Fondamenti di Programmazione

Prof. Ing. Loris Penserini, PhD

elpense@gmail.com

https://dblp.org/pid/p/LorisPenserini.html

Informatica: Modellare il Mondo Reale

L'Informatica (o Computer Science) è la scienza che studia la gestione dell'informazione tramite procedure automatizzate. Questo è realizzabile attraverso: una consolidata teoria dell'informazione, linguaggi formali di rappresentazione, e sistemi elettronici per l'esecuzione automatica.

Attraverso specifici linguaggi di programmazione, si possono scrivere algoritmi (sequenze di istruzioni) che un calcolatore elettronico è in grado di elaborare in modo automatico al fine di fornire delle informazioni utili all'utente partendo da dati iniziali.

Ciascun paradigma di programmazione fornisce allo sviluppatore di software un differente approccio concettuale per implementare il pensiero computazionale, cioè la definizione della strategia algoritmica utile per digitalizzare e **risolvere un problema del mondo reale**.

L'evoluzione delle tecnologie digitali nei settori della Robotica e dell'AI, spingono i ricercatori a sviluppare nuovi paradigmi di programmazione…

Programmazione

La maggior parte dei linguaggi di programmazione si basano sul «**paradigma imperativo**», in cui il programma consiste in un insieme di istruzioni dette anche «**direttive**» o «**comandi**».

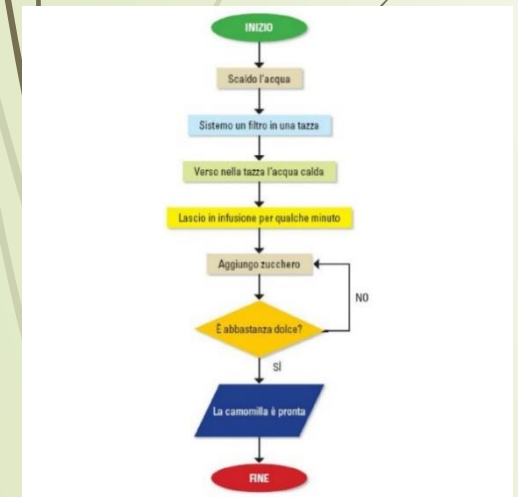
La programmazione imperativa viene generalmente contrapposta a quella dichiarativa, in cui un programma consiste in un insieme di «affermazioni» (non «comandi») che l'esecutore è tenuto a considerare vere e/o rendere vere. Un esempio di paradigma dichiarativo è la programmazione logica (es. Prolog, LTL, CTL, ecc.).

Come vedremo in seguito, i concetti introdotti dalla programmazione strutturata sono alla base di numerosi altri linguaggi di programmazione, non ultimo quello orientato agli oggetti.

Comunicare con il Calcolatore

Indipendentemente dall'hardware, dal S.O. e dai linguaggi di programmazione, queste sono le fasi necessarie per comunicare con un elaboratore.

ALGORITMO



PROGRAMMATORE

LINGUAGGIO ASSEMBLER

LINGUAGGIO MACCHINA

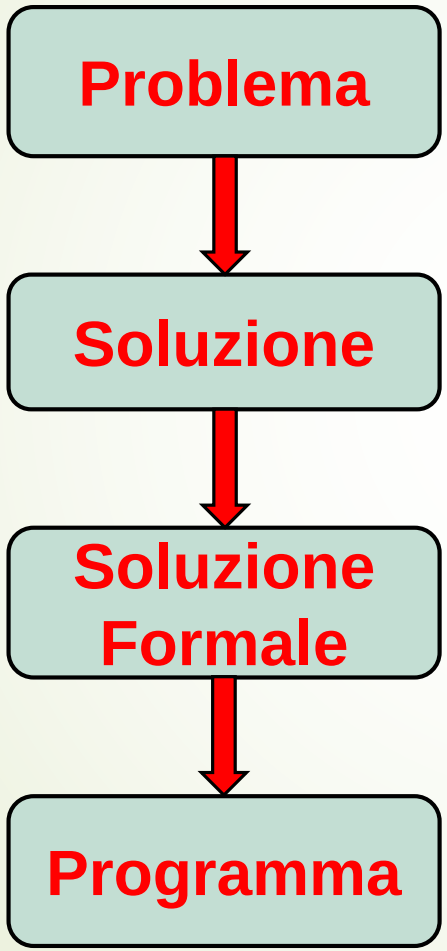
Scrivere un Algoritmo

A **design-time** un problema per essere risolto da un «elaboratore elettronico» o «esecutore» deve essere rappresentato con una formalizzazione dell'**ALGORITMO**, cioè nella logica del linguaggio formale con la quale si descrive il modello informatico di un problema del mondo reale.

L'algoritmo è definito da un insieme finito di passi (operazioni) che portano a risolvere un problema informatico, dove si hanno dei dati in ingresso (input) e si producono dei risultati (output).

La soluzione ad un problema del mondo reale può avere differenti algoritmi risolutori, tuttavia la programmazione cerca di trovare quello più efficiente.

Dall'Algoritmo al Programma



Esperienza da cui si ricava l'idea

Metodi formali e strumenti automatici per scrivere l'Algoritmo

Paradigmi di programmazione e relativi linguaggi

Algoritmo e Programma

Trovata la soluzione migliore al problema, questa viene formalizzata nell'**algoritmo** che è composto da una sequenza finita di istruzioni che sono comprensibili dall' «**esecutore**» ovvero l'elaboratore elettronico.

Infine, l'algoritmo deve essere tradotto in un linguaggio formale comprensibile all' «esecutore»: il **programma**.

Proprietà di un Algoritmo

E' composto da una sequenza ***finita*** di operazioni/istruzioni che vengono svolte dall' «esecutore» in un ***processo sequenziale***.

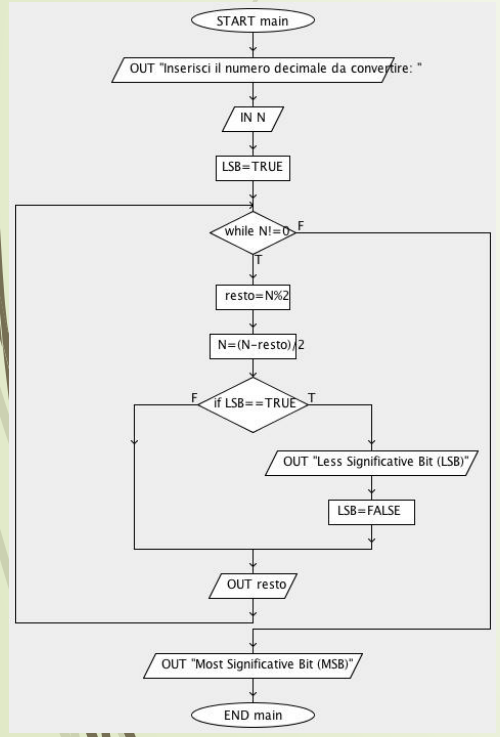
Ogni operazione deve produrre, se eseguita, un effetto ***osservabile*** che possa essere descritto.

Ogni operazione deve produrre lo stesso effetto ogni volta che viene eseguita a partire dagli stessi input e dalle stesse condizioni iniziali (***determinismo***).

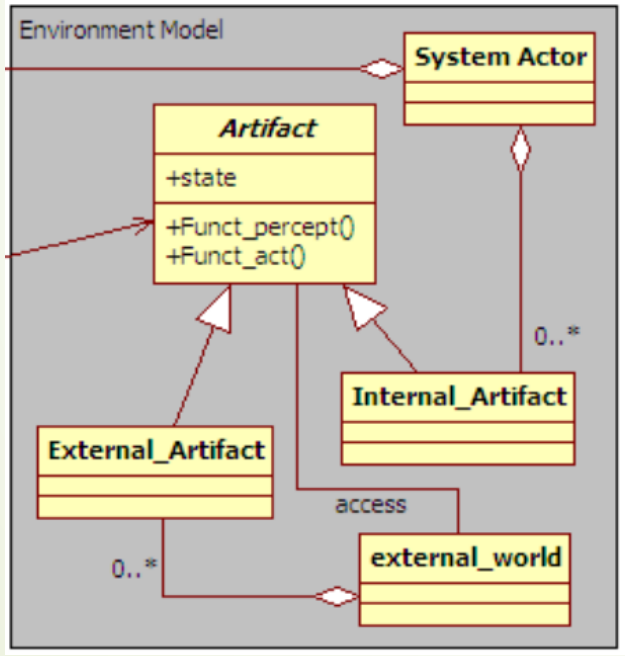
Paradigmi di programmazione…

Alcuni dei principali paradigmi di programmazione e relativi livelli di astrazione dell'ingegneria del software.

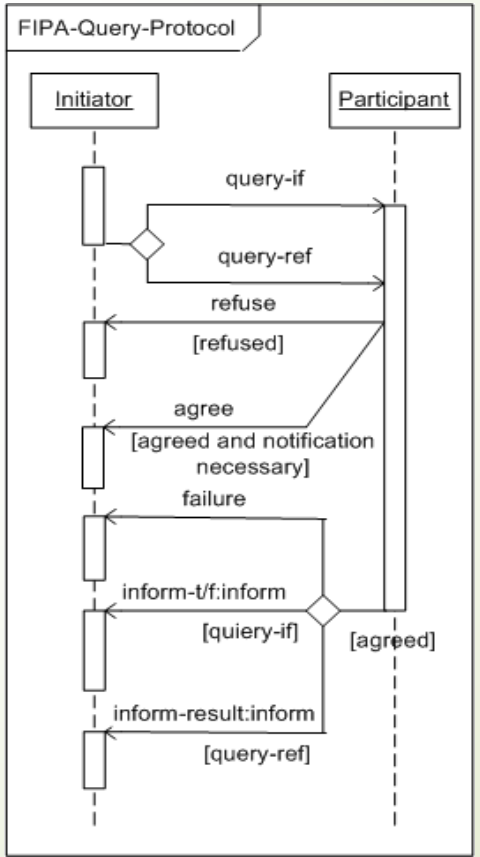
Flow-Chart (Structured Prog.)



UML (Object Oriented Prog.)



Agent-UML (Agent Oriented Prog.)



Scratch/Blockly (Block based Prog.)

Le fasi del ciclo di vita del software

Nell'ingegneria del software si spiegano le fasi di sviluppo di un'applicazione software come fosse un prodotto con un suo «ciclo di vita».

Il modello più noto di sviluppo del software è il «**modello a cascata**» (*waterfall model*) in cui ci sono una serie di fasi a cascata e ciascuna fase riceve ingressi dalla fase precedente e produce uscite, che sono a loro volta ingressi per la fase seguente.

- **Studio di fattibilità**
- **Analisi e specifica dei requisiti**
- **Progettazione**
- **Programmazione/Implementazione e test**
- **Installazione, Integrazione e test di sistema**
- **Manutenzione**

Altri modelli si sono evoluti nel tempo: «**modello trasformazionale**» e «**modello a spirale**».

Paradigmi per Sistemi Complessi

Il software di oggi è più complesso che in passato, perché gli si richiede di funzionare in modo autonomo in ambienti dinamici, aperti e imprevedibili. Le metodologie di ingegneria del software che si ispirano ai sistemi autonomi o *self-adaptive* (*SAS*), offrono una soluzione promettente per far fronte al problema di sviluppo di sistemi complessi.

Uno dei paradigmi di programmazione più recente è quello orientato agli agenti: **AOP** (*Agent Oriented Programming*) e il relativo **AOSE** (*Agent Oriented Software Engineering*):

- **AOP: Jade**, **Jadex**, **Jake**, **3APL**, **Alan**, **JEAP**, **…**

[Morandini et al., 2008] [Penserini et al., 2007b] [Pagliarecci et al., 2007] [Panti et al., 2003]

- **AOSE: MaSE**, **GAIA**, **Tropos4AS**,**...**

[Penserini et al., 2007a] [Morandini et al., 2009]

Le fasi del «ciclo di vita» nella metodologia Tropos4AS sono un ibrido tra mod. trasformazionale (adotta approccio MDA) e mod. a spirale (si reitera sulle fasi precedenti) [Morandini et al., 2017]

Dal Sorgente al Programma Eseguibile

IL CALCOLATORE NON E' IN GRADO DI COMPRENDERE DIRETTAMENTE UN LINGUAGGIO AD ALTO LIVELLO, cioè un linguaggio utile a semplificare il lavoro del programmatore.

Per cui, il testo del programma scritto in un linguaggio di programmazione di alto livello, detto programma **sorgente** (source), deve essere tradotto in **linguaggio macchina** per poter essere eseguito dall'elaboratore.

Assembler

Questo è il linguaggio con il quale si comunica con lo specifico processore del PC in uso. Per esempio, la famiglia dei Motorola (es. MC68000, …) e la famiglia degli INTEL (es. 8086, …, 80386, …, Pentium, ecc.)

Esempio:

```
if (D0 == 5)
       D1++;
D2 = D0;
```

Linguaggio di Alto Livello (C, C++, Java, …)

```
        CMPI.L  #5,D0
        BNE     SKIP
        ADDQ.L  #1,D1
SKIP    MOVE.L  D0,D2
```

Assembly del Processore Motorola 68000

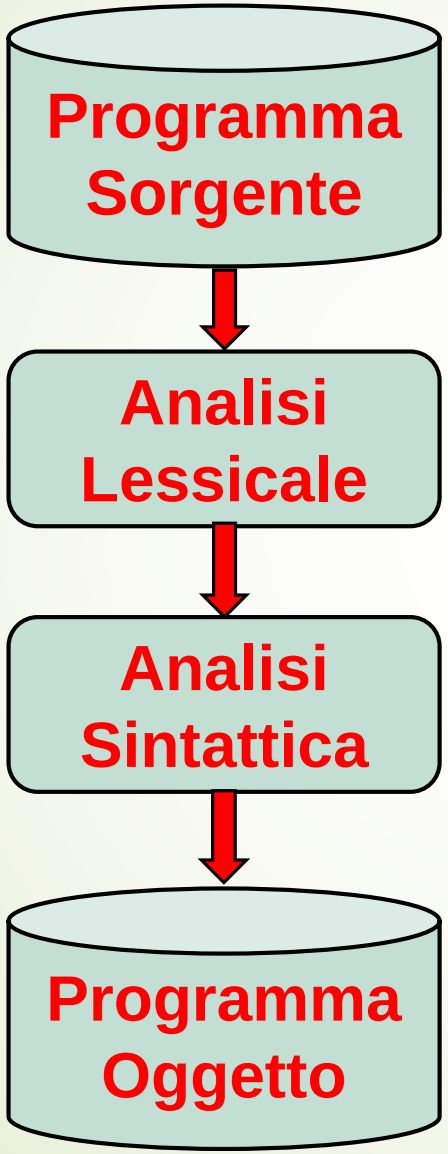
Tradurre il Programma

Il processo di traduzione può essere di due tipologie:

Interpretazione: un'applicazione che prende il nome di interprete considera il testo sorgente, istruzione per istruzione, e lo traduce mentre lo esegue; su questo principio lavorano linguaggi quali il Basic, PHP, Python e altri linguaggi per la gestione di basi di dati e per le applicazioni Web.

Compilazione: un'applicazione (compilatore) che trasforma l'intero programma sorgente in linguaggio macchina, memorizzando in un file il risultato del proprio lavoro. Così un programma compilato una sola volta, può essere eseguito senza bisogno di altri interventi, quante volte si vuole. Il risultato della compilazione si chiama programma oggetto (object). Linguaggi come: C/C++ e Java.

