CONDIZIONALI

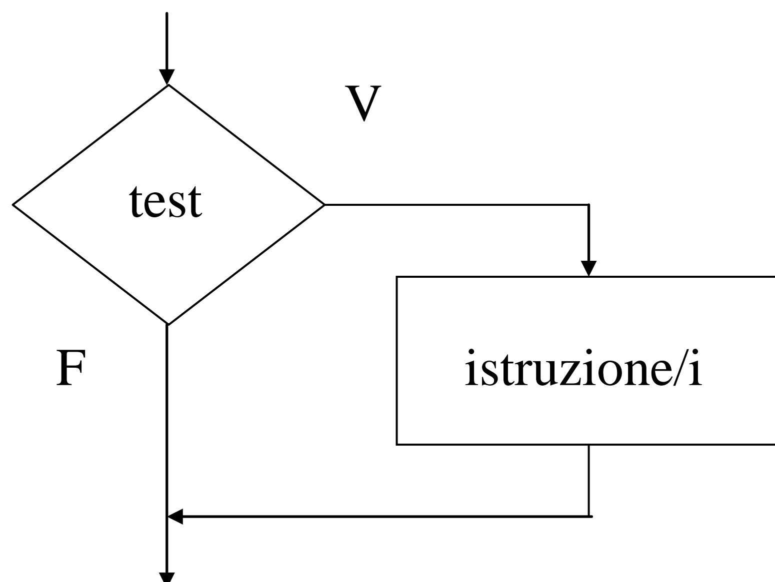
Se test è vero allora
  istruzione/i 1
altrimenti
  istruzione/i 2

If condizione *Then*
  istruzione/i
Else
  istruzione/i
End If



Se test è vero allora
  istruzione/i

If condizione *Then*
  istruzione/i
End If



Esercizio 2: riscrivere l'algoritmo per il calcolo del valore assoluto supponendo di disporre soltanto dell'istruzione condizionale If…Then…

Passo	Istruzione
P1:	Leggi X
P2:	val_assoluto ⟵ X
P3:	Se X < 0 allora
	val_assoluto ⟵ val_assoluto*(−1)
P4:	Visualizza val_assoluto

oppure, risparmiando una variabile:

Passo	Istruzione
P1:	Leggi X
P2:	Se X < 0 allora
	X⟵X∗(−1)
P3:	Visualizza X

SUDDIVISIONE IN SOTTOPROBLEMI

L'algoritmo è costituito da una sequenza di *sottoproblemi elementari* (le istruzioni) che devono essere *comprensibili* e *direttamente eseguibili* dall'esecutore.

Problemi molto complessi possono richiedere per la loro soluzione una sequenza di sottoproblemi elementari particolarmente elevata e tali da rendere il relativo algoritmo poco maneggevole, per cui si potrebbero avere difficoltà

- nel seguire il ragionamento
- nel trovare gli eventuali errori logici
- nell'apportare modifiche in un momento successivo alla stesura dell'algoritmo stesso.

In questi casi è opportuno semplificare l'algoritmo arrestando la scomposizione del problema in *sottoproblemi terminali* anche non elementari (cioè non direttamente eseguibili) ma dei quali si conosce la soluzione.

Algoritmi	Sottoproblemi terminali elementari	Istruzioni direttamente eseguibili
	Sottoproblemi terminali non elementari	Per essere eseguibili vanno scomposti in sottoproblemi terminali elementari

Problema 2: determinare il massimo tra tre numeri reali X, Y, Z

Per risolvere questo problema si può utilizzare l'algoritmo per il sottoproblema elementare di ricerca del massimo tra due numeri.

P1: Leggi X

P2: Leggi Y

P3: Leggi Z

P4: Ricerca il massimo tra X ed Y e metti il risultato in M

P5: Ricerca il massimo tra M e Z e metti il risultato in maximum

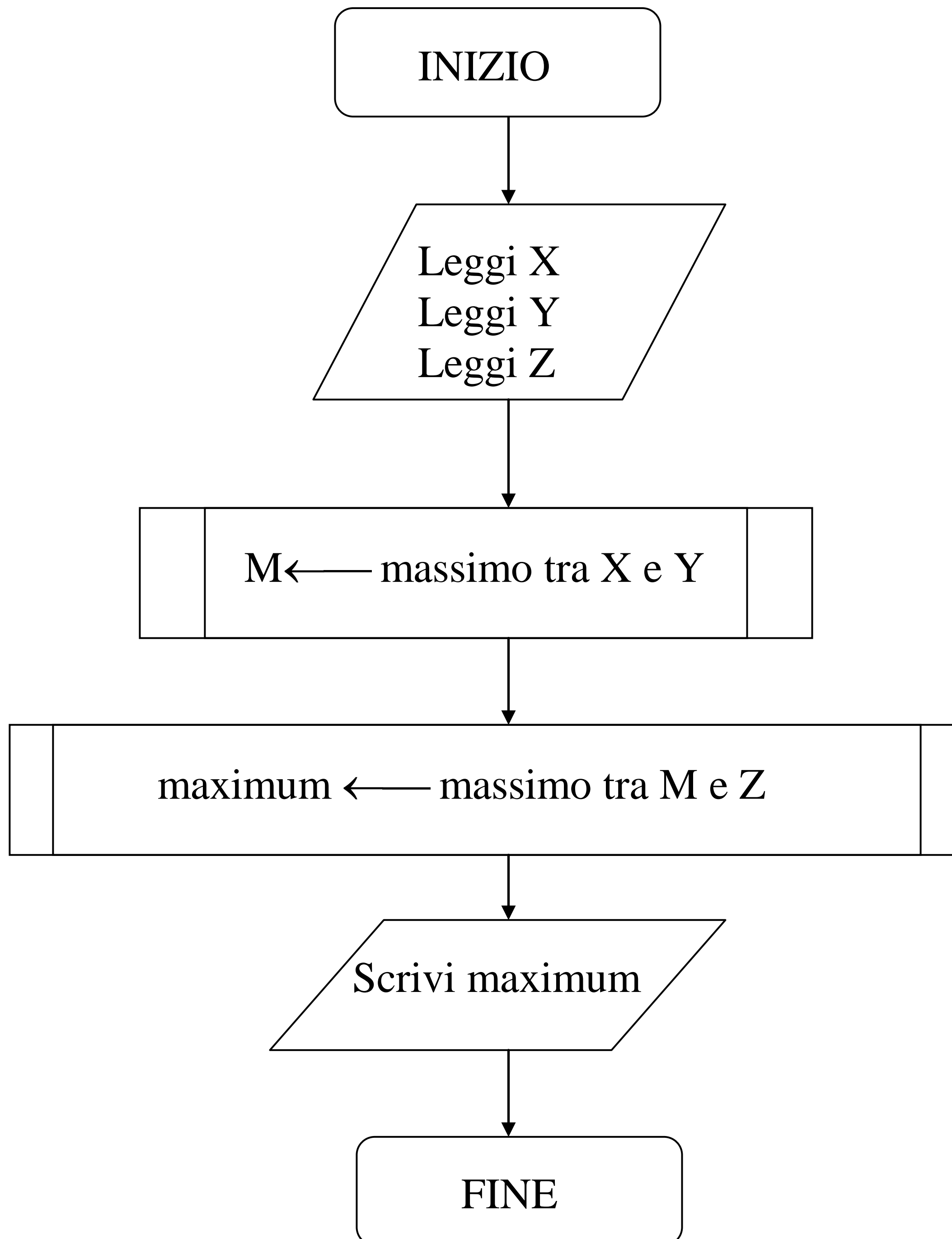
P6: Scrivi maximum

I passi P4 e P5 corrispondono alla risoluzione di sottoproblemi terminali non elementari. Pertanto, affinché questo algoritmo sia eseguibile dall'esecutore, al posto di P4 e P5 si devono considerare i passi degli algoritmi che consentono di risolvere tali sottoproblemi.

