

ALGORITMI E MACCHINA DI VON NEUMANN

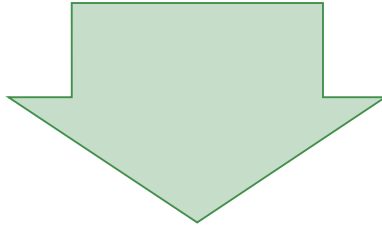
- **Il concetto di Algoritmo**
- **La Macchina di Von Neumann**

Risoluzione del Problema

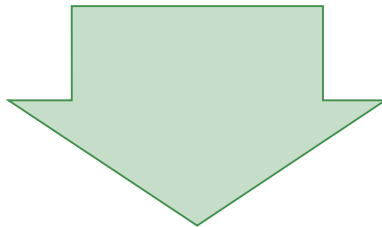
- **Precisa formulazione del problema**
- **Ricerca dell'**Algoritmo**, cioè di**

**un procedimento non ambiguo,
formato da
numero finito di azioni
sufficientemente semplici
che risolva il problema dato**

Dati del Problema



ALGORITMO



RISULTATO

Esempio :

sostituzione della ruota di un'auto

istruzioni:

solleva l'auto
svita i bulloni
togli la ruota
metti la ruota di scorta
avvita i bulloni
abbassa l'auto

Un algoritmo viene descritto mediante un

LINGUAGGIO

- Flow chart (diagramma di flusso)
- Tipo Pascal, tipo Matlab, ...
-
-

Convenzioni linguistiche del

FLOW CHART

inizio

denota l'inizio dell'algoritmo

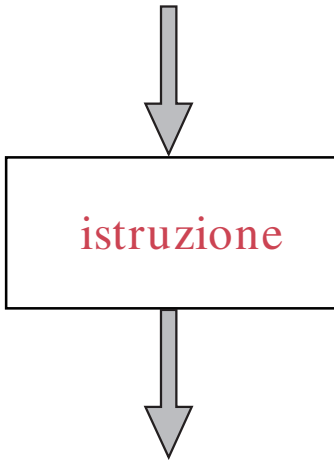


fine

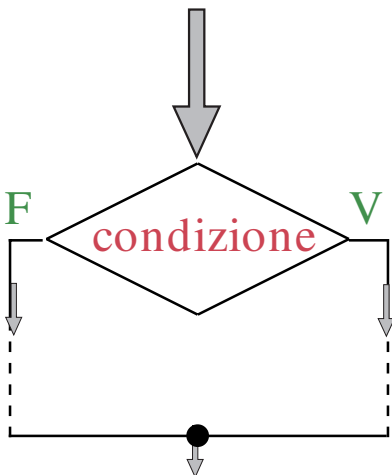
denota la fine dell'algoritmo

Convenzioni linguistiche del

FLOW CHART

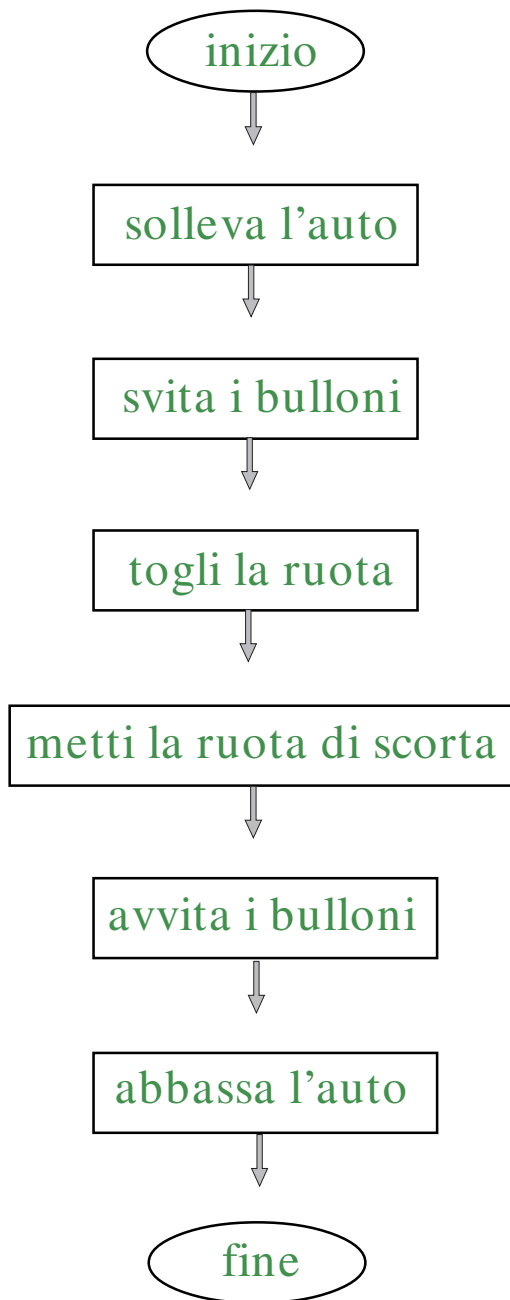


denota una istruzione



denota che l'esecuzione
dipende dal valore della
condizione (vero o falso)

Flow chart dell'algoritmo per il cambio della ruota



La **non ambiguità** di una istruzione dipende dalle

*CARATTERISTICHE
DELL'ESECUTORE*

l'istruzione

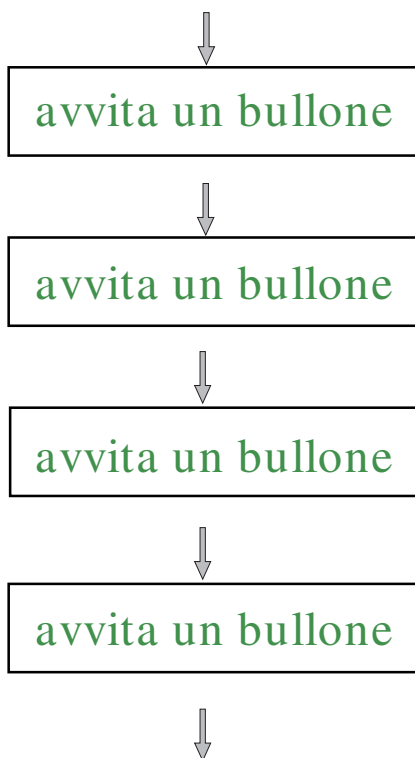
avvita i bulloni

è sufficientemente specificata?

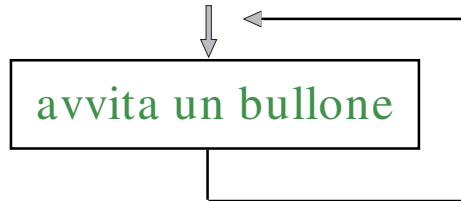
l'istruzione

avvita i bulloni

può essere sostituita dalla sequenza
di istruzioni più semplici

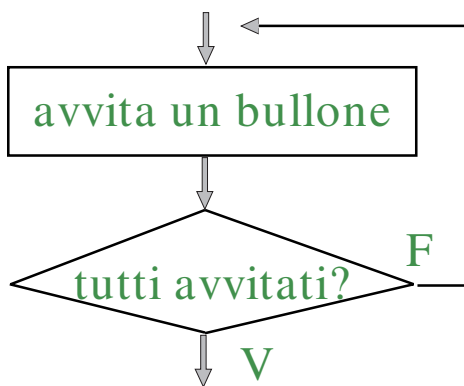


Tale sequenza si può rappresentare
con la struttura di tipo:



ma in questa forma
il ciclo è senza fine!

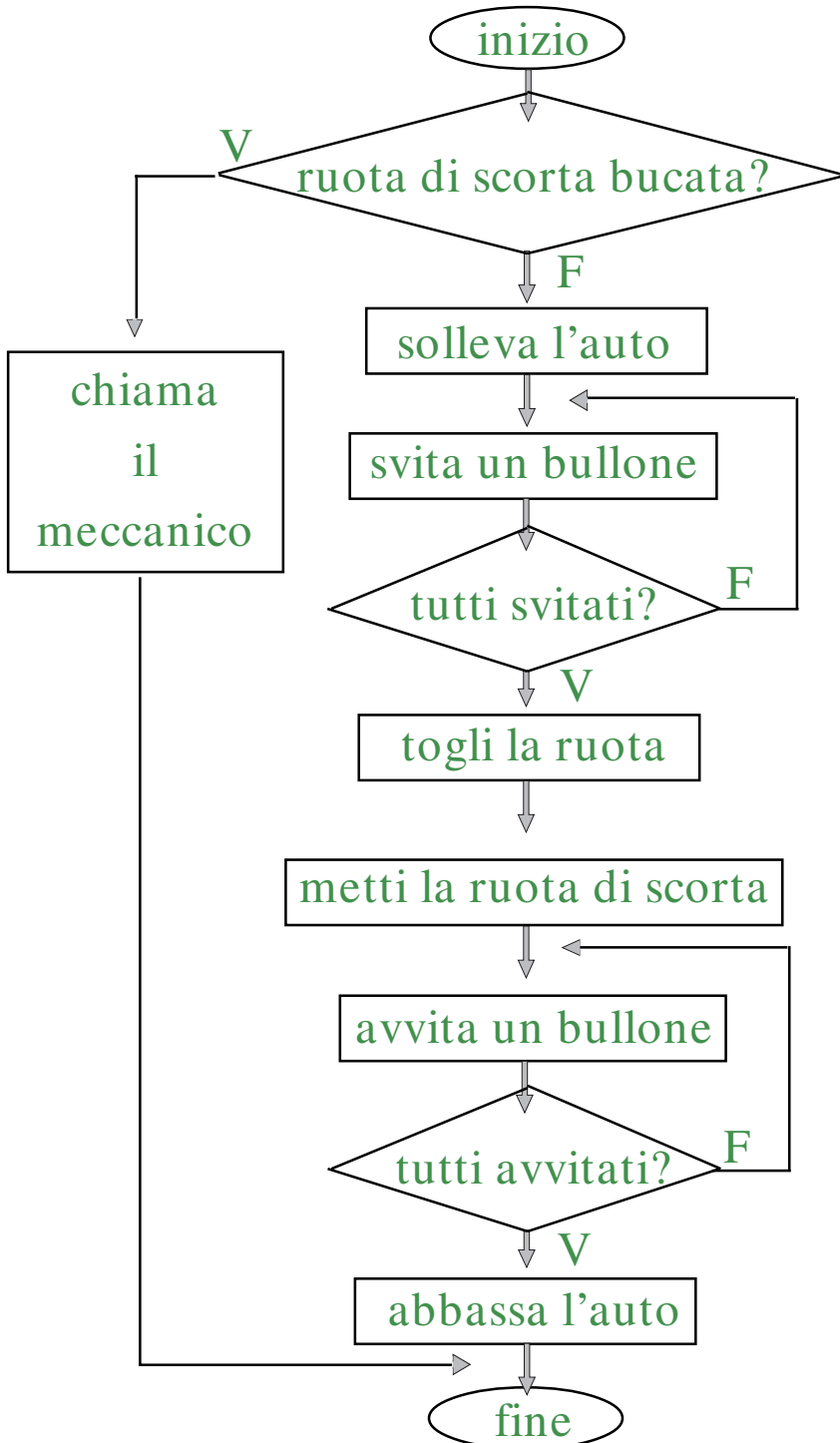
SOLUZIONE



struttura di

TIPO ITERATIVO

algoritmo per il cambio della ruota



Esempio :

preparazione di una torta

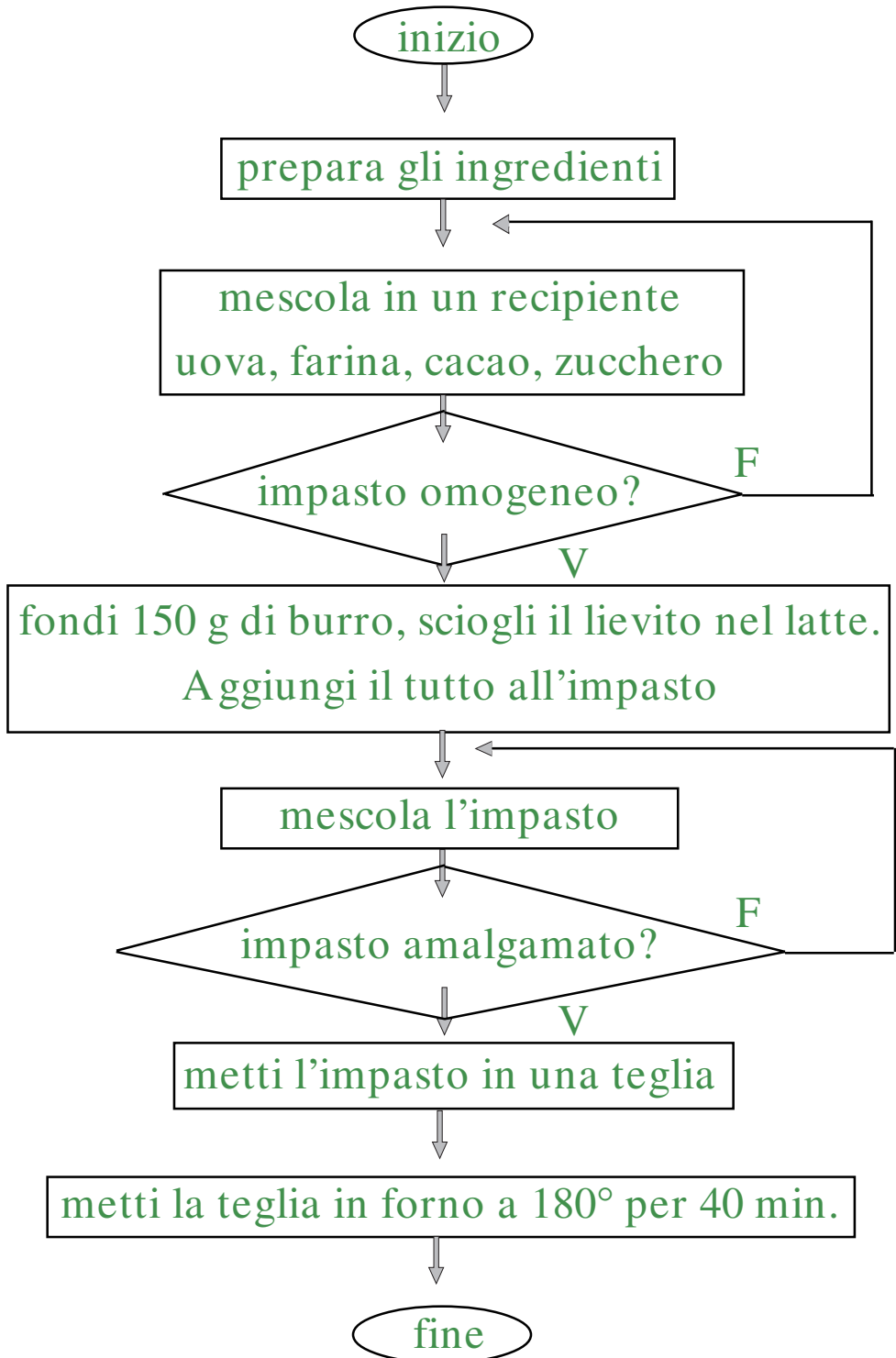
ingredienti: dati del problema

0,25 l latte, 300 g farina, 3 uova, 200 g zucchero, 100 g cacao, 160 g burro, 20 g lievito.

la ricetta: istruzioni

Mescolare farina, uova, zucchero e cacao fino ad ottenere un impasto omogeneo. Aggiungere 150 g di burro fuso e il lievito sciolto nel latte e mescolare fino ad amalgamare il tutto.

Mettere il composto in una teglia imburrata (10 g di burro) e far cuocere in forno per 40 min. alla temperatura di 180 gradi.



Definizione di Algoritmo

Un **algoritmo** è un procedimento per la risoluzione di una classe di problemi, costituito da un **insieme finito di direttive non ambigue** che specificano una **sequenza finita di operazioni** da eseguire su un **insieme finito di dati**

ETIMOLOGIA: da “Al Khuwarizmi”, nome del matematico persiano del IX sec. che descrisse gli algoritmi per le operazioni aritmetiche sui numeri decimali.

Il concetto di

NON AMBIGUITÀ

impone che siano note a priori **le capacità logiche ed operative** dell'esecutore dell'algoritmo

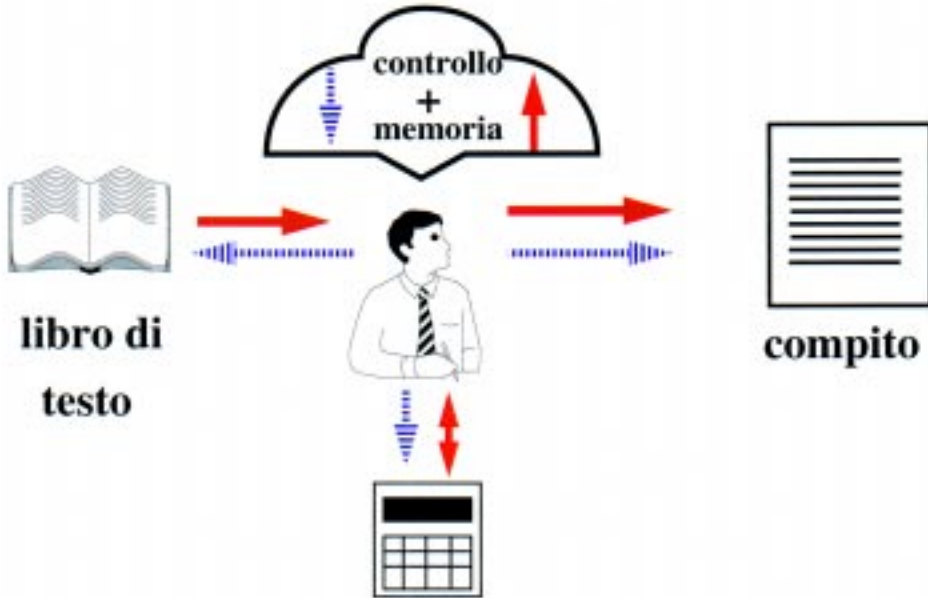
INOLTRE

un algoritmo definisce **non solo il flusso delle operazioni** da compiere, **ma anche i dati** su cui tali operazioni vanno eseguite

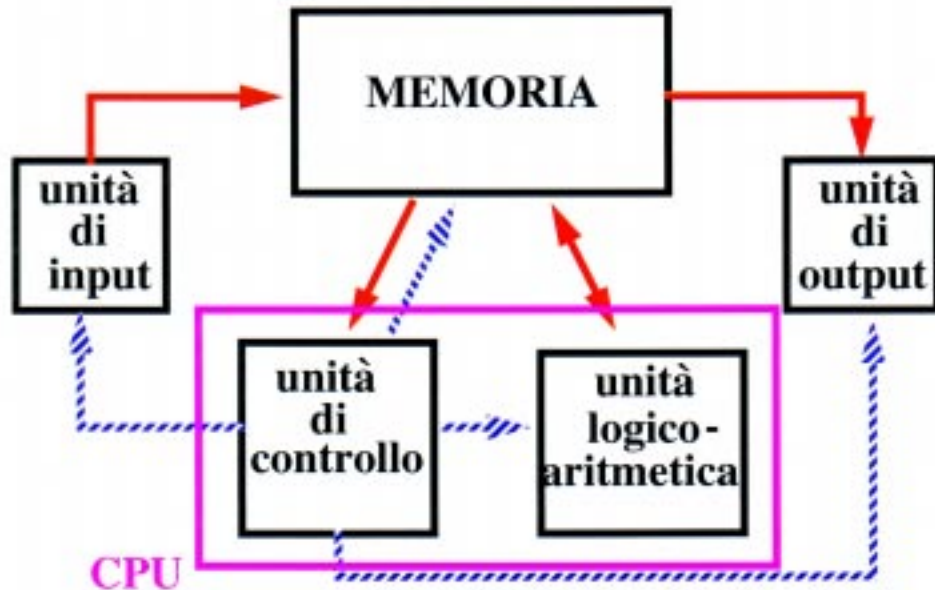
Un esecutore di algoritmi
deve essere quindi in grado di:

- immagazzinare dati (di input intermedi e di output) e istruzioni
- effettuare operazioni aritmetiche e logiche su tali dati
- controllare il flusso delle operazioni

Schema uomo



Schema macchina di Von Neumann



→ dati e istruzioni

controllo

I componenti fondamentali di un calcolatore sono:

- Memoria
- Unità di controllo
- Unità logico-aritmetica
- Unità di input e output

La memoria

è il supporto fisico
che permette di immagazzinare
informazioni (istruzioni e dati)

La memoria è organizzata come
una lista sequenziale di

LOCAZIONI (O CELLE)

Ogni locazione è costituita
da una **sequenza finita**
di componenti elementari (BIT)
ciascuno dei quali
può rappresentare una cifra binaria
(0 oppure 1)

esempio: locazione a 8 bit

1	1	1	0	1	1	1	0
---	---	---	---	---	---	---	---



BIT

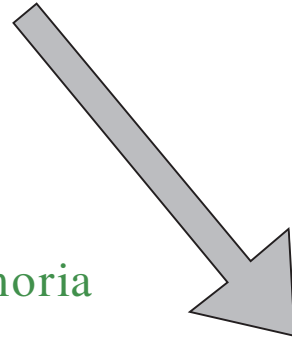


LOCAZIONE

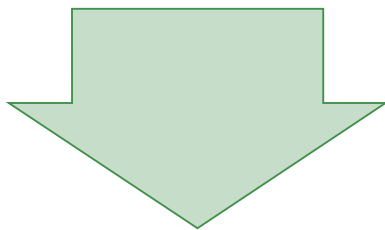
Le locazioni sono
univocamente individuate
in memoria da