Interpretazione e Compilazione



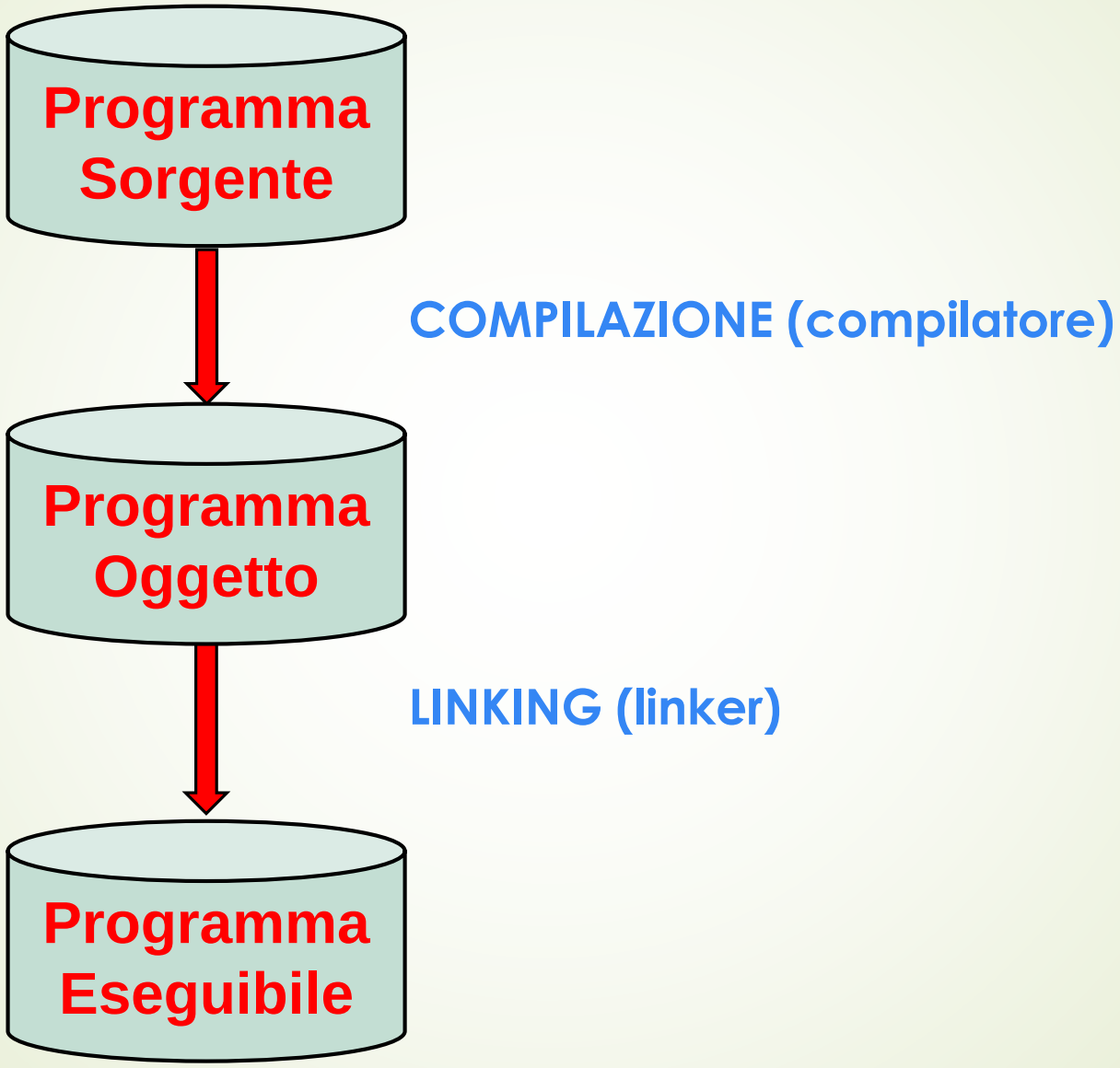
Testo del programma

Controllo della correttezza dei termini e dei simboli

Controllo della correttezza dei costrutti sintattici

Programma tradotto in linguaggio macchina

Compilazione



Pregi e difetti

Velocità di esecuzione: i programmi compilati hanno in genere prestazioni migliori (nella compilazione si possono attuare processi di ottimizzazione dell'eseguibile).

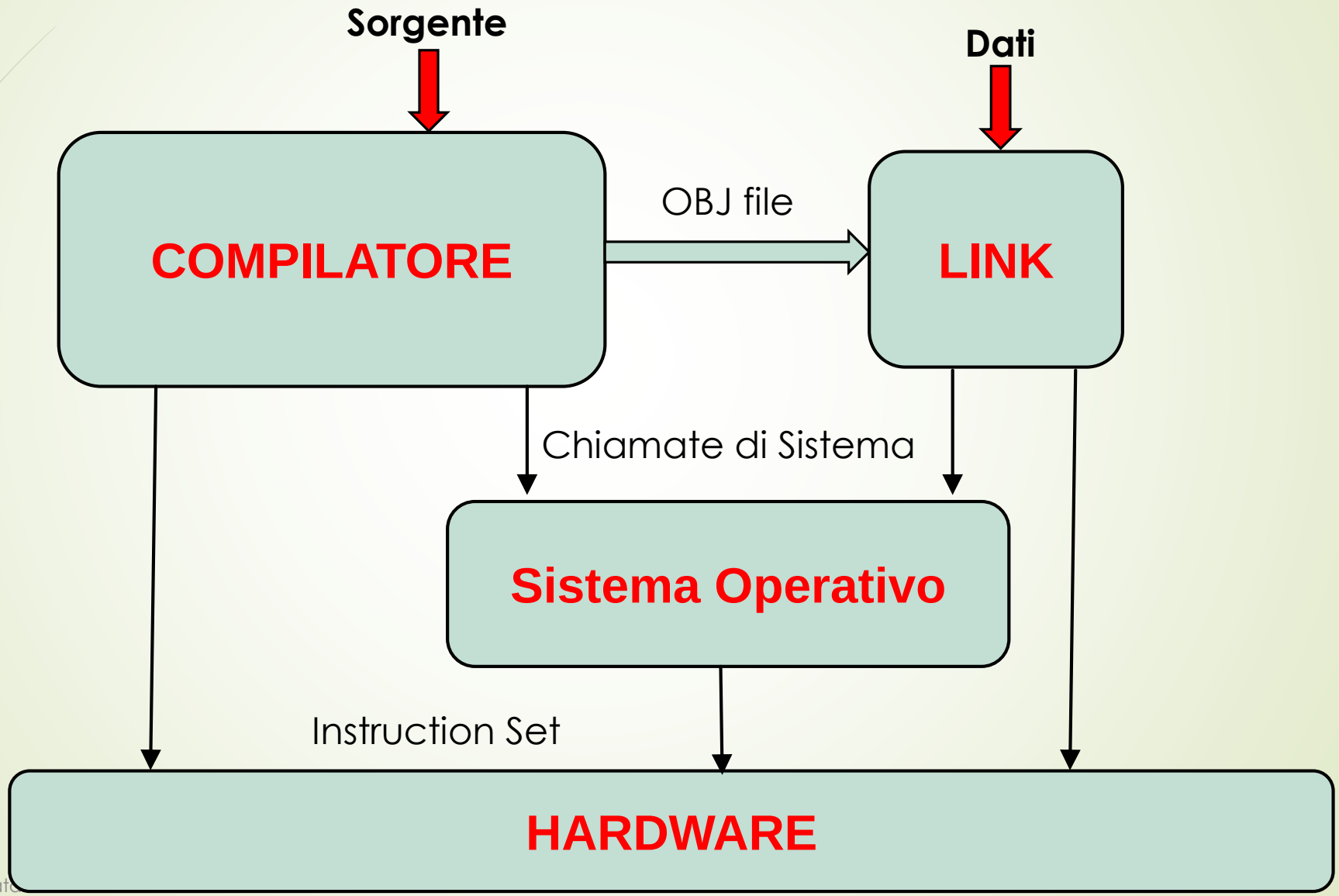
Messa a punto del programma: gli interpreti permettono di correggere gli errori non appena vengono scoperti, senza bisogno di ricompilare interamente il programma.

Sistemi Ibridi

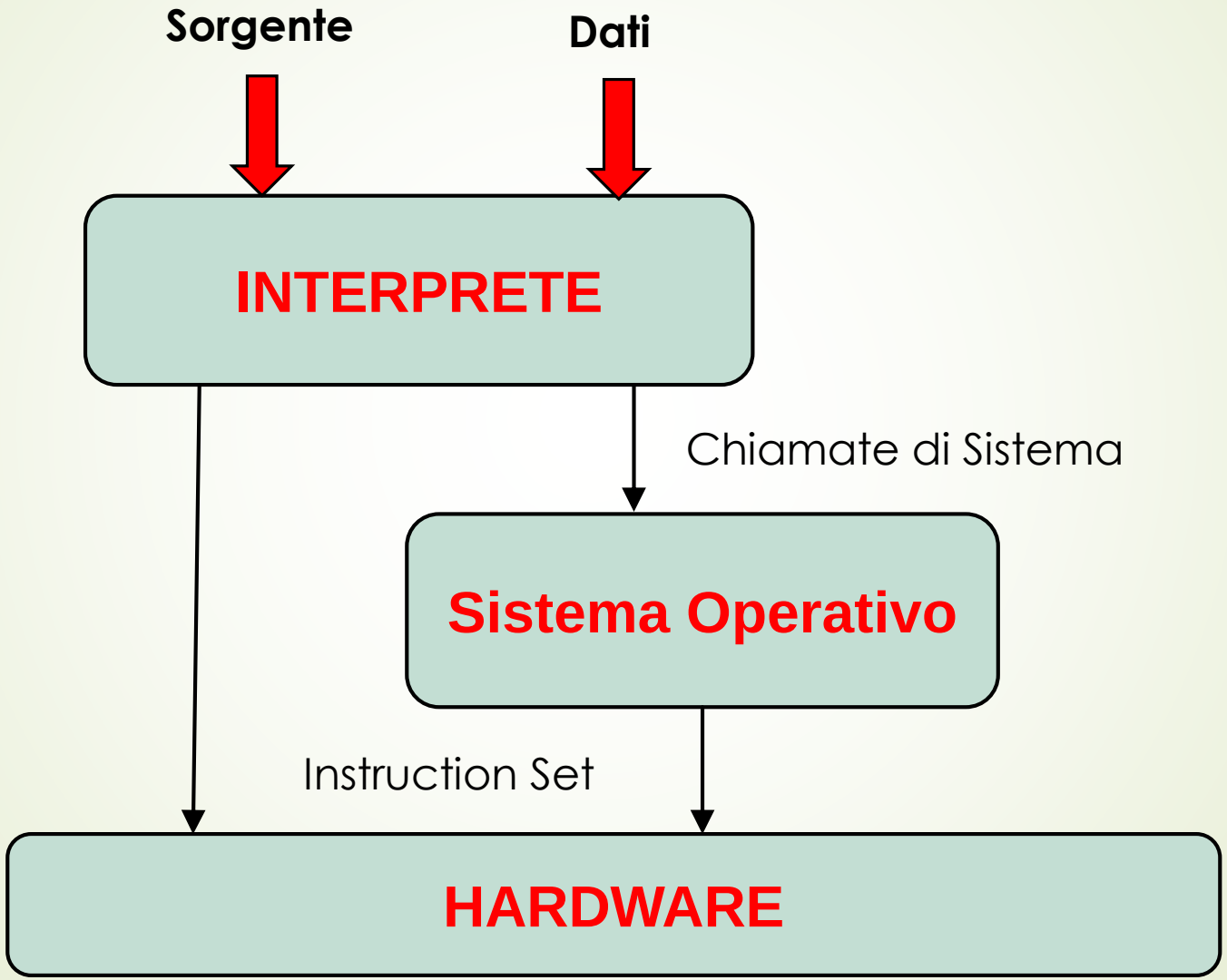
Per esempio nel linguaggio JAVA, la compilazione genera un file oggetto chiamato ByteCode che viene «interpretato» dall'interprete Java (Java Virtual Machine), per dare portabilità al codice.

Tuttavia, il ByteCode può anche essere tradotto in un eseguibile con il linker. In quest'ultimo caso, l'eseguibile è vincolato a funzionare sul particolare hardware e S.O. per il quale è stato generato.

Compilazione

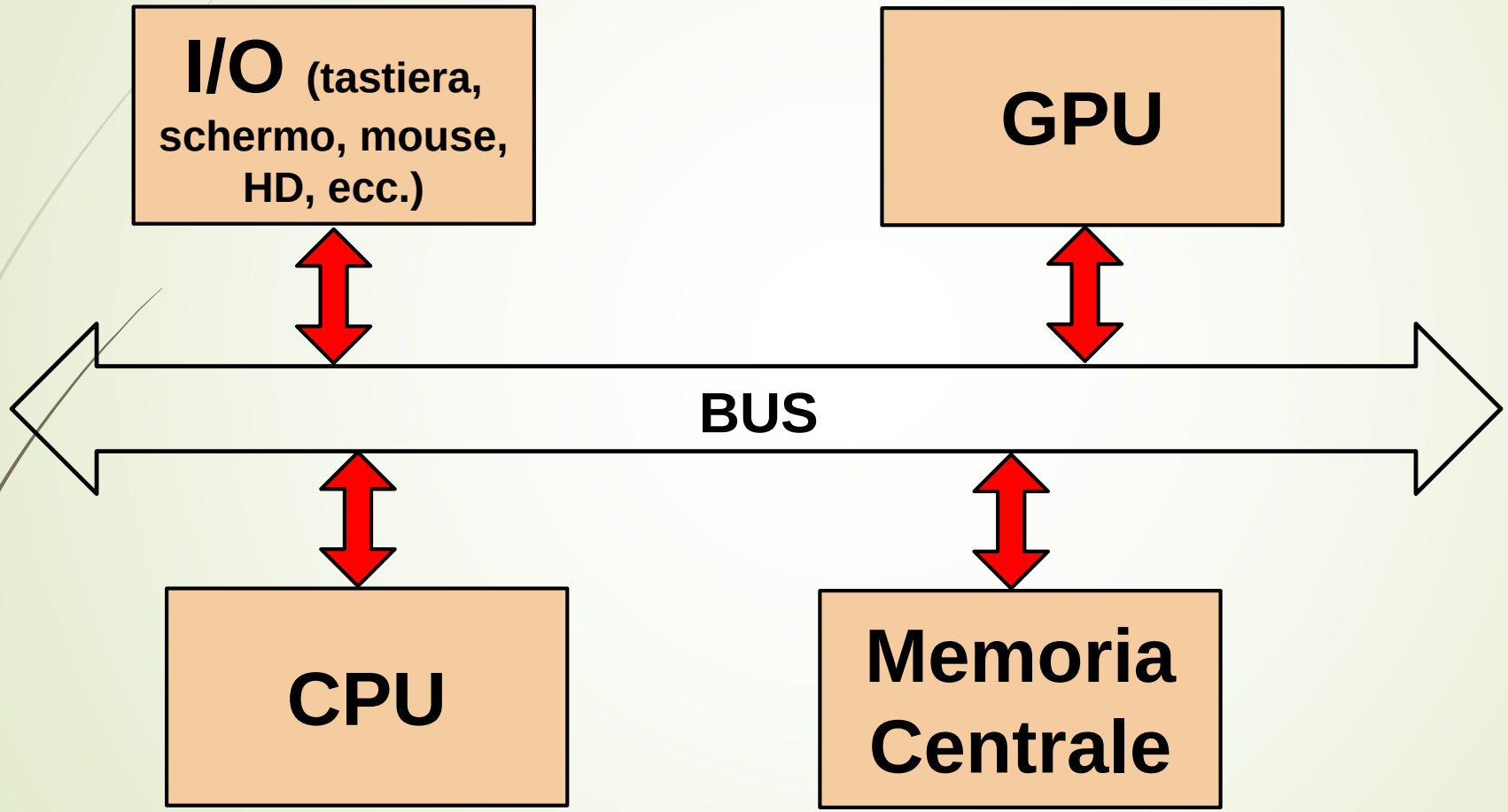


Interpretazione



Eseguire un Algoritmo nel Sistema di Elaborazione (overview)