Quando un algoritmo viene tradotto in un programma per venir eseguito da un calcolatore, i sottoproblemi terminali elementari corrispondono alle istruzioni del programma, mentre ai sottoproblemi terminali non elementari corrispondono *sottoprogrammi*, a loro volta costituiti da istruzioni o sottoprogrammi.

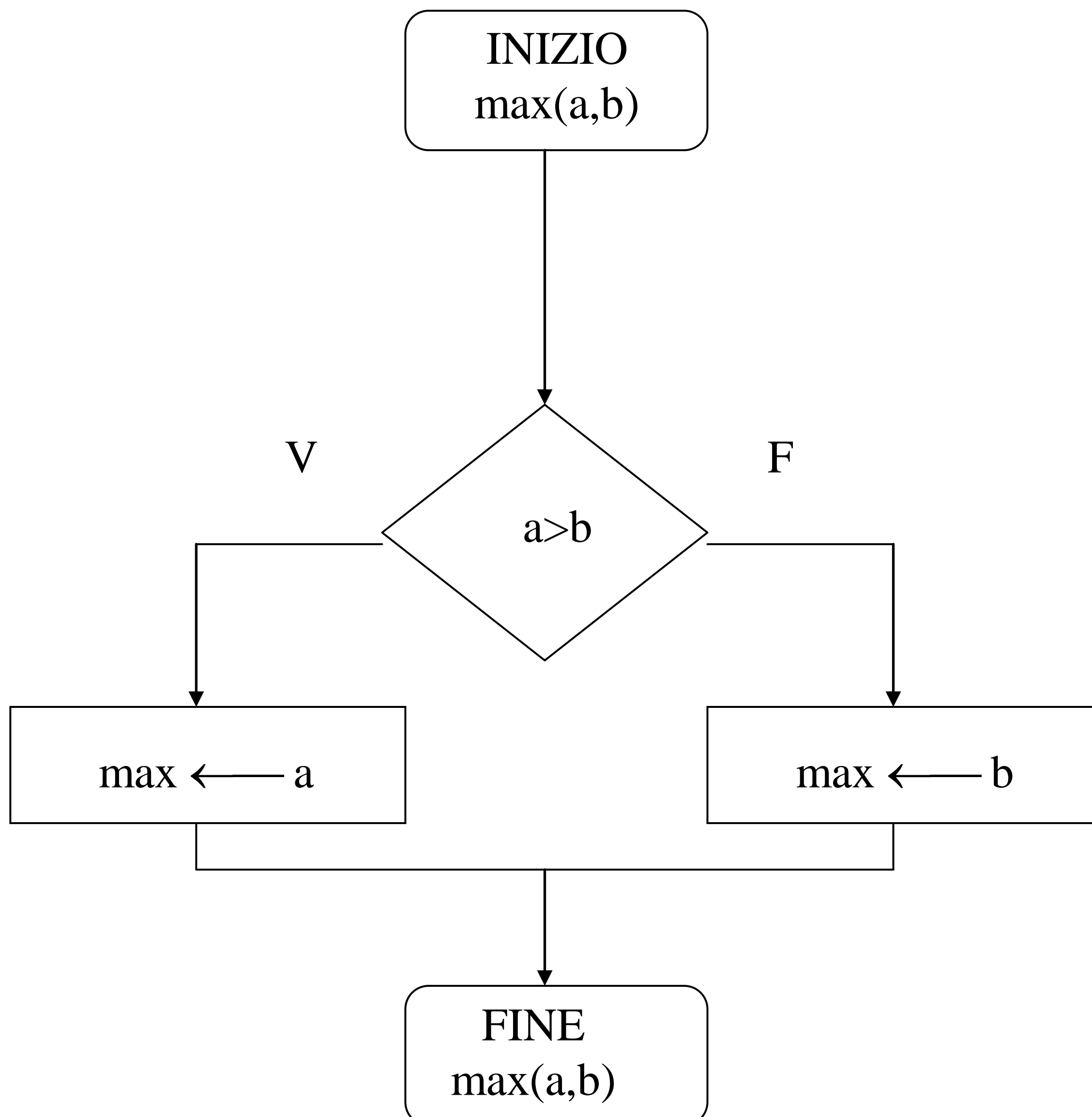
I sottoprogrammi vengono rappresentati in un flow-chart da un blocco di questo tipo.

Il diagramma a blocchi del Problema 2 è quindi il seguente:

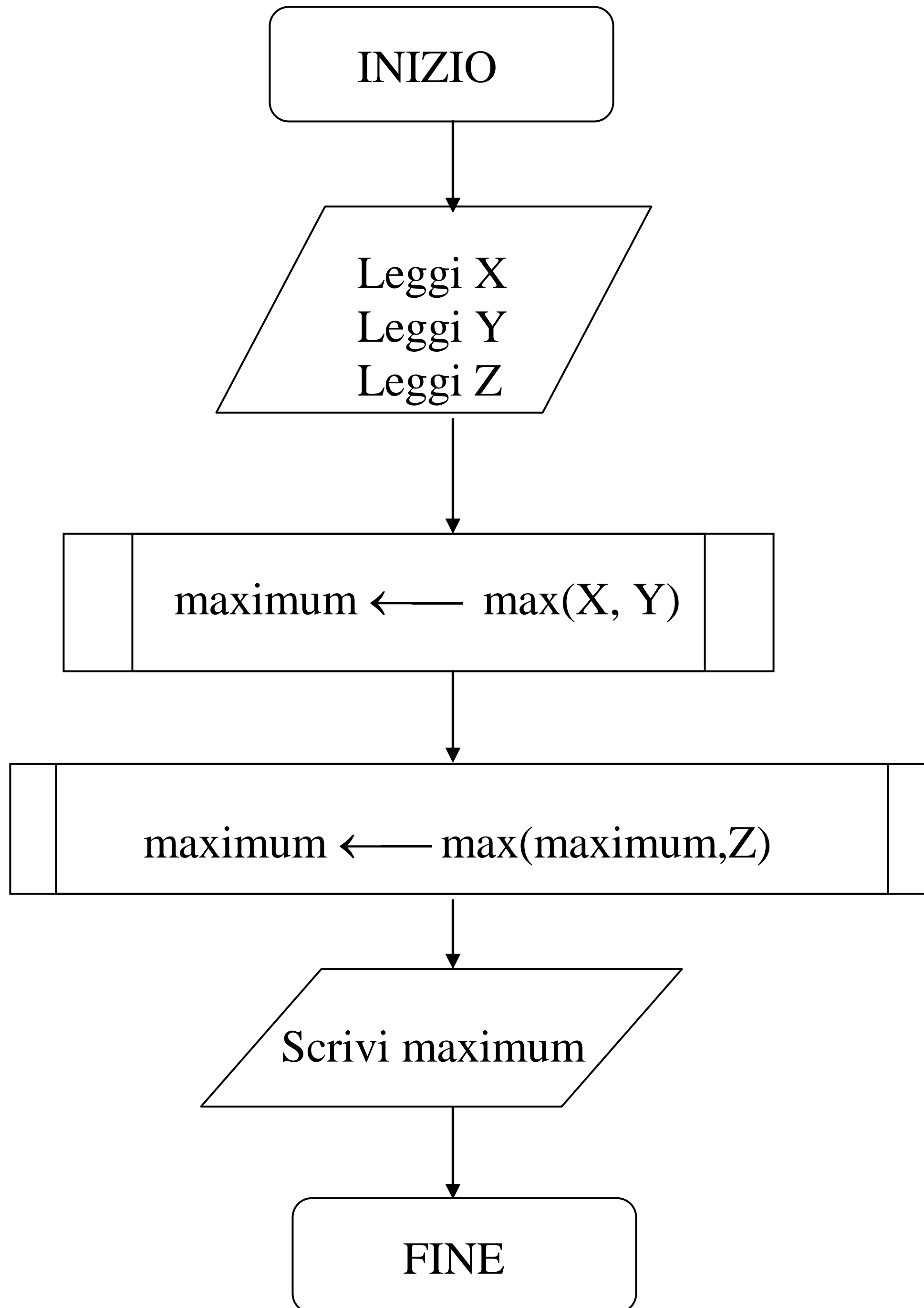


Il problema della ricerca del massimo tra due numeri può venire trattato mediante una *funzione*, max(a,b), in cui a e b sono detti *parametri*.