INDIRIZZI

schema della memoria



1	0	0	0	1	0	1	0	0010
1	1	0	1	0	1	1	0	0011
0	0	0	1	1	1	0	1	0100
0	0	1	0	0	0	1	1	0101
1	1	0	1	1	0	0	1	0110
1	1	1	0	1	0	0	1	0111



LOCAZIONE

=

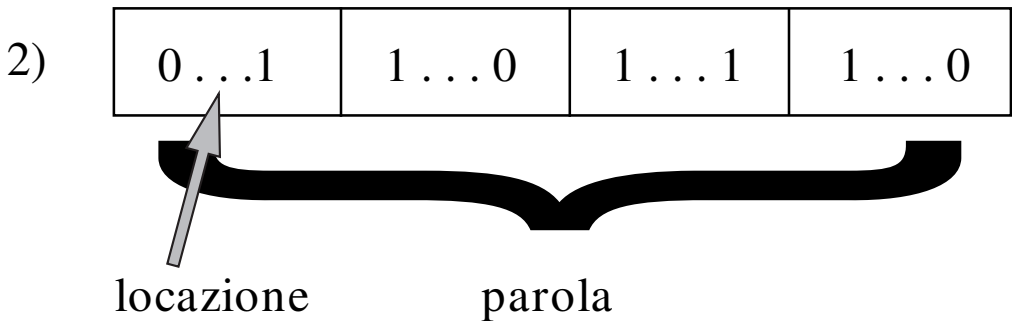
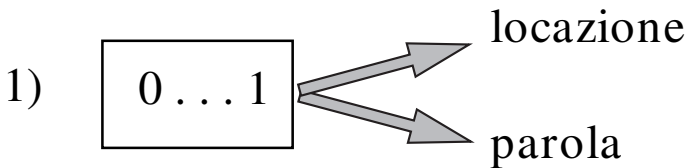
minima quantità di bit
indirizzabile

in anni recenti molti costruttori
di calcolatori hanno standardizzato
locazioni a 8 bit chiamate byte
(1 byte = 8 bit)

Le locazioni possono essere
raggruppate in
parole (o word)

parola
 =
 minimo numero di bit
 necessari per memorizzare un
 dato o una istruzione

Esempi:



Operazioni sulla memoria

lettura

(prelevare il contenuto di una o più locazioni di memoria)

scrittura

(definire il contenuto di una o più locazioni di memoria)

Parametri della memoria

N: capacità

(numero di parole)

W: ampiezza delle parole

(generalmente misurata in byte)

C: tempo di accesso

(tempo per prelevare un dato da una locazione)

Il costo della memoria è proporzionale a:

$$\frac{WN}{C}$$

Caratteristiche della memoria di alcuni calcolatori

calcolatore	parola	capacità (MB)
Sun Sparc	32 bit	64-512
PC 80836-486	32 bit	4-32
PC Pentium	32 bit	64-512
Intel i860	32 bit	8-16
ws HP9000	32 bit	32-400
ws IBM RS 6000	32 bit	128-1024
ws IBM Power2	64 bit	128-768
Digital 5900	32 bit	32-256
ws DEC alpha	64 bit	128-1024

1 Mbyte = 10^6 bytes

Unità di controllo:

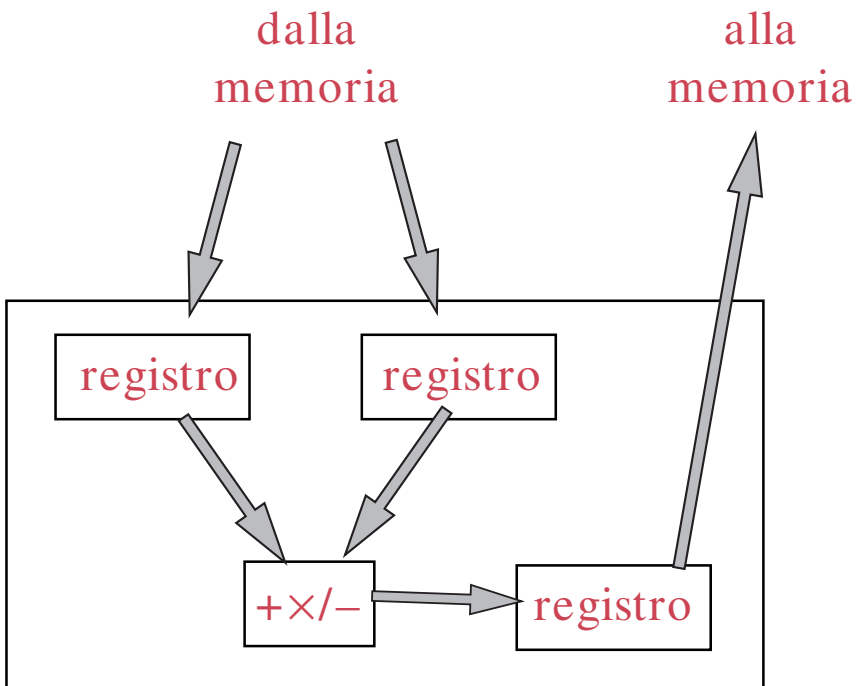
è il coordinatore di tutte
le attività del calcolatore

- preleva dalla memoria le istruzioni
- le interpreta
- trasferisce i dati dalla memoria a speciali aree di memoria presenti nell'unità logico-aritmetica (registri)
- determina la successiva istruzione da eseguire

Unità logico-aritmetica

è l'esecutore delle operazioni
sui dati, sia aritmetiche,
sia di confronto

- gli operandi sono memorizzati nei **registri**



unità logico-aritmetica

$$\begin{array}{c} \text{unità di controllo} \\ + \\ \text{unità logico-aritmetica} \\ = \\ \text{CPU} \\ \text{(Central Processing Unit)} \end{array}$$

Caratteristica della CPU:

velocità operativa

(tempo di esecuzione di una istruzione)

misure della velocità operativa:

- **tempo di clock (in MHz)**
valori tipici sono 66, 100, 133, 200, 233, 300, 400
- **MIPS**
Milioni di Istruzioni Per Secondo
- **MFLOPS**
Milioni di operazioni FLOating-Point al Secondo

Unità di Input-Output

è l'interfaccia del calcolatore
con il mondo esterno

- consente l'interazione con l'uomo (video, tastiera, stampante, mouse, ...)
- consente l'accesso alle reti telematiche (Ethernet, Internet, ...)

La documentazione del software

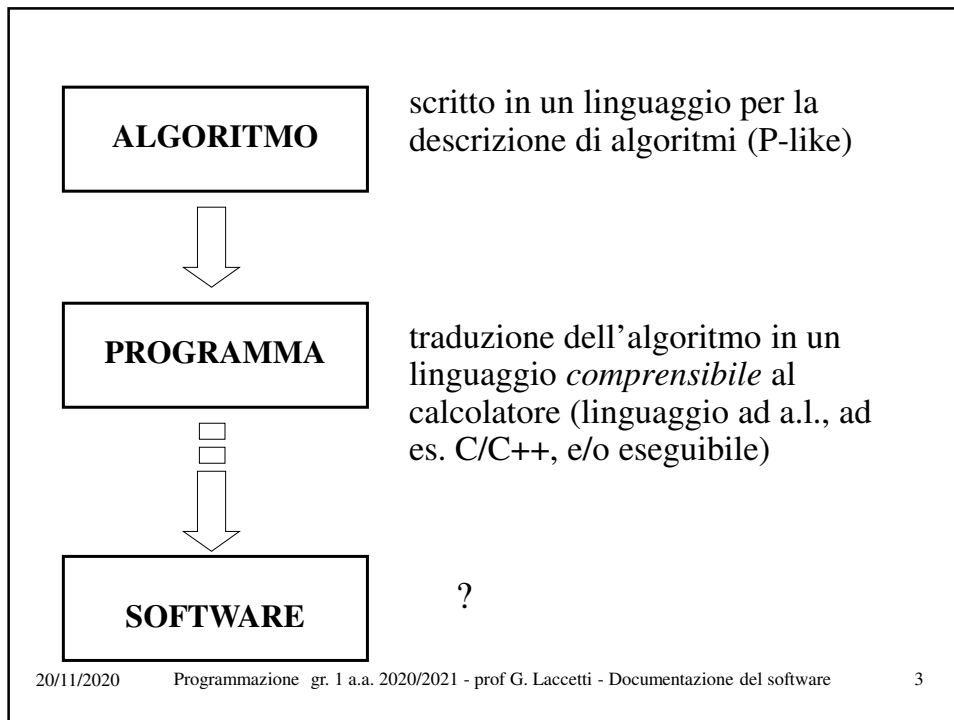
Slides basate su materiale presente in
A. Murli – Lezioni di Laboratorio di Programmazione e Calcolo, cap. 8

20/11/2020 Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software 1

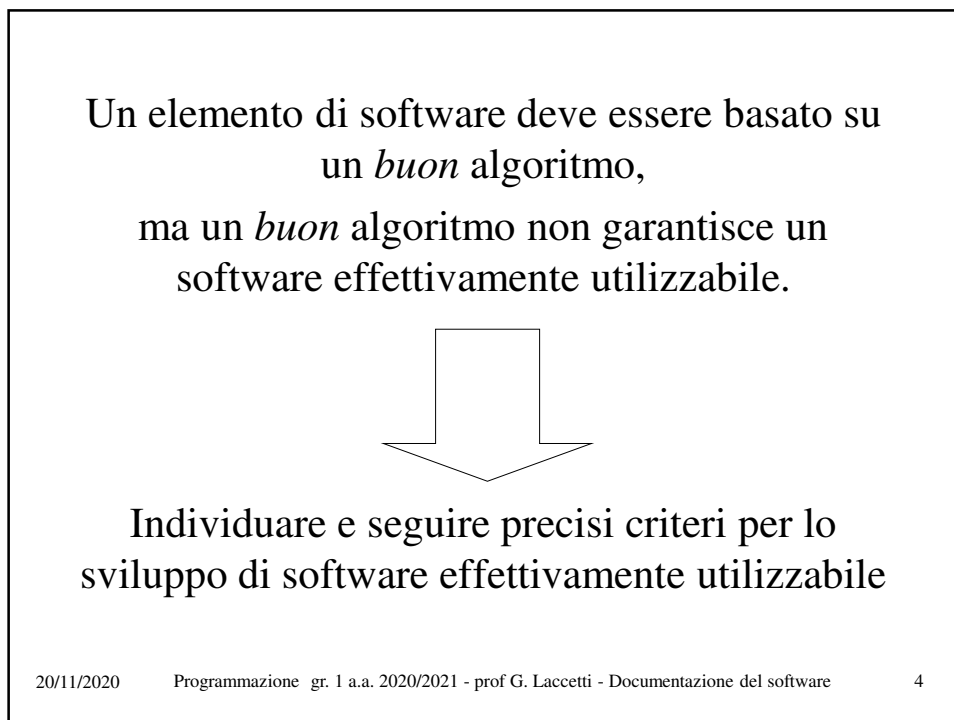
1

20/11/2020 Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software 2

2



3



4

Criteri di valutazione del software

- efficienza
- affidabilità
- robustezza
- bontà della documentazione
- semplicità di utilizzo
- flessibilità
- modularità
- trasportabilità
-

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

5

5

... Il software è generalmente buono solo tanto
quanto lo è la documentazione di supporto ...

(B.Ford, in "Problems and Methodologies in Mathematical Software Production", P.Messina e A.
Murli eds. ...)

20/11/2020

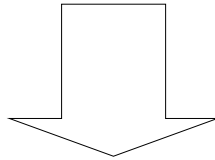
Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

6

6

Documentazione del software

prima interfaccia utente-software



- istruzioni per l'uso
- informazioni sull'organizzazione interna
- trasferimento delle esperienze di progetto e valutazione

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

7

7

Una documentazione deve contenere:

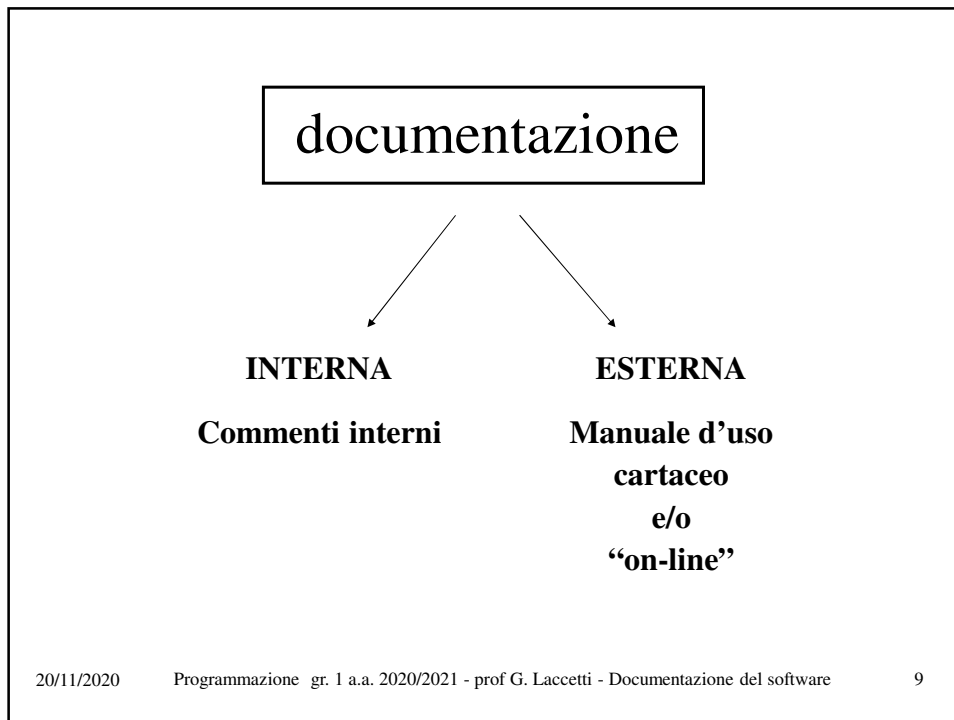
- una chiara descrizione della classe di problemi risolubili dalla procedura e delle restrizioni sulla sua applicabilità
- una precisa descrizione del tipo e dello scopo di ogni parametro della procedura e di ogni restrizione su di esso
- un programma test che illustri l'uso della procedura
- una indicazione dell'accuratezza e dell'efficienza della procedura

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

8

8



9

convenzioni sulla documentazione esterna

- scopo
- specifiche
- descrizione
- riferimenti bibliografici
- lista dei parametri
- indicatori di errore
- procedure ausiliarie
- complessità di tempo e di spazio
- accuratezza fornita
- esempio di uso

20/11/2020 Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software 10

10

Esempio di documentazione esterna per la procedura “ProdScal” (procedura in linguaggio C)

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

11

11

Procedura ProdScal

Scopo

Calcolo del prodotto scalare di due vettori.

Specifiche

```
void ProdScal (int n, float a[], float b[], float *r)
```

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

12

12

Descrizione

a) Background del problema

Il problema del calcolo del prodotto scalare di due vettori è una delle operazioni fondamentali dell'algebra lineare.

Indicando con $\mathbf{A}=(a_0, \dots, a_{n-1})$ e $\mathbf{B}=(b_0, \dots, b_{n-1})$ i due vettori, il loro prodotto scalare p è uno scalare definito come:

$$p = \sum_{i=0}^{n-1} a_i * b_i$$

b) Breve descrizione dell'algoritmo

L'algoritmo implementato calcola il risultato utilizzando la definizione su riportata di prodotto scalare. L'algoritmo in Pascal-like è, dunque, il seguente:

```
p:=0
for i:=0 to n-1 do
  p:=p+a(i)*b(i)
endfor
```

Riferimenti bibliografici

A. Murli, G. Laccetti, et al., Laboratorio di Programmazione I, pag ... Liguori, 2003.

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

13

13

Lista dei parametri

int n : In input contiene la dimensione **n** dei due vettori. Invariato in output.
float a[n] : In input **a[i]** deve contenere a_i , $i=0, \dots, n-1$. Invariato in output.
float b[n] : In input **b[i]** deve contenere b_i , $i=0, \dots, n-1$. Invariato in output.
float *r : In output contiene un puntatore al risultato del prodotto scalare.

Indicatori di errore

Nessuno

Procedure ausiliarie

Nessuna

Raccomandazioni sull'uso

Si ricorda che, affinché il prodotto scalare possa essere calcolato, è necessario che i due vettori di input abbiano la stessa dimensione.

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

14

14

Complessità computazionale

a) Complessità di tempo

L'algoritmo richiede n addizioni e moltiplicazioni (operazioni predominanti); pertanto ha complessità di tempo asintotica $T(n) = O(n)$

b) Complessità di spazio

L'algoritmo utilizza due array monodimensionali di dimensione n ; pertanto ha complessità di spazio asintotica: $S(n) = O(n)$.
Non usa memoria aggiuntiva per array locali.

Accuratezza fornita

Dipende dalla precisione del sistema aritmetico floating-point utilizzato

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

15

15

Esempio d'uso

a) Esempio di programma chiamante

```
#include <stdio.h>
#include <malloc.h>

/* Prototipo della function */
void ProdScal(int n, float a[ ], float b[ ], float *r);

main()
{
    /* Dichiarazione delle variabili */
    int n, i;
    float *a, *b, ris;

    printf("Inserire la dimensione dei vettori\n");
    scanf("%d", &n);

    /* Allocazione dinamica degli array */
    a=(float *)malloc(n*sizeof(float));
    b=(float *)malloc(n*sizeof(float));
```

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

16

16

```

printf("Inserire i %d elementi del primo vettore\n",n);
for (i=0; i<n; i++)
    scanf("%f", &a[i]);

printf("Inserire i %d elementi del secondo vettore\n",n);
for (i=0; i<n; i++)
    scanf("%f", &b[i]);

/* Chiamata della function */
ProdScal(n,a,b,&ris);

/* Stampa del risultato */
printf("Risultato del prodotto scalare: %f\n",ris);

/* Rilascio della memoria per gli array */
free(a);
free(b);
}

```

20/11/2020 Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

17

17

b) Esempio di esecuzione

```

Inserire la dimensione dei vettori
4
Inserire i 4 elementi del primo vettore
1
2
3
4
Inserire i 4 elementi del secondo vettore
1
2
3
4
Risultato del prodotto scalare: 30.0

```

20/11/2020 Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

18

18

documentazione interna

parte integrante del testo della procedura (in
linguaggio di programmazione , ad es
C/C++)

- linee di commento iniziali
- linee di commento per:
 - dichiarazioni di tipo
 - sequenze di istruzioni

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

19

19

Nelle linee di commento iniziali in genere si riporta una
parte della documentazione esterna, ad esempio le sezioni

- scopo
- specifiche
- lista dei parametri
- indicatori di errore
- procedure ausiliarie
- esempio di uso

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

20

20

```

void ProdScal (int n, float a[], float b[], float *r)

/* Scopo */
/* Calcolo del prodotto scalare di due vettori. */
/* Specifiche */
/* void ProdScal (int n, float a[], float b[], float *r) */
/* Lista dei parametri */

/* int n      : In input contiene la dimensione n dei due vettori. Invariato in output. */
/* float a[n]: In input contiene gli elementi di a .Invariato in output. */
/* float b[n]: In input contiene gli elementi di b.Invariato in output. */
/* float *r   : In output contiene un puntatore al risultato del prodotto scalare. */

/* Indicatori di errore */
/* Nessuno */

/* Procedure ausiliarie */
/* Nessuna */

/* Raccomandazioni sull'uso */
/* Si ricorda che, affinché il prodotto scalare possa essere calcolato, è necessario che */
/* i due vettori di input abbiano la stessa dimensione. */
/* ESEMPIO D'USO */
/* ..... */

```

Listing della function ProdScal in C/C++

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

21

21

IN GENERALE

- **Scopo**
breve descrizione del problema che la procedura risolve (2-3 righe)
- **Specifiche**
testata della procedura e dichiarazione dei parametri di I/O
- **Descrizione**
 - background del problema
 - breve descrizione dell'algoritmo
- **Riferimenti bibliografici**
testi, riviste, manuali per approfondire il background e/o l'algoritmo
- **Lista dei parametri**
elenco dei parametri di chiamata della procedura, con indicazioni su: I/O, tipo, dimensione e breve descrizione

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

22

22

- **Indicatori di errore**
spiegazione delle situazioni di errore previste e significato dei relativi parametri di uscita
- **Procedure ausiliarie**
elenco di tutte le eventuali procedure richiamate
- **Complessità di T/S**
 - complessità asintotica di tempo dell'algoritmo usato ed eventuali tempi di esecuzione di esempi test.
 - complessità asintotica di spazio dell'algoritmo usato ed eventuale occupazione di memoria aggiuntiva richiesta dalla routine
- **Accuratezza fornita**
eventuale stima sull'accuratezza del risultato
- **Esempio di uso**
 - Esempio di programma che richiama la procedura
 - Esempi di esecuzione con tutti i dati di input e di output

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

23

23

CONCLUSIONI

- Il software è progettato e realizzato per essere usato da altri
- Documentazione necessaria !
- Il software è quindi una implementazione *funzionante e opportunamente e adeguatamente documentata* di un algoritmo per la risoluzione di un problema (o classe di ..) in uno specifico ambiente di calcolo

" ... un algoritmo non permette **direttamente** la risoluzione di un problema usando il computer; ...
solo la *implementazione dell'algoritmo* (il software!) risolve il problema ..."