Dal punto di vista hardware



Macchina di Von Neumann

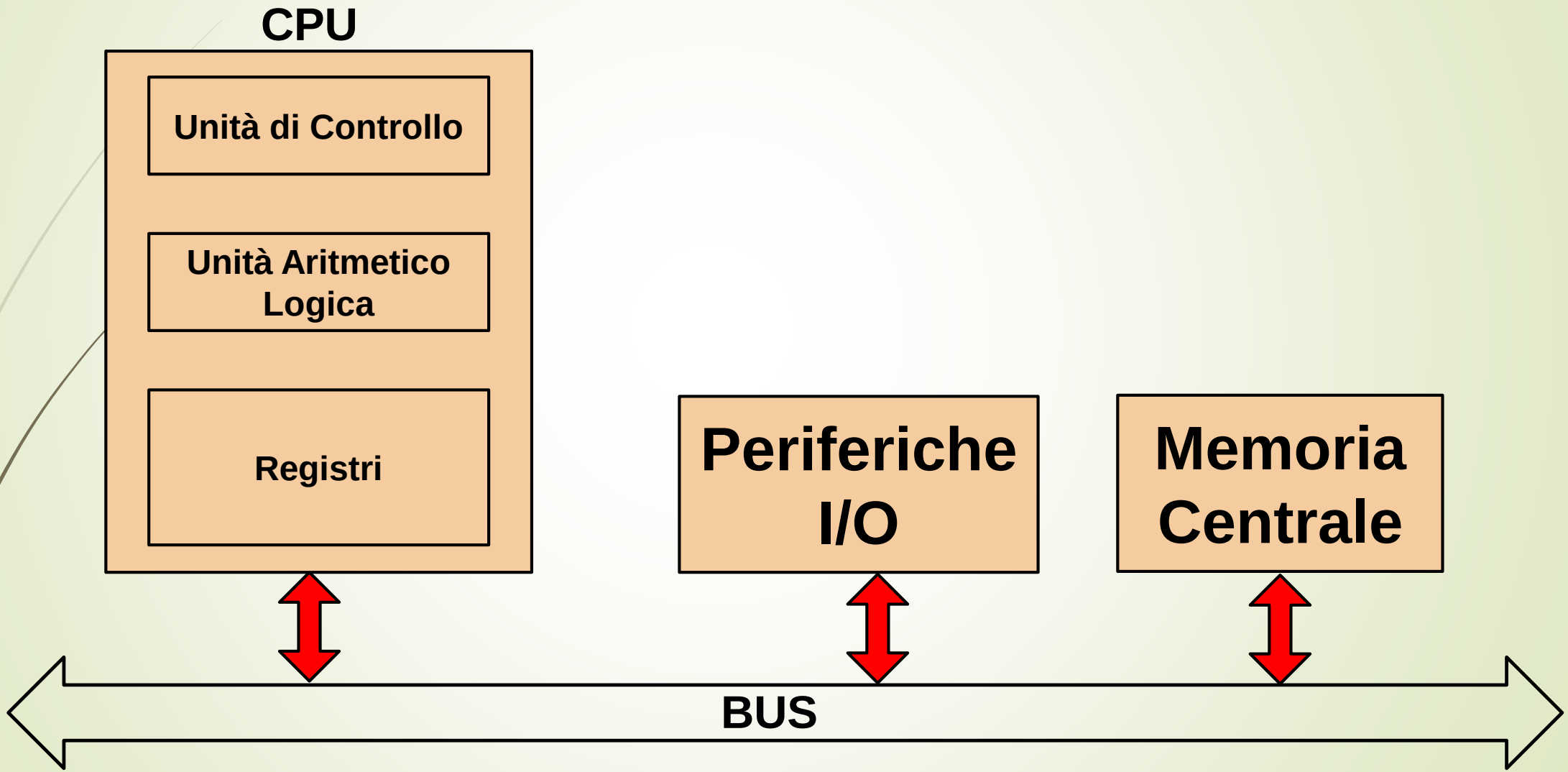
Tipi di BUS

BUS DATI: utilizzato per trasferire i dati tra RAM e CPU e tra CPU e Periferiche I/O

BUS INDIRIZZI: identifica la posizione delle celle di memoria in cui la CPU va a leggere/scrivere

BUS DI CONTROLLO: percorso da segnali di controllo che consentono di selezionare le unità coinvolte in un trasferimento dati (sorgente e destinazione), e quindi definire la direzione dello scambio (scrittura o lettura).

Il Processore



CPU

Unità di Controllo: legge le istruzioni dalla memoria e ne determina il tipo

Unità Aritmetico Logica (ALU): Esegue le operazioni necessarie per eseguire le istruzioni

Registri

- Sono memorie ad alta velocità usate per mantenere risultati temporanei
- Determinano il parallelismo (pipelining) della CPU
- Esistono vari tipi di registri con funzioni specifiche come ad esempio: Program Counter (PC), Instruction Register (IR), Program Status Word (PSW) che contiene info sullo stato di esecuzione del programma, MAR contiene l'indirizzo della locazione di memoria da leggere/scrivere, MDR necessario per scambiare informazioni tra memoria e CPU, ecc.

Frequenza di Clock: Velocità con cui la CPU scandisce le operazioni. Si misura in Hertz (Hz) che significa «volte al secondo»: 2Hz => 2 volte al secondo

Periodo di clock: intervallo di tempo tra un fronte e l'altro della sequenza di impulsi (inverso della frequenza di clock)

Il periodo di clock deve essere sufficientemente lungo da consentire a tutti i segnali elettrici di arrivare.

Esecuzione delle Istruzioni

L'esecuzione di una qualsiasi istruzione segue un sequenza di azioni che tecnicamente prende il nome di ***«Ciclo di Fetch-Decode-Execute»*** come segue:

- **FETCH**: prende l'istruzione corrente dalla memoria e la inserisce nel registro delle istruzioni IR
- Incrementa il program counter (PC) in modo che contenga l'indirizzo dell'istruzione successiva
- **DECODE**: Determina il tipo di istruzione corrente
- Se l'istruzione usa una parola in memoria determina dove si trova
- Se necessario, carica la parola in un registro della CPU
- **EXECUTE**: esegue l'istruzione
- Ricomincia dal nuovo valore del PC

La Macchina di Turing (overview)

La Tesi di Church-Turing

L'uso della Macchina di Turing (MdT) introduce in modo intuitivo l'idea che un problema sia risolubile attraverso un procedimento quando sia possibile costruire una MdT in grado di eseguire un algoritmo che descrive, attraverso passi elementari, il procedimento risolutivo.

Per cui, il concetto di algoritmo è il procedimento risolto dalla MdT astratta.

La Tesi di Church-Turing afferma che: La classe delle funzioni computabili, secondo il concetto intuitivo di algoritmo, coincide con la classe delle funzioni Turing-computabili, cioè computabili con una MdT.

Sulla base della Tesi di Church-Turing possiamo quindi individuare all'interno dell'insieme di tutti i possibili problemi il sottoinsieme dei problemi che sono computabili, cioè quelli ai quali può essere associata una MdT che risolve il problema. Inoltre l'algoritmo risolutivo di un problema computabile può essere identificato con la sequenza delle azioni che vengono eseguite da una MdT.

Automa

Abbiamo precedentemente affermato che un «esecutore» o «elaboratore elettronico» deve eseguire in modo **automatico** l'algoritmo.

Per cui definiamo con il termine «**automa**» un dispositivo in grado di eseguire da solo, cioè in modo automatico, senza l'intervento di una persona, una sequenza di azioni stabilite in precedenza. Inoltre, l'automa è in grado di acquisire informazioni dall'esterno e produrre informazioni verso l'esterno e, infine, al suo interno può assumere stati diversi.

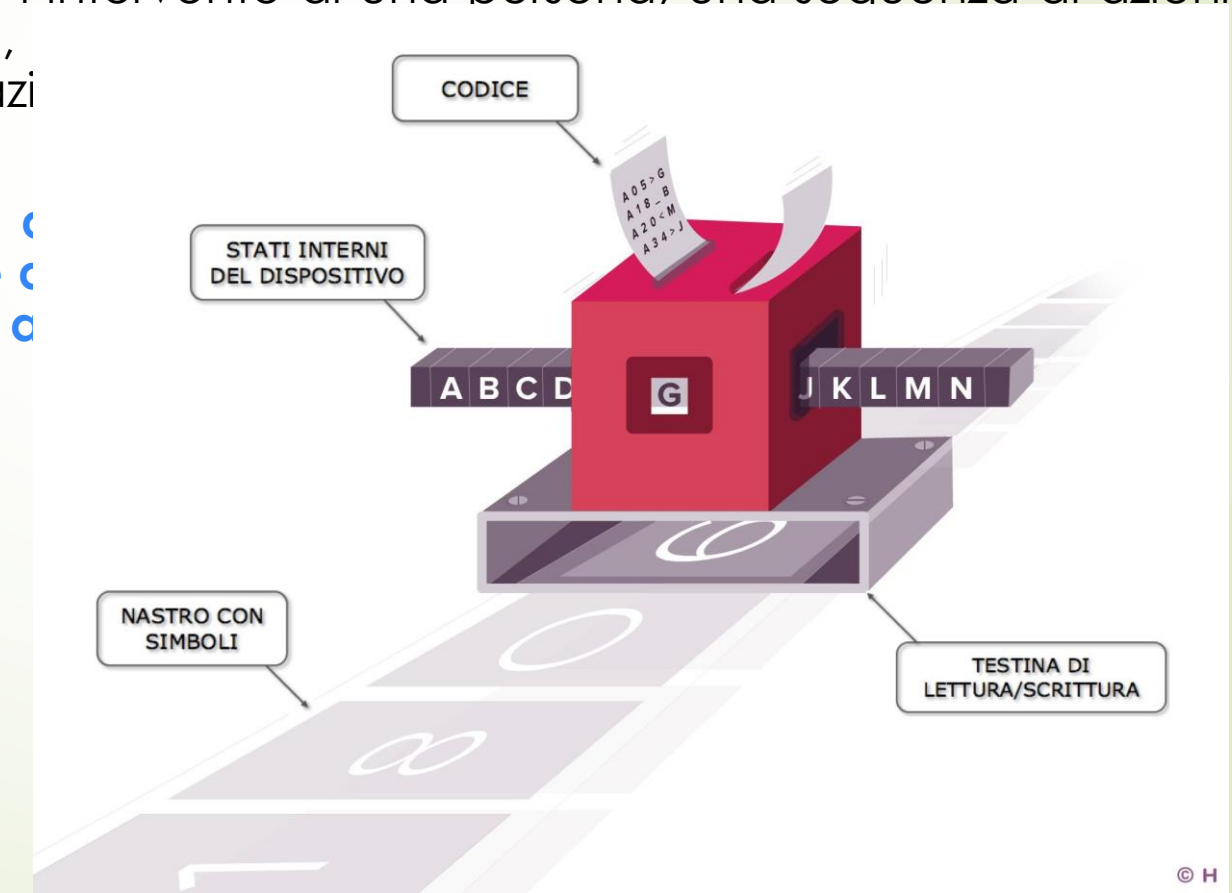
La MdT è un automa costituita da un meccanismo di controllo, da un nastro di lunghezza infinita nei due sensi e da una testina di lettura/scrittura che può effettuare due tipi di movimento a sinistra o a destra.

Automa

Abbiamo precedentemente affermato che un «esecutore» o «elaboratore elettronico» deve eseguire in modo **automatico** l'algoritmo.

Per cui definiamo con il termine «**automa**» un dispositivo in grado di eseguire da solo, cioè in modo automatico, senza l'intervento di una persona, una sequenza di azioni stabilite in precedenza. Inoltre, dall'esterno e produrre informazi assumere stati diversi.

La MdT è un automa costituita lunghezza infinita nei due sensi e due tipi di movimento a sinistra o



Formalizzazione della MdT

MdT = (A, I, S, s0, F, d)

A: alfabeto dei simboli utilizzati dalla MdT (es. «blank» rappresenta la cella vuota)

I: l'insieme dei simboli di input che possono essere letti dal nastro

S: l'insieme degli stati della macchina

S0: lo stato iniziale della macchina

F: l'insieme degli stati finali della macchina (sottoinsieme di S)

d: funzione di transizione che associa ad una coppia (simbolo-input, stato) una terna (stato, simbolo-output, movimento-testina):

$$(i_t, s_{t-1}) \rightarrow (s_t, u_t, m_t)$$

s_t : stato successivo; u_t : simbolo scritto sul nastro; m_t : movimento testina

Viene prodotta una «**matrice di transizione**» che schematizza con una tabella la «**funzione di transizione**».

Formalizzazione della MdT

Matrice delle transizioni.

Stati / Input	s_1	s_2	s_3
i_1	s_{11},u_{11},m_{11}	s_{12},u_{12},m_{12}	s_{13},u_{13},m_{13}
i_2	s_{21},u_{21},m_{21}	s_{22},u_{22},m_{22}	s_{23},u_{23},m_{23}
i_3	s_{31},u_{31},m_{31}	s_{32},u_{32},m_{32}	s_{33},u_{33},m_{33}
i_4	s_{41},u_{41},m_{41}	s_{42},u_{42},m_{42}	s_{43},u_{43},m_{43}

In ogni istante la MdT assume una configurazione esprimibile con la quintupla: simbolo letto, stato, nuovo stato, simbolo scritto e movimento della testina. E' chiamata «**mossa**» della MdT il passaggio da una configurazione all'altra.

Esempio di utilizzo della MdT

Problema: calcolare il quoziente e il resto della divisione di un numero per 2

Definizione formale della MdT per risolvere il «problema»

A = (blank, #, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9) , con # situazione di fine input

I = (#, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

S = (R0, R1, FINE) con R0: lo stato iniziale e FINE: lo stato finale

F = (FINE)

All'inizio, sul nastro si trovano scritte le cifre che compongono il dividendo. Alla fine dell'elaborazione, si trovano le cifre del quoziente e, immediatamente a destra, quella del resto (che sarà 0 oppure 1). La MdT opera la divisione partendo dalle cifre più significative del dividendo (da sinistra a destra) e scrivendo al posto di ogni cifra il quoziente della cifra diviso 2, tenendo conto dell'eventuale resto precedente.