Il diagramma a blocchi della funzione max sarà il seguente:



Utilizzando questa funzione il diagramma di flusso relativo al Problema 2 diviene:



In generale i sottoprogrammi possono essere costituiti da

- *funzioni*, se restituiscono un risultato direttamente utilizzabile in un'espressione
- *procedure* o *subroutine*, se non restituiscono un risultato direttamente utilizzabile in un'espressione

ALCUNE OSSERVAZIONI

- Le istruzioni che compongono un sottoprogramma vengono scritte *una volta soltanto*.
- Un programma può richiamare un sottoprogramma *più volte* ed ogni volta l'esecutore esegue tutte le istruzioni che compongono il sottoprogramma stesso.
- Ad ogni chiamata di un sottoprogramma è possibile passare allo stesso dati diversi su cui operare attraverso i *parametri*.
- Definendo un sottoprogramma (cioè scrivendo le istruzioni dell'algoritmo corrispondente) è come se si dotasse l'esecutore della capacità di risolvere, come sottoproblemi elementari, dei problemi più complessi.
- I linguaggi di programmazione evoluti dispongono abitualmente di *librerie di funzioni* che consentono di trattare come sottoproblemi elementari dei problemi che non lo sono affatto (ad esempio il calcolo della radice quadrata di un numero)

In generale, con problemi complessi, è indispensabile individuare sottoproblemi più semplici, da risolvere separatamente.

- Programmazione *top-down*: si parte da un algoritmo molto generico che descrive globalmente la soluzione del problema per passare via via ad algoritmi sempre più dettagliati che descrivono le singole operazioni particolari (problemi terminali elementari)

- Programmazione *bottom-up*: si parte della realizzazione di algoritmi che risolvono specifici problemi (moduli); questi vengono poi collegati tra loro creando una struttura più complessa fino ad arrivare al programma finale.

La programmazione di tipo top-down consente

- di avere una visione più chiara delle varie parti del programma
- di sviluppare il programma per fasi
- di affidare eventualmente lo sviluppo dei singoli moduli a programmatori diversi
- di utilizzare eventuali moduli già esistenti
- di verificare il programma ad ogni livello di realizzazione (test più semplici ed efficienti)

Problema 3: determinare il massimo di n numeri reali.

Anche in questo caso ci si riconduce alla ricerca del massimo tra due numeri.

Tralasciando per il momento le istruzioni di input, il problema potrebbe essere risolto in questo modo:

P1:	Trovare il massimo tra i primi due numeri
P2:	Trovare il massimo tra il terzo numero ed il risultato del sottoproblema precedente
P3:	Trovare il massimo tra il quarto numero ed il risultato del sottoproblema precedente
	………………………………………
Pn-1	Trovare il massimo tra l’n-esimo numero ed il risultato del sottoproblema precedente

Il problema è così risolto poiché ad ogni passo ci si riconduce alla soluzione del Problema 1.

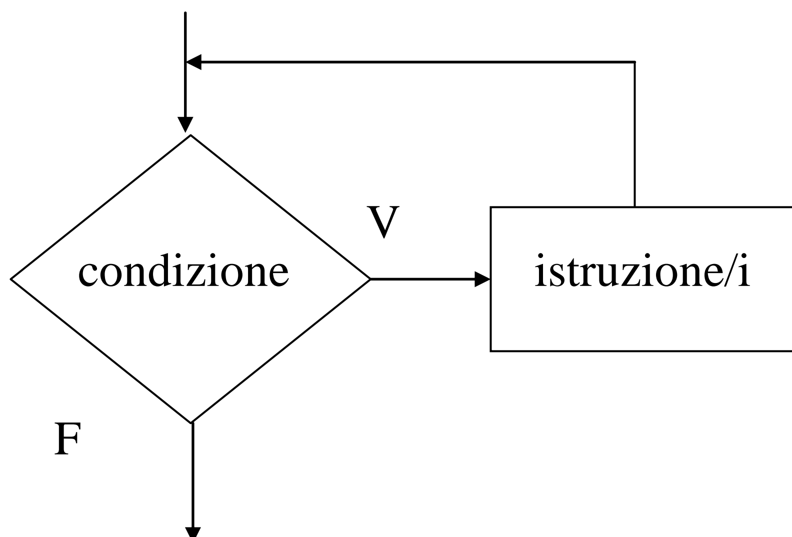
Per ripetere l’esecuzione del calcolo del massimo tra due numeri si dovrà eseguire un ciclo, nel modo seguente:

P1:	Trovare il massimo tra i primi due numeri
P2:	Finché ci sono numeri da verificare, ripetere il passo P3
P3:	Trovare il massimo tra il nuovo numero da esaminare ed il più grande trovato in precedenza

La struttura: *Finché condizione ripetere*
istruzione/i

è presente in tutti i linguaggi strutturati e viene chiamata *struttura di tipo while* o *ciclo while*.

- La condizione costituisce un test per il quale si verifica il valore logico (vero o falso).
- Fintantoché il valore logico del predicato è vero si ripete l'esecuzione dell'istruzione.
- Dopo ogni esecuzione dell'istruzione viene rivalutato il test ed il ciclo termina quando tale test risulta falso.
- Una volta che il ciclo sia terminato, vengono eseguite le istruzioni successive nella sequenza.
- Affinché il ciclo abbia termine l'istruzione (o le istruzioni) contenute all'interno del ciclo devono prima o poi fare in modo che la condizione assuma il valore logico falso.



Do While condizione
istruzione/i
Loop

La risoluzione completa del Problema 3 richiede di indicare come effettuare l'input dei dati, in modo che l'algoritmo funzioni per un numero qualsiasi di numeri tra i quali cercare il massimo.

1° caso - Si richiede all'utente di indicare quanti sono i dati.

L'algoritmo sarà allora:

```
Scrivi "Quanti sono i numeri da esaminare?"
Leggi n
If n > 0 Then
   Leggi m
   cont ⟵ 1                    cont è un contatore.
   Do While cont < n
      Leggi x
      cont ⟵ cont + 1
      m ⟵ max(m,x)
   Loop
   Scrivi "Il massimo è" m
End If
```

A sua volta, ammettendo ora che l'esecutore "conosca" la relazione d'ordine nell'insieme dei numeri reali, l'algoritmo di determinazione del massimo tra due numeri sarà:

```
If a>b Then
   max ⟵ a
Else
   max ⟵ b
End If
```

2° caso – Si chiede all'utente se vuole continuare o meno.

L'algoritmo diventa allora:

Scrivi "Questo programma serve per trovare il massimo in una sequenza di numeri"

Scrivi "Inserisci il primo numero"

Leggi m

Scrivi "Vuoi inserire altri numeri? (S/N)"

Leggi risposta

Do While risposta = "S"

    Leggi x

    m ⟵ max(m,x)

    Scrivi "Vuoi inserire altri numeri ? (S/N)"

    Leggi risposta

Loop

Scrivi "Il massimo è" m

PROGRAMMAZIONE STRUTTURATA

L’efficienza di un programma è in genere misurata in termini di

- tempo di esecuzione
- memoria richiesta
- affidabilità
- facilità di manutenzione
- estensibilità

Un programma chiaro e semplice consente

- facilità nella fase di verifica e correzione degli errori
- facilità nello sviluppo, nella manutenzione e nell’aggiornamento anche da parte di persone diverse dall’autore
- riduzione degli errori in fase di realizzazione

Alcune buone regole di programmazione:

- commentare adeguatamente il programma
- scegliere nomi di variabili significativi
- aumentare la modularità
- standardizzazione
- altre semplici regole (indentazione, eliminazione di parti ridondanti, …)

Ma prima di tutto:

SEMPLIFICARE LA STRUTTURA DEL PROGRAMMA

TEOREMA DI BÖHM-JACOPINI (1966)

Qualunque algoritmo descrivibile con diagrammi a blocchi è descrivibile utilizzando soltanto le seguenti *strutture di controllo*:

1) **BLOCCO**: sequenza di istruzione

2) **SELEZIONE**: If ... Then ... Else ...