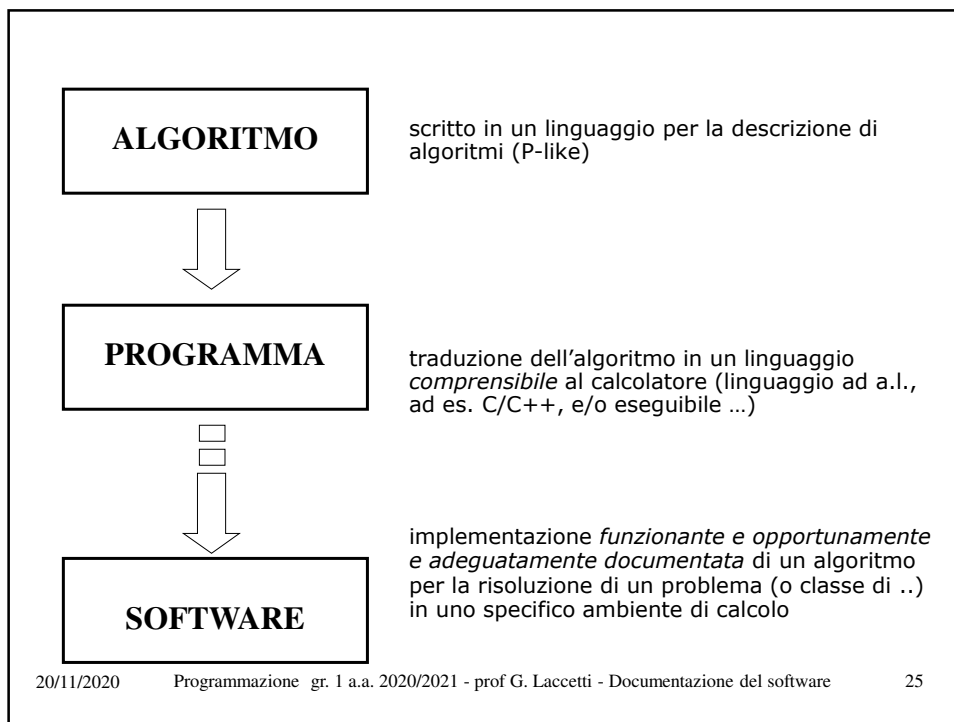
(W.J. Cody, in "Problems and Methodologies in Mathematical Software Production",
P.Messina e A.Murli eds. ...)

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

24

24



25



26

APPENDICE

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

27

27

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

28

28

Esempio di documentazione esterna per la procedura
“TRISOL” (procedura in linguaggio FORTRAN)

SCOPO

TRISOL risolve un sistema di equazioni lineari della forma

$$\mathbf{T}\mathbf{x} = \mathbf{b}$$

Dove T è una matrice triangolare inferiore.

SPECIFICHE

Versione in singola precisione

SUBROUTINE TRISOL (T, LDT, N, B, IFAIL)

INTEGER LDT, N, IFAIL

REAL T(LDT, N), B(N)

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

29

29

SCOPO

Background del problema

I sistemi di equazioni $\mathbf{T}\mathbf{x}=\mathbf{b}$, in cui la matrice T è triangolare, sono particolarmente semplici da risolvere.

Nel caso T sia triangolare inferiore, assumendo che

$$\mathbf{T}_{ii} \neq 0$$

$i=1,\dots,N$, le incognite possono essere calcolate per sostituzione in avanti, dalla relazione

$$x_i = \frac{b_i - \sum_{k=1}^{i-1} T_{ik} x_k}{T_{ii}} \quad (1)$$

$i=1,\dots,N$

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

30

30

Breve descrizione dell'algoritmo

L'algoritmo implementato non utilizza la (1). Esso adotta la convenzione di accedere agli elementi della matrice per colonne.

Poiché il Fortran memorizza le matrici per colonne, tale approccio genera accessi sequenziali alla memoria, mentre la (1) causerebbe accessi a locazioni di memoria non contigue. Invece di sommare i termini $T_{ik}x_k$ $k=1,\dots,i-1$, e poi sottrarre la somma da b_i , i termini sono sottratti da b_i uno alla volta, cioè, calcolato x_k si effettuano le sottrazioni $b_i - T_{ik}x_k$ $i=k+1,\dots,N$.

Tale modifica non altera la stabilità dell'algoritmo.

L'algoritmo prevede il controllo della non singolarità di T .

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

31

31

RACCOMANDAZIONI SULL'USO DELLA ROUTINE

Il parametro di input LDT deve essere uguale al numero di righe della matrice T , come dichiarato nel programma chiamante.

Il parametro di input N deve essere uguale al numero di equazioni.

Nel caso N sia molto grande, usare altre routine che tengano conto della struttura triangolare di T , nella sua memorizzazione.

Nel caso T sia sparsa, usare altre routine ad hoc.

Notare che in output, il vettore delle soluzioni è memorizzato in B (i termini noti vengono persi)

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

32

32

RIFERIMENTI BIBLIOGRAFICI

J.J. Dongarra, J.R. Bunch, C.B. Moler, G.W. Stewart: "LINPACK USER'S GUIDE", SIAM, 1979.

LISTA DEI PARAMETRI DI INPUT-OUTPUT

IN INPUT

- T** Array reale. Contiene la matrice dei coefficienti del sistema. Gli elementi uguali a zero, cioè quelli al di sopra della diagonale principale non sono utilizzati. I corrispondenti elementi possono essere usati per memorizzare altre informazioni.
- LDT** Intero. È il massimo numero di righe dell'array T (leading dimension).
- N** Intero. È l'ordine del sistema.
- B** Array reale. Contiene i termini noti del sistema.

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

33

33

IN OUTPUT

- B** Array reale. Contiene la soluzione, se l'indicatore IFAIL=0, altrimenti è inalterato.
- IFAIL** Intero. Contiene zero se il sistema è non singolare, altrimenti contiene l'indice del primo elemento diagonale della matrice T uguale a zero.

INDICATORI DI ERRORE

IFAIL=0 Denota la terminazione senza errore.

IFAIL=I Implica che T è singolare e $T_{ii} = 0$

ROUTINE AUSILIARIE RICHIAMATE

Nessuna.

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

34

34

TEMPO DI ESECUZIONE

L'algoritmo richiede n divisioni e:

$$\sum_{i=1}^N i - 1 = 1/2 N (N-1) \approx N^2/2$$

addizioni e moltiplicazioni. Pertanto, ha complessità asintotica di tempo

$$O(N^2/2)$$

MEMORIA RICHIESTA

L'algoritmo utilizza un array bidimensionale per memorizzare la matrice, pertanto ha una complessità asintotica di spazio

$$O(N^2)$$

Non usa memoria aggiuntiva per array locali

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

35

35

ACCURATEZZA FORNITA

L'accuratezza del risultato dipende dal condizionamento della matrice T e dalla precisione del sistema aritmetico floating-point utilizzato.

20/11/2020

Programmazione gr. 1 a.a. 2020/2021 - prof G. Laccetti - Documentazione del software

36

36

L'approccio ricorsivo

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

1

1

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

2

2

L'approccio ricorsivo

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

3

3

Metodologie di sviluppo di algoritmi

- **approccio incrementale**
- **approccio divide et impera (divide and conquer)**
- **approccio ricorsivo**

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

4

4

- L'idea incrementale può essere sintetizzata come la risoluzione di una sequenza di istanze dello stesso problema, a partire dall'istanza più semplice fino alla soluzione del problema dato
- La risoluzione di una istanza viene effettuata attraverso una stessa operazione che coinvolge ad ogni passo la soluzione della istanza precedente

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

5

5

- Problemi di calcolo di sommatorie e produttorie rientrano nell'ambito dei problemi descrivibili da *formule ricorrenti lineari*:

$$y_i = a_i y_{i-1} + b_i$$

$i > 0; a_i, b_i \text{ dati}; y_0 = \text{valore prefissato}$



formula ricorrente del *primo ordine*

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

6

6

- Formula di Fibonacci (formula ricorrente del secondo ordine):

$$y_i = a_i y_{i-1} + b_i y_{i-2} + c_i$$

$i > 0; a_i, b_i, c_i, \text{dati}; y_0 = \text{valore prefissato}$

- `frlc_2 (n,1.,1.,0.,1.)`

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

7

7

- L'applicazione dell'approccio incrementale alla soluzione delle formule ricorrenti
- e' detta *approccio iterativo*
- il conseguente algoritmo e' un *algoritmo iterativo*

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

8

8

- L'idea di fondo dell'approccio **divide et impera** e' quella di suddividere il problema in due o piu' sottoproblemi (risolvibili piu' semplicemente), applicando ancora a tali sottoproblemi la stessa tecnica suddivisione.
- Un algoritmo basato sul **divide et impera** genera una sequenza di istanze piu' semplici del problema, fino all'istanza che non e' ulteriormente divisibile, e che ha **soluzione banale**
- *In genere la soluzione del problema di partenza deve poi essere ricostruita a partire dalla soluzione banale*

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

9

9

- L' **approccio ricorsivo** e' un modo, alternativo all'*iterazione*, per denotare la *ripetizione di una azione*
- Dal punto di vista del linguaggio, l'approccio ricorsivo si realizza mediante una *function* (o una *procedure*) che al suo interno contiene una chiamata alla *function* (o *procedure*) stessa; tale processo e' detto **chiamata ricorsiva** (o **autoattivazione**)

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

1
0

10

- Consideriamo il problema del calcolo della somma dei primi n numeri naturali

$$s_{nat} = \sum_{i=1}^n i$$

il problema si può formulare come

$$s_{nat} = n + \sum_{i=1}^{n-1} i$$

cioè come :

- n + “soluzione del problema della somma dei primi $(n-1)$ numeri naturali”
- Il problema del calcolo della somma dei primi $(n-1)$ numeri naturali è una istanza più semplice del problema di partenza!

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

11

11

- La risoluzione di tale istanza può essere rimandata, a sua volta, alla risoluzione dell'istanza più semplice:

$$\sum_{i=1}^{n-1} i = (n-1) + \sum_{i=1}^{n-2} i$$

e così via, fino al sottoproblema banale, cioè:

$$\sum_{i=1}^1 i = 1$$

- Tale procedimento si sintetizza con e allora

$$s_{nat} = s_n$$

$$s_k = k + s_{k-1} \quad k > 1 \quad s_1 = 1$$

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

12

12

- Utilizzando l'approccio ricorsivo, l'algoritmo incrementale può essere:

```
function somma_ric (n) : integer
var n: integer
begin
  if (n=1) then
    /*soluzione problema banale*/
    somma_ric:= 1
  else
    /*autoattivazione*/
    somma_ric:= n+somma_ric(n-1)
  endif
end
```

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

13

13

- Il calcolo della soluzione di un problema esprimibile mediante una formula ricorrente viene descritto in modo immediato nell'approccio ricorsivo:
- il **caso banale** è la **cond. iniziale** ($s_1 = 1$ nell'esempio)
- la chiamata/e ricorsiva riproduce la struttura della formula (*n + soluzione del problema della somma dei primi n-1 numeri naturali*, nell'esempio)

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

14

14

- Sequenza attivazioni della function **somma_ric**
(si consideri $n=5$)

```
somma_ric(5)=5+somma_ric(4)
  ||
  4+somma_ric(3)
    ||
    3+somma_ric(2)
      ||
      2+somma_ric(1)
```

- A questo punto non è stato calcolato nessun risultato !!
- Sono state effettuate 5 chiamate alla function **somma_ric**
(1 attivazione e 4 autoattivazioni) ma **nessuna attivazione ha portato a termine l'esecuzione !**

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

15

15

```
somma_ric(5)=5+somma_ric(4)
  ||
  4+somma_ric(3)
    ||
    3+somma_ric(2)
      ||
      2+somma_ric(1)
        ||
        1
```

```
somma_ric(5)=5+somma_ric(4)
  ||
  4+somma_ric(3)
    ||
    3+3
```

```
somma_ric(5)=5+somma_ric(4)
  ||
  4+6
somma_ric(5)=5+10
=15
```

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

16

16

```

function fatt(n):integer
  var n: integer
  begin
    fatt:= 1
    for i:= 2 to n do
      fatt:= fatt * i
    endfor
  end

  function fatt_ricorsivo(n):integer
  var n :integer
  begin
    if (n<=1) then
      /*soluzione problema banale*/
      fatt_ricorsivo:=1
    else
      /*autoattivazione*/
      fatt_ricorsivo:=fatt_ricorsivo(n-1)*n
    endif
  end
end

```

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

17

17

Massimo Comun Divisore (Great Common Divisor, GCD) (basato sull'algoritmo di Euclide)

GCD tra 2 interi - caso banale: uno dei due interi = 0 → GCD=l'altro

altrimenti $GCD(M,N) = GCD(N, \text{resto di } M/N = M \bmod N)$

```

function gcd(m,n): integer
  var m,n: integer
  begin
    if (m mod n = 0) then
      gcd:= n
    else
      gcd:= gcd(n, m mod n)
    end
  end
end

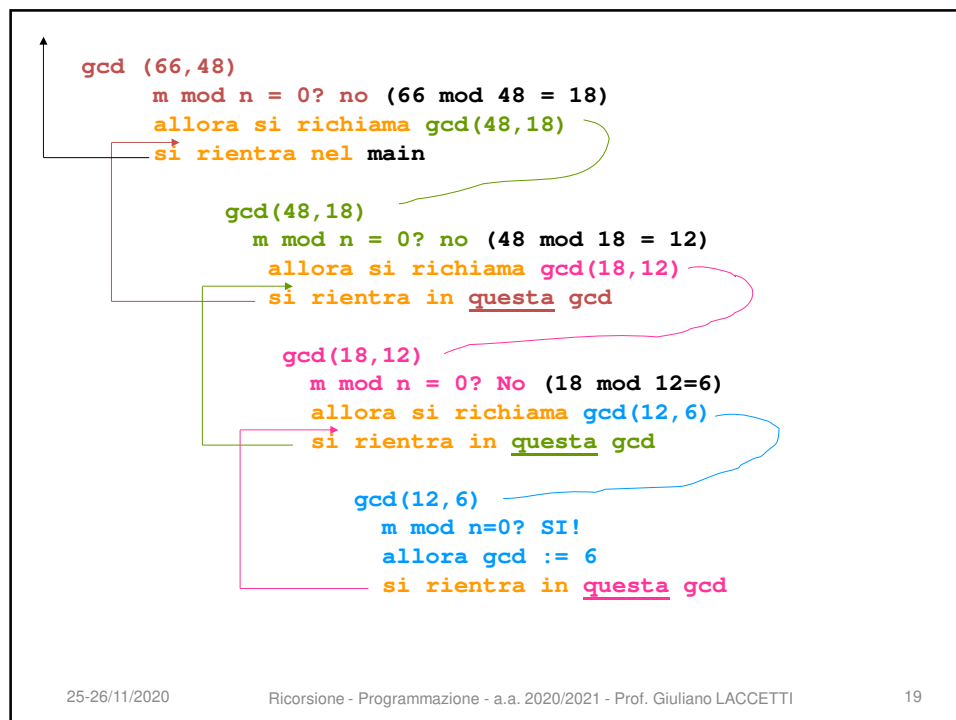
```

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

18

18



19

Massimo Comun Divisore (Great Common Divisor, GCD) (basato sull'algoritmo di Dijkstra)

L'idea: se $m > n$, $\text{GCD}(m, n) = \text{GCD}(m-n, n)$.

Perché?

Se m/d e n/d sono entrambe divisioni esatte (senza resto), allora $(m-n)/d$ è una divisione esatta. Questo porta al seguente algoritmo:

$$\text{for } m, n > 0 \quad \text{gcd}(m, n) = \begin{cases} m & \text{se } m = n \\ \text{gcd}(m - n, n) & \text{se } m > n \\ \text{gcd}(m, n - m) & \text{se } m < n \end{cases}$$

25-26/11/2020 Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI 20

20

Ricerca binaria – versione ricorsiva

```
procedure ricerca_binaria (in:elenco, ...
lunghezza, chiave; out: trovato, posiz)
var elenco: array (1..lunghezza) of character
var chiave: character
var lunghezza, posiz: integer
var trova_pos_ricorsiva: integer function
begin
    posiz:=trova_pos_ricorsiva(elenco,1,lunghezza,chiave)
    if (posiz=0) then
        trovato:=false
    else
        trovato:=true
    endif
end
```

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

21

21

```
function trova_pos_ricorsiva(elenco,primo,ultimo,chiave)
: integer
var elenco: array(primo .. ultimo) of character
var chiave: character
var mediano, primo,ultimo: integer
begin
    mediano:=(primo+ultimo)/2
    if (primo=ultimo .or. chiave=elenco(mediano)) then
        if (chiave=elenco(primo)) then
            trova_pos_ricorsiva:= primo
        else
            if (chiave=elenco(mediano)) then
                trova_pos_ricorsiva:= mediano
            else
                trova_pos_ricorsiva:= 0
            endif
        endif
    else
        if (chiave<elenco(mediano)) then
            trova_pos_ricorsiva:=
            trova_pos_ricorsiva(elenco,primo,mediano-1,chiave)
        else
            trova_pos_ricorsiva:=
            trova_pos_ricorsiva(elenco,mediano+1,ultimo,chiave)
        endif
    end
end
```


25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

22

22

Riepilogo

- Algoritmo ricorsivo: algoritmo la cui soluzione è trovata in termini di soluzioni di versioni “più piccole” dello stesso problema
- Un problema che si risolve “con un ciclo” si può risolvere con la **ricorsione** ( ripetizione di un set di istruzioni, proprio come in un ciclo)
- Quello che cambia è che prima di terminare tale sequenza ripetitiva, la sequenza stessa si interrompe, e ne inizia una nuova (nuova **istanza** della stessa sequenza, **con diversi-nuovi valori dei parametri**)

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

23

23

Esercizio n. 1

- Scrivere una **function** ricorsiva di tipo **logico sorted** che restituisce il valore **TRUE** se gli elementi di un array **A** di interi di dimensione **n**, (passati come parametri) sono in ordine crescente.

```
function sorted (in:n,A) : logical
...
  if (n=1) then
    sorted := TRUE
  else if ( A(n-1)> A(n) ) then
    sorted := FALSE
  else
    sorted := sorted (n-1, A)
  endif
endif
```

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

24

24

Esercizio n. 2

- Scrivere una **function** ricorsiva di tipo **intero lowest** che restituisce il valore dell'indice dell'elemento più piccolo di un array **A** di dimensione **n** (passati come parametri). La prima chiamata dal **main** è del tipo
min := lowest (n-1, A, n)

```
function lowest (in: n, A, index) : integer
....
  if (n=0) then
    lowest := index
  else if ( A(n)< A(index) ) then
    lowest := lowest (n-1, A, n)
  else
    lowest := lowest (n-1, A, index)
  endif
endif
```

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

25

25

Esercizio n. 3

- Scrivere una **function** ricorsiva di tipo **intero ric_seq_ricorsiva** che restituisce il valore della posizione di una **chiave** in un array **A** di dimensione **n** (passati come parametri). (ricerca sequenziale). L'assenza della chiave è indicata dal valore 0

```
function ric_seq_ricorsiva (in: n, A, chiave) : integer
...
  if (n=0) then
    i := 0
  else if ( A(n) = chiave ) then
    i := n
  else
    i := ric_seq_ricorsiva (n-1, A, chiave)
  endif
endif
ric_seq_ricorsiva := i
.....
```

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

26

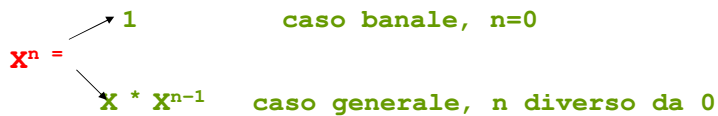
26

Esercizio n. 4

- Scrivere una **function** ricorsiva di tipo **reale**
x_elevato_n (in: **x,n**) che restituisce il valore di X^n

```
Ricorda: function x_elevato_n (x,n) : real
    var x: real
    var n: integer
```

.....