R0 -> stato con resto precedente uguale a zero

R1 -> stato con resto precedente uguale a uno

Esempio di utilizzo della MdT

La funzione di transizione opera in questo modo:

(R0, cifra pari) -> (R0, cifra pari : 2, Destra)

(R1, cifra pari) -> (R0, (cifra pari + 10) : 2, Destra)

(R0, cifra dispari) -> (R1, cifra pari : 2, Destra)

(R1, cifra dispari) -> (R1, (cifra pari + 10) : 2, Destra)

(R1, #) -> (FINE, 1, Destra)

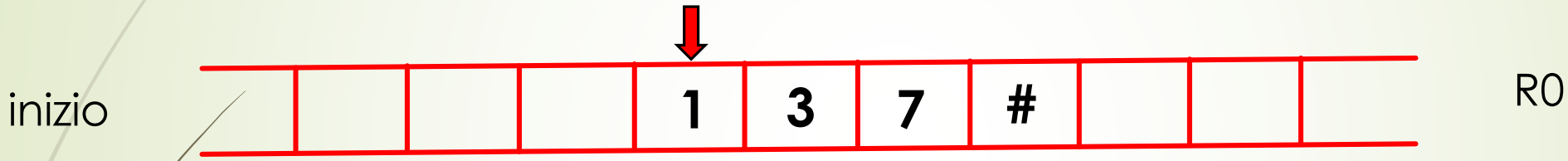
(R0, #) -> (FINE, 0, Destra)

	R0	R1
#	FINE, 0, D	FINE, 1, D
0	R0, 0, D	R0, 5, D
1	R1, 0, D	R1, 5, D
2	R0, 1, D	R0, 6, D
3	R1, 1, D	R1, 6, D
4	R0, 2, D	R0, 7, D
5	R1, 2, D	R1, 7, D
6	R0, 3, D	R0, 8, D
7	R1, 3, D	R1, 8, D
8	R0, 4, D	R0, 9, D
9	R1, 4, D	R1, 9, D

Project Work

Problema: calcolare il quoziente e il resto della divisione per 2 del numero 137

Scandire le operazioni necessarie per la MdT definita precedentemente per questo tipo di problemi.



Project Work - Soluzione

Codifica dell'Informazione (overview)

Rappresentazione delle Informazioni

L'informazione digitale all'interno dell'elaboratore è rappresentata in codifica binaria 0 e 1 (**bit**).

Le cifre binarie all'interno di un calcolatore vengono trattate a gruppi o pacchetti contenenti un numero costante di bit: in particolare, per essere elaborate, le cifre binarie vengono raggruppate in sequenze o stringhe di 8 bit. Una stringa di 8 bit prende il nome di **byte**.

Per il trattamento dei dati, gli elaboratori operano su sequenze composte da un numero fisso di byte. Tali stringhe di byte prendono il nome di **parole**.

Rappresentazione dei numeri

Per i numeri decimali si usano le seguenti rappresentazioni:

Virgola fissa (fixed point)

Es. **1.8** **0.00347** **32.321**

Virgola mobile (floating point)

Es. **5E-3** **10E+3** **2.5E-4**

Quest'ultimo modo di rappresentazione si chiama notazione scientifica o rappresentazione esponenziale. La lettera **E** sta al posto di «10 elevato a» ed è seguita dall'esponente a cui elevare la base 10: la potenza di 10 va poi moltiplicata per il numero (**mantissa**) che precede la lettera E.

Per esempio: **2.5E-4 ➔ 2.5 x 10^{-4} = 0.00025**

Project Work

Fornire una rappresentazione esponenziale dei seguenti numeri:

0.00125

1.5455

77300000

99554433

Rappresentazione dei numeri in Memoria

La rappresentazione interna dei numeri in memoria RAM subisce delle limitazioni dovute alle dimensioni fisiche della cella di memoria.

Con il termine **precisione** della rappresentazione interna dei numeri si indica il numero di byte utilizzati per la rappresentazione dei numeri che può variare per diversi sistemi di elaborazione in commercio.

- **precisione semplice** (o precisione singola): quando i numeri reali sono rappresentati, per esempio, con 4 byte (cioè 32 bit) ,
- **precisione doppia**: quando i numeri reali sono rappresentati, per esempio, con 8 byte (cioè 64 bit).

Rappresentazione Alfanumerica

Le informazioni esprimibili mediante una combinazione di lettere, cifre o caratteri speciali devono avere una corrispondenza binaria affinché un elaboratore riesca a riconoscere e a trattare questo tipo di informazione digitale.

L'associazione di una combinazione binaria del byte ad un determinato simbolo (lettera, cifra o carattere speciale) e chiamata **codifica**.

La prima codifica nella storia dell'informatica si chiama **ASCII** (*American Standard Code for Information Interchange*), proposto dall'ingegnere dell'IBM, Bob Berner nel 1961, e fu poi pubblicato dall'*American National Standards Institute* nel 1963.

Codifica ASCII (estesa)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`	128	80	Ç	160	A0	á	192	C0	└	224	E0	α
1	01	Start of heading	33	21	!	65	41	A	97	61	a	129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
2	02	Start of text	34	22	"	66	42	B	98	62	b	130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
3	03	End of text	35	23	#	67	43	C	99	63	c	131	83	â	163	A3	ú	195	C3	├	227	E3	π
4	04	End of transmit	36	24	$	68	44	D	100	64	d	132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
5	05	Enquiry	37	25	%	69	45	E	101	65	e	133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
6	06	Acknowledge	38	26	&	70	46	F	102	66	f	134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
7	07	Audible bell	39	27	'	71	47	G	103	67	g	135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
8	08	Backspace	40	28	(	72	48	H	104	68	h	136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i	137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j	138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k	139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l	140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m	141	8D	ì	173	AD	¡	205	CD	═	237	ED	ø
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n	142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o	143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
16	10	Data link escape	48	30	0	80	50	P	112	70	p	144	90	É	176	B0	░	208	D0	╨	240	F0	≡
17	11	Device control 1	49	31	1	81	51	Q	113	71	q	145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
18	12	Device control 2	50	32	2	82	52	R	114	72	r	146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
19	13	Device control 3	51	33	3	83	53	S	115	73	s	147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
20	14	Device control 4	52	34	4	84	54	T	116	74	t	148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u	149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v	150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
23	17	End trans. block	55	37	7	87	57	W	119	77	w	151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
24	18	Cancel	56	38	8	88	58	X	120	78	x	152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
25	19	End of medium	57	39	9	89	59	Y	121	79	y	153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z	154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{	155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|	156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}	157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~	158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	⌂	159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	□

La nuova codifica Unicode

Così come avvenne per altri standard dell'informatica (es. TCP/IP) anche la codifica ASCII estesa (8bit) risultò alla fine degli anni '80 non più sufficiente a codificare i simboli di tutte le lingue…

Nel 1991 nacque la codifica **Unicode** (estensione dell'ASCII) che a sua volta ebbe delle evoluzioni: UTF-8 (8bit), UTF-16 (16bit) e UTF-32 (32 bit).

L'obiettivo generale di Unicode è di creare una codifica che comprenda tutti i caratteri, con tutte le variazioni possibili, di tutte le lingue esistenti, oltre ai simboli utilizzati in matematica e nelle scienze.

Sia internamente ad un elaboratore, sia in ambiente distribuito, ogni carattere viene convertito in bit per essere trasmesso. Più bit vengono utilizzati, più caratteri differenti possono essere utilizzati e più lunga sarà la sequenza di bit da trasmettere.

Le tabelle dei codici Unicode sono disponibili sul sito
http://www.unicode.org/charts

Project Work

Prendiamo in considerazione la stringa di testo «**Ciao, mondo!**» contenente 12 caratteri (si considerano anche spazi e punto esclamativo).

Calcolare quanto spazio di memoria occuperebbe la stessa stringa di testo nelle codifiche ASCII standard e UTF-32.

Project Work - soluzione

Project Work - soluzione

Utilizzando le tabelle di codifica UTF disponibili in rete, aprite una pagina vuota di MS-Word e generate i simboli desiderati nel seguente modo:

«codice esadecimale» ALT+X

20AC (ALT+X) => €

0683 (ALT+X) => ڃ

Algebra Booleana (overview)

Algebra Booleana

Nella programmazione si usa molto spesso la logica degli enunciati e i concetti di base dell'**algebra delle proposizioni** che viene anche chiamata algebra booleana dal nome del matematico inglese George Boole (1815-1864). Da qui poi nacque la **Logica Proposizionale** su cui si basano molti linguaggi formali usati nell'Intelligenza Artificiale

[Panti et al., 2001] [Aldewereld et al., 2008] [Morandini et al., 2009]

Un **enunciato** rappresenta una proposizione che può essere vera o falsa, ma non entrambe le cose. La verità o falsità di un enunciato è anche detta **valore di verità**.

Esempi:

«Oggi c'è il sole!», «Domani si parte» => sono enunciati

«Speriamo che sia promosso», «come è andato il viaggio?» => non sono enunciati perché non sono né veri né falsi.

Enunciati Composti

In alcuni casi, gli enunciati possono essere composti, cioè formati da sottoenunciati collegati tra loro da **connettivi logici**.

Esempio:

«Domani si parte oppure si resta a casa» => è un enunciato composto da due sottoenunciati, «Domani si parte» e «Domani si resta a casa», collegati tra loro dal connettivo «oppure»

Per un enunciato composto il valore di verità è definito dai valori di verità dei suoi sottoenunciati e dal connettivo logico che li unisce.

Congiunzione - AND

In alcuni casi, gli enunciati possono essere collegati dal connettivo «e» che in informatica è rappresentato da **AND** (forma inglese) e che viene chiamato **congiunzione**.

Se indichiamo con le lettere **p** e **q** i due enunciati, la tabella o valore di verità relativa alla loro congiunzione (**p AND q**) risulta:

p	q	p AND q
V	V	V
V	F	F
F	V	F
F	F	F

Dove «V» e «F» sono i valori di verità, «Vero» e «Falso».

Disgiunzione - OR

In alcuni casi, gli enunciati possono essere collegati dal connettivo «oppure» che in informatica è rappresentato da **OR** (forma inglese) e che viene chiamato **disgiunzione**.

Se indichiamo con le lettere **p** e **q** i due enunciati, la tabella o valore di verità relativa alla loro disgiunzione (**p OR q**) risulta:

p	q	p OR q
V	V	V
V	F	V
F	V	V
F	F	F

Dove «V» e «F» sono i valori di verità, «Vero» e «Falso».

Disgiunzione Esclusiva - XOR

In alcuni casi, gli enunciati possono essere collegati dal connettivo «o esclusivo» che in informatica è rappresentato da **XOR** (forma inglese) e che viene chiamato **disgiunzione esclusiva**.

Se indichiamo con le lettere **p** e **q** i due enunciati, la tabella o valore di verità relativa alla loro disgiunzione (**p XOR q**) risulta:

p	q	p XOR q
V	V	F
V	F	V
F	V	V
F	F	F

Dove «V» e «F» sono i valori di verità, «Vero» e «Falso».

OR – XOR differenze semantiche

Nella lingua italiana la particella «o» può assumere due significati diversi:

- «p o q o entrambi» => disgiunzione OR
- «p o q ma non entrambi» => disgiunzione esclusiva XOR

Per esempio:

- «ho vinto 100€ oppure ho vinto una bicicletta» => si intende anche che possono essere vere entrambe
- «ora sta piovendo oppure ora non sta piovendo» => si intende anche che non possono essere vere (o false) entrambe

Negazione - NOT

Per cui dato un enunciato **p** è possibile ricavare un altro enunciato dato dalla negazione del primo: **NOT p => negazione** di **p**

Nel linguaggio naturale siamo soliti dire «non è vero che …» oppure semplicemente anteponendo la parola «non» davanti all'enunciato.

p	NOT p
V	F
F	V

Dove «V» e «F» sono i valori di verità, «Vero» e «Falso».

Project Work

Utilizzando le tabelle di verità, rispondere alle seguenti domande:

- Se a = 3 e b = 5 l'espressione: (a < 2) AND (b > 7) produce una proposizione vera o falsa?
- Se a = 6 e b = 15 l'espressione: (a > 2) OR (b > 11) produce una proposizione vera o falsa?
- Se a = 6 e b = 15 l'espressione: (a > 2) XOR (b > 11) produce una proposizione vera o falsa?
- Se a = 6 e b = 15 l'espressione: (a > 2) AND (NOT(b < 11)) produce una proposizione vera o falsa?
- Per quali valori interi di q la seguente espressione è vera
  (q > 10) AND (q < 15)

Project Work - soluzione

Formalizzare la Soluzione del Problema: «linguaggi di progetto»

Linguaggi Formali di Progetto

I metodi formali più utilizzati per definire un progetto per la soluzione algoritmica del problema sono:

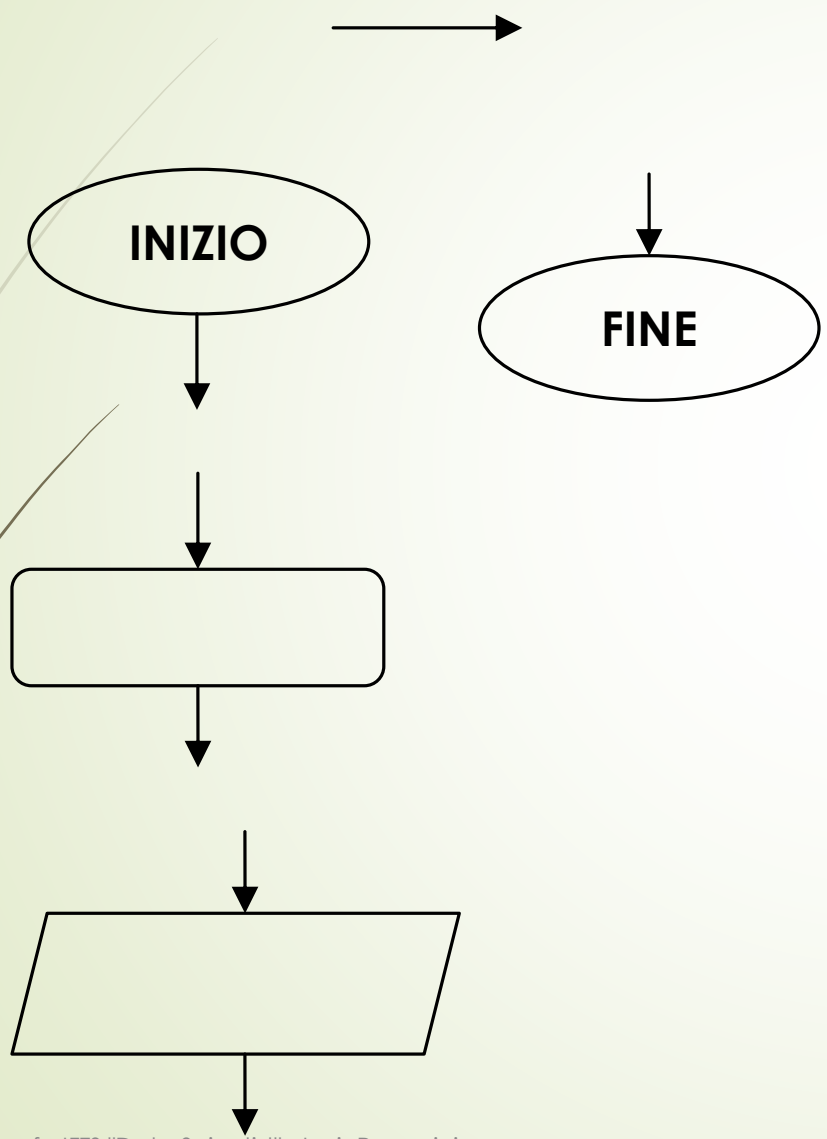
Pseudo-codice

- Vantaggi
  - Più diretto
- Svantaggi
  - Meno astratto
  - Interpretazione più complessa

Flow-Chart

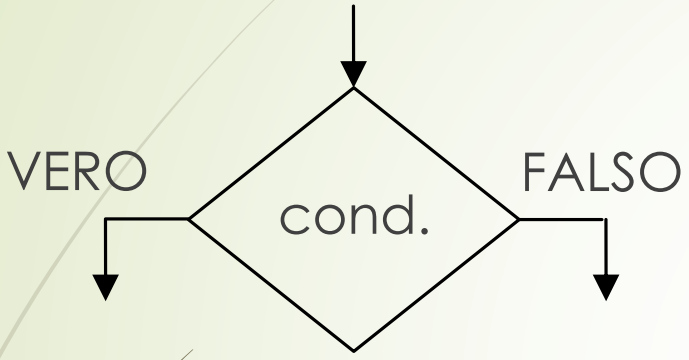
- Vantaggi
  - Più intuitivo perché grafico
  - Più astratto
- Svantaggi
  - Acquisire la semantica dei simboli grafici

Notazione Grafica – Flow Chart

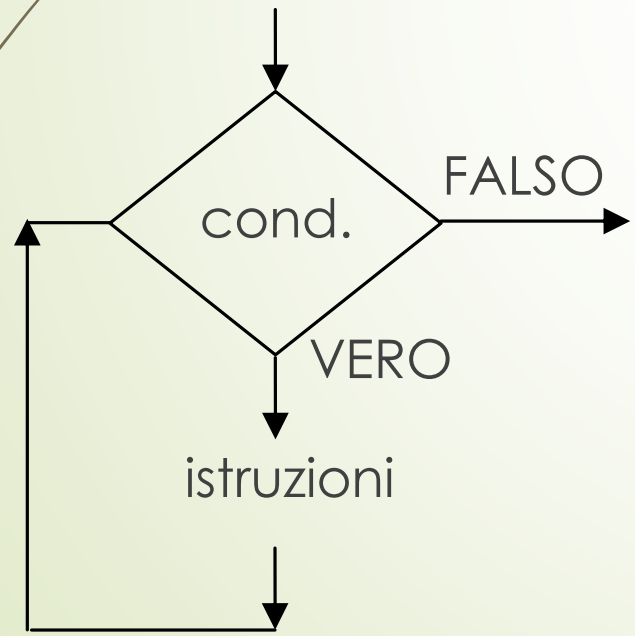


- Rappresenta la direzione del flusso del programma
- Indicano i punti di inizio e di fine dell'Algoritmo
- Indica un'elaborazione, per esempio un assegnazione
- Indica sia un **input** e sia un **output** , si assume per default che l'input dei dati avviene per tastiera e l'output è a video.