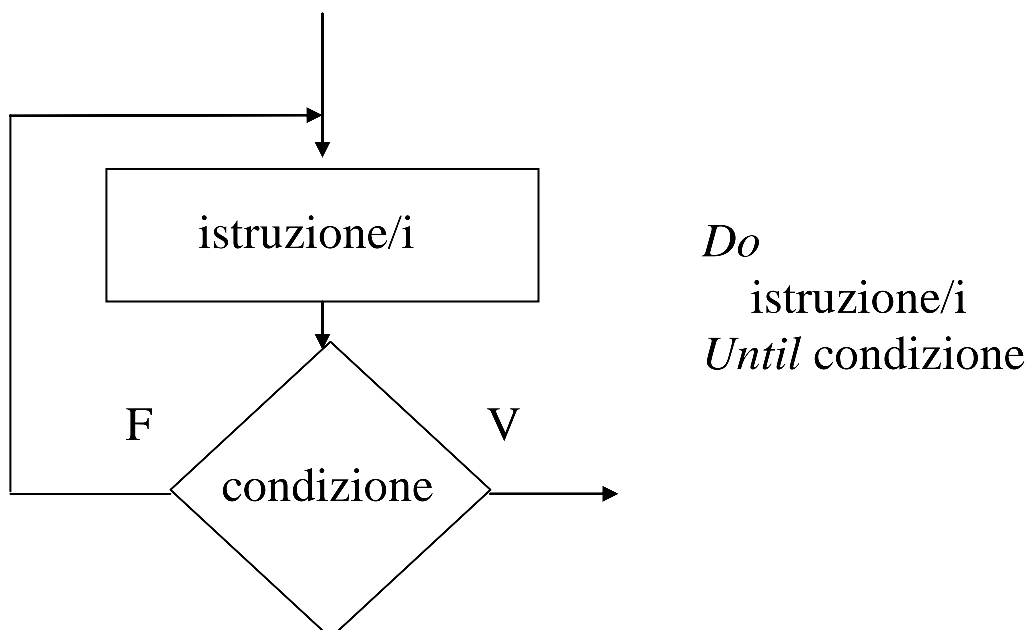
3) **FRASE ITERATIVA**: Do While ...

a condizione di usare, se necessario, variabili ausiliarie e duplicazioni di blocchi.

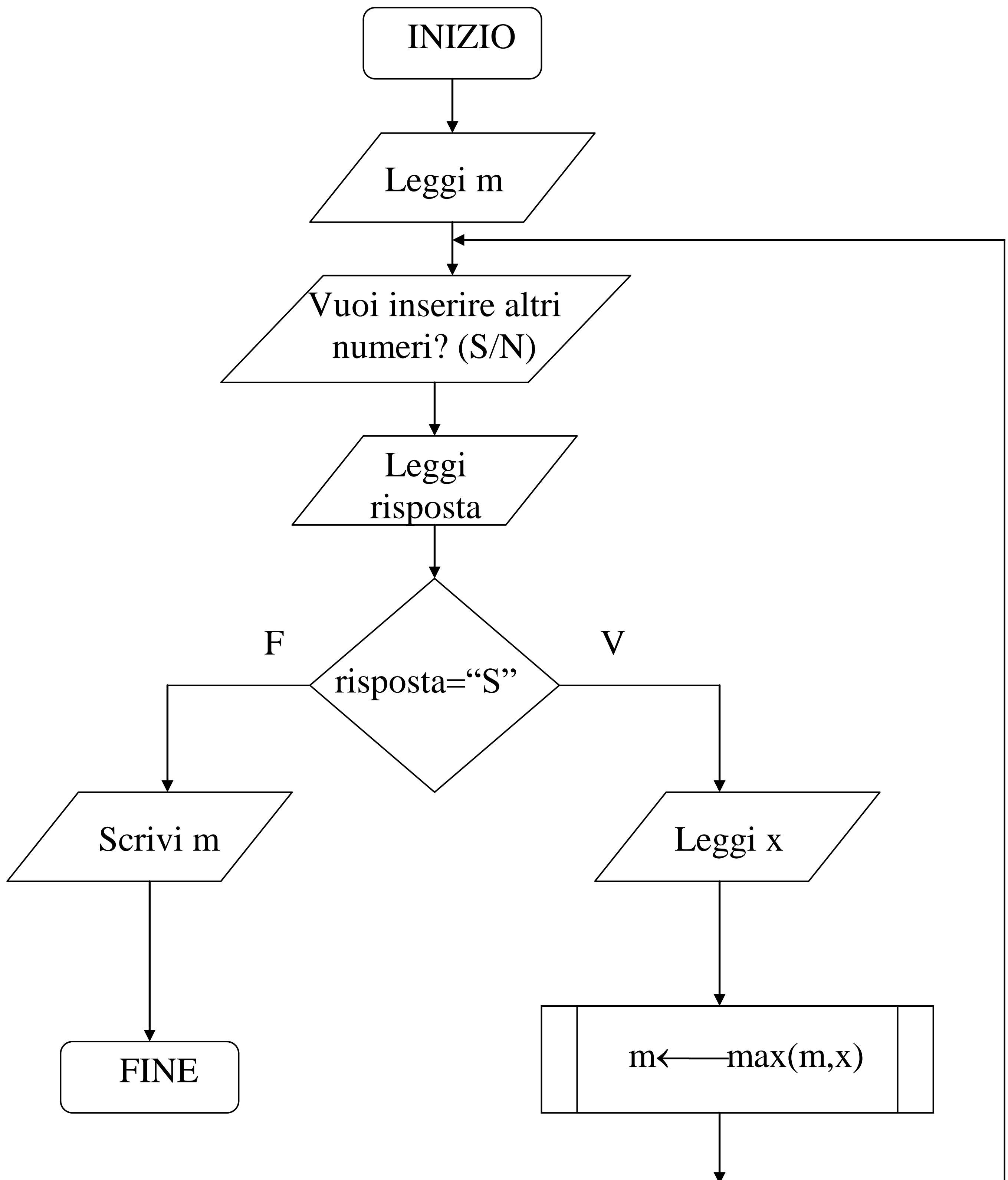
Questo teorema costituisce la base della:

PROGRAMMAZIONE STRUTTURATA

NB: le strutture precedenti hanno la caratteristica fondamentale di avere *un unico punto di entrata* ed *un unico punto di uscita*.
Oltre alla selezione ed al ciclo while esistono anche altre strutture di controllo, ad esempio



Il Problema 3 precedente potrebbe essere risolto anche con un algoritmo che non utilizza le sole strutture di controllo viste:



La programmazione strutturata nasce inizialmente come critica all'uso dell'istruzione GOTO. Questa istruzione consente di spostare il flusso di istruzioni eseguite ad un qualunque altro punto del programma. Ad esempio:

```
i = 1
mostra: Visualizza i
i = i + 1
If i < 10 Then
  Goto mostra
End If
```

In versione strutturata si ha, in modo equivalente ma più chiaro:

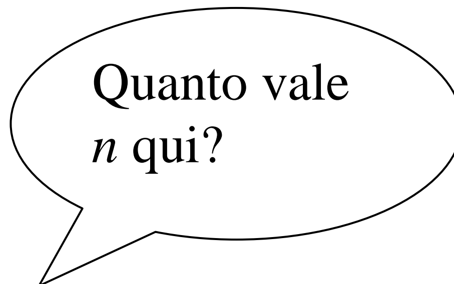
```
i = 1
Do While i < 10
  Visualizza i
  i = i + 1
Loop
```