25-26/11/2020

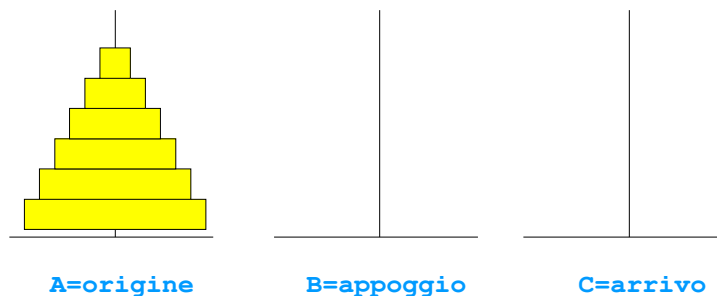
Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

27

27

Esercizio n. 5 - La torre di Hanoi - 1

- Spostare $n=64$ **golden disks** (qui consideriamo ad es. $n=6$) da un piolo all'altro (ad esempio da A a C) rispettando **certe regole precise**



1. Si può muovere un solo disco alla volta
2. Il terzo piolo serve come piolo di "appoggio"
3. Nessun disco può essere appoggiato su uno di diametro inferiore

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

28

28

Esercizio n. 5 - La torre di Hanoi - 2

- **Il problema si può scomporre nei seguenti sottoproblemi:**

1. Spostare i 5 dischi "che stanno sopra" (*top disks*) da **origine** a **appoggio**
2. Spostare il disco di sotto da **origine** a **arrivo**
3. Spostare i 5 dischi da **appoggio** a **arrivo**

25-26/11/2020

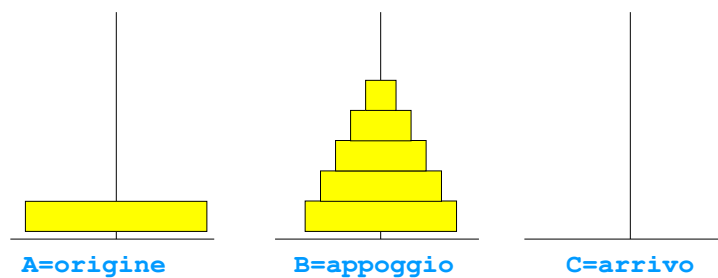
Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

29

29

Esercizio n. 5 - La torre di Hanoi - 3

1. Spostare i 5 *top disks* da **origine** a **appoggio**



25-26/11/2020

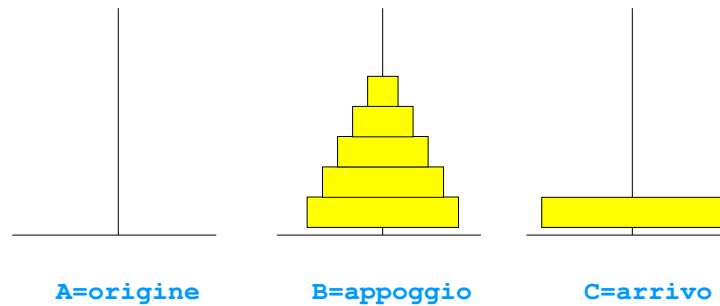
Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

30

30

Esercizio n. 5 - La torre di Hanoi - 4

2. Spostare il disco **di sotto** da **origine** a **arrivo**



25-26/11/2020

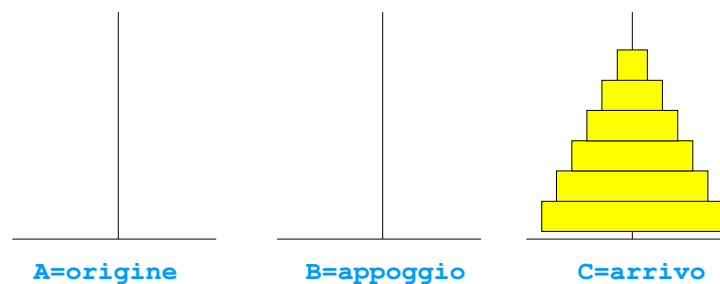
Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

31

31

Esercizio n. 5 - La torre di Hanoi - 5

3. Spostare i 5 dischi da **appoggio** a **arrivo**



Problema risolto!! (n=6)

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

32

32

Esercizio n. 5 - La torre di Hanoi - 6

- Questa è la strada ricorsiva: spezzare il problema in una istanza di dimensioni più piccole dello stesso problema e proseguire con la suddivisione fino al caso banale
- Un possibile esempio di testata di una procedura ricorsiva per il problema della torre di Hanoi, potrebbe essere

procedure tower_of_hanoi (in:n; in/out: origine, arrivo, temp)

dove **n** è il numero di dischi, **origine**, **temp** e **arrivo** i 3 pioli di cui abbiamo parlato, con ovvio significato.

Se **n = 1** allora sposta il disco da **origine** a **arrivo**

Se **n > 1**, dividi il problema in 3 sottoproblemi:

1. Usando lo stesso algoritmo, sposta i "**top**" **n-1** dischi da **origine** a **temp** (in questa fase, il piolo **arrivo** funge da appoggio)
2. Sposta l'ultimo disco rimasto da **origine** a **arrivo**
3. Sposta i "**top**" **n-1** dischi da **temp** a **arrivo** (**origine** funge da appoggio)

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

33

33

Esercizio n. 5 - La torre di Hanoi - 7

procedure tower_of_hanoi(in:n; in/out: origine, arrivo, temp)

- in questa versione **origine**, **arrivo** e **temp** sono 3 array di dimensione **n** ; pensando i dischi "**etichettati**" da 1 a **n**, **n** è il disco di diametro maggiore.
- Il contenuto dei 3 array dopo ogni trasferimento (**operazione di assegnazione** **arrivo(n) := origine(n)**) riflette la situazione corrente dei 3 pioli.

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

34

34

Esercizio n. 5 - La torre di Hanoi - 8

```
procedure tower_of_hanoi (in:n; in/out origine, arrivo,
                        temp)
begin
  if n = 1 then
    arrivo(n) := origine(n)
  else
    tower_of_hanoi(n-1, origine, temp, arrivo)
    arrivo(n) := origine(n)
    tower_of_hanoi(n-1, temp, arrivo, origine)
  endif
end

end procedure
```

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

35

35

Esercizio n. 5 - La torre di Hanoi - 9

- Volendo **"leggere"** tutte le **"mosse"** da compiere, ad esempio (vedi Dromey) si può, invece di aggiornare i contenuti degli array, richiedere una **visualizzazione (print)** della **"mossa"** corrente

```
... ..
if n = 1 then
  print ("da" arrivo "a" origine)
else
  tower_of_hanoi(n-1, origine, temp, arrivo)
  print ("da" arrivo "a" origine)
  tower_of_hanoi(n-1, temp, arrivo, origine)
endif
... ..
```

- In questo caso **origine**, **temp** e **arrivo** potrebbero essere ad esempio semplici variabili di tipo alfanumerico.

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

36

36

Esercizio n. 5 - La torre di Hanoi - 10

- Prevedere, nella versione con gli array, una procedura che ad ogni "mossa" visualizzi la situazione dei 3 pioli

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

37

37

Esercizio n. 6

Sia **A** una matrice di dimensioni **nxn**, contenente numeri interi.

Progettare in P-like un **algoritmo ricorsivo**, sotto forma di **function** di tipo **logical** (**function diagonale_nulla (A,riga)**, con **A** array 2D di tipo integer, di dimensioni **nxn**, implementazione della matrice A), che restituisca **TRUE** se tutti gli elementi della diagonale principale sono nulli, **FALSE** altrimenti.

La chiamata nel main è del tipo

matrice_diagonale:= diagonale_nulla (A,n)

25-26/11/2020

Ricorsione - Programmazione - a.a. 2020/2021 - Prof. Giuliano LACCETTI

38

38

Il problema della leading dimension

3/11/2020

Programmazione 9CFU - gr. 1 - a.a. 2020/2021 - prof. Giuliano Laccetti - LDA

1

1

Memorizzazione elementi array 2D per righe (tipica ad es. del C)

```
var: A [4,3]: array of integer  
var: N,M: integer  
read N, M → ad es. (3,2)
```

```
procedure alpha (TAB,N,M)  
var TAB[N,M]:array of integer
```

A(1,1)		TAB(1,1)
A(1,2)		TAB(1,2)
A(1,3)		TAB(2,1)
A(2,1)		TAB(2,2)
A(2,2)		TAB(3,1)
A(2,3)		TAB(3,2)
A(3,1)		
A(3,2)		
A(3,3)		
A(4,1)		
A(4,2)		
A(4,3)		

«disallineamento» tra elementi
array chiamante e array *procedure*
a causa del diverso numero di colonne nella
dichiarazione di dimensionamento

3/11/2020

Programmazione 9CFU - gr. 1 - a.a. 2020/2021 - prof. Giuliano Laccetti - LDA

2

2

Memorizzazione elementi array 2D per righe (tipica ad es. del C)

```
var: A [4,3]: array of integer
var: N,M: integer
read N,M → ad es. (3,2)
LDA:= 3
alpha(A,LDA)
```

```
procedure alpha (TAB, LDA, N,M)
var TAB[N,LDA]:array of integer
```

A(1,1)		TAB(1,1)
A(1,2)		TAB(1,2)
A(1,3)		
A(2,1)		TAB(2,1)
A(2,2)		TAB(2,2)
A(2,3)		
A(3,1)		TAB(3,1)
A(3,2)		TAB(3,2)
A(3,3)		
A(4,1)		
A(4,2)		
A(4,3)		

Corretto allineamento
elementi array chiamante – array *procedure*
mediante uso della leading dimension LDA

Algoritmo di merge tra 2 array ordinati

Un'ampia discussione sul tema, così come diverse varianti dell'algoritmo qui presentato, si trovano in:

Laboratorio di Programmazione I, pag. 180 e segg.
Algoritmi fondamentali, pag. 186 e segg.

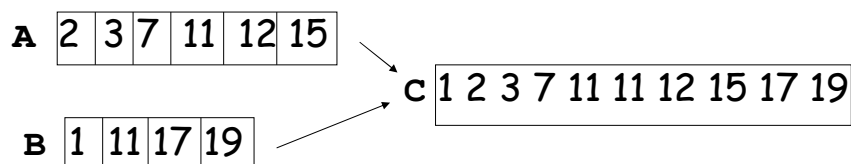
Algoritmo di merge tra 2 array ordinati

Un'ampia discussione sul tema, così come diverse varianti dell'algoritmo qui presentato, si trovano in:

Laboratorio di Programmazione I, pag. 180 e segg.
Algoritmi fondamentali, pag. 186 e segg.

3

- dati 2 array **A** e **B**, di dimensione **n** ed **m**, rispettivamente, entrambi di tipo intero, *costruire un unico array C, fusione dei 2 array dati, che mantenga l'ordinamento* (merge tra 2 array ordinati)



4

- Per ottenere un unico array **C** ordinato (di dimensione, ovviamente **n+m**), l'**operazione base** da effettuare è **confrontare**, a partire dal primo elemento di ciascun array, un elemento di **A** con uno di **B**; supponendo che l'elemento di **A** sia minore di quello di **B**, si inserisce tale elemento di **A** in **C**; il successivo confronto avviene tra l'elemento di **B** (sempre "*quello*") ed il *successivo* di **A**

(analoghe azioni se invece è l'elemento di **B** a risultare minore)

Per ottenere l'avanzamento sull'uno o altro array, basterà aggiornare un indice.

Anche l'indice di **C** dovrà essere aggiornato ogni volta che si effettua un inserimento

- Un primo ragionamento può essere:
Abbiamo tre array, **A**, **B**, **C**, usiamo un indice diverso per ciascuno di essi
ad es. **i** per **A**; **j** per **B**; **k** per **C** **allora abbiamo**

```

i:=1; j:=1, k:=1
... ..
while i<=n AND j<=m do
    if ( A(i) < B(j) ) then
        C(k):=A(i)
        i:=i+1
    else
        C(k):=B(j)
        j:=j+1
    endif
    k:=k+1
endwhile
... ..

```

- Il ciclo **while..do** va ripetuto fintanto che ci sono ancora elementi sia in **A** sia in **B**, cioè, come da condizione, **i<=n** e **j<=m**
- Quando ciò non è più vero, dopo aver verificato quale dei 2 array ha già “contribuito” con tutti i suoi elementi a **C**, non resta che copiare i rimanenti elementi dell’altro, così come sono, in **C**.

```

if i>n then                /* array A "esaurito" */
  for h:=j to m do
    C(k) := B(h)          /* copia dei restanti elementi di B in C */
    k:=k+1
  endfor
else                        /* array B "esaurito" */
  for h:=i to n do
    C(k) := A(h)          /* copia dei restanti elementi di A in C */
    k:=k+1
  endfor

```

ESERCIZI da fare

1. Progettare in P-like, sotto forma di **procedure**, un algoritmo di merge tra array ordinati

```
procedure merge_tra_2Array_ordinati (in: A, B, n, m out: C)
```

- 1bis. Progettare anche un possibile chiamante per questa procedura

2. Progettare inoltre una versione dell’algoritmo che, fondendo i 2 array, elimini eventuali “doppioni”

Fine merge

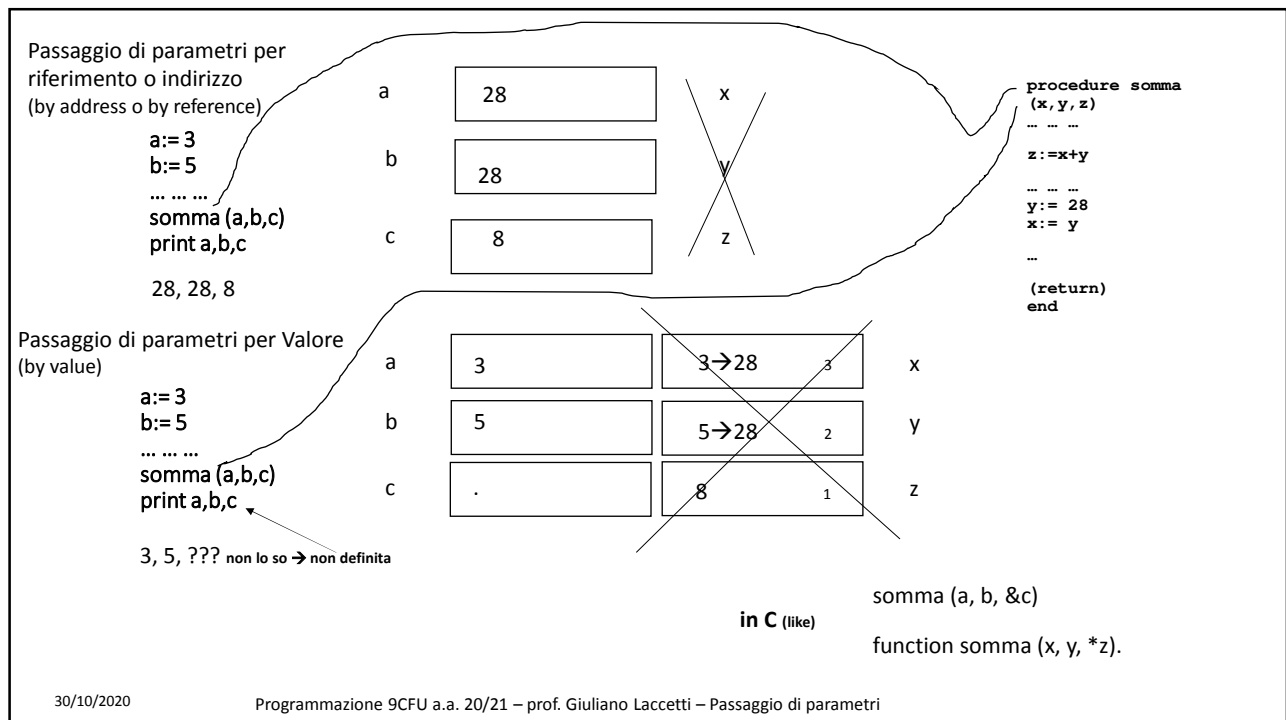
Passaggio di parametri

1. riferimento
2. valore

30/10/2020

Programmazione 9CFU a.a. 20/21 – prof. Giuliano Laccetti – Passaggio di parametri

1



30/10/2020

Programmazione 9CFU a.a. 20/21 – prof. Giuliano Laccetti – Passaggio di parametri

2

Algoritmo di ricerca binaria

A
A
A
A
A
A
A
A
A
A
A
A
A
A

12/11/2020

A
A
A
A
A
A
A
A
A
A
A
A
A
A

23/10/2019

Programmazione 9CFU gr. 1 – a.a. 2019/2020 – Ricerca Binaria – prof. Giuliano Laccetti


Algoritmo di ricerca di un elemento in un array ordinato: ricerca binaria

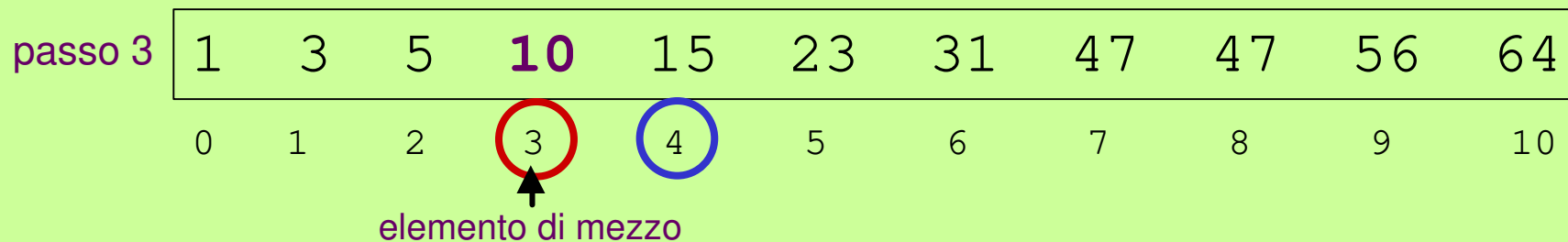
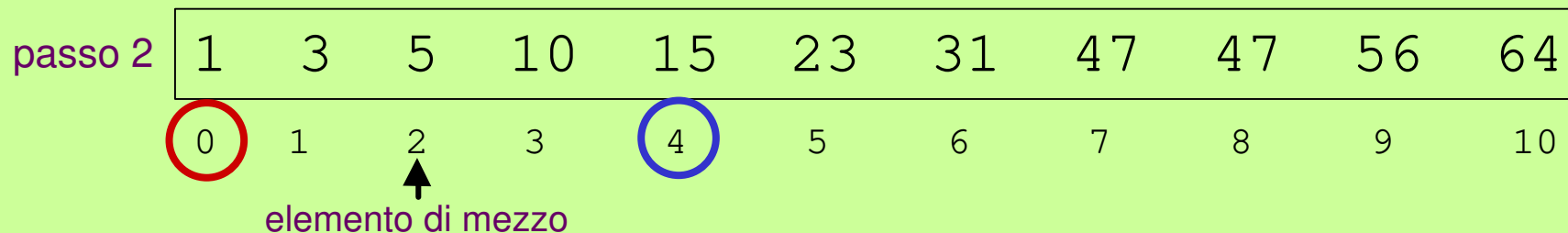
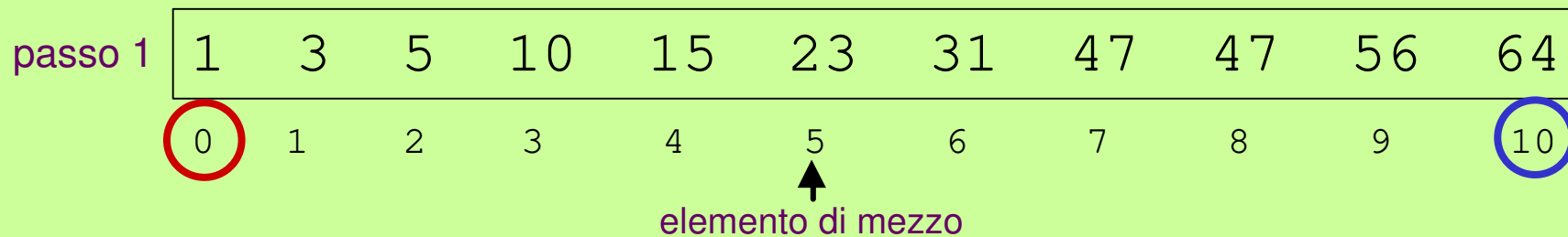
- L'algoritmo di ricerca sequenziale ha una complessità (caso peggiore e caso medio) proporzionale a n e viene pertanto denominato *ricerca lineare*
- Un algoritmo molto più efficiente è quello che usiamo per cercare, ad es., in un elenco del telefono
- Tale algoritmo si basa sull'ipotesi che l'array sia ordinato (per es. in senso crescente) e viene denominato *ricerca binaria*

Ricerca binaria: algoritmo

valore da cercare: 10

primo 

ultimo 






Ricerca binaria: algoritmo

valore da cercare: 8

primo 




ultimo 

passo 1

1	3	5	10	15	23	31	47	47	56	64
	1	2	3	4		6	7	8	9	



elemento di mezzo

passo 2

1	3	5	10	15	23	31	47	47	56	64
	1		3		5	6	7	8	9	10



elemento di mezzo

passo 3

1	3	5	10	15	23	31	47	47	56	64
0	1	2			5	6	7	8	9	10

elemento di mezzo

passo 4

1	3	5	10	15	23	31	47	47	56	64
0	1			4	5	6	7	8	9	10

Ricerca binaria: algoritmo in P-like

```
procedure ricerca_binaria ( in: A, n, elemento;
    out: trovato, posizione_elemento)
    var primo, ultimo, medio: integer
    var: posizione_elemento: integer
    var: trovato: logical
    var: A(1..n) :array of tipobase
    var: elemento: tipobase

    primo:= 1
    ultimo:= n
    trovato:= FALSE
    posizione_elemento:= -1

    while ( (primo<=ultimo).AND. not trovato) do
        medio = (primo + ultimo) / 2
        if ( elemento = A(medio) ) then
            trovato:= TRUE
            posizione_elemento:= medio
        else if ( elemento < A(medio) ) then
            ultimo = medio - 1;
        else
            primo = medio + 1;
        endif
    endif
endwhile
```

Ricerca binaria: complessità di tempo

- Prendiamo come passo elementare *un ciclo* (include due assegnazioni)
- Ad ogni ciclo la lunghezza del sottoarray si dimezza
- Se non si trova prima il valore cercato dopo approssimativamente $\log_2 n$ cicli il sottoarray si riduce a 1 o 2 elementi
- Al ciclo successivo `primo` supera `ultimo` e l'algoritmo termina
- Caso peggiore e caso medio proporzionale a $\log_2 n$

Ricerca calcolata (hash)

Slides basate su

R.G. Dromey - *Algoritmi Fondamentali* – Gruppo Editoriale Jackson

E. Horowitz et al. – *Strutture dati in C* – McGraw-Hill

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

1

1

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

2

2

Ricerca calcolata (hash)

Slides basate su

R.G. Dromey - *Algoritmi Fondamentali* – Gruppo Editoriale Jackson

E. Horowitz et al. – *Strutture dati in C* – McGraw-Hill

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

3

3

Ricerca calcolata (hash)

Un metodo di ricerca molto più veloce della
ricerca binaria, e che spesso esamina solo 1 o 2
elementi prima di concludersi con successo !!