Notazione Grafica: semantica decisionale

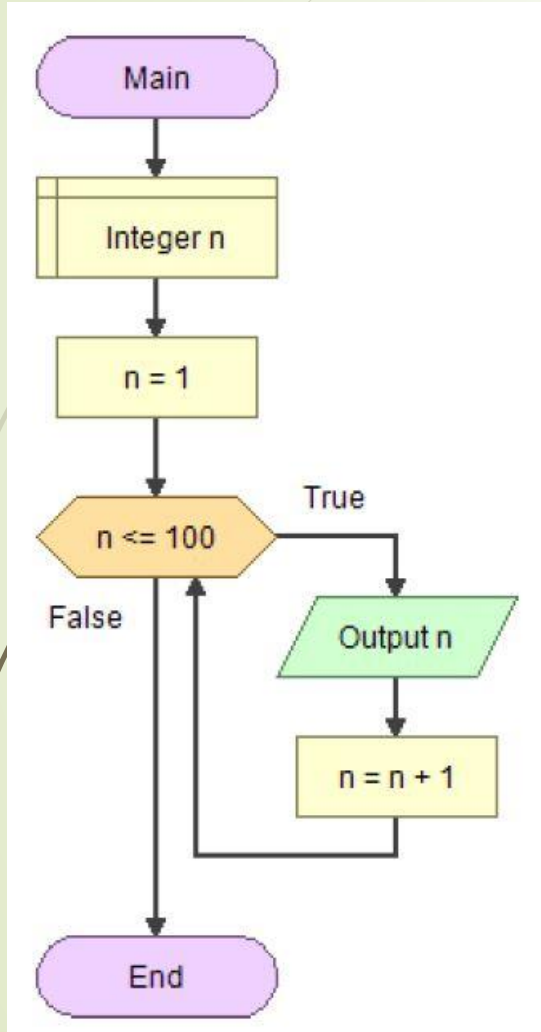


- Simbolo di decisione (**IF**): se la condizione «cond.» assume valore booleano «true» allora il flusso del programma percorre il ramo del VERO, altrimenti si passa sul ramo del FALSO.



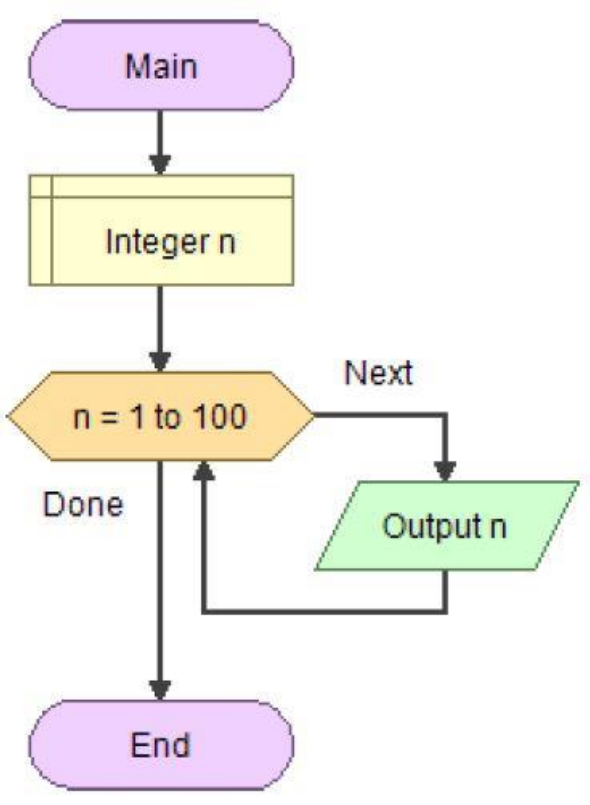
- Indica un ciclo/ripeti (**while**, **for**, …), cioè si ripete il blocco di «istruzioni» fino a quando la «cond.» è VERA. Quando la «cond.» risulta falsa allora si esce dal ciclo, cioè si percorre il ramo del FALSO.

Costrutti di Flowgorithm



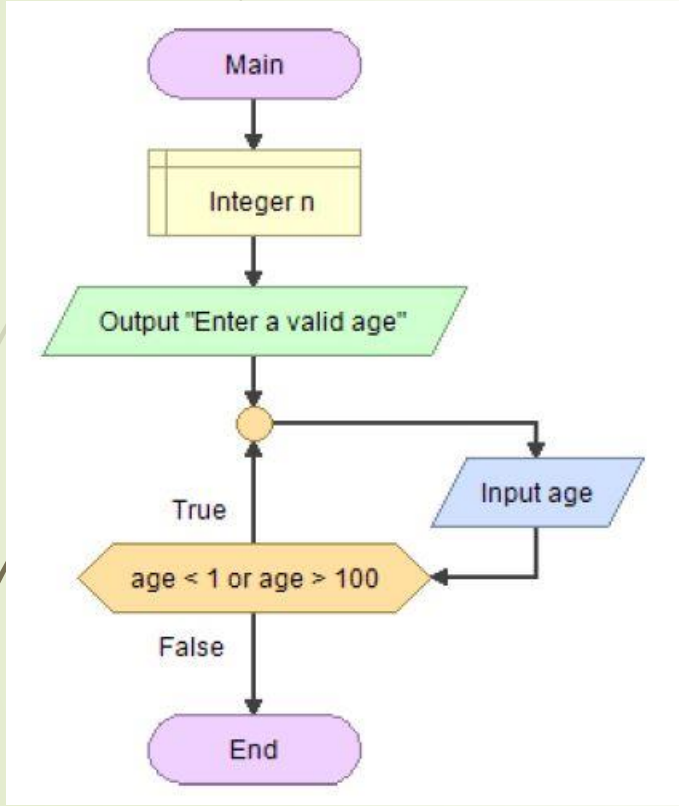
- **Ciclo While**: valuta una espressione booleana e, se vera, esegue le istruzioni contenute al suo interno. Cicla fino a quando la condizione diventa falsa.

Costrutti di Flowgorithm



- **Ciclo for**: incrementa una variabile all'interno di un intervallo di valori assegnati.

Costrutti di Flowgorithm



- Indica un ciclo DO. L'esempio mostra che si accetta solo un input valido, cioè se l'età è compresa tra 1 e 100, altrimenti si ripete il ciclo. Rispetto ad un ciclo «While» le istruzioni al suo interno vengono eseguite almeno una volta, prima di verificare la condizione.

Progettare la soluzione

Problema

Realizzare un algoritmo che riceva in input due numeri interi e determini quale dei due è più grande.

Soluzione informale

Si utilizza un'operazione di confronto tra i due numeri e in base al risultato si stampa in uscita il messaggio per l'utente.

Soluzione formale

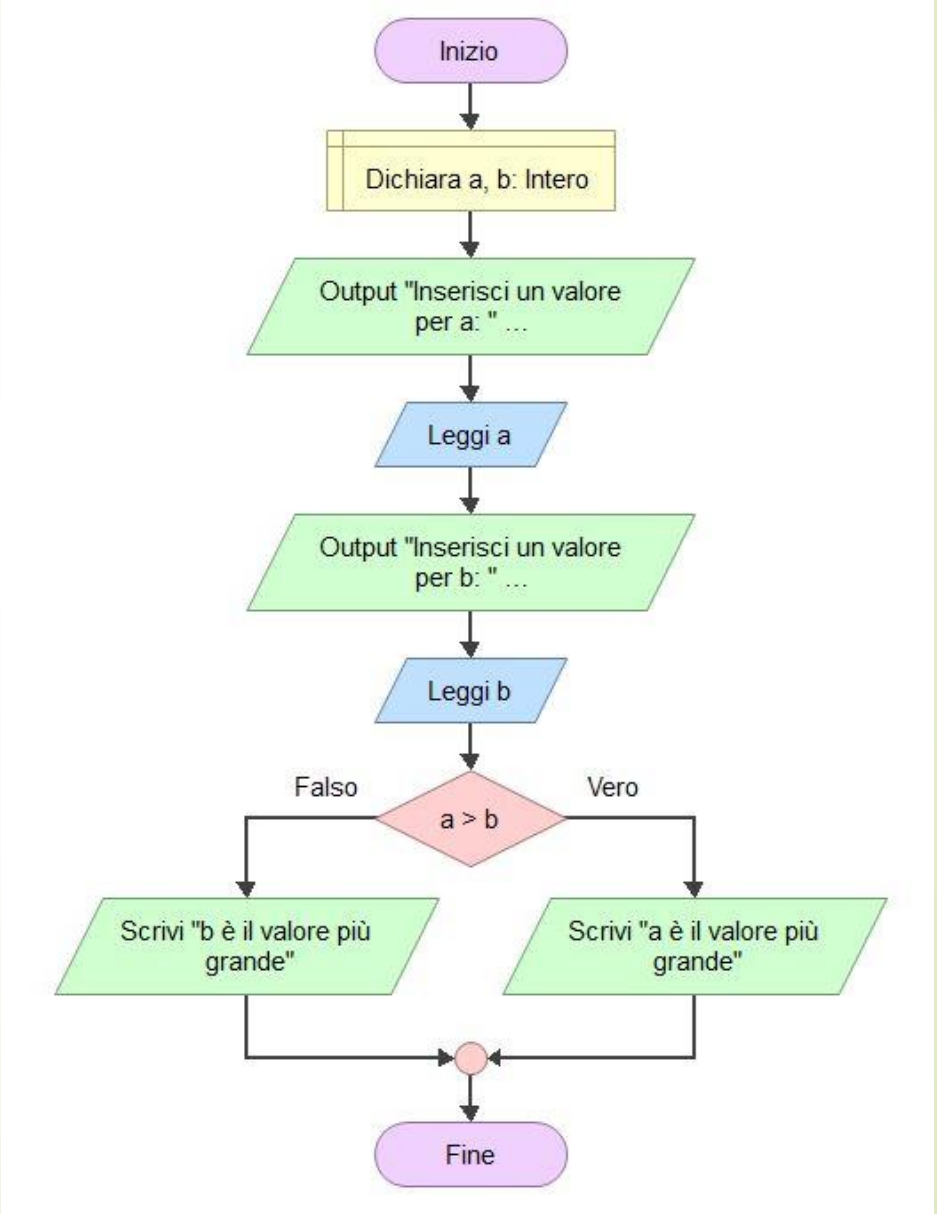
Con linguaggi di progetto: *pseudo-codice* e con *Flow-Chart*

Il progetto

pseudo-codice

```
0   Funzione Inizio
1       Dichiara a, b: Intero
2
3       Scrivi "Inserisci un valore per a: "
4       Leggi a
5       Scrivi "Inserisci un valore per b: "
6       Leggi b
7       Se a > b
8           Scrivi "a è il valore più grande"
9       Altro
10          Scrivi "b è il valore più grande"
11      Fine
12  Fine
```

Flow-Chart



Programmazione Strutturata «strutture di controllo»

Programmazione

La maggior parte dei linguaggi di programmazione si basano sul «**paradigma imperativo**», in cui il programma consiste in un insieme di istruzioni dette anche «**direttive**» o «**comandi**».

La programmazione imperativa viene generalmente contrapposta a quella dichiarativa, in cui un programma consiste in un insieme di «affermazioni» (non «comandi») che l'esecutore è tenuto a considerare vere e/o rendere vere. Un esempio di paradigma dichiarativo è la programmazione logica (es. Prolog, LTL, CTL, ecc.).

Come vedremo in seguito, i concetti introdotti dalla programmazione strutturata sono alla base di numerosi altri linguaggi di programmazione, non ultimo quello orientato agli oggetti.

Programmazione Strutturata

Si fa risalire alla fine degli anni ‘60, quando si decretò (con Edsger Dijkstra) la fine del «salto incondizionato» (*goto*) come strumento fondamentale per la definizione degli algoritmi.

In particolare, con Jacopini e Bohm, si dimostrò che qualsiasi algoritmo può essere espresso con tre tipi di strutture di controllo: sequenza, selezione e ciclo.

La programmazione strutturata è costituita da strutture di controllo e da funzioni (o metodi) che vengono richiamate nell’ordine corretto a partire dalla funzione principale chiamata «**main**».

I linguaggi come il Pascal, C, Fortran, ecc. che permettono una programmazione strutturata, consentono di costruire programmi ordinati, basati sulle strutture di controllo e sull’organizzazione modulare del codice.

Selezione

```
if (condizione)
{
    //istruzioni da eseguire quando la «condizione» è «vera»
}
else
{
    //istruzioni da eseguire quando la «condizione» è «falsa»
}
```

Si possono avere anche blocchi di if annidati utilizzando:

« **if(cond1) … else if(cond2) …** »

Selezione Multipla

```
Switch (espressione)
{
    case valore1:
        //istruzioni da eseguire quando «espressione» = «valore1»
        break;
    case valore2:
        //istruzioni da eseguire quando «espressione» = «valore2»
        break;
    ....
    default: //istruzioni
        break;
}
```

Ripetizione: «While» e «do…While»

```
While (condizione)
{
    //istruzioni da eseguire quando «condizione» è «vera»
}


do
{
    //istruzioni da eseguire quando «condizione» è «vera», la prima
    //volta in ogni caso
} While (condizione)
```

Ripetizione: «for»

```
for (inizializzazione; condizione; aggiornamento)
{
    //istruzioni da eseguire quando «condizione» è «vera»
}
```

«inizializzazione» viene eseguita una sola volta prima di entrare nel ciclo for, mentre la «condizione» viene valutata tutte le volte prima di eseguire il blocco di istruzioni. Se è «falsa» le istruzioni non vengono eseguite. Al termini di ogni ciclo, viene eseguito «aggiornamento» e successivamente viene rivalutata la condizione.

Progettare la soluzione

Problema

Realizzare un algoritmo che riceva in input due numeri interi e determini quale dei due è più grande.

Soluzione informale

Si utilizza un'operazione di confronto tra i due numeri e in base al risultato si stampa in uscita il messaggio per l'utente.