Nel suo articolo *Go-to Statements Considered Harmful*, pubblicato nel 1968, Dijkstra ha fortemente criticato l'uso del GOTO, principalmente per le ragioni che seguono. Usando i GOTO:

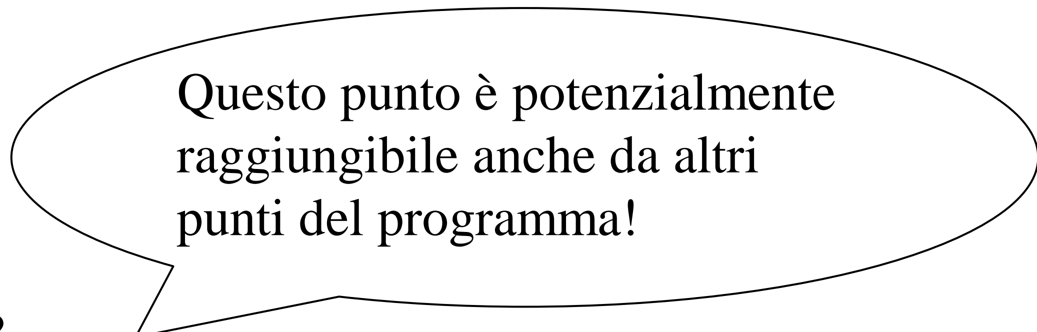
- Non si è sempre in grado di capire il valore di una variabile di ciclo al termine del ciclo.
- Non si è sempre in grado di capire lo stato del programma in presenza di un'istruzione potenzialmente raggiungibile con un goto.
- Non si è sempre in grado di capire il progresso di un programma.

Esempi:

```
Do While n<>0
   istruzioni
  If condizione Then
      Goto fuori
  End If
Loop
fuori: istruzione
```

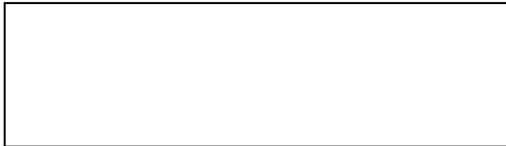



```
top: If (n = 0) Then
   Goto bottom
End If
istruzione1
Goto top;
bottom: istruzione2
```

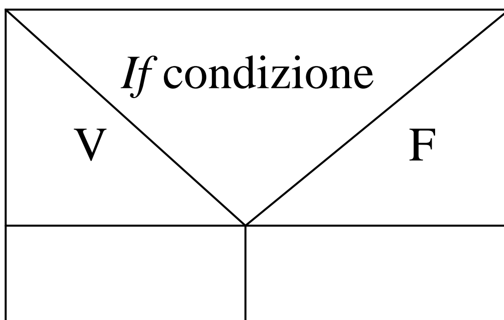


DIAGRAMMI A STRUTTURA

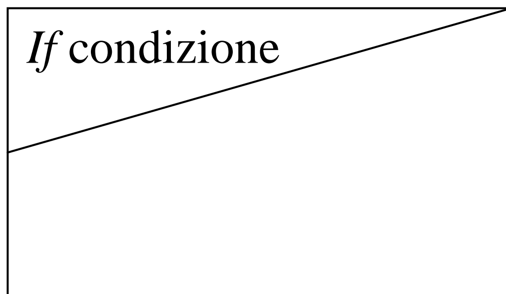
Un'alternativa ai diagrammi a blocchi sono i *diagrammi a struttura*.



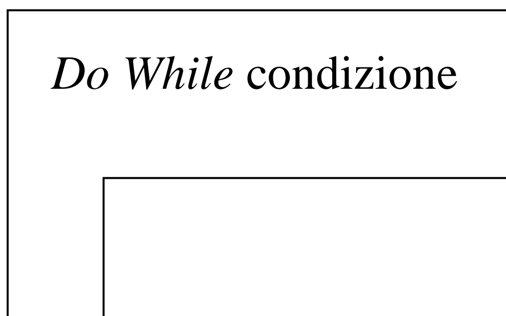
Blocco semplice (istruzioni di assegnazione, input/output, …)



Blocco condizionale:
If … Then … Else …

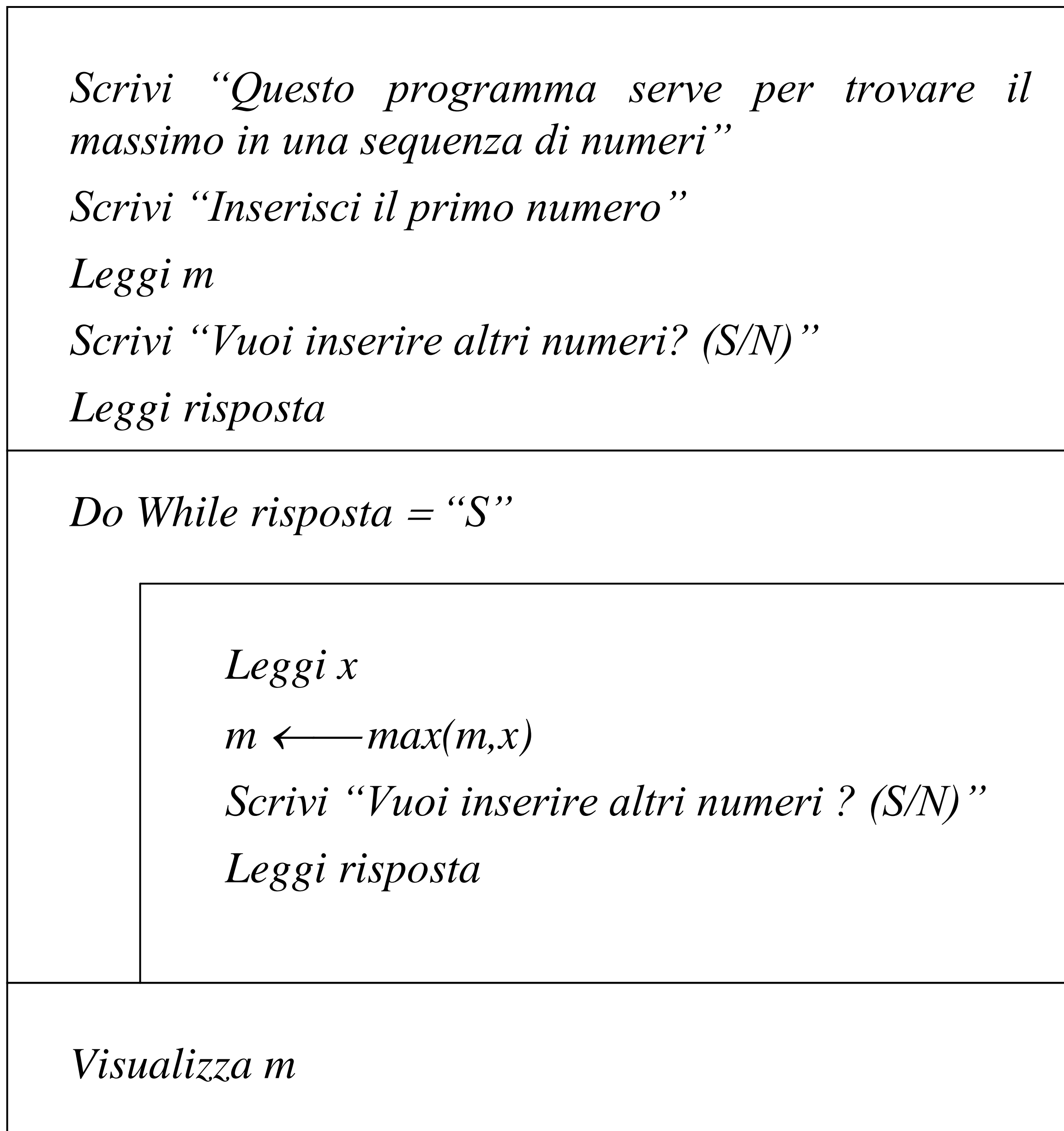


Blocco condizionale:
If … Then …



Blocco per rappresentare un ciclo *while*

Con i diagrammi a struttura l'algoritmo di determinazione del massimo tra n numeri (con n inserito dall'utente) viene espresso nella forma che segue.