Slides basate su

R.G. Dromey - *Algoritmi Fondamentali* – Gruppo Editoriale Jackson

E. Horowitz et al. – *Strutture dati in C* – McGraw-Hill

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

4

4

Supponiamo di avere il seguente array di numeri interi,
e supponiamo di voler cercare il 44

10 12 20 23 27 30 31 39 42 44 45 49 53 57 60

(come abbiamo già detto svariate volte, in realtà vorremo cercare informazioni "associate" al 44, cioè 44 è un campo *chiave*)

Potremmo trovare tale valore con un solo accesso? Per *magia* potremmo tentare immediatamente con la 10ma posizione?

IMPOSSIBILE !!!

E se invece il 44 occupasse la 44ma posizione?

POSSIBILE !!!

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

5

5

(Tale approccio è però palesemente impraticabile. Si pensi ad esempio di voler memorizzare in un array una nostra piccola rubrica telefonica personale, diciamo di una ventina di numeri, di cui uno è 5552737. Non costruiremmo mai un array di 5552737 elementi, solo per memorizzarne 20 !!!).

Torniamo però al nostro esempio

10 12 20 23 27 30 31 39 42 44 45 49 53 57 60

Abbiamo un insieme di 15 numeri tra 10 e 60

1. *normalizzazione* . Ad esempio, cioè, "applichiamo" una trasformazione al numero che vogliamo cercare: 60 diventa "15", 20 diventa "5" , e così via (*divisione per 4*)

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

6

6

Applicando questa trasformazione (arrotondando il risultato della divisione), ci si accorge che alcuni valori condividono lo stesso "indice" nell'intervallo 1-15, mentre altri non vengono usati

		3		5	6	7	8		10	11	12	13	14	15
		↑		↑	↑	↑	↑		↑	↑	↑	↑	↑	↑
		(10,12)		(20)	(23)	(27)			(39)		(49)		(57)	
									(30,31)	(42,44,45)	(53)		(60)	

Riducendo l'estensione, si sono però così introdotte le occupazioni multiple, le *collisioni*.

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

7

7

- Un tale schema di normalizzazione introdurrà situazioni di collisione molto più "pesanti", in casi, ad esempio, in cui il valore max sia molto più grande di tutti gli altri: 6000 invece di 60, nel nostro miniesempio
- Si ha bisogno, dunque, di una trasformazione che non sia causata da forti irregolarità nella distribuzione dei dati originali nell'array risultante.
- **2. hashing** Calcolare i valori dell'insieme originale modulo la dimensione dell'array, 15 nel nostro esempio, e poi aggiungendo 1 (supponiamo per semplicità che tutti i valori delle chiavi siano positivi).
- Questa trasformazione viene usualmente chiamata *hashing*, cioè *calcolata*

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

8

8

Applicando questa *funzione hash* (il modulo dimensione array) al nostro esempio, si ottiene

30	31			49	20			23	39	10		12		44
↑								↑				↑		
(45,60)								(53)				(27,42,57)		

Questo risultato è un po' deludente, perché ci sono ancora parecchie collisioni; le condizioni "generalì" diciamo così, però, sono migliori, in quanto non ci sono forti irregolarità nella distribuzione

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

9

9

- Proviamo ad aumentare la dimensione dell'array "risultato", diciamo, ad esempio, del 20% (tale valore è *accettabile* !)

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
57	20		60	23		44	45	27		10	30	12			53			
	↑			↑							↑	↑						
	39			42							49	31						

Ancora ci sono collisioni, ma questa volta non sono multiple !

Comunque: come *gestire* le collisioni ?

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

10

10

- Le collisioni possono essere risolte memorizzando l'elemento "che collide" nella prima posizione "libera" in avanti!

In tal caso per il nostro esempio si avrà:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
57	20		60	23		44	45	27		10	30	12			53			
	↑			↑							↑	↑						
	39			42							49	31						

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

11

11

- Le collisioni possono essere risolte memorizzando l'elemento "che collide" nella prima posizione "libera" in avanti!

In tal caso per il nostro esempio si avrà:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
57	20	39	60	23	42	44	45	27		10	30	12	31	49	53			

Valutiamo il numero medio di accessi per recuperare un elemento da questa tabella hash:

- 11 elementi localizzati con 1 accesso
- 3 elementi localizzati con 2 accessi
- 1 elemento localizzato con 4 accessi

tempo medio = 1.4 accessi !!!

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

12

12

Progettazione di una procedura di hashing, che restituisca la posizione della chiave cercata (quindi un numero) nella *tabella hash*

1. Calcolare il valore della funzione hash (modulo dimensione tabella)
2. Se la chiave non è nella posizione della tabella corrispondente al valore calcolato, allora:
 - ricerca lineare in avanti dalla posizione corrente della tabella modulo dimensione tabella (si considera l'array/tabella *circolare*)

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

13

13

- Supponendo di cercare **key**, in una tabella **hash_table** di dimensione **n**, il Passo 1. si può realizzare:

posizione_candidata := (key mod n)+1
- ed il test per decidere se **key** si trova o meno nella posizione **posizione_candidata** (prima parte Passo 2.) potrebbe essere del tipo:

if hash_table(posizione_candidata) != key then
- Per *rifare il giro* basterà calcolare

posizione := (posizione + 1) mod n

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

14

14

Come arrestarsi? - 1

- L'elemento **key** è stato trovato **OPPURE**
- è stata incontrata una posizione vuota
- (hint: come al solito, è buona norma non utilizzare un valore del dominio di **key** per indicare una posizione vuota; comunque, per l'algoritmo si potrà utilizzare ad esempio una variabile **empty** che verrà definita in fase di implementazione finale)

if hash_table(posizione_candidata)=empty then

La ricerca allora sarà del tipo:

- fino a che **key** non è trovato e la posizione corrente non è **empty**, si può passare alla locazione successiva (modulo dimensione tabella)

E se la tabella fosse piena, e **key** non presente?

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

15

15

Come arrestarsi? - 2

- Non appena si ritorna alla **posizione_candidata** da cui si è partiti, senza aver trovato **key**, sicuramente **key** non è presente. Allora la ricerca termina per:

1. **key** trovato
2. **key** non presente e posizione corrente = **empty**
3. **key** non presente e tabella *piena*

Controllare per ogni nuova posizione 3 cose è pesante; si possono riunire il 1 ed il 3 test, utilizzando ad esempio una *sentinella*. Più precisamente si può memorizzare temporaneamente **key** nella **posizione_candidata** calcolata **dopo** aver verificato che il valore presente in quella posizione è diverso da **key**. Il valore originale sarà rimesso a posto una volta completata la ricerca.

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

16

16

Come arrestarsi? - 3

- Per terminare il ciclo sarà utilizzata una variabile logica **ancora_da_cercare** che sarà **false** quando **key** è stato trovato oppure si è incontrata una posizione **empty**
- Un'altra variabile logica, **trovato**, si userà per distinguere le condizioni di terminazione. A questa variabile si assegna un valore **SOLO** alla fine della ricerca.

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

17

17

```
procedure hash_search(in: hash_table, size_hash_table,  
                     key, empty; out:posizione, trovato)
```

1. Calcolare **posizione_candidata** (funzione hash modulo dim. array) per **key**
2. Inizializzazione variabile logica **trovato** per terminare la ricerca
3. Se **key** è nella posizione **posizione_candidata** allora:
 - (a) passare alla condizione di terminazione, altrimenti
 - (b) porre una sentinella per controllare la condizione *tabella piena*
4. Finché non è soddisfatta la condizione di terminazione
 - (a) calcolare la posizione successiva
 - (b) se **key** è nella posizione corrente, allora
 - (b.1) andare alla condizione di terminazione e aggiornare **trovato** altrimenti
 - (b.2) se la posizione corrente è **empty** segnalare terminazione
5. Ripristinare tabella
6. Fine procedura

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

18

18

```

procedure hash_search(in: hash_table, size_hash_table, key,
                      empty; out: posizione, trovato)

begin procedure hash_search
  ... ..
  ancora_da_cercare := true
  trovato := false
  posizione_candidata := key mod size_hash_table
  posizione_corrente := posizione_candidata
  if hash_table(posizione_candidata) = key then
    ancora_da_cercare := false
    trovato := true
    temp := hash_table(posizione_candidata)
  else
    temp := hash_table(posizione_candidata)
    hash_table(posizione_candidata) := key
  endif
endif

```

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

19

19

```

while (ancora_da_cercare)
  posizione_corrente := (posizione_corrente+1) mod size_hash_table
  if hash_table(posizione_corrente)=key then
    ancora_da_cercare := false
    if posizione_corrente != posizione_candidata then
      trovato := true
    endif
  else
    if hash_table(posizione_corrente)=empty then
      ancora_da_cercare := false
    endif
  endif
endwhile
hash_table(posizione_candidata) := temp

end procedure hash_search

```

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

20

20

Fattore di carico

- Fattore di carico α = frazione di posizioni occupate nella tabella
- Si può dimostrare che in una ricerca con esito positivo si esamineranno in media $[1+(1/(1-\alpha))]/2$ locazioni.
In una tabella in cui $\alpha = 80\%$, ad esempio, il numero di "confronti" medio prima di trovare l'elemento desiderato sarà circa 3, indipendentemente dalla dimensione della tabella !!
- Allo stesso modo si può dimostrare che il costo medio per una ricerca con esito negativo è $[1+(1/(1-\alpha)^2)]/2$; cioè in una tabella in cui $\alpha = 80\%$, il numero di "confronti" medio prima di incontrare una posizione "vuota" sarà 13, sempre indipendentemente dalla dimensione della tabella !!

11/12/2020

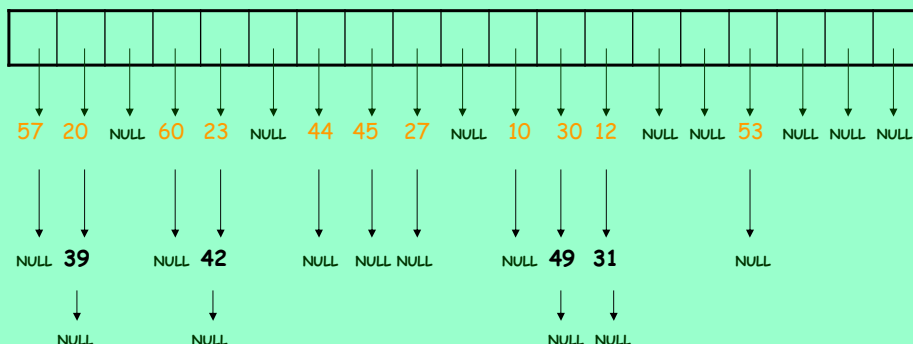
Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

21

21

Hash Table con linked list

- Per rendere più efficiente ancora la ricerca, specialmente in casi con esito negativo, si può pensare alla tabella hash come un array di puntatori
- in ogni elemento dell'array, cioè, ci sarà un puntatore ad una linked list delle key che dovrebbero occupare "quella" posizione nell'array/tabella.



11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

22

22

Esercizi

- Realizzare una procedura per la ricerca in una tabella hash che tenga conto di tutti i possibili casi discussi (ovviamente sarà necessario, ai fini della sperimentazione di tale procedura, avere a disposizione una procedura per “costruire” una tabella hash su cui poi operare la ricerca)
- Realizzare una procedura per la ricerca in una tabella hash organizzata con puntatori a linked list

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

23

23

Rif.:

- Dromey cap. 5, par. 5.8 - pag. 241-251

11/12/2020

Algoritmo di ricerca calcolata (hash) - Programmazione 9CFU gr. 1 aa 2020/2021
prof. Giuliano LACCETTI

24

24

Counting sort

- Dato un array 1D A , di tipo intero, di dimensione n , contenente n numeri interi (non necessariamente distinti) compresi tra 1 e k , progettare un algoritmo, sotto forma di procedura, per ordinare l'array con il *counting sort*
- 1. costruire un array *conta* di k elementi, in cui in *conta* ($A(j)$) ci sia il numero di occorrenze di $A(j)$ in A , per $j=1, \dots, n$
- 2. scandire l'array *conta*, per $i=1, \dots, k$ e scrivere il valore i in A per *conta*(i) volte

```
procedure counting_sort (A, n, k )
.....
  for i:= 1 to k do
    conta(i) := 0
  endfor
  for j := 1 to n do
    conta (A(j)) := conta( A(j) ) +1
  endfor
  for j:= 1 to n do
    A(j) := 0
  endfor
  j := 1
  for i:= 1 to k do
    for s:= 1 to conta(i) do
      A(j) := i
      j := j+1
    endfor
  endfor
.....
```

Algoritmo di exchange sort

Ampia discussione di tale algoritmo, così come differenti versioni, si trovano in:

Algoritmi fondamentali, pag. 203 e segg.

Lezioni di Laboratorio di Programmazione e Calcolo, pag. 258 e segg.

3/11/2020

Programmazione 9CFU - gr. 1 - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di exchange sort

1

1

3/11/2020

Programmazione 9CFU - gr. 1 - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di exchange sort

2

2

Algoritmo di exchange sort

Ampia discussione di tale algoritmo, così come differenti versioni, si trovano in:

Algoritmi fondamentali, pag. 203 e segg.

Lezioni di Laboratorio di Programmazione e Calcolo, pag. 258 e segg.

3/11/2020

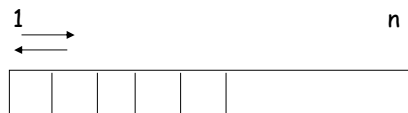
Programmazione 9CFU - gr. 1 - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di exchange sort

3

3

• dato un array di n interi, ordinarlo in ordine non decrescente, con il metodo di *scambio* (*exchange sort*)

Si confrontano 2 elementi adiacenti e, se il caso, si scambiano




3/11/2020

Programmazione 9CFU - gr. 1 - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di exchange sort

4


4

A partire ad es. dal primo elemento, lo si confronta con il successivo, si effettua lo scambio se non sono in ordine




2	4	5	3	11	1	21	17
---	---	---	---	----	---	----	----

 no scambio



2	4	5	3	11	1	21	17
---	---	---	---	----	---	----	----

 no scambio



2	4	5	3	11	1	21	17
---	---	---	---	----	---	----	----

 scambio

2	4	3	5	11	1	21	17
---	---	---	---	----	---	----	----

3/11/2020

Programmazione 9CFU - gr. 1 - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di exchange sort

5

5

Alla fine della prima "passata" si ottiene

2	4	3	5	1	11	17	21
---	---	---	---	---	----	----	----

Il risultato è stato di far slittare l'elemento più grande all'ultimo posto, cioè nella sua posizione corretta (per l'ordinamento). L'array ovviamente non è ancora in ordine, bisogna ripetere lo stesso procedimento

2	3	4	1	5	11	17	21
---	---	---	---	---	----	----	----

2	3	1	4	5	11	17	21
---	---	---	---	---	----	----	----

2	1	3	4	5	11	17	21
---	---	---	---	---	----	----	----

3/11/2020

Programmazione 9CFU - gr. 1 - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di exchange sort

6

6

Algoritmo di exchange sort - 1

```
for j:= 1 to n-1 do
  if (A(j)> A(j+1))then
    "scambio"
  endif
endfor
```

1							n-1	
2	4	3	5	1	11	17	21	

Algoritmo di exchange sort - 2

```
for j:= 1 to n-2 do
  if (A(j)> A(j+1))then
    "scambio"
  endif
endfor
```

1							n-2	
2	3	4	1	5	11	17	21	

Algoritmo di exchange sort

```
procedure exchange_sort (in: n; in/out: A)
  var i, j, n : integer
  var A : array [1..n] of real
  var ordinato: logical
  begin
    ordinato:= FALSE
    i:= 1
    while (i<n) AND (NOT ordinato) do
      ordinato:= TRUE
      for j:= 1 to n-i do
        if (A(j)>A(j+1))then
          "scambio"
          ordinato:= FALSE
        endif
      endfor
      i:=i+1
    endwhile
  end
end exchange_sort
```

3/11/2020

Programmazione 9CFU - gr. 1 - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di exchange sort

9

9

Algoritmo di exchange sort: complessità di tempo

- operazioni di confronto;
- c'è un 1 ciclo **for** innestato in un **while ..do**, che "dipende" dal valore di una variabile indice gestita dal **while..do**
- il **ciclo for** interno viene eseguito **ALMENO** 1 volta
→ **n-1 confronti** **(caso migliore)**
- per ogni valore di **i** verranno effettuati **n-i** confronti quindi, nel **caso peggiore**,
³ $(n-1) + (n-2) + \dots + 2 + 1$ confronti, cioè
 $(n^2 - n) / 2 - 1$ confronti
 $T(n) = O(n^2)$ confronti **(caso peggiore)**

3/11/2020

Programmazione 9CFU - gr. 1 - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di exchange sort

10

10

Algoritmo di exchange sort: complessità di tempo

- operazioni di scambio;
- se l'array è già in ordine NON vengono effettuati scambi (caso migliore);
- nel caso peggiore ci sono tanti scambi quanti confronti
($(n^2 - n)/2 - 1$)
ricordare il numero di scambi per il selection sort, e confrontarlo con questo!
- nel caso medio si effettuano $n(n-1)/4$ confronti

Molti **scambi** nell'ordinamento di array di grandi dimensioni con dati disposti casualmente (poco efficiente, lo scambio è una operazione "costosa")

- Vantaggioso, invece, nel caso di array non grande e con pochi elementi "fuori posto"

3/11/2020

Programmazione 9CFU - gr. 1 - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di exchange sort

11

11

- confrontare il selection sort e l'exchange sort, applicati a diverse disposizioni degli elementi dell'array, utilizzando un contatore del numero di scambi e confronti effettuati
- progettare in P-like una versione dell'exchange sort che metta in ordine l'array a partire dal più piccolo

3/11/2020

Programmazione 9CFU - gr. 1 - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di exchange sort

12

12

EXCHANGE SORT SBAGLIATO

```
procedure exchange_sort_sbagliato (in:n; in/out:A)

  var i, j, n : integer
  var A : array [1..n] of real
  begin
    for i:=1 to n-1 do
      for j:= 1 to n-i do
        if (A(j)>A(j+1))then
          "scambio"
        endif
      endfor
    endfor
  end

end exchange_sort_sbagliato
```

3/11/2020

Programmazione 9CFU - gr. 1 - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di exchange sort

13

13

Fine exchange sort

3/11/2020

Programmazione 9CFU - gr. 1 - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di exchange sort

14

14

Algoritmo di insertion sort

Slides basate su materiale presente in
A.Murli, G.Laccetti et al. - Laboratorio di Programmazione I, pag. 176 e segg
.G.R.Dromey - Algoritmi Fondamentali, pag. 208 e segg.

13/11/2020

Programmazione - prof. G. LACCETTI - a.a. 2020/2021 - Algoritmo di insertion sort

1

1

13/11/2020

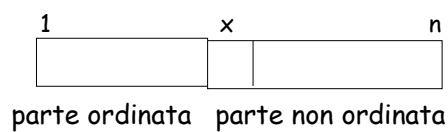
Programmazione - prof. G. LACCETTI - a.a. 2020/2021 - Algoritmo di insertion sort

2

2

- dato un array di n interi, ordinarlo in ordine non decrescente, con il metodo di *inserzione*

l'ordinamento con il metodo di *insertion sort* è uno dei modi più naturali per ordinare informazioni. E' tipico ad esempio del giocatore di carte che ordina le carte in suo possesso.