Soluzione formale

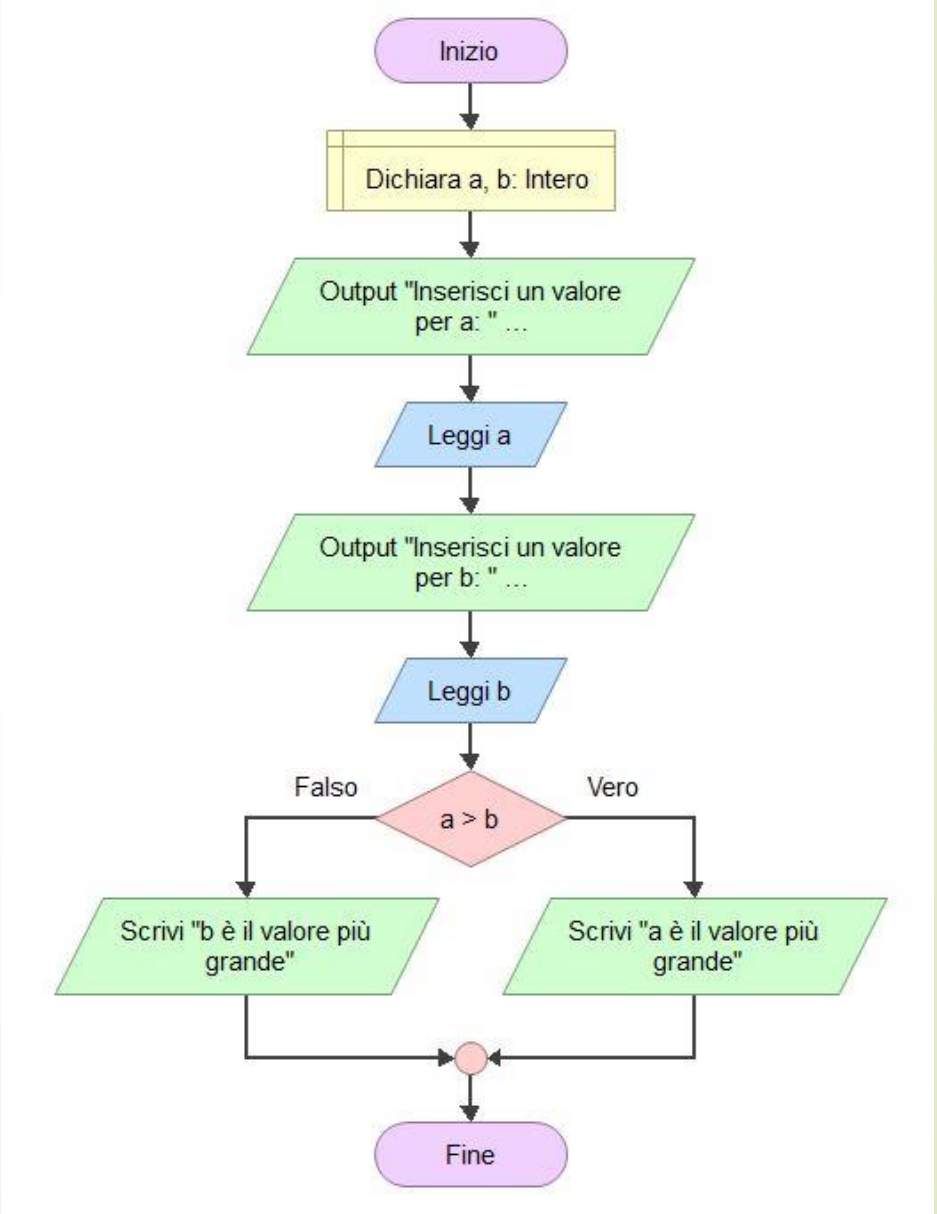
Con linguaggi di progetto: *pseudo-codice* e con *Flow-Chart*

Il progetto

pseudo-codice

```
0   Funzione Inizio
1       Dichiara a, b: Intero
2
3       Scrivi "Inserisci un valore per a: "
4       Leggi a
5       Scrivi "Inserisci un valore per b: "
6       Leggi b
7       Se a > b
8           Scrivi "a è il valore più grande"
9       Altro
10          Scrivi "b è il valore più grande"
11      Fine
12  Fine
```

Flow-Chart



Il Programma

Confronta.java

```java
import java.io.*;

public class Confronta {

    public static void main(String[] args) throws IOException{
        int a,b;
        String str;

        InputStreamReader input = new InputStreamReader(System.in);
        BufferedReader tastiera = new BufferedReader(input);

        System.out.println("Inserisci il primo numero 'a': ");
        str = tastiera.readLine();
        a = Integer.parseInt(str);

        System.out.println("Inserisci il secondo numero 'b': ");
        str = tastiera.readLine();
        b = Integer.parseInt(str);

        if(a > b) {
            System.out.println("Il numero maggiore è 'a' e vale: "+a);
        }else if(a < b || a == b){
            System.out.println("Il numero maggiore è 'b' e vale: "+b);
        }
    }
}
```

Codifica della soluzione nel linguaggio JAVA

Operatori e Variabili

Variabile

Rappresenta un contenitore nel quale parcheggiare dei dati che dovranno essere utilizzati nelle operazioni dell'Algoritmo. Per cui, una variabile è individuata da un identificatore o nome della variabile, e contiene dati che variano durante l'esecuzione dell'Algoritmo.

Su una variabile (var) si possono eseguire queste operazioni:

DICHIARAZIONE: si dichiara il nome «var» e il tipo di dato che contiene, per es. «intero»

ASSEGNAZIONE: si assegna un valore preciso «var = 12»

UTILIZZO: si utilizza la variabile per risolvere espressioni o per stampare in output un risultato: «var2 = var + 5» oppure «stampa var»

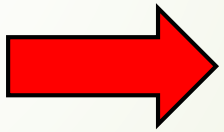
Una variabile a run-time rappresenta una cella di memoria, per cui nella fase di dichiarazione l'esecutore apprende quanto spazio allocare alle variabili a seconda del tipo di dato che contengono.

Campo d'azione

In molti linguaggi strutturati/oggetti, le parentesi graffe individuano i «blocchi di istruzioni», per cui una variabile definita all'interno di un blocco ha validità (o visibilità) solo in quel blocco o nei sotto-blocchi.

Lo spazio di validità (o visibilità) di una variabile all'interno di un programma è anche detto «scope» o «campo d'azione».

```
{
      int a = 1;
      {
            a = a + 5;
            int b = a;
            System.out.println(''b_1 = '' + b);
      }
     a = a  + 1;
      int b = a;
      System.out.println (''b_2 = '' + b);
}
```



b_1 = 6

b_2 = 7

Tipi di Dati Numerici (Java)

I tipi di dati che può contenere una variabile sono in genere simili per tutti i linguaggi, anche se poi ogni linguaggio può avere delle variazioni. Per esempio in Java non esiste il tipo «*unsigned*».

Numeri interi

Tipo	Dimensione	Valori
Byte	8 Bit	Da -128 a 127
Short	16 Bit	Da -32.768 a 32.767
Int	32 Bit	Da -2.147.483.648 a 2.147.483.647
Long	64 Bit	Da - 2^{63} a 2^{63}-1

Numeri in virgola mobile

Tipo	Dimensione	Precisione
Float	32 Bit	Singola
Double	64 Bit	Doppia

Tipi di Dati Testuali e Booleani (Java)

I tipi di dati che può contenere una variabile sono in genere simili per tutti i linguaggi, anche se poi ogni linguaggio può avere delle variazioni.

Dati Testuali

Tipo	Dimensione	Note
Char	16 Bit	Codifica Unicode
String	variabile	Si possono usare sequenze di escape
boolean	---	Valori: true o false

Tipi di Dati in Flowgorithm

I tipi di dati che si possono utilizzare in Flowgorithm (p.35 - manuale):

Integer

Real

String

Boolean

Operatori in Java

Gli operatori matematici e logici per il confronto, possono variare a seconda del tipo di linguaggio.

=	>	<	!	~	?:					
==	<=	>=	!=	&&	||	++	–	–		
+	-	*	/	&	|	^	%	<<	>>	>>>
+=	-=	*=	/=	&=	|=	^=	%=	<<=	>>=	>>>=

Operatori in Flowgorithm

Gli operatori matematici e logici per il confronto, possono variare a seconda del tipo di linguaggio.

Operator	C Family	BASIC Family	Mathematics (Unicode)
Equality	==	=	=
Inequality	!=	<>	≠
Less Than or Equal	<=	<=	≤
Greater Than Or Equal	>=	>=	≥
Logical Not	!	not	¬
Logical And	&&	and	∧
Logical Or	||	or	∨
Multiply	*	*	×
Divide	/	/	÷
Modulo	%	mod	(no symbol)

Realizzare Algoritmi con procedimento iterativo

Progettare la soluzione

Problema (pari/dispari)

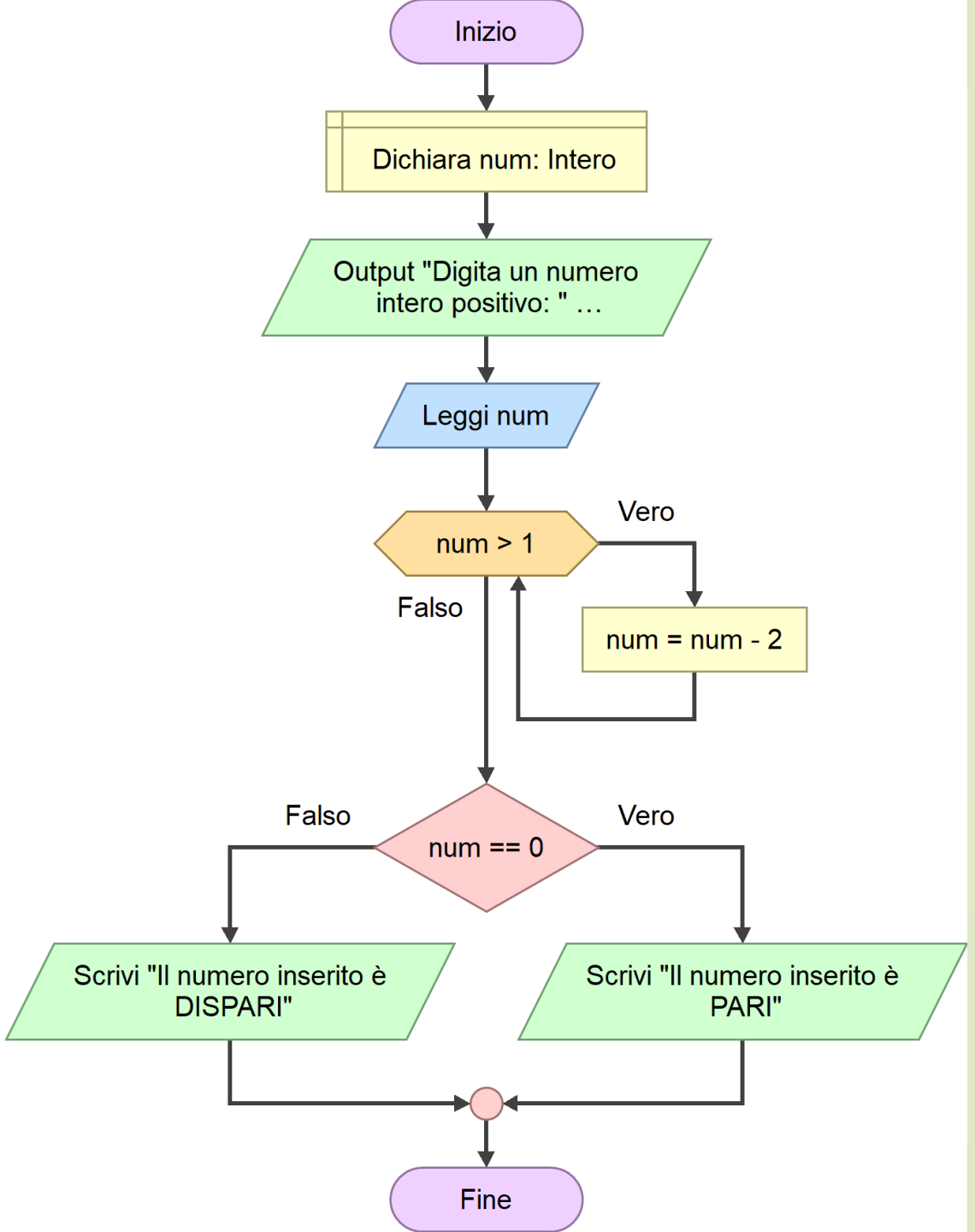
Realizzare un algoritmo che dato un numero intero positivo verifica se è dispari o pari e ne comunichi il risultato.

Idea (informale)

Un numero pari è divisibile per 2, per cui tolgo 2 al numero inserito tante volte fino a quando ho resto 1 o 0. Se il resto è 0 allora il numero iniziale inserito sarà un numero pari e viceversa.

Progetto della soluzion

```
0   Funzione Inizio
1       Dichiara num: Intero
2
3       Scrivi "Digita un numero intero positivo: "
4       Leggi num
5       Mentre num > 1
6           Assegna num = num - 2
7       Fine
8       Se num == 0
9           Scrivi "Il numero inserito è PARI"
10      Altro
11          Scrivi "Il numero inserito è DISPARI"
12      Fine
13  Fine
```



Progetto della soluzione: Java

PariDispari.java

```java
import java.io.BufferedReader;

public class PariDispari {

    public static void main(String[] args) throws IOException{
        int num = 0;
        String str;

        InputStreamReader input = new InputStreamReader(System.in);
        BufferedReader tastiera = new BufferedReader(input);

        System.out.print("Digita un numero intero positivo: ");
        System.out.print("");
        str = tastiera.readLine();
        num = Integer.parseInt(str);

        while(num > 1) {
            num = num - 2;
        }
        if(num == 0) {
            System.out.println("Il numero inserito è PARI");
        } else {
            System.out.println("Il numero inserito è DISPARI");
        }
    }
}
```

Progettare la soluzione

Problema (min)

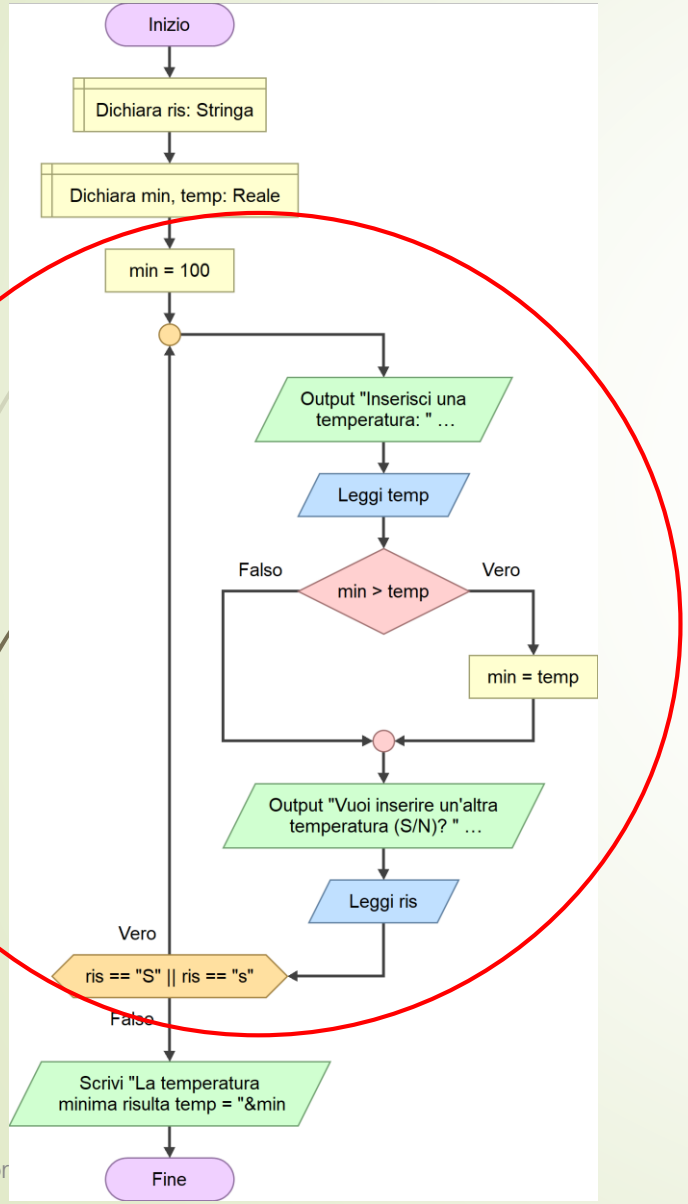
Realizzare un algoritmo che riceva in input una serie di temperature e produca in output la temperatura minima.