Esercizio 3: calcolare il prodotto tra due numeri interi positivi, supponendo che l'esecutore sia in grado di eseguire le sole operazioni di somma e sottrazione.

Dati due numeri interi positivi n ed m, si ha ovviamente:

$$n \bullet m = \underbrace{m + m + \ldots + m}_{n \text{ volte}}$$

L'algoritmo procederà sommando *m* per *n* volte ed accumulando via via il totale in una variabile *z*.

Leggi n

Leggi m

$z \longleftarrow 0$

Do While $n > 0$

  $z \longleftarrow z + m$

  $n \longleftarrow n - 1$

Loop

Visualizza z

Per testare l'algoritmo si possono provare degli esempi di esecuzione (*traccia dell'algoritmo*), ad esempio con *n*=3, *m*=2:

z	n	m	n > 0
	3	2	
0			V
2	2		V
4	1		V
6	0		F

Problema 4: dati due numeri interi positivi *m* ed *n*, determinare il loro massimo comun divisore (MCD).

Ricordiamo che:

Def.: Dati due interi positivi *n* e *m*, *n* è detto *divisore* di *m* se esiste un intero positivo *q* tale che $q \bullet n = m$.

In generale,

Def: Dati due interi *m*, *n* con *n* diverso da zero, esiste un'unica coppia di interi *q* e *r*, detti quoziente e resto, tali che $m = n \bullet q + r$, con $0<=r<|n|$.

Un possibile algoritmo per trovare il massimo comun divisore è:

P1:	Scomponi i due numeri in fattori primi
P2:	Individua i fattori le cui basi sono comuni ad entrambi i numeri
P3:	Scegli tra questi quelli che hanno il minimo esponente
P4:	Moltiplica questi ultimi tra loro ottenendo così il MCD

Questo non è tuttavia un algoritmo agevole, in particolare per la necessità di individuare i fattori primi dei numeri considerati.

Indicati con *m* ed *n* i due interi positivi di cui si vuole trovare il massimo comun divisore, un metodo migliore consiste nell'esaminare sequenzialmente, via via, tutti i numeri naturali compresi tra 1 ed il minimo tra *m* ed *n* e, ogni volta che uno di questi risulti un divisore sia di *m* sia di *n*, nel salvarlo in una opportuna variabile. Al termine questa variabile conterrà il massimo comun divisore tra *m* ed *n*.

P1:	Leggi m
P2:	Leggi n
P3:	k ⟵ min(m,n)
P4:	cont ⟵ 1
P5:	Finché cont ≤ k Se cont è divisore di m e di n allora MCD ⟵ cont cont ⟵ cont + 1
P6 :	Scrivi MCD

Un classico operatore è la *divisione intera*, qui indicata con \

17 \ 3 = 5 13 \ 4 = 3

Quindi per vedere se cont è divisore di m si può:

- calcolare a ⟵ m \ cont
- verificare se a • cont = m

Un altro operatore comune è il *modulo*, qui indicato con *mod* restituisce il resto della divisione intera tra due numeri, ad esempio

17 mod 5 = 2

Per vedere se cont è divisore di m è quindi sufficiente verificare se

m mod cont = 0

L'algoritmo completo diventa:

```
Leggi m; Leggi n
If m < n Then
   k ⟵ m
Else
   k ⟵ n
End If
cont ⟵ 1
Do While cont <= k
   a ⟵ m \ cont
   b ⟵ n \ cont
   If (a * cont = m) and (b * cont = n) Then
      mcd ⟵ cont
   End If
   cont ⟵ cont + 1
Loop
Visualizza mcd
```

NB: utilizzando l'operatore mod, il ciclo while diviene

```
Do While cont <= k
   If (m mod cont = 0) and (n mod cont = 0) Then
      mcd ⟵ cont
   End If
   cont ⟵ cont + 1
Loop
```

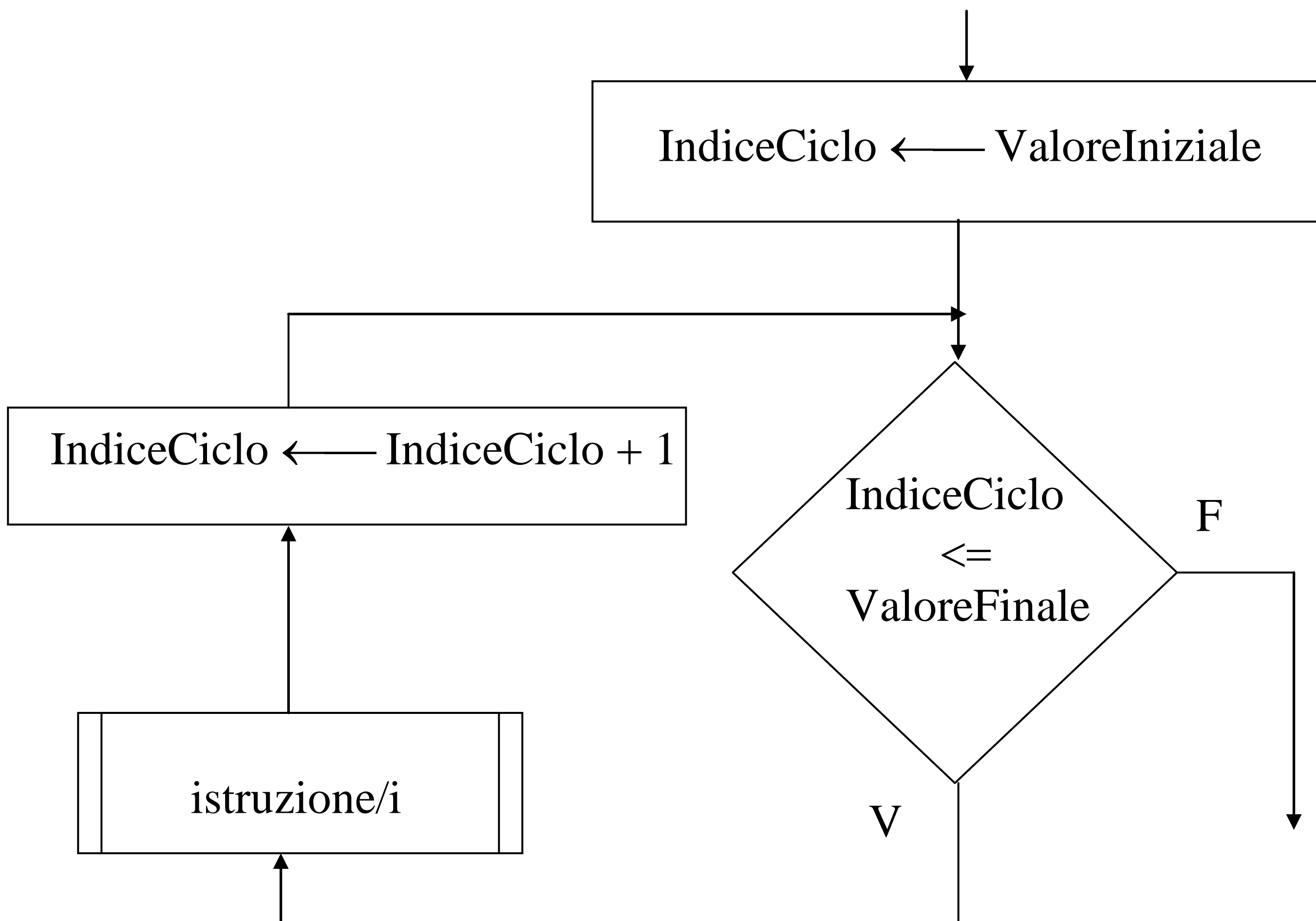
Nel precedente ciclo Do While la variabile cont viene incrementata di 1 ad ogni iterazione.