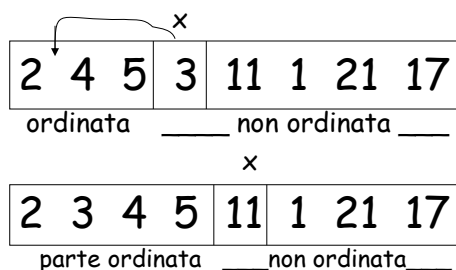
13/11/2020

Programmazione - prof. G. LACCETTI - a.a. 2020/2021 - Algoritmo di insertion sort

3

3

Si preleva il primo elemento della parte non ordinata e lo si inserisce *al suo posto* nella parte ordinata



13/11/2020

Programmazione - prof. G. LACCETTI - a.a. 2020/2021 - Algoritmo di insertion sort

4

4

Algoritmo di insertion sort

```
for i:= 2 to n do
  scegliere prox elemento da inserire (x:=a(i))
  inserire x nella parte ordinata, al posto giusto
endfor
```

Per fare spazio all'inserimento di x, tutti gli elementi maggiori di x, nella parte ordinata, devono essere spostati in avanti di un posto. Si parte da j:=i

```
while (x < a(j-1) ) do
  a(j) := a(j-1)
  j := j-1
endwhile
```

13/11/2020

Programmazione - prof. G. LACCETTI - a.a. 2020/2021 - Algoritmo di insertion sort

5

5

- Problema di terminazione nel caso in cui x sia minore di tutti gli a(j) della parte ordinata, cioè $j=1, \dots, i-1$



```
while ((x < a(j-1)) AND (j >= 2)) do
  oppure
```

Uso di una sentinella: determinare il minimo dell'array e inserirlo al primo posto, PRIMA di iniziare il processo di inserzione

13/11/2020

Programmazione - prof. G. LACCETTI - a.a. 2020/2021 - Algoritmo di insertion sort

6

6

```

                                Algoritmo di insertion sort
procedure insertion_sort (in: n; in/out: a)
  var i, j, n, : integer
  var x : real
  var a : array [1..n] of real
  begin
    mettere il minimo al primo posto
    for i:= 3 to n do
      x := a(i)
      j:=i
      while (x < a(j-1) ) do
        a(j):= a(j-1)
        j := j-1
      endwhile
      a(j) := x
    endfor
  end
end insertion_sort

```

13/11/2020

Programmazione - prof. G. LACCETTI - a.a. 2020/2021 - Algoritmo di insertion sort

7

7

Algoritmo di insertion sort: complessità di tempo

- operazioni di confronto;
 - esame del solo ciclo **for** contenente il **while .. do**
 - il **while .. do** interno viene eseguito **ALMENO 1** volta per ogni valore di **i** del **for** esterno
 → $n-2$ confronti (caso migliore)
 - per ogni valore di **i** **AL PIU'** verranno effettuati **i-1** confronti quindi
 $T(n) = O(n^2)$ confronti (caso peggiore)
- precisamente $2+3+\dots+n-1$ confronti, cioè $(n^2 - n)/2 - 1$ confronti

13/11/2020

Programmazione - prof. G. LACCETTI - a.a. 2020/2021 - Algoritmo di insertion sort

8

8

Fine insertion sort

ALGORITMO DI QUICKSORT

Rif. (studiare): Dromey – Algoritmi Fondamentali, algoritmo 5.6, pag. 221 e segg.

10/12/2020

Algoritmo di quicksort - Programmazione 9CFU - gr. 1 a.a. 2020/2021
prof. Giuliano LACCETTI

1

1

10/12/2020

Algoritmo di quicksort - Programmazione 9CFU - gr. 1 a.a. 2020/2021
prof. Giuliano LACCETTI

2

2

ALGORITMO DI QUICKSORT (Hoare, 1962)

Uno dei *top ten algorithms* del XX secolo

10/12/2020

Algoritmo di quicksort - Programmazione 9CFU - gr. 1 a.a. 2020/2021
prof. Giuliano LACCETTI

3

3

**Idea (simile a quella dello shell sort):
spostamenti dei dati a "grandi" distanze**

(ad es. exchange sort scambia dati adiacenti !)

Si vuole riuscire ad avere dopo un primo passo

piccoli elementi	elementi grandi
------------------	-----------------

un array in cui le 2 metà` contengono a sinistra elementi piu`
piccoli di quelli a destra

10/12/2020

Algoritmo di quicksort - Programmazione 9CFU - gr. 1 a.a. 2020/2021
prof. Giuliano LACCETTI

4

4

Bisogna "scegliere" un elemento in base al quale
partizionare l'array

21	34	17	13	18	9	39	2	37	30
----	----	----	----	----	---	----	---	----	----

Si sceglie ad esempio (elemento di mezzo)

Poi si procede con i confronti da sinistra E da destra
"verso l'interno"

10/12/2020

Algoritmo di quicksort - Programmazione 9CFU - gr. 1 a.a. 2020/2021
prof. Giuliano LACCETTI

5

5

21	34	17	13	18	9	39	2	37	30
----	----	----	----	----	---	----	---	----	----

Ad es. la prima volta si esaminano 21 e 30 : il 30 deve stare a
dx di 18 e sta bene così ;
il 21 si deve spostare: bisogna
trovare un elemento da spostare a sx.

Procedendo da destra verso l'interno si arriva al 2 e si scambiano
21 e 2

2	34	17	13	18	9	39	21	37	30
---	----	----	----	----	---	----	----	----	----

poi si scambiano 9 e 34

2	9	17	13	18	34	39	21	37	30
---	---	----	----	----	----	----	----	----	----

< 18

>= 18

ora le 2 metà sono "separate" !

6

6

Abbiamo adesso 2 partizioni che possono essere trattate indipendentemente, cioè se ordiniamo i primi 4 valori e quindi gli altri 6, alla fine l'intero array sarà ordinato.

Ognuna delle 2 partizioni può essere vista come istanza simile, ma di dimensioni più piccole, del problema originale, da risolvere allo stesso modo (del problema originale).

Ripetendo il processo fino ad arrivare a partizioni di dimensione 1 (ovviamente già ordinate!), si arriva naturalmente all'ordinamento dell'intero array

```
while (dimensione partizioni > 1) do  
  
    "scegliere partizione da trattare"  
  
    "selezionare dalla partizione  
    corrente un valore di  
    discriminazione"  
  
    "suddividere la partizione corrente in  
    2 partizioni più piccole parzialmente  
    ordinate"  
  
endwhile
```

Esistono diversi modi di scegliere l'elemento di partizione, ad esempio il primo elemento del sottoarray oppure l'elemento mediano; quest'ultima scelta non peggiora le prestazioni dell'altra scelta nel caso di elementi disposti in maniera casuale, ma le migliora nel caso di elementi ordinati o ordinati in senso inverso.

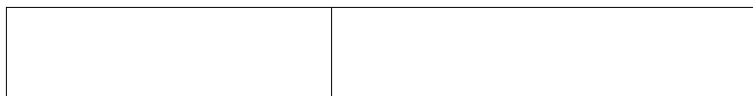
**Se allora upper e lower indicano i limiti dell'array per una data partizione,
 $middle := (lower + upper) / 2$**

Ad ogni passo quindi i dati correnti sono suddivisi in 2 partizioni, delle quali se ne esamina subito una



è necessario "salvare" le informazioni sui limiti dell'altra, da trattare in seguito.

Ad esempio,



partizione sx esaminata subito

**partizione destra esaminata
in seguito: bisogna salvare i "limiti"**

- A mano a mano che il procedimento va avanti, avremo la necessità di “salvare” un gran numero di partizioni, (la cui dimensione sarà sempre più piccola...).

- La tecnica di “ripescaggio” delle partizioni accantonate sarà quella, una volta arrivati a partizione di dimensione 1 (quindi ordinata!), di riprendere in considerazione l'ultima partizione salvata



Utilizzo struttura dati stack

Tra le 2 partizioni “create” ad un certo passo, quale scegliere da trattare subito e quale da “conservare”?

Per risparmiare memoria conviene esaminare prima la partizione di dimensione minore, salvando i limiti di quella maggiore. Così facendo si riduce il numero di elementi da aggiungere allo stack!

```

while (dimensione partizioni > 1) do

    "scegliere la partizione più piccola da
    trattare subito"

    "selezionare l'elemento mediano della
    partizione come valore di discriminazione"

    "suddividere la partizione corrente in 2
    partizioni più piccole parzialmente
    ordinate"

    "salvare la partizione di dimensione
    maggiore per un esame successivo"

endwhile

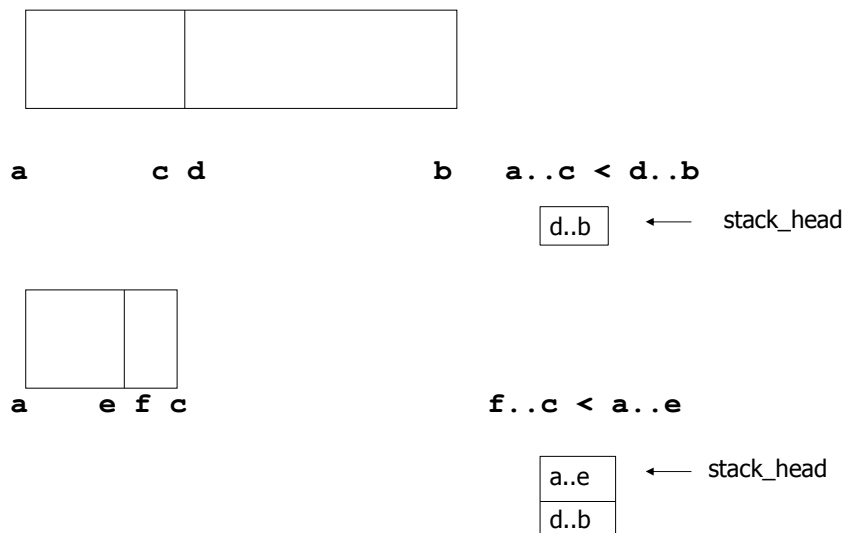
```

10/12/2020

Algoritmo di quicksort - Programmazione 9CFU - gr. 1 a.a. 2020/2021
prof. Giuliano LACCETTI

15

15

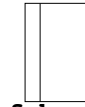


10/12/2020

Algoritmo di quicksort - Programmazione 9CFU - gr. 1 a.a. 2020/2021
prof. Giuliano LACCETTI

16

16



fgh c

f..g < h..c

h..c
a..e
d..b

← stack_head

f..g == dimensione 1

A questo punto, si ricomincia il processo di partizione dell'ultima partizione memorizzata : h .. c , estraendola dallo stack.

Una volta ridotte tutte le partizioni a dimensione 1, non ci saranno più "limiti" nello stack, e l'algoritmo può ritenersi concluso.

10/12/2020

Algoritmo di quicksort - Programmazione 9CFU - gr. 1 a.a. 2020/2021
prof. Giuliano LACCETTI

17

17

Nell'algoritmo, come condizione di partenza possiamo inserire nello stack inizio e fine dell'array originario

Inserimento nello stack degli indici di inizio e fine (ad esempio 1,n) dell'array da ordinare

while stack !vuoto

estrazione dallo stack dei limiti di una partizione

while partizione corrente dim > 1

- . *selezionare elemento centrale*
- . *partizionare in 2 rispetto al centrale*
- . *inserire nello stack i limiti della sottopartizione di dim maggiore*

endwhile

endwhile

10/12/2020

Algoritmo di quicksort - Programmazione 9CFU - gr. 1 a.a. 2020/2021
prof. Giuliano LACCETTI

18

18

```

creazione stack
push (head,1)
push (head,n)
while (head != NULL) do
    right:= pop(head)
    left := pop(head)
    while(right > left)do
        "partiziona rispetto a middle:=(right+left)/2"
        if new_right < middle then
            push(head,new_left)
            push(head,right)
        else
            push(head,left)
            push(head,new_right)
        endif
    endwhile
endwhile

ricordare di aggiornare left e right prima di iterare di nuovo

```

10/12/2020

Algoritmo di quicksort - Programmazione 9CFU - gr. 1 a.a. 2020/2021
prof. Giuliano LACCETTI

19

19

complessità algoritmo di quicksort **(operazione = confronto)**

caso peggiore $O(n^2)$  **max o min come discriminante**

caso medio $O(n \log_2 n)$

10/12/2020

Algoritmo di quicksort - Programmazione 9CFU - gr. 1 a.a. 2020/2021
prof. Giuliano LACCETTI

20

20

Quicksort ricorsivo

- Si applica sempre lo stesso meccanismo alle successive partizioni risultanti: la partizione di segmenti sempre più piccoli continua fino ad arrivare a un segmento di un solo elemento.
- Lo stesso procedimento viene applicato ad istanze sempre più piccole del problema di partenza



- **natura ricorsiva del procedimento !!**

```
.....  
partition ( in/out: a, leftlower, rightupper;  
            out: leftupper, rightlower)  
quicksort_ricorsivo (a, leftlower, leftupper)  
quicksort_ricorsivo (a, rightlower, rightupper)  
.....
```

```

procedure quicksort_ricorsivo (.....)
.....
  if leftlower < rightupper then
    partition (.....)
    "test su quale partizione esaminare prima"
    quicksort_ricorsivo (.....)
    quicksort_ricorsivo (.....)
  endif
.....

```

Algoritmo di selection sort

Ampia discussione sull'algoritmo, così come differenti versioni, si trovano in:

Laboratorio di Programmazione I, pag. 178 e segg.

Algoritmi fondamentali, pag. 196 e segg.

5/11/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di selection sort

1

5/11/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di selection sort

2

Algoritmo di selection sort

Ampia discussione sull'algoritmo, così come differenti versioni, si trovano in:

Laboratorio di Programmazione I, pag. 178 e segg.

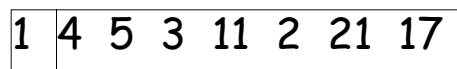
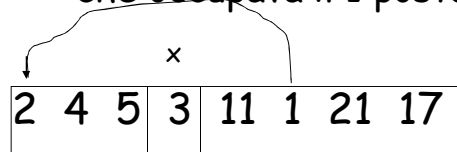
Algoritmi fondamentali, pag. 196 e segg.

- dato un array di n interi, ordinarlo in ordine non decrescente, con il metodo di *selezione*

ad ogni passo del procedimento, si trova ad es., il più piccolo elemento, e lo si pone in ordine.

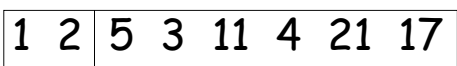
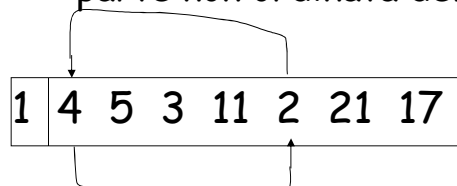


Si individua l'elemento minimo, lo si pone al primo posto, scambiandolo con l'elemento che occupava il 1 posto



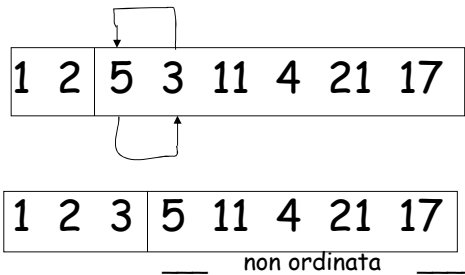
non ordinata

Si procede allo stesso modo sulla restante parte non ordinata dell'array

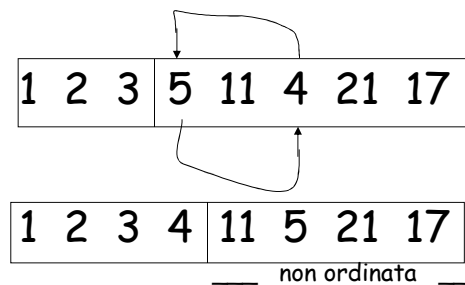


non ordinata

Si procede allo stesso modo sulla restante parte non ordinata dell'array



Si procede allo stesso modo sulla restante parte non ordinata dell'array



Algoritmo di selection sort - 1

```
min:= A(1)
p:=1
for j:= 2 to n do
    if (A(j)<min)then
        min:=A(j)
        p:=j
    endif
endfor
"scambio tra A(1) e A(p)"
```

Algoritmo di selection sort - 2

```
min:= A(2)
p:=2
for j:= 3 to n do
    if (A(j)<min)then
        min:=A(j)
        p:=j
    endif
endfor
"scambio tra A(2) e A(p)"
```

Algoritmo di selection sort

```
procedure selection_sort (in: n; in/out: A)
  var i, j, n, p : integer
  var A : array [1..n] of real
  var min: real
  begin
    for i:= 1 to n-1 do
      min:=A(i)
      p:=i
      for j:= i+1 to n do
        if (A(j)<min)then
          min:=A(j)
          p:=j
        endif
      endfor
      "scambio tra A(i) e A(p)"
    endfor
  end
end selection_sort
```

Algoritmo di selection sort: complessità di tempo

operazioni di confronto

•vi sono 2 cicli `for` innestati, in cui quello interno "dipende" da quello esterno

• al 1° passo, per $i=1$, il ciclo interno viene eseguito $n-1$ volte;

• al 2° passo, per $i=2$, il ciclo interno viene eseguito $n-2$ volte;

• ...

• per $i=n-1$, il ciclo interno viene eseguito 1 volta

→ $(n-1) + (n-2) + \dots + 1$ confronti

Algoritmo di selection sort: complessità di tempo

operazioni di scambio

- l'operazione di scambio fa parte delle operazioni del ciclo for più esterno, quindi si effettuano $n-1$ scambi

Progettare un algoritmo in P-like sotto forma di function **logical function array_ordinato(A, n)**

- che esamini un array e restituisca il valore TRUE se l'array è ordinato, FALSE altrimenti

```

function array_ordinato (A,n) : logical
  var i : integer
  var A : array [1..n] of real
  begin
    i:=1
    while i<n AND A(i)<=A(i+1)do
      i:= i+1
    endwhile
    if i=n then
      array_ordinato:= TRUE
    else
      array_ordinato:= FALSE
    endif
  end
end array_ordinato

```

Fine selection sort

ALGORITMO DI SHELLSORT

(diminuzione di incrementi)

(D. Shell)

rif.: R.G. Dromey - Algoritmi Fondamentali, Jackson Libri
leggere e studiare bene la sezione dedicata a questo Algoritmo 5.5, pag. 214 e segg.

20-24/11/20

Programmazione gr. 1 - a.a. 2020/2021 prof. Giuliano LACCETTI
shellsort

1

1

20-24/11/20

Programmazione gr. 1 - a.a. 2020/2021 prof. Giuliano LACCETTI
shellsort

2

2

ALGORITMO DI SHELLSORT (diminuzione di incrementi) (D. Shell)

Nell'algoritmo di exchange sort si confrontano gli elementi *adiacenti*, cioè a distanza "1" tra loro.

Osservando un array ordinato e il corrispondente array di partenza (disordinato), si può notare che gli elementi sono "spostati" *in media* di $n/3$ posti



Algoritmo che confronta gli elementi a distanza "più grande" di 1 (per portare prima ciascun elemento vicino alla sua posizione definitiva)

20-24/11/20

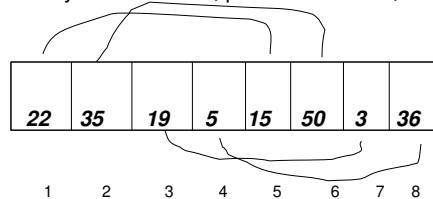
Programmazione gr. 1 - a.a. 2020/2021 prof. Giuliano LACCETTI
shellsort

3

3

IDEA:

confrontare gli elementi dell'array a distanza $n/2$, poi a distanza $n/4$, ... via via fino a distanza 1

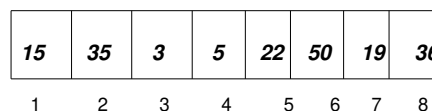


Si confrontano, al 1 passo, gli elementi a distanza $n/2=4$ e si scambiano di posto se necessario

Ad es. la prima volta si esaminano 22 e 15 : il $15 \leq 22$
e si scambiano i 2 elementi;

poi si confrontano 35 e 50 e non si effettuano scambi, ...