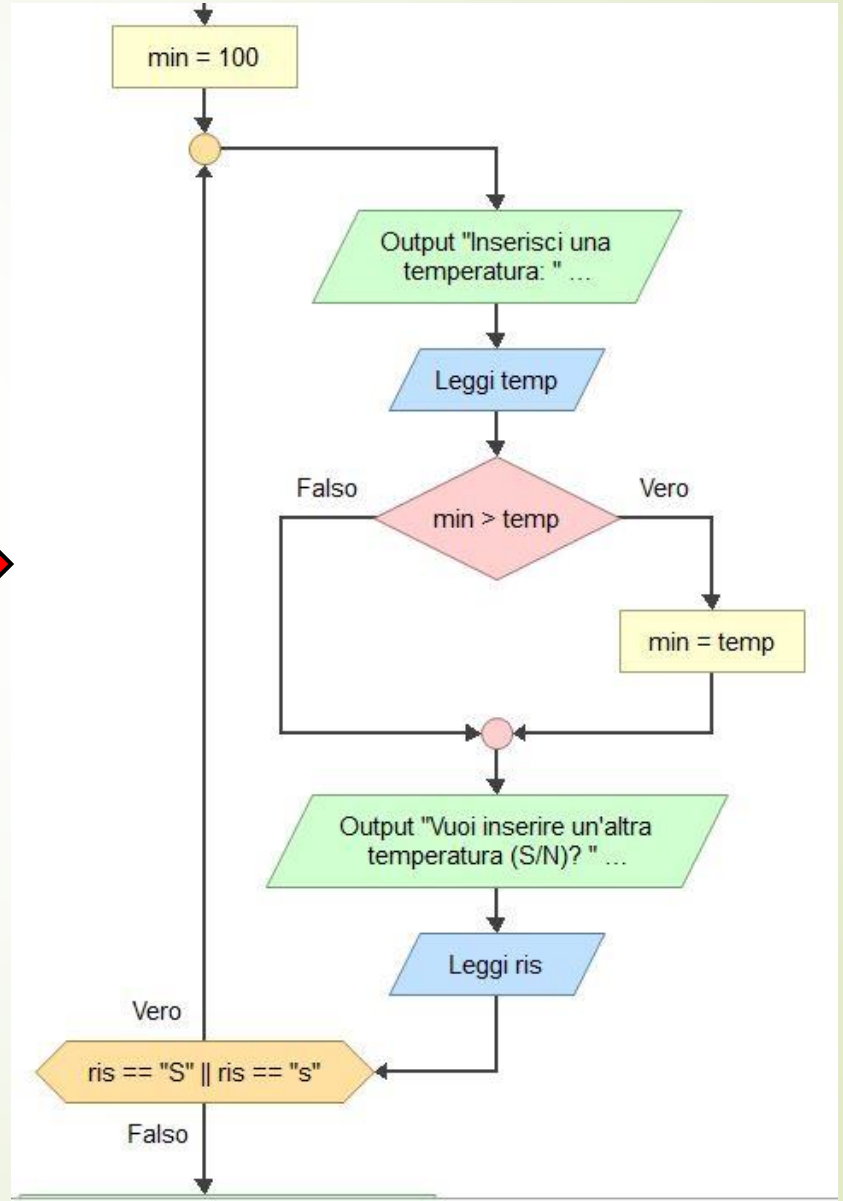
Idea (informale)

Realizzo l'input dentro un ciclo e, ad ogni inserimento, testo con una domanda per l'utente se vuole continuare ad inserire altre temperature.

Inizializzo una variabile «min» ad un valore iniziale alto, così da non perturbare la ricerca del minimo sull'insieme dato. Così «min» servirà da accumulatore del minimo relativo, operando un confronto tutte le volte che l'utente inserisce una temperatura. Al termine del ciclo, o fase di inserimento, «min» conterrà il minimo assoluto.

Progetto della soluzione: Flow-Chart





Progetto della soluzione: pseudocodice

```
0   Funzione Inizio
1       Dichiara ris: Stringa
2       Dichiara min, temp: Reale
3
4       Assegna min = 100
5       Ciclo
6           Scrivi "Inserisci una temperatura: "
7           Leggi temp
8           Se min > temp
9               Assegna min = temp
10          Fine
11          Scrivi "Vuoi inserire un'altra temperatura (S/N)? "
12          Leggi ris
13      Fai ris == "S" || ris == "s"
14      Scrivi "La temperatura minima risulta temp = "&min
15  Fine
```

Progetto della soluzione: Java

Min.java

```
1  import java.io.*;
2
3  public class Min {
4
5      public static void main(String[] args) throws IOException{
6          double min = 100.00;
7          double temp;
8          String str;
9          String ris = "S";
10
11         InputStreamReader input = new InputStreamReader(System.in);
12         BufferedReader tastiera = new BufferedReader(input);
13
14         do {
15             System.out.print("Inserisci una temperatura: ");
16             str = tastiera.readLine();
17             temp = Double.parseDouble(str);
18             if(min > temp) {
19                 min = temp;
20             }
21             System.out.println("Vuoi inserire un'altra temperatura (S/N)? ");
22             ris = tastiera.readLine();
23
24         } while(ris.equals("S") || ris.equals("s"));
25
26         System.out.println("La temperatura minima risulta temp = " + min);
27     }
28 }
```

Project Work

Problema (min e max)

Realizzare un algoritmo che riceva in input una serie di temperature e produca in output la temperatura minima e quella massima.

L'utente conosce il numero (ntemp) dei valori di temperature da inserire.

Project Work - Soluzione

Problema (min e max)

Realizzare un algoritmo che riceva in input una serie di temperature e produca in output la temperatura minima e quella massima.

L'utente conosce il numero (ntemp) dei valori di temperature da inserire.

Idea (informale)

L'input «ntemp» determina il numero dei cicli. Utilizzo il primo inserimento di «temp» per inizializzare sia «min» che «max», che serviranno da accumulatori del minimo e del massimo relativi, operando un confronto tutte le volte che l'utente inserisce una temperatura.

Al termine del ciclo, o fase di inserimento, «min» e «max» conterranno il minimo e il massimo assoluti.

Project Work - Soluzione

Project Work - Soluzione

Project Work

Problema (potenza)

Realizzare un algoritmo che riceva in input un numero reale, la «base», e un numero intero positivo, l' «esponente», e calcoli e stampi l'elevamento a potenza. Ovviamente senza utilizzare eventuali operatori già preconfezionati del linguaggio adottato. Per esempio:

base = 3

esponente = 2 => $3^2 = 9$

Project Work - Soluzione

Project Work - Soluzione

Project Work - Soluzione

Algoritmi con «Funzioni» (overview)

Concetto di «Funzione»

Nell'ingegneria del software, concetti come: usabilità, modularità, manutenibilità, affidabilità, testabilità, portabilità, ecc. sono chiamati «requisiti non funzionali» del sistema e ne determinano gli «obiettivi di qualità» (es. «soft-goal» in Tropos, [Penserini et al., 2007a]).

Nella programmazione, che deve considerare/implementare i requisiti del sistema, si adottano «convenzioni» per meglio raggiungere anche questi obiettivi di qualità.

Una di queste convenzioni, molto usata nella programmazione «**top-down**», è quella di procedere per astrazioni successive:

- Prima si realizzano i moduli fondamentali del sistema, trascurando i dettagli
- Poi si procede a realizzare i moduli con i dettagli

Ogni modulo svolge una specifica funzione all'interno del problema, cioè il programma deve essere scomposto in moduli funzionalmente indipendenti che vengono chiamati «**Funzioni**», «**metodi**», o «**procedure**» a seconda del linguaggio usato.

Passaggio di parametri tra «Funzioni»

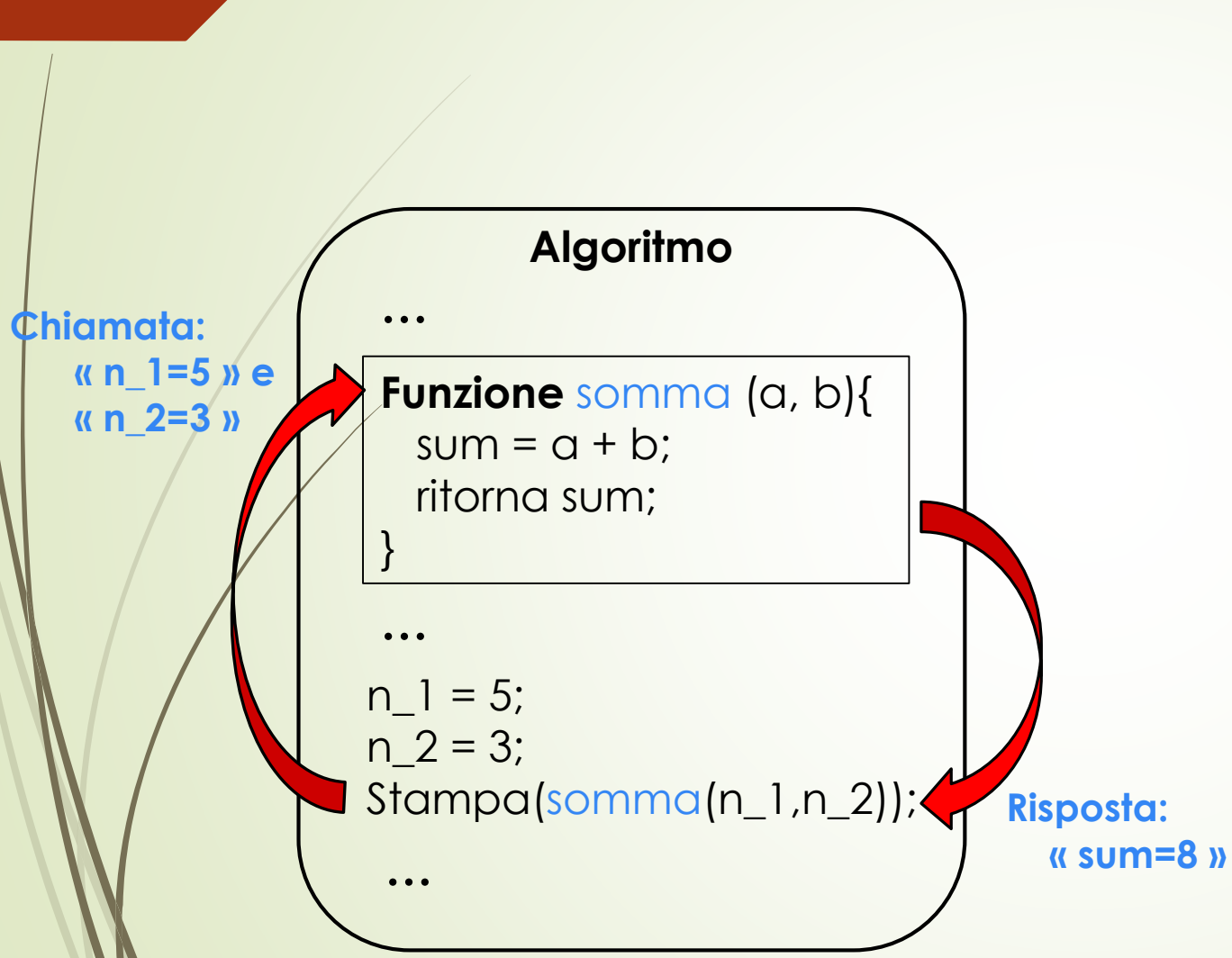
Lo stile di usare un codice modulare, premia sicuramente l'usabilità e la manutenzione del software, per cui è importante poter riutilizzare sottoprogrammi provenienti da altri progetti. Per questo motivo, **le funzioni/sottoprogrammi consentono il passaggio di parametri «da» e «verso» il modulo chiamante.**

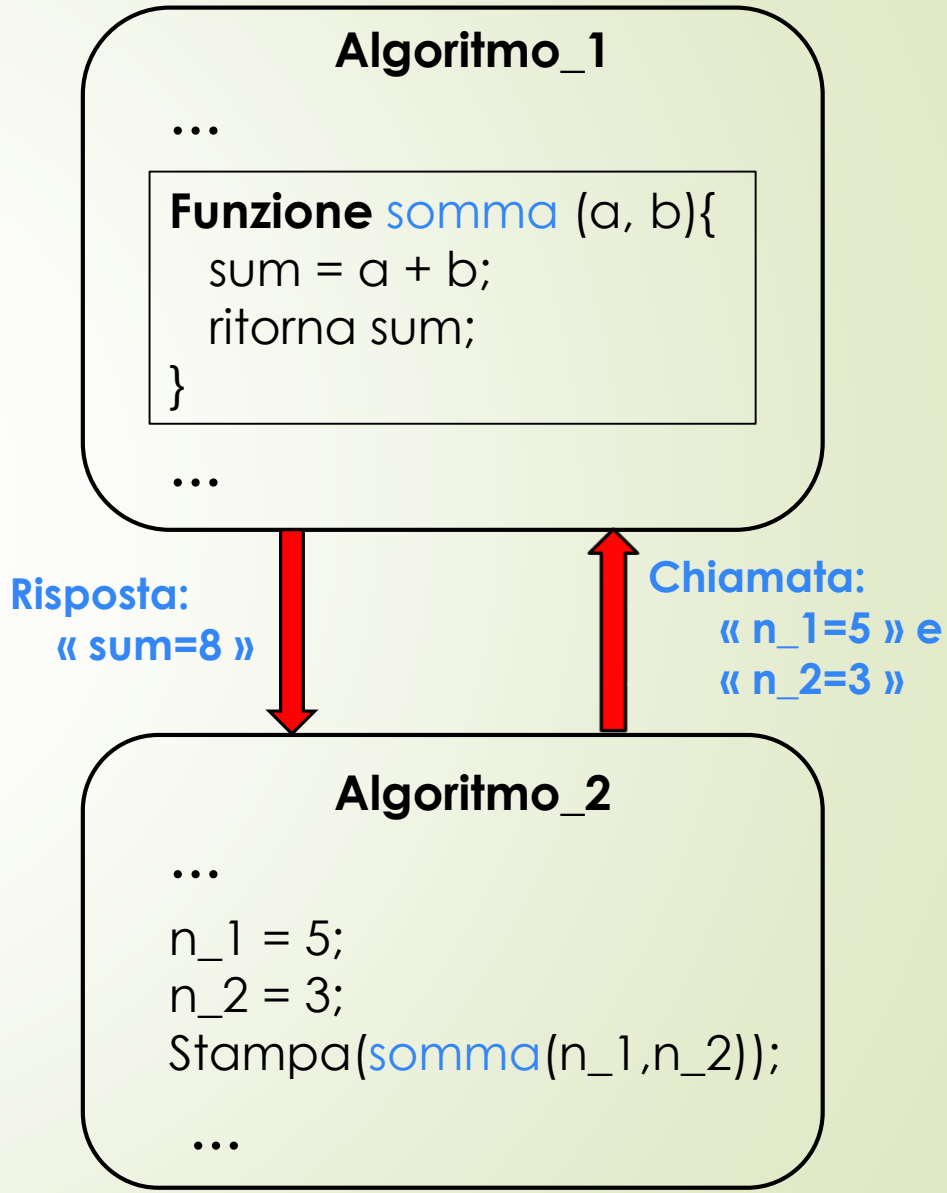
Il chiamante può essere lo stesso modulo che richiama una sua funzione interna, ma si possono avere chiamate a funzioni di librerie/moduli esterni.