Esiste una struttura di controllo che effettua automaticamente l'aggiornamento del contatore, il *ciclo For/Next*,

For IndiceCiclo = ValoreIniziale *To* ValoreFinale

istruzione/i

Next IndiceCiclo



Nel ciclo For/Next sono quindi implicitamente contenuti i seguenti tipi di istruzioni:

- di inizializzazione: IndiceCiclo ⟵ ValoreIniziale
- di controllo: IndiceCiclo <= ValoreFinale
- di modifica del contatore: IndiceCiclo ⟵ IndiceCiclo + 1

Sono quindi "praticamente" equivalenti le strutture cicliche:

```
For IndiceCiclo = ValoreIniziale To ValoreFinale
   istruzione/i
Next IndiceCiclo
```

```
IndiceCiclo = ValoreIniziale
Do While IndiceCiclo <= ValoreFinale
   istruzione/i
   IndiceCiclo = IndiceCiclo + 1
Loop
```

L'algoritmo di ricerca del massimo comun divisore diventa:

```
Leggi m; Leggi n
If m < n Then
   k ⟵ m
Else
   k ⟵ n
End If
For cont = 1 To k
   If (m mod cont = 0) and (n mod cont = 0) Then
      mcd ⟵ cont
   End If
Next cont
Visualizza mcd
```

Un altro algoritmo: l'algoritmo di Euclide – versione 1

Per calcolare in modo di gran lunga più efficiente il MCD si può usare l'algoritmo di Euclide (367 a.C – 283 a.C.). Una prima versione si basa sul seguente teorema:

Dati m, n, q *interi positivi, se* q *divide* m *ed* n*, allora divide anche la loro differenza.*

Dimostrazione: Sia $m>n$. Allora, esistono h, k interi positivi tali che $m=h \bullet q$, $n=k \bullet q$. Dunque $m\text{-}n=(h\text{-}k) \bullet q$ con $h\text{-}k$ intero positivo.

Quindi, se $m>n$, $MCD(m,n) = MCD(n, m\text{-}n)$

Ne segue un facile algoritmo di calcolo del MCD:

```
Leggi m
Leggi n
Do While m>0
    if n>m then
        z = m
        m = n
        n = z
    End If
    m = m-n
Loop
Visualizza n
```

Esempio:

MCD(50,12) = MCD (38, 12) = MCD(26, 12)=MCD(14, 12) =

MCD(12, 2) = MCD(10, 2)= MCD(8,2) = MCD(6, 2) =

MCD (4, 2) = MCD(2,2) = 2

L'algoritmo di Euclide – versione 2

Un'altra versione dell'algoritmo di Euclide per il calcolo del massimo comun divisore si basa sul seguente teorema:

Dati m, n *numeri interi non negativi con* m>n, *siano* q *e* r, *rispettivamente quoziente e resto della divisione di* m *per* n. *Allora,* c *divide* m *e* n *se e soltanto se divide* n *e* r.

Dimostrazione: Se c divide m e n, allora $m=h \bullet c$, $n=k \bullet c$. Dunque, poiché $m = n \bullet q+r$, si ha $r = m - n \bullet q = (h-k \bullet q) \bullet c$, per cui c divide anche r. Si può dimostrare facilmente anche il verso contrario dell'enunciato.

Quindi, se $m>n$, $MCD(m,n) = MCD(n, r)$

Ne segue un altro algoritmo di calcolo del massimo comun divisore:

```
Leggi m
Leggi n
Do While n <> 0
        z = m mod n
        m = n
        n = z
Loop
Visualizza m
```

Esempio:

MCD(50,12) = MCD (12, 2) = 2

Esercizio 4: scrivere un algoritmo che realizzi la divisione intera di due numeri interi positivi fornendo in output il quoziente ed il resto, utilizzando le sole operazioni aritmetiche di addizione e sottrazione.

Presi a, b numeri interi positivi, q sarà il quoziente ed r il resto di *a:b (cioè* $a = b \cdot q + r$, con q, r numeri naturali e $r < b$*)*.

L'algoritmo procede per sottrazioni successive di b da a fintantoché la quantità ottenuta non diviene minore di b, costituendo così il resto.

Leggi a

Leggi b

$q \longleftarrow 0$

Do While $a >= b$

  $a \longleftarrow a - b$

  $q \longleftarrow q + 1$

Loop

$r \longleftarrow a$

Visualizza q, r

Esercizio 5: calcolare il fattoriale di un numero naturale

$n! = n \cdot (n-1) \cdot (n-2) \ \ldots \cdot 1$

```
Leggi n
If n >= 0 Then
   fattoriale ⟵ 1
   If n > 0 Then
      For i = 1 To n
         fattoriale = fattoriale * i
      Next i
   End If
   Visualizza fattoriale
Else
   Visualizza "n deve essere non negativo"
End If
```

Istruzione di inizializzazione della variabile fattoriale

VARIABILI STRUTTURATE

Le variabili finora utilizzate venivano indicate soltanto con dei *nomi* che individuavano *celle di memoria centrale* nelle quali venivano memorizzati i dati sui quali opera l'algoritmo.

Nel corso dell'esecuzione tali variabili possono assumere valori *diversi*. È possibile anche definire delle *costanti*, che corrispondono sempre a celle di memoria centrale, il cui valore riamane però costante per tutta la durata dell'esecuzione dell'algoritmo.

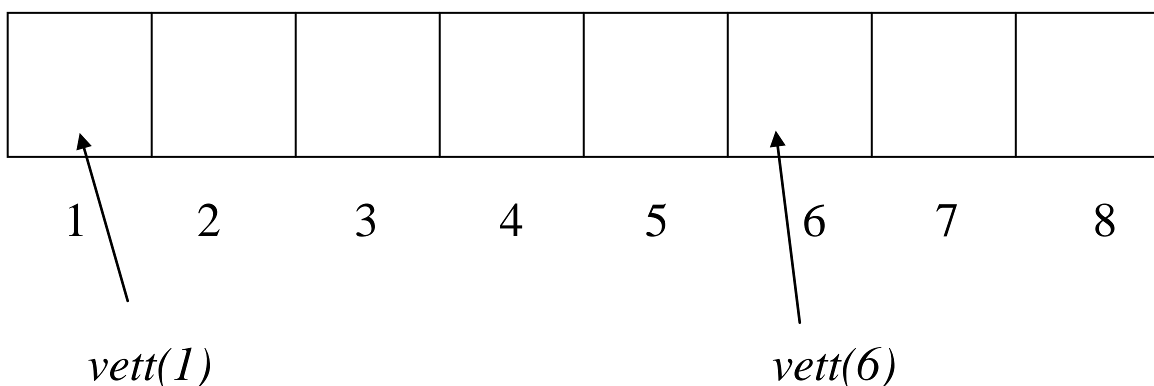
La maggior parte dei linguaggi di alto livello rende disponibili anche delle variabili *strutturate*, che consentono di riferirsi a più variabili tra loro correlate come se si trattasse di un'unica variabile aggregata.

Un primo esempio di variabile strutturata è il *vettore* (o *array*).

L'array è costituito da una sequenza di celle di memoria *consecutive* ed *omogenee* (cioè tutte dello stesso tipo).

Al vettore viene assegnato un *unico* nome e le singole celle che lo costituiscono vengono identificate mediante il nome del vettore ed un *indice*, cioè un numero intero che indica il numero d'ordine del singolo elemento del vettore.

Esempio: vettore costituito da 8 elementi contenenti numeri interi di nome *vett*: *vett(1)* identifica il primo elemento, *vett(6)* il sesto.