Dopo questo primo passo abbiamo $n/2 (=4)$ catene di lunghezza 2 ordinate



20-24/11/20

Programmazione gr. 1 - a.a. 2020/2021 prof. Giuliano LACCETTI
shellsort

4

4

Passo successivo: confronto (ed eventuali scambi) tra elementi a distanza $n/4(=2)$

Passo successivo: confronto (ed eventuali scambi) tra elementi a distanza $n/8(=1)$

In realtà quello che dobbiamo fare è utilizzare un algoritmo di ordinamento applicato ad ogni singola "catena"

Algoritmo di insertion sort

```
inc:=n
while (inc > 1) do
  inc:= inc/2
  "insertion sort di catene a distanza inc"
endwhile
```

Il numero di catene da ordinare è sempre = **inc**; allora

```
inc:=n
while (inc > 1) do
  inc:= inc/2
  for j:= 1 to inc do
    "insertion sort di catene a distanza inc"
  endfor
endwhile
```

20-24/11/20

Programmazione gr. 1 - a.a. 2020/2021 prof. Giuliano LACCETTI
shellsort

5

5

```
inc:=n
while (inc > 1) do
  inc:= inc/2
  for j:= 1 to inc do
    k:= j+inc
    while (k<=n) do
      x:= a(k)
      "trovare la posiz. current per x"
      a(current):= x
      k:= k+inc
    endwhile
  endfor
endwhile
```

"trovare la posiz. current per x":

partendo da **current:= k** il primo elemento da confrontare con **x** si trova nella posizione precedente **previous**, con **previous:= current-inc**. I membri della catena precedenti si trovano ovviamente con **previous:= previous-inc**

20-24/11/20

Programmazione gr. 1 - a.a. 2020/2021 prof. Giuliano LACCETTI
shellsort

6

6

Nell'alg. di insertion sort per eseguire l'inserimento si usa un loop del tipo

```
while (x<a(previous)) do
```

questa volta però non abbiamo il “trucco” della sentinella, e allora dobbiamo controllare che

```
previous >= j    allora
```

```
while (previous>=j and x<a(previous)) do
```

attenzione però a questo  accesso all'array a

20-24/11/20

Programmazione gr. 1 - a.a. 2020/2021 prof. Giuliano LACCETTI
shellsort

7

7

```
while (previous >= j and not inserted) do
```

la variabile logica inserted indica se il confronto $x > a(\text{previous})$ è vero o falso, ed è aggiornata all'interno del corpo del loop.

```
inc:=n
while (inc > 1) do
  inc:= inc/2
  for j:= 1 to inc do
    k:= j+inc
    while (k<=n) do
      inserted:= false
      x:= a(k)
      current:= k
      previous:= current-inc
      while (previous>=j and not inserted)
        if(x<a(previous)) then
          a(current):=a(previous)
          current:= previous
          previous:= previous-inc
        else
          inserted:= true
        endif
      endwhile
      a(current):= x
      k:= k+inc
    endwhile
  endfor
endwhile
```

20-24/11/20

Programmazione gr. 1 - a.a. 2020/2021 prof. Giuliano LACCETTI
shellsort

8

8

Il tipo strutturato **record**

1

2

Il tipo strutturato primitivo più comune nei linguaggi di programmazione è l'array. (Rappresentazione immediata di dati organizzati "a tabella")

In molti problemi i dati presentano naturalmente una organizzazione diversa da quella tabellare, oppure è più conveniente rappresentare le relazioni tra i dati con una struttura diversa

In molte applicazioni i dati, pur essendo organizzati a tabella, non sono tutti dello stesso tipo



Tipo strutturato **record**
(tipo strutturato statico più generale)

Dichiarazione in P-like

```
var <variabile> : record  
    <campo_1>:<tipo>  
    .....  
    <campo_k>:<tipo>  
end
```

I campi campo_1, .. campo_k sono nomi che identificano le componenti individuali del record.

```
var data: record  
    giorno: integer  
    mese: character  
    anno: integer  
end  
  
var indirizzo: record  
    via:record  
        strada: character  
        numero_civico: integer  
    end  
    cap: integer  
    comune: character  
    provincia: character  
end
```

Le componenti di una variabile record sono denotabili in modo esplicito mediante i selettori di record

```
data.giorno:= 5
data.mese := 'marzo'
data.anno := 2009

indirizzo.via.strada := 'Cintia'
indirizzo.via.numero_civico:= 11
indirizzo.cap:=80126
indirizzo.comune:= 'Napoli'
indirizzo.provincia:= 'NA'
```

24/11/2020

Programmazione gr. 1- a.a. 2020/2021 - prof. Giuliano LACCETTI
record

7

7

```
var vettore: record
    dimensione : array (1..2) of integer
    componente : array (1..3) of real
end
```

```
vettore.dimensione(1):=3
vettore.dimensione(2):=1
vettore.componente(1):=0.
```

... ..

... ..

24/11/2020

Programmazione gr. 1- a.a. 2020/2021 - prof. Giuliano LACCETTI
record

8

8

La struttura record da un lato può essere vista una generalizzazione dell'array (no a vincolo di omogeneità di tipo tra componenti)

dall'altra presenta una limitazione per un accesso più rigido alle componenti (bisogna denotare sempre in maniera esplicita il selettore di record)

Il tipo derivato risulta utile nel caso di struttura
record

```
type data_g_m_a : record  
  giorno: integer  
   mese  : character  
   anno  : integer  
end
```

```
var data_nascita, data_laurea: data_g_m_a  
data_nascita.giorno := 17  
data_laurea.anno := 2009
```

tipo di dato **complex**

```
type complex: record  
    Re: real  
    Im: real  
end
```

```
var x,y,z: complex  
  
procedure somma_complex (in x,y; out:z)  
    ... ..  
    z.Re := x.Re + y.Re  
    z.Im := x.Im + y.Im  
    ... ..  
end
```

La “*creazione*” di nuovi tipi di dato può essere vista come una vera e propria “*disciplina*” che costituisce quello che si chiama Abstract Data Type (**ADT**)



Formalizzazione della creazione di un nuovo tipo di dato, seguendo l’usuale definizione per cui un tipo di dato è “caratterizzato” dai valori che esso può assumere e dalle operazioni che su di esso si possono effettuare

Esempio: ADT complex

type complex: record	
Re: real	“definizione” del
	tipo
Im: real	
end	
 function/procedure: ...	
... ..	operazioni sul
... ..	tipo
... ..	

ADT Unit (terminologia pascal)

File contenente la “definizione” del tipo e tutte le functions e/o procedures che realizzano le operazioni su di esso

(Analogia con interfacce e classi di java, con astrazione ed information hiding ...)

fine struttura record

LA COMPLESSITÀ COMPUTAZIONALE

parte 1

Materiale tratto da

A.Murli – Lezioni di Laboratorio di Programmazione e Calcolo
Cap. 5

A. Murli, G. Laccetti et al. - Laboratorio di Programmazione I par. 3.3

17/11/2020

Programmazione 9CFU - gr. 1 prof G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1 1

1

17/11/2020

Programmazione 9CFU - gr. 1 prof G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1 2

2

LA COMPLESSITÀ COMPUTAZIONALE

parte 1

Materiale tratto da

A. Murli – Lezioni di Laboratorio di Programmazione e Calcolo
Cap. 5

A. Murli, G. Laccetti et al. - Laboratorio di Programmazione I par. 3.3

17/11/2020

Programmazione 9CFU - gr. 1 prof G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1 3

3

Esempio:

Risolvere l'equazione

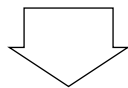
$$7x = 21$$

in un sistema aritmetico floating-point con

$$\beta = 10 \text{ e } t = 6$$

algoritmo 1

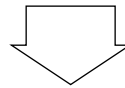
$$x = 21/7 = 3$$



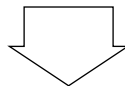
1 divisione

algoritmo 2

$$\begin{aligned} x &= 21 * 1/7 = \\ 21 * 0.142857 &= \\ 0.299997 * 10 \end{aligned}$$



1 molt. + 1 div.



l'algoritmo 1 è più efficiente

17/11/2020

Programmazione 9CFU - gr. 1 prof G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1

4

4

Esempio:

Calcolare $f(x) = x^{16}$

algoritmo 1

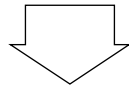
1) $x^2 = x * x$

2) $x^3 = x^2 * x$

3) $x^4 = x^3 * x$

.....

15) $x^{16} = x^{15} * x$



15 moltiplicazioni

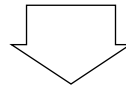
algoritmo 2

1) $x^2 = x * x$

2) $x^4 = x^2 * x^2$

3) $x^8 = x^4 * x^4$

4) $x^{16} = x^8 * x^8$



4 moltiplicazioni

L'algoritmo 2 è più efficiente

17/11/2020

Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1

5

5

In generale, calcolare

$f(x) = x^n$, $n = 2^m$

richiede



Algoritmo 1

$n - 1$ operazioni
moltiplicando x per
se stesso
ripetutamente

Algoritmo 2

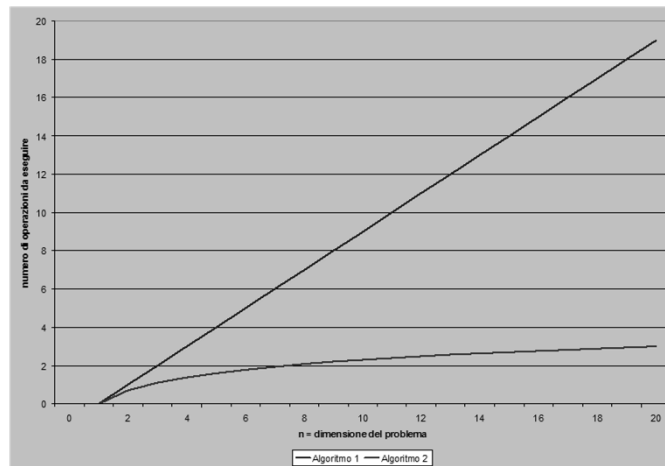
$\log_2 n$ operazioni
elevando x al
quadrato
ripetutamente

17/11/2020

Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1

6

6



Confronto tra il numero di operazioni eseguite per il calcolo di x^n , al variare di n dall'algoritmo 1 e dall'algoritmo 2

17/11/2020

Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1

7

7

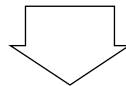
Problema

Valutare per un fissato valore di x il polinomio

$$a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

Si può procedere nel seguente modo:

$p_1 = a_3 * x * x * x$	3M
$p_2 = a_2 * x * x$	2M
$p_3 = a_1 * x$	1M
$p_4 = p_1 + p_2 + p_3 + a_0$	3A



6 moltiplicazioni e 3 addizioni

M = 1 moltiplicazione o divisione

A = 1 somma o sottrazione

17/11/2020

Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1

8

8

Caso generale

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

Algoritmo 1

```
p := a0
for i := n to 1, step -1 do
  y := x
  for j := 2 to i do
    y := y * x
  endfor
  p := p + ai * y
endfor
```

$$(n + (n-1) + \dots + 1) M = [n(n+1)/2] M$$
$$(1 + 1 + \dots + 1) A = n A$$

17/11/2020

Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1

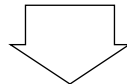
9

9

$$a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

un procedimento più efficiente:

$y_1 = x * x$	1M
$y_2 = y_1 * x$	1M
$p_1 = a_3 * y_2$	1M
$p_2 = a_2 * y_1$	1M
$p_3 = a_1 * x$	1M
$p_4 = p_1 + p_2 + p_3 + a_0$	3A



5 moltiplicazioni e 3 addizioni

17/11/2020

Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1

10

10

Caso generale

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

Algoritmo 2

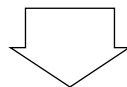
```
p := a0
y := x
for j := 1 to n do
  if (j ≠ 1) then
    y := y * x
  endif
  p := p + aj * y
endfor
```

$$(1 + 2(n-1))M = (2n-1)M$$
$$(1 + 1 + \dots + 1)A = nA$$

Un procedimento ancora più efficiente:

$$((a_3 x + a_2)x + a_1)x + a_0 = a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

$y_1 = a_3 * x$	1M
$y_2 = y_1 + a_2$	1A
$y_3 = y_2 * x$	1M
$y_4 = y_3 + a_1$	1A
$y_5 = y_4 * x$	1M
$y_6 = y_5 + a_0$	1A



3 moltiplicazioni e 3 addizioni

Caso generale

Algoritmo di HORNER

```
p := an
for i := n-1 to 0, step -1 do
    p := p * x + ai
endfor
```

$n M + n A$

**L'Algoritmo di HORNER è
il più efficiente**

17/11/2020

Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1

13

13

L'efficienza di un algoritmo
dipende dal numero di
operazioni richieste per ottenere
la soluzione del problema

17/11/2020

Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1

14

14

Problema

Memorizzazione di una matrice “sparsa”

$$A = \begin{pmatrix} 7 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 2 & 8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -6 \\ -3 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \\ -1 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 8 & 0 & 3 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

A: matrice 8 x 8

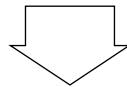
7/11/2020

Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1

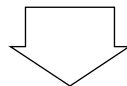
15

15

Memorizzazione mediante
array bidimensionale



Sono necessarie $8 \times 8 = 64$ locazioni di memoria



Spreco di memoria

7/11/2020

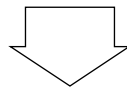
Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1

16

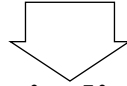
16

Memorizzazione mediante
3 array (elementi, colonna, riga)

- $A' = (7, -1, 2, 8, 1, -6, -3, 2, 4, -1, 7, 9, 8, 3, 2)$
 A' = elementi non nulli di A riga per riga
- $J = (1, 4, 1, 2, 5, 8, 1, 7, 6, 1, 4, 1, 5, 7, 3)$
 J : indice di colonna in A dell'elemento $A'(i)$
- $I = (1, 1, 2, 2, 3, 3, 4, 4, 5, 6, 6, 7, 7, 7, 8)$
 I : indice di riga in A dell'elemento $A'(i)$



Sono necessarie $15 + 15 + 15 = 45$ locazioni



Risparmio di memoria

7/11/2020

Programmazione 9CFU - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte I

17

17

Un algoritmo che utilizza
questo schema
di memorizzazione è più efficiente
di un algoritmo che utilizza lo schema di
memorizzazione mediante un array 2D

7/11/2020

Programmazione 9CFU - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte I

18

18

L'efficienza di un algoritmo
dipende dallo spazio richiesto
dai dati su cui opera

17/11/2020 Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1 21

21

Per valutare l'efficienza
di un algoritmo si misurano:

- il **numero di operazioni** effettuate
- lo **spazio di memoria** richiesto dai dati
(di input, intermedi e di output)

=

**Costo computazionale
dell'algoritmo**

17/11/2020 Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessita' Computazionale parte 1 22

22

In generale
per misurare il costo computazionale
di un algoritmo si definiscono:

- una funzione
complessità di tempo $T(n)$
- una funzione
complessità di spazio $S(n)$

n = dimensione del problema

$T(n)$ è il **numero delle operazioni
più significative** che si
effettuano nell'esecuzione
dell'algoritmo

Il tempo di esecuzione di un algoritmo
 τ , è proporzionale a $T(n)$

$$\tau = \kappa T(n) \mu \quad (\kappa = \text{cost})$$

$T(n)$	μ
# operazioni più significative	tempo di esecuzione di 1 operazione

μ è funzione del **periodo di clock**

Il periodo di clock dipende dalla tecnologia in senso stretto

Il tempo di esecuzione di un algoritmo allora dipende:

- **dall'algoritmo** (attraverso $T(n)$)
- **dal calcolatore** (attraverso μ)

$S(n)$ è il **numero delle variabili utilizzate dall'algoritmo**

Lo **spazio di memoria** occupato dai **dati di input, intermedi e di output** è **proporzionale** a $S(n)$

Complessità Computazionale

Fine Parte 1

7/11/2020 Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale parte 1 27

Complessità Computazionale

Fine Parte 1

7/11/2020 Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale parte 1 27

Complessità Computazionale

Fine Parte 1

7/11/2020 Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale parte 1 27

Complessità Computazionale

Fine Parte 1

7/11/2020 Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale parte 1 27

Complessità Computazionale

Fine Parte 1

7/11/2020 Programmazione 9CFU - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale parte 1 27

19/11/2020

LA COMPLESSITÀ COMPUTAZIONALE

parte 2

Materiale tratto da

A.Murli – Lezioni di Laboratorio di Programmazione e Calcolo
Cap. 5

A. Murli, G. Laccetti et al. - Laboratorio di Programmazione I par. 3.3

Programmazione I - gr. I prof G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

1

1

19/11/2020

Programmazione I - gr. I prof G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

2

2

LA COMPLESSITÀ COMPUTAZIONALE

parte 2

Materiale tratto da

A. Murli – Lezioni di Laboratorio di Programmazione e Calcolo
Cap. 5

A. Murli, G. Laccetti et al. - Laboratorio di Programmazione I par. 3.3

19/11/2020

Programmazione I - gr. 1 prof G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

3

3

Esempio

Valutazione di

$$p_n(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$$

- **Algoritmo 1**

$$T(n) = [n(n+1)/2] M + n A$$

$$S(n) = n + 4$$

- **Algoritmo 2**

$$T(n) = (2n-1) M + n A$$

$$S(n) = n + 4$$

- **Algoritmo di Horner**

$$T(n) = n M + n A$$

$$S(n) = n + 3$$

19/11/2020

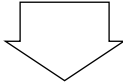
Programmazione I - gr. 1 prof G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

4

4

Hostrowski (1954) ha dimostrato che
sono necessarie **almeno**
 n moltiplicazioni e n addizioni
per valutare un polinomio
di **grado** $n \leq 4$

(In seguito tale risultato è stato esteso
ai polinomi di **grado qualsiasi**)



**L'algoritmo di HORNER
è ottimale**

19/11/2020 Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2 5

5

In generale

il costo computazionale
di un algoritmo si
valuta al **crescere**
della dimensione
del problema

19/11/2020 Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2 6

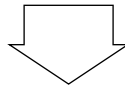
6

Esempio

Algoritmo 1

$$T(n) = (n^2/2 + n/2) M + n A$$

n	$T(n)$
2	$(2+1) M + 2 A$
10	$(50+5) M + 10 A$
100	$(5.000+50) M + 100 A$
1.000	$(500.000+500) M + 1.000 A$



**al crescere di n
il termine dominante è $n^2/2$**

19/11/2020

Programmazione I - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

7

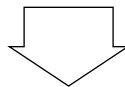
7

Esempio

Algoritmo di Horner

$$T(n) = n M + n A$$

n	$T(n)$
2	$2 M + 2 A$
10	$10 M + 10 A$
100	$100 M + 100 A$
1.000	$1.000 M + 1.000 A$



**al crescere di n
il termine dominante è n**

19/11/2020

Programmazione I - gr. 1 prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

8

8

In generale si studia
il comportamento delle funzioni
 $T(n)$ e $S(n)$ al **crescere** di n
(complessità asintotica)

ovvero

si considerano
i termini dominanti per $n \rightarrow \infty$

(generalmente trascurando
le costanti moltiplicative)

19/11/2020
Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

9

9

Esempio

- Algoritmo 1**
 $T(n) = (n^2/2 + n/2) M + n A$ $n^2/2 = \text{termine dominante}$

$T(n) = O(n^2)$
 n^2 complessità asintotica

- Algoritmo di Horner**
 $T(n) = n M + n A$ $n = \text{termine dominante}$

$T(n) = O(n)$
 n complessità asintotica

19/11/2020
Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

10

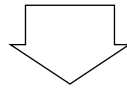
10

In generale si pone:

$$T(n) = O(f(n)) \quad (f(n) > 0)$$

se

$$\lim_{n \rightarrow \infty} T(n) / f(n) = \text{cost} \neq 0$$



$f(n)$ complessità di tempo asintotica

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

11

11

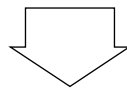
Problema

Determinare un limite superiore ed inferiore per la complessità asintotica di un algoritmo

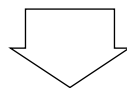
Esempio

Algoritmo 1

$$T(n) = (n^2/2 + n/2) M + n A$$



$$T(n) \leq n^2 \quad \forall n \geq 2$$
$$(T(2) = 5, \quad T(3) = 9, \quad T(4) = 14)$$



n^2 limite superiore per $T(n)$

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

12

12

se esistono due costanti $c, n_0 > 0$ tali che:

se esistono due costanti $c, n_0 > 0$ tali che:

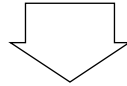
$$T(n) \leq cf(n) \quad \forall n \geq n_0$$

e

$$\forall g(n) : T(n) \leq g(n) \quad \forall n \geq n_g$$

si ha

$$g(n) \geq cf(n) \quad \forall n \geq n_g$$



$cf(n)$ limitazione inferiore per $T(n)$

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

13

13

se esistono due costanti $c, n_0 > 0$ tali che:

se esistono due costanti $c, n_0 > 0$ tali che:

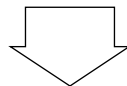
$$T(n) \leq ch(n) \quad \forall n \geq n_0$$

e

$$\forall g(n) : T(n) \leq g(n) \quad \forall n \geq n_g$$

si ha

$$g(n) \leq ch(n) \quad \forall n \geq n_g$$



$ch(n)$ limitazione superiore per $T(n)$

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

14

14

Limitazioni sulle dimensioni dei problemi risolvibili da algoritmi con alcuni valori tipici di $T(n)$

compl. di tempo massima dimensione (approx.) del problema risolvibile in

$T(n)$	1 sec	1 min	1 ora
$\log_2 n$	$2^{10^{14}}$	$2^{6 \cdot 10^{15}}$	$2^{3.6 \cdot 10^{17}}$
n	10^{14}	$6 \cdot 10^{15}$	$3.6 \cdot 10^{17}$
$n \log_2 n$	$3 \cdot 10^{12}$	$1.5 \cdot 10^{14}$	$1.3 \cdot 10^{16}$
n^2	10^7	$8 \cdot 10^7$	$6 \cdot 10^8$
n^3	$4.8 \cdot 10^4$	$3.7 \cdot 10^5$	$7.5 \cdot 10^5$
2^n	43	54	63
$n!$	16	18	19

Su un calcolatore con velocità di 100 Tflops (10^{14} operazioni floating-point al sec.)