Nella OOP questa tecnica avviene in modo chiaro a diversi livelli:

- di classe: cioè metodi/funzioni dello stesso modulo/classe
- di oggetto: cioè metodi/funzioni dello stesso gruppo di moduli/classi che costituiscono il programma/oggetto
- di package: cioè metodi/funzioni esterne, cioè scritte da terze parti

Passaggio di parametri tra «Funzioni»





Utilizziamo le Funzioni

Problema (potenza)

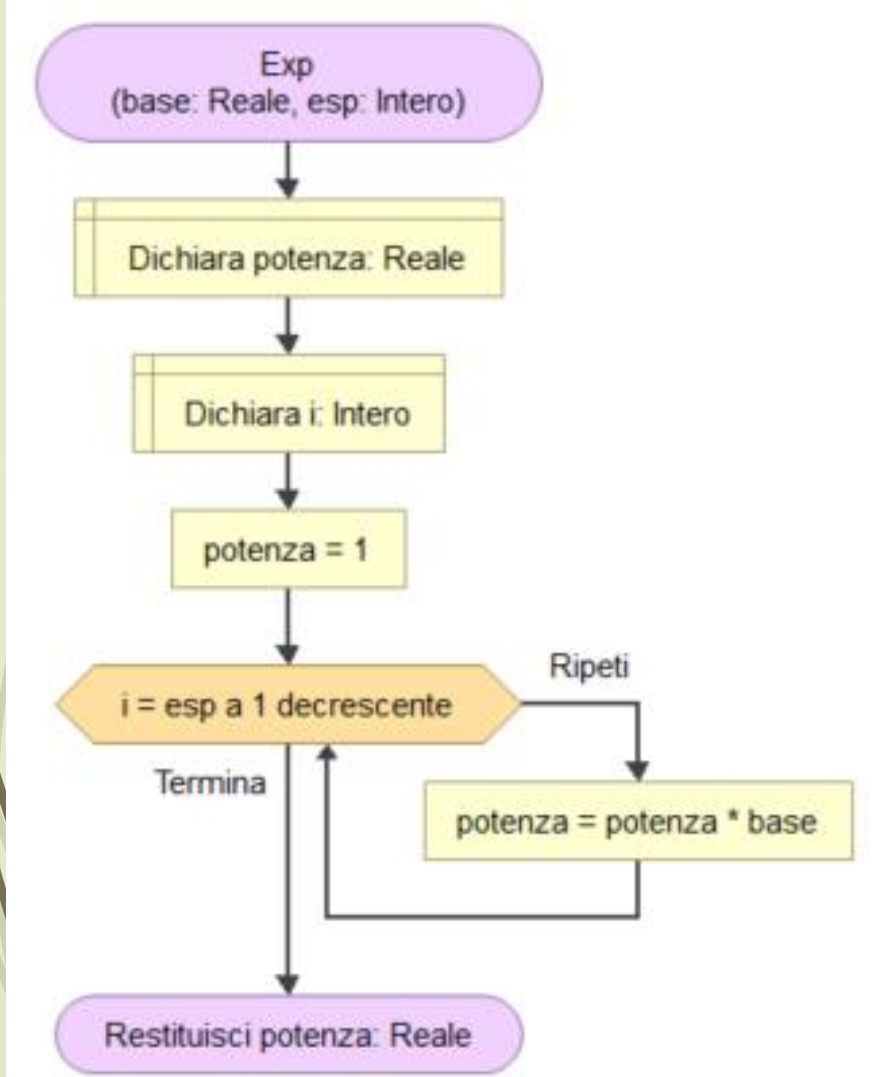
Realizzare un algoritmo che riceva in input un numero reale, la «base», e un numero intero positivo, l' «esponente», e calcoli e stampi l'elevamento a potenza. Ovviamente senza utilizzare eventuali operatori già preconfezionati del linguaggio adottato. Per esempio:

base = 3

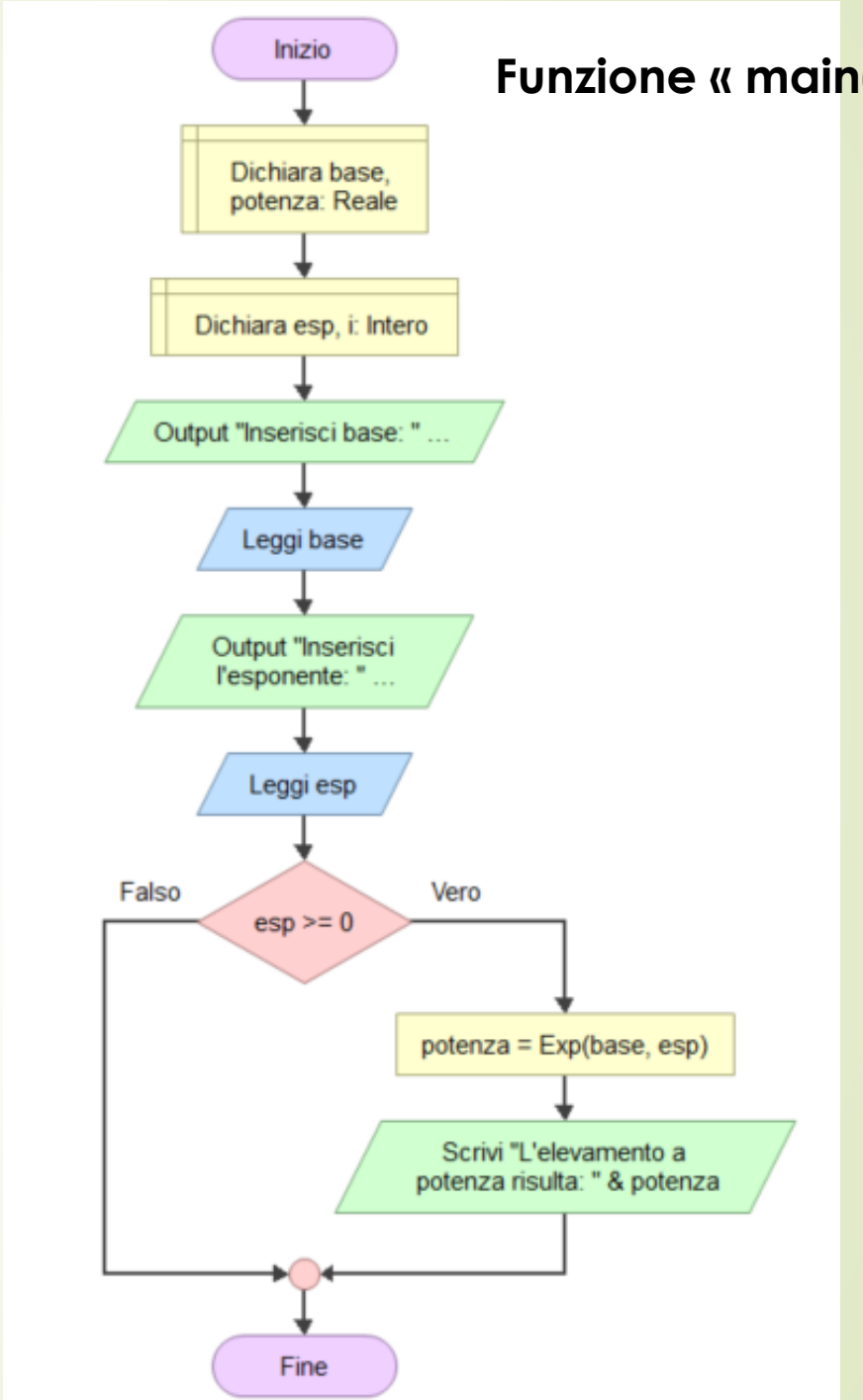
esponente = 2 **=>** **$3^2 = 9$**

Ex. Potenza- Soluzione

Sottoprogramma: « Exp(…) »



Funzione « main(…) »



Ex. Potenza- Soluzione

Sottoprogramma: « Exp(…) »

```
14  Funzione Exp (base: Reale, esp: Intero)
15      Dichiara potenza: Reale
16      Dichiara i: Intero
17
18      Assegna potenza = 1
19      Per i = esp a 1 decrescente
20          Assegna potenza = potenza * base
21      Fine
22  Restituisci potenza: Reale
```

Funzione « main(…) »

```
0   Funzione Inizio
1       Dichiara base, potenza: Reale
2       Dichiara esp, i: Intero
3
4       Scrivi "Inserisci base: "
5       Leggi base
6       Scrivi "Inserisci l'esponente: "
7       Leggi esp
8       Se esp >= 0
9           Assegna potenza = Exp(base, esp)
10          Scrivi "L'elevamento a potenza risulta: " & potenza
11      Fine
12  Fine
```

Ex. Potenza

Frammento di codice della soluzione in Java.

```java
public class Power {

    public Power() throws IOException{
        double base = 0;
        int esp;
        String str;

        InputStreamReader input = new InputStreamReader(System.in);
        BufferedReader tastiera = new BufferedReader(input);

        System.out.print("Inserisci la 'base': ");
        str = tastiera.readLine();
        base = Double.parseDouble(str);

        System.out.print("Inserisci l' 'esponente': ");
        str = tastiera.readLine();
        esp = Integer.parseInt(str);
        System.out.print("");

        if(esp >= 0) {
            System.out.print("L'elevamento a potenza risulta: " + Exp(base, esp));
        }
    }

    public double Exp(double base, int esp) {
        double potenza = 1;
        int i;

        for(i = esp; i > 0; i--) {
            potenza = potenza * base;
        }
        return potenza;
    }

```

La logica della Ricorsione (overview)

La Ricorsione

Tecnicamente una funzione è ricorsiva quando per calcolare la soluzione richiama se stessa.

Lo scopo della ricorsione è quello di risolvere un problema facendo riferimento allo stesso problema su scala ridotta. La condizione di terminazione si ottiene quando si identifica uno più casi semplici con soluzione immediata. Per cui in generale la ricorsione ha la seguente logica:

```
if (raggiunto il caso semplice?)
    restituisci la soluzione;
else
    usa la ricorsione su dati ridotti;
```

Applicazioni della Ricorsione

La ricorsione è spesso utilizzata per risolvere problemi concettualmente complessi, specialmente in combinazione con la OOP e la AOP. Per esempio, in tutte quelle applicazioni di AI (e non) dove si usano strutture dati ad Albero e a Grafo, le operazioni di lettura/scrittura/modifica seguono spesso approcci ricorsivi, poiché sono più intuitivi e chiari da comprendere.

Tuttavia, l'approccio ricorsivo spesso necessita di molta memoria per allocare metodi/funzioni che si richiamano, mentre nell'approccio iterativo che usa solo strutture di controllo (if, for, while, ecc.) si riescono ad ottimizzare meglio gli algoritmi in termini di velocità di risposta.

Esempio

Problema (fattoriale)

Realizzare un algoritmo che riceva in input un numero intero positivo, e calcoli e stampi il relativo fattoriale. Per esempio:

Il Fattoriale del numero 3, e si indica così « 3! » è uguale a moltiplicare tutti i numeri naturali a partire dal numero dato, come segue:

3! = 3 * 2 * 1 = 6

5! = 5 * 4 * 3 * 2 * 1 = 120

Logica Iterativa

Fatt.java

```java
import java.io.*;

public class Fatt {

    public Fatt() throws IOException{
        int n,i;
        int fat=1;

        System.out.print("Calcola il FATTORIALE di: ");
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        String line = input.readLine();
        n = Integer.parseInt(line);
        i = n;

        System.out.print(i + "! = " + i);
        while(i>0){
            fat = fat*i;
            i--;
            if(i!=0)
                System.out.print(" * " + i);
        }
        System.out.println();
        System.out.print("Il FATTORIALE di " + n + " è: " + fat);
    }

    public static void main(String[] args) throws IOException{
        new Fatt();
    }

}
```

Logica Ricorsiva

FattR.java

```java
import java.io.*;

public class FattR {

    public FattR() throws IOException{
        int n;

        System.out.print("Calcola il FATTORIALE di: ");
        BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
        String line = input.readLine();
        n = Integer.parseInt(line);

        if(n == 0)
            System.out.print("Il FATTORIALE di zero: 0! = 1");
        else if(n == 1)
            System.out.print("Il FATTORIALE di uno: 1! = 1");
        else if(n > 1) {
            System.out.print(n + "! = ");
            System.out.print(" = " + Fat(n));
        }
    }

    public int Fat(int n) {
        if(n == 1) {
            System.out.print(n);
            return 1;
        }else {
            System.out.print(n + " * ");
            return n * Fat(n-1);
        }
    }

    public static void main(String[] args) throws IOException{
        new FattR();
    }
}
```

Spunti Bibliografici dell'Autore

[Morandini et al., 2017] Mirko Morandini, Loris Penserini, Anna Perini, Alessandro Marchetto: Engineering requirements for adaptive systems. Requirements Engineering Journal, 22(1): 77-103 (2017)

[Morandini et al., 2009] Morandini M., Penserini L., and Perini A. (2009b). Operational Semantics of Goal Models in Adaptive Agents. In 8th Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS'09). IFAAMAS.

[Morandini et al., 2008] Morandini, M., Penserini, L., and Perini, A. (2008b). Automated mapping from goal models to self-adaptive systems. In Demo session at the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), pages 485–486.

[Aldewereld et al., 2008] Huib Aldewereld, Frank Dignum, Loris Penserini, Virginia Dignum: Norm Dynamics in Adaptive Organisations. NORMAS 2008: 1-15

[Penserini et al., 2007a] Loris Penserini, Anna Perini, Angelo Susi, John Mylopoulos: High variability design for software agents: Extending Tropos. ACM Trans. Auton. Adapt. Syst. 2(4): 16 (2007)

[Penserini et al., 2007b] Loris Penserini, Anna Perini, Angelo Susi, Mirko Morandini, John Mylopoulos: A design framework for generating BDI-agents from goal models. AAMAS 2007: 149

[Pagliarecci et al., 2007] Francesco Pagliarecci, Loris Penserini, Luca Spalazzi: From a Goal-Oriented Methodology to a BDI Agent Language: The Case of Tropos and Alan. OTM Workshops (1) 2007: 105-114

[Penserini et al., 2006a] Loris Penserini, Anna Perini, Angelo Susi, John Mylopoulos: From Stakeholder Intentions to Software Agent Implementations. CAiSE 2006: 465-479

[Penserini et al., 2006b] Loris Penserini, Anna Perini, Angelo Susi, John Mylopoulos: From Capability Specifications to Code for Multi-Agent Software. ASE 2006: 253-256

[Panti et al., 2003] Maurizio Panti, Loris Penserini, Luca Spalazzi: A critical discussion about an agent platform based on FIPA specification. SEBD 2000: 345-356

[Panti et al., 2001] Maurizio Panti, Luca Spalazzi, Loris Penserini: A Distributed Case-Based Query Rewriting. IJCAI 2001: 1005-1010

GRAZIE!