MATRICI O ARRAY MULTIDIMENSIONALI

Un *array multidimensionale* è formato da elementi (singole celle) identificati da più indici simultaneamente, ognuno in corrispondenza di una delle dimensioni della matrice.

Esempio: vendite(i,j) i = 1, …, 50 (prodotti) j = 1, …, 12 (mesi)

è una *matrice* che può venire utilizzata per memorizzare le quantità vendute di 50 prodotti diversi nei 12 mesi dell'anno; vendite(27,3) conterrà quindi la quantità venduta del 27-esimo prodotto nel mese di marzo.

Esercizio 6: scrivere un algoritmo che legge una sequenza di numeri positivi e terminante con uno zero e ne stampa la somma.

L'algoritmo richiederà un ciclo, ma questo non potrà essere realizzato con un For/Next, poiché non è noto a priori quanti numeri verranno letti.

somma = 0

Visualizza "Inserisci un numero positivo o 0 per terminare"

Leggi x ← Lettura fuori ciclo

Do While x > 0

  somma = somma + x

  Visualizza "Inserisci un numero positivo o 0 per terminare"

  Leggi x

Loop

Visualizza somma

Esercizio 7: somma degli elementi di un vettore.

Si supponga che i dati siano già stati inseriti in un vettore *vett* il cui numero di elementi *n* è noto.

For i = 1 To n

somma = somma + vett(i)

Next i

Visualizza somma

Problema 5: leggere una sequenza di numeri interrotta da uno zero e visualizzarli in ordine opposto a quello di lettura.

È necessario utilizzare un vettore, che indicheremo con vett.

```
i = 0
Leggi x
Do While x <> 0
    i = i + 1
   vetti(i) = x
   Leggi x
Loop
Do While  i > 0
   Visualizza vett(i)
    i = i − 1
Loop
```

NB: alla fine del primo ciclo *i* contiene il numero di elementi letti

Una variante della struttura For/Next consente di indicare un incremento, cioè di quanto è incrementato l'indice ad ogni ciclo:

For IndCiclo = ValoreIniziale *To* ValoreFinale [*Step* Incremento]

   istruzione/i

Next IndCiclo

Se l'incremento non è specificato, si assuma che esso valga uno.

Una variante dell'ultima parte dell'algoritmo è quindi:

```
For j =  i To 1 Step –1
   visualizza vett(j)
Next j
```

RICERCA SEQUENZIALE IN UN VETTORE

Problema 6: cercare in un vettore la prima posizione (l'indice) occupata da un elemento assegnato in input

L'algoritmo scorre il vettore fin quando o trova l'elemento o non ci sono più elementi del vettore da esaminare.

Supponiamo che il vettore sia già stato letto e sia quindi memorizzato in memoria centrale (indichiamo con *n* il numero di elementi), ad esempio con la seguente sequenza di istruzioni:

```
Leggi n
For i = 1 To n
   Leggi vett(i)
Next i
```

Per cercare il valore contenuto nella variabile x viene usato un ciclo *Do While*. Il ciclo avrà termine quando una delle seguenti due condizioni si verifica:

- si è trovato l'elemento
- non ci sono più elementi da esaminare

Utilizziamo quindi la variabile booleana *blnTrovato*, che verrà inizializzata al valore *False* ed assumerà il valore *True* se l'elemento viene trovato all'interno del vettore.

```
Leggi x
blnTrovato = False
i = 1
Do While (i <= n) And (blnTrovato = False)
   If vett(i) = x Then
      blnTrovato = True
```

```
    Else
        i = i + 1
    End If
Loop
If blnTrovato = True
    Visualizza "La posizione è " & i
Else
    Visualizza "L'elemento non è presente nel vettore"
End If
```

Non viene usato un ciclo For/Next perché è inutile continuare la ricerca fino ad esaurire tutti gli elementi del vettore qualora si sia già trovato l'elemento cercato. È possibile usare un ciclo For/Next qualora si vogliano reperire tutte le posizioni in cui l'elemento cercato è presente nel vettore, ad esempio:

```
Leggi x
For i = 1 To n
    If vett(i) = x Then
        visualizza "L'elemento cercato è presente in posizione " & i
    End If
Next i
```

NB: alcuni linguaggi contengono istruzioni in grado di interrompere l'esecuzione di un ciclo For/Next o Do While, che potrebbero quindi essere inserite all'interno di una struttura condizionata *If* situata all'interno del ciclo. Il loro uso tuttavia può rendere non agevole la lettura e comprensione dell'algoritmo codificato.

ORDINAMENTO (SORT) DI UN VETTORE

Problema 7: scrivere un algoritmo che ordini gli elementi di un vettore in ordine crescente.

Supponiamo che il vettore sia già stato letto e sia quindi memorizzato in memoria centrale. Chiameremo il vettore *vett* ed indicheremo con n il numero di elementi.

Per ordinare il vettore, iniziamo cercando di posizionare in testa al vettore l'elemento più piccolo. A tale fine, confrontiamo l'elemento *vett(1)* successivamente con *vett(2)*, *vett(3)*, …, *vett(n)*. Ogni volta che viene individuato un elemento *vett(i)* più piccolo dell'elemento *vett(1)* questi due elementi vengono scambiati, cioè il contenuto dell'elemento in posizione *vett(i)* viene posto nella variabile *vett(1)* e viceversa (si utilizzerà a tale scopo una variabile ausiliaria).

Al termine di questa operazione avremo in posizione 1 l'elemento più piccolo del vettore. La stessa operazione verrà poi rifatta per il sottovettore del vettore iniziale costituito dagli elementi con indice compreso tra *2* ed *n*. In questo modo al termine avremo in posizione *2* l'elemento più piccolo tra tutti quelli del vettore escluso l'elemento minimo, già posizionato in *vett(1)*.

L'algoritmo continua in questo modo fino a considerare soltanto il sottovettore composto dagli elementi con indice compreso tra *n–1* ed *n*.

Al termine di tutte queste operazioni il vettore risulterà ordinato.

È evidente quindi che per realizzare questo algoritmo si utilizzeranno due cicli:

- *Ciclo esterno in cui i varia da 1 a n–1*: individua l'elemento del vettore nel quale si inserirà il minimo degli elementi con indice compreso tra *i* ed *n*;

- *Ciclo interno in cui j varia da i+1 ad n*: consente di confrontare l'elemento i-esimo con tutti gli elementi successivi.

For i = 1 To n–1	Ciclo For/Next esterno
For j = i+1 To n	Ciclo For/Next interno
If vett(i) > vett(j) Then	
a = vett(i)	Scambio di contenuto tra *vett(i)* e *vett(j)* usando la variabile ausiliaria *a*
vett(i) = vett(j)	
vett(j) = a	
End If	
Next j	
Next i	

Esercizio 8: scrivere un algoritmo che memorizzi in una matrice bidimensionale gli elementi del triangolo di Tartaglia, per un certo numero di righe.

Indicheremo con T(i,j) la matrice destinata a contenere gli elementi del triangolo di tartaglia.

La relazione che permette di costruire il j-esimo elemento sulla riga i-esima del triangolo di Tartaglia a partire dagli elementi della riga precedente è

$$T(i,j) = T(i-1,j-1) + T(i-1,j)$$

L'algoritmo sarà allora:

Leggi n

$T(1,1) = 1$

For $i = 2$ *To* n

    $T(i,1) = 1$

    For $j = 2$ *To* $i - 1$

        $T(i,j) = T(i-1,j-1) + T(i-1,j)$

    Next j

    $T(i,i) = 1$

Next i

NB: il triangolo di Tartaglia (Niccolò Fontana, 1499 circa – 1557) è noto anche come triangolo di (Blaise) Pascal (1623 – 1662), di (Omar) Khayyam (1048 – 1131), o di Yang Hui (1238 – 1298), ma pare fosse noto già a Pingala (2° secolo A.C.)