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

15

15

Algoritmo = tecnologia ! - 1

- Problema di dimensione 10^{10}
- Operazione principale : flop (floating-point operation)
- Algoritmo con $T(n)=O(n^2) \approx n^2$ flop
- Algoritmo con $T(n)=O(n \log_2 n) \approx n \log_2 n$ flop
- PC 10 Gflops
- SC 100 Tflops (=100000 Gflops)

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

16

16

Algoritmo = tecnologia ! - 2

Algoritmo con $T(n) = O(n^2)$ ($n=10^{10}$)

$(10^{10})^2 \text{ flop} / 10^{10} \text{ flops} = 10^{10} \text{ sec}$ PC (≈ 300 anni)

$(10^{10})^2 \text{ flop} / 10^{14} \text{ flops} = 10^6 \text{ sec}$ SC (≈ 12 giorni)

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2 17

17

Algoritmo = tecnologia ! - 3

Algoritmo con $T(n) = O(n \log_2 n)$ ($n=10^{10}$)

$10^{10} \times \log_2 10^{10} \text{ flop} / 10^{10} \text{ flops} \approx 23 \text{ sec}$ PC

$10^{10} \times \log_2 10^{10} \text{ flop} / 10^{14} \text{ flops} \approx 0.002 \text{ sec}$ SC

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2 18

18

Algoritmo = tecnologia ! - 4

Un miglioramento delle prestazioni si ottiene

- con un miglioramento tecnologico in senso stretto (da PC a SC)
- con un miglioramento dell'algoritmo (da $O(n^2)$ a $O(n \log_2 n)$)

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

19

19

E' semplice constatare che un algoritmo la cui struttura è

- ```
for i:=1 to n do
 q operazioni dominanti
endfor
```

ha una complessità di tempo  $T(n)=q n$ , cioè proporzionale a  $n$

- ```
for i:= 1 to n do
  q operazioni dominanti
endfor
for j:= 1 to n do
  q operazioni dominanti
endfor
```

ha una complessità di tempo $T(n)=2q n$, cioè proporzionale a n

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

20

20

E' semplice constatare che un algoritmo la cui struttura è

- ```

 for i:= 1 to n do
 for j:= 1 to n do
 q operazioni dominanti
 endfor
 endfor

```

ha una complessità di tempo  $T(n)=q n^2$ , cioè proporzionale a  $n^2$

- ```

      for i:= 1 to n do
        for j:= 1 to i do
          q operazioni dominanti
        endfor
      endfor

```

ha una complessità di tempo $T(n)=q+2q+3q+\dots+nq = q \frac{n(n+1)}{2}$, cioè proporzionale a n^2

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

21

21

E' semplice constatare che un algoritmo la cui struttura è

- ```

 for i:= 1 to n do
 for j:= 1 to n do
 for k:= 1 to n do
 q operazioni dominanti
 endfor
 endfor
 endfor

```

ha una complessità di tempo  $T(n)=q n^3$ , cioè proporzionale a  $n^3$ .

- ```

      i := 0
      repeat
        i := i+1
        q operazioni dominanti
      until 2i >= n

```

ha una complessità di tempo $T(n)=q \log_2 n$, cioè proporzionale a $\log_2 n$.

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

22

22

Esempi di semplici algoritmi e relativa complessità

Prodotto scalare – vettore - $O(n)$

```
... ..  
for i := 1 to n do  
    B(i) := a * B(i)  
endfor
```

Prodotto matrice – vettore - $O(n^2)$ ($O(n*m)$)

```
for i := 1 to n do  
    C(i) := 0  
    for j := 1 to m do  
        C(i) := C(i) + A(i, j) * B(j)  
    endfor  
endfor
```

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2 23

23

Esempi di semplici algoritmi e relativa complessità

Prodotto matrice – matrice - $O(n^3)$ ($O(n*p*m)$)

```
for i := 1 to n do  
    for k := 1 to p do  
        C(i, k) := 0  
        for j := 1 to m do  
            C(i, k) := C(i, k) + A(i, j) * B(j, k)  
        endfor  
    endfor  
endfor
```

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

24

24

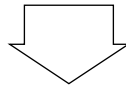
Ordinamento di un array - algoritmo di exchange sort

Nel caso di un array di lunghezza n con gli elementi in ordine decrescente (caso peggiore) sono necessari

$$\underbrace{(n-1)}_{\text{passo1}} + \underbrace{(n-2)}_{\text{passo2}} + \dots + \underbrace{1}_{\text{passo } n-1} \text{ confronti}$$

e

$$\underbrace{(n-1)}_{\text{passo1}} + \underbrace{(n-2)}_{\text{passo2}} + \dots + \underbrace{1}_{\text{passo } n-1} \text{ scambi}$$



$$n(n-1) / 2 \text{ confronti e scambi}$$

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

25

25

Nel caso di un array di lunghezza n con gli elementi in ordine crescente (caso migliore) sono necessari

$$(n-1) \text{ confronti}$$

e

$$0 \text{ scambi}$$

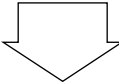
19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

26

26

Per alcuni algoritmi la complessità
di tempo dipende dai dati di input
oltre che
dalla dimensione del problema



la complessità di tempo è
compresa fra quella del caso
migliore e quella del caso peggiore

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

27

27

COMPLESSITÀ COMPUTAZIONALE

Fine Parte 2

19/11/2020

Programmazione I - gr. I prof. G. Laccetti a.a. 2020/2021 - Complessità Computazionale - parte 2

28

28

Complessità computazionale – Come «contare» il numero di operazioni in due casi tipici

```
for i:= 1 to n do
  for j:=1 to n do
    operazione
  endfor
endfor
```

Il ciclo interno con indice j viene ripetuto n volte, in corrispondenza degli n valori che l'indice del ciclo, j , assume, da 1 a n . Questo accade per ciascun valore dell'indice del ciclo esterno, i , che assume n valori, da 1 a n . Quindi n valori di j per ciascun valore di i ; in totale quindi l'operazione interna al ciclo più interno, viene ripetuta n volte per ciascun valore di i ; **quindi in totale $n \cdot n$ volte, n^2 volte.**

```
for i:= 1 to n do
  for j:= i+1 to n do
    operazione
  endfor
endfor
```

Il ciclo interno con indice j viene ripetuto un numero di volte che dipende dal valore dell'indice del ciclo esterno, i ; per calcolare il numero di volte che l'operazione viene eseguita, si DEVE RAGIONARE ESCLUSIVAMENTE IN QUESTO MODO
per i che vale 1 il ciclo interno con indice j , e quindi l'operazione al suo interno, viene ripetuto $n-1$ volte, in corrispondenza degli $n-1$ valori che assume j , a partire da 2 fino ad n ;
per i che vale 2 ... $n-2$ volte;
per i che vale 3 ... $n-3$ volte;
.... ;
per i che vale $n-2$... 2 volte;
per i che vale $n-1$... 1 volta;
per i che vale n ... 0 volte (in quanto il valore di partenza di j , $n+1$, è > del valore di arrivo, n).

L'operazione all'interno del ciclo più interno quindi viene eseguita in totale:

$(n-1) + (n-2) + (n-3) + \dots + 2 + 1$ volte, cioè la somma dei primi $n-1$ numeri naturali, la cui espressione in forma chiusa è $n \cdot (n-1) / 2 = n^2/2 - n/2$

ALCUNE NOTE SUL CALCOLO DI FORMULE RICORRENTI

Corso di Programmazione 9CFU gr. 1 a.a. 2020-2021

Prof Giuliano Laccetti

(argomenti discussi il 5 e il 6 novembre 2020)

I problemi del calcolo di sommatorie e di produttorie (cioè una sequenza di prodotti) rientrano nell'ambito dei problemi descrivibili da *formule ricorrenti lineari*, cioè da relazioni della forma:

$$y_i = a_i y_{i-1} + b_i \quad (3.3.1)$$

$i > 0$; a_i, b_i dati; $y_0 =$ un valore prefissato

La (3.3.1) è una regola che definisce una successione di valori; ogni valore è calcolabile in termini dei valori precedenti. Il primo valore (y_0) è prefissato ed è detto *condizione iniziale*. La (3.3.1) è una formula ricorrente del *primo ordine*, poichè ogni valore dipende solo dal valore al passo precedente.

Supponiamo per semplicità che $a_i = a$, $b_i = b$ per tutti gli i . La seguente procedura visualizza sul dispositivo di output i primi $n+1$ valori della (3.3.1)

dati di input: n, y_0, a, b

dato di output: nessuno

costrutto ripetitivo: for

operazione ripetuta (al generico passo i): applicazione della formula ricorrente

```
procedure frlc_1(in:n,a,b,y_zero)
var a,b,y_zero,y: real
var n: integer
begin
    y := y_zero
    print y
    for i :=1 to n do
        y := a*y+b
        print y
    endfor
end
```

In modo analogo la function `frlcn_1` calcola il valore y_n dell' n -simo elemento della successione generata dalla (3.3.1).

```
function frlcn_1(n,a,b,y_zero): real
var a,b,y_zero: real
var n: integer
begin
    frlcn_1 := y_zero
    for i:=1 to n do
        frlcn_1 := a*frlcn_1+b
    endfor
end
```

Si noti che non è necessario memorizzare tutti i valori calcolati, ma solo il precedente. Ciò è evidenziato nella seguente versione della procedura `frlc_1`, che fornisce sul dispositivo di uscita anche il modulo della differenza tra due valori consecutivi.

```
procedure frlc_1(in:n,a,b,y_zero)
var a,b,y_zero,y_att,y_prec: real
var n: integer
begin
    y_prec := y_zero
    print y_prec
    for i:=1 to n do
        y_att := a*y_prec+b
        print y_att,abs(y_att-y_prec)
        y_prec := y_att
    endfor
end
```

```

        endfor
    end

```

Poichè la formula ricorrente è del primo ordine, la *memoria* necessaria all'algoritmo è costituita dalla sola variabile `y_prec`, il cui valore viene aggiornato al termine di ogni passo del ciclo.

Una formula ricorrente (lineare) del secondo ordine ha la forma:

$$y_i = ay_{i-1} + by_{i-2} + c$$

$$y_0, y_1 = \text{due valori prefissati}$$

dove per semplicità a, b e c sono costanti. Si noti che una formula del secondo ordine ha due condizioni iniziali.

La seguente procedura visualizza sul dispositivo di uscita i primi n elementi della successione y_i (e $|y_i - y_{i-1}|$).

```

procedure frlc_2(in:n,a,b,c,y_zero,y_uno)
var a,b,c,y_zero,y_uno,y_att,y_prec_1,y_prec_2: real
var n: integer
begin
    y_prec_2 := y_zero
    y_prec_1 := y_uno
    print y_prec_1,y_prec_2
    for i:=1 to n do
        y_att := a*y_prec_1+b*y_prec_2+c
        print y_att,abs(y_att-y_prec_1)
        y_prec_2 := y_prec_1
        y_prec_1 := y_att
    endfor
end

```

Poichè la formula ricorrente è del secondo ordine, la *memoria* necessaria all'algoritmo è costituita dalle sole due variabili `y_prec_1` e `y_prec_2`, i cui valori vengono aggiornati al termine di ogni passo del ciclo.

Una formula ricorrente del secondo ordine molto nota è la formula di Fibonacci¹:

$$y_i = y_{i-1} + y_{i-2}$$

$$i \geq 2, y_0 = 0, y_1 = 1.$$

Il valore y_i è detto i -simo numero di Fibonacci.

I primi n numeri di Fibonacci possono essere visualizzati utilizzando la procedura generale `frlc_2` richiamata con i seguenti valori dei parametri attuali

```
frlc_2(n,1.0,1.0,0.0,1.0)
```

Data l'importanza della formula di Fibonacci, di seguito viene presentata una function specifica, la function `fibonacci`, che calcola l' n -simo numero di Fibonacci utilizzando la formula ricorrente.

```
function fibonacci(n): integer
```

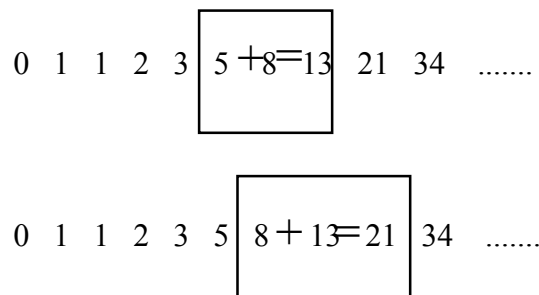
¹ dal nome del matematico italiano Leonardo Pisano, detto Fibonacci (filius Bonacci), che la presentò nel 1202, nel suo libro Liber Abbaci.

```

var n, fib_prec_1, fib_prec_2: integer
begin
  if n=1 or n=0
  then
    fibonacci := n
  else
    fib_prec_1 := 1
    fib_prec_2 := 0
    for i:=2 to n do
      fibonacci := fib_prec_1+fib_prec_2
      fib_prec_2 := fib_prec_1
      fib_prec_1 := fibonacci
    endfor
  endif
end

```

La sequenza computazionale descritta dall'algorithm precedente è sintetizzata graficamente in figura.



E' banale generalizzare quanto detto finora al caso di formule ricorrenti lineari di ordine superiore a due.

Il discorso può anche estendersi al caso non lineare, cioè a una relazione del tipo

$$y_i = f(y_{i-1}, i) \quad i > 0; \quad f \text{ data}; \quad y_0 = \text{un valore prefissato}, \quad (3.3.2)$$

detta *formula ricorrente non lineare* del primo ordine, e a

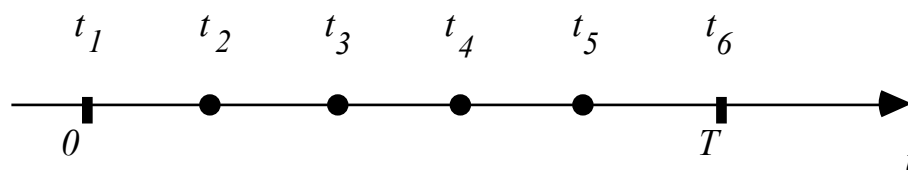
$$y_i = f(y_{i-1}, y_{i-2}, \mathbf{K}, y_{i-k}, i) \quad (3.3.3)$$

$$i > 0; \quad f \text{ data}; \quad y_0, y_1, \mathbf{K}, y_{k-1} = k \text{ valori prefissati},$$

che è una *formula ricorrente non lineare di ordine k*; la funzione f è la *funzione di iterazione* della formula ricorrente.

L'applicazione dell'idea incrementale alla soluzione di (3.3.3), e quindi anche di (3.3.1) e (3.3.2), è detta *approccio iterativo*, e il conseguente algoritmo è un *algoritmo iterativo*.

Una formula ricorrente è anche detta *equazione alle differenze* (il termine nasce dalla presenza delle differenze $i-1, \mathbf{K} i-k$) o anche *sistema dinamico a tempo discreto* (il termine nasce dal fatto che l'indice i può essere inteso come una discretizzazione del tempo) e ha una grande importanza come strumento per costruire modelli matematici di fenomeni reali. La soluzione è la successione dei valori y_i , che può sempre essere calcolata mediante un algoritmo iterativo, se sono note le condizioni iniziali. In questo senso, l'iterazione è una struttura temporale. In particolare, quando si parla di tempo discreto si intende che, invece di considerare un certo intervallo temporale, per esempio $[0, T]$, si utilizza una griglia definita su tale intervallo.



griglia temporale uniforme con sei punti

I valori y_i possono essere considerati come una campionatura (sulla griglia) di una funzione definita sull'intervallo $[0, T]$, cioè y_i è il valore della funzione al tempo (discreto) t_i .

Per esempio, la formula di Fibonacci fu introdotta per risolvere il seguente problema: quante coppie di conigli possono essere prodotte da una singola coppia in un anno, nell'ipotesi che ogni coppia generi una nuova coppia ogni mese, che ogni nuova coppia sia fertile dopo un mese e che nessun coniglio muoia?

La formula di Fibonacci² è la descrizione (*modello matematico*) dell'evoluzione, a tempo discreto, di una popolazione di conigli; t_i è l' i -simo mese e y_i è il numero di coppie di conigli all' i -simo mese. La popolazione è determinata da due fattori diversi: primo, poichè nessun coniglio muore, tutte le coppie di conigli y_{i-1} in vita nel mese precedente lo saranno ancora nel mese successivo; secondo, ogni coppia di sufficiente anzianità genera una nuova coppia: il numero di queste coppie è y_{i-2} .

Un altro famoso esempio di modello a tempo discreto per descrivere l'evoluzione di una popolazione è quello introdotto da Malthus³. Egli affermò che una popolazione, in assenza di fattori che ne limitano la crescita, cresce in un dato periodo di tempo con una rapidità fissata e l'incremento è proporzionale al numero di individui. Ciò significa che se una popolazione passa in un anno da 100 a 150 individui, allora nello stesso periodo di tempo passerebbe da 1000 a 1500 individui. La formula ricorrente che descrive questo comportamento è

$$y_i = y_{i-1} + 0.5 \cdot y_{i-1}, \text{ con } y_0 = \text{un valore prefissato}$$

La quantità 0.5 è il *tasso relativo di crescita* annuale della popolazione (spesso dato in termini percentuali, 50% nell'esempio); y_i è la popolazione all' i -simo anno.

In generale, la formula di Malthus a tempo discreto è

$$y_i = y_{i-1} + a \cdot h \cdot y_{i-1}$$

con y_0 = condizione iniziale

dove h è il passo di griglia (cioè l'intervallo di tempo tra due elementi consecutivi della griglia, supposta uniforme) e a è il tasso relativo di crescita per unità di tempo.

Si noti che se $a < 0$ allora la popolazione decresce e a è detto *tasso di decrescita*.

² Le applicazioni e le proprietà della formula di Fibonacci sono incredibilmente numerose. Esiste una rivista scientifica, il cui titolo è *The Fibonacci Quarterly*, interamente dedicata all'argomento.

³ Thomas R. Malthus (1766-1834), economista inglese considerato il fondatore dello studio quantitativo delle popolazioni con il suo *Essay on the principle of population as it affects the future improvement of society* (1798).

Le equazioni alle differenze possono anche essere risolte in modo diretto, invece di ricorrere a un algoritmo iterativo. Nel caso lineare (e del primo ordine) esiste un modo molto semplice per esprimere la soluzione attraverso una formula diretta, per qualunque fissato i .

Per esempio, la formula $y_i = 5y_{i-1}$, con $y_0 = 1$ dà luogo a

$$y_0 = 1$$

$$y_1 = 5y_0 = 5 \cdot 1 = 5$$

$$y_2 = 5y_1 = 5 \cdot 5y_0 = 5^2 y_0 = 25 \cdot 1 = 25$$

$$y_3 = 5y_2 = 5 \cdot 5y_1 = 5 \cdot 5 \cdot 5y_0 = 5^3 y_0 = 125$$

L

Quindi la soluzione generale è $y_i = 5^i y_0$.

La formula $y_i = ay_{i-1} + b$, con $y_0 = c$ dà luogo a

$$y_0 = c$$

$$y_1 = ay_0 + b = ac + b$$

$$y_2 = ay_1 + b = a^2c + ab + b$$

$$y_3 = a^3c + a^2b + ab + b$$

L

Ricordando l'espressione della sommatoria geometrica (vedi par.3.1.6), si ha che la soluzione generale è

$$y_i = \begin{cases} c + ib & , \quad a = 1 \\ a^i c + \frac{1-a^i}{1-a} b & , \quad a \neq 1 \end{cases}$$

La formula ricorrente di Fibonacci ha la soluzione data in modo diretto dalla seguente espressione, che non verrà qui derivata (vedi per esempio T.H. Cormen, C.E. Leiserson, R.L. Rivest "Introduzione agli algoritmi", Jackson Libri, citato in bibliografia)

$$y_i = \left(\frac{1+\sqrt{5}}{2} \right)^i + \left(\frac{1-\sqrt{5}}{2} \right)^i \approx 2^{0.694i}$$

In generale, è possibile risolvere in modo diretto le formule ricorrenti lineari di qualsiasi ordine, mentre non esistono metodi generali per esprimere la soluzione di formule ricorrenti non lineari.

Consideriamo alcuni semplici esempi di problemi descritti da una formula ricorrente.

ESEMPIO:

il gioco della catena di Sant'Antonio consiste nel

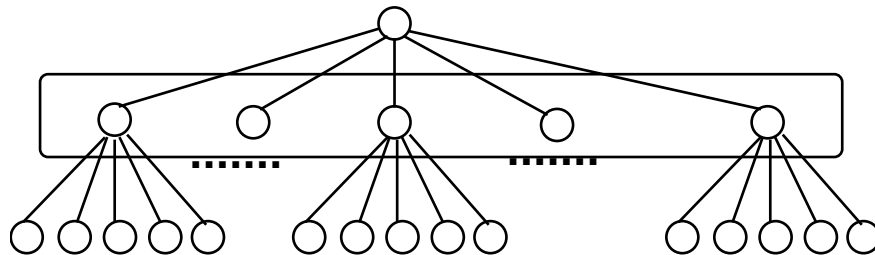
ricevere una lettera con una lista di, diciamo, sei nomi e i relativi indirizzi;

inviare 10 euro al primo nome della lista;

cancellare il primo nome della lista e aggiungere il proprio nome alla fine;

inviare una copia della lettera così modificata ad altri cinque conoscenti.

La dinamica del gioco può essere visualizzata utilizzando una struttura ad albero per organizzare le varie lettere che vengono inviate durante il gioco, tra le quali esiste appunto una relazione gerarchica.



Ogni elemento dell'albero rappresenta in questo caso una lettera. Le lettere che appaiono in un certo livello dell'albero sono dette lettere della stessa generazione (quelle evidenziate dal rettangolo in figura sono della prima generazione).

Supponiamo di voler calcolare quante lettere ci sono alla i -sima generazione; da tale valore dipende la quantità di denaro che guadagneremmo partecipando al gioco.

Denotiamo con y_i il numero di lettere della generazione i e conveniamo di denotare con y_0 l'unica lettera che riceviamo (cioè $y_0 = 1$); questa rappresenta l'inizio della nostra partecipazione al gioco e costituisce il livello 0 dell'albero (la radice dell'albero). Il numero delle lettere che noi spediamo è allora y_1 , le lettere spedite dai conoscenti che contattiamo è y_2 , e così via, generazione dopo generazione. La relazione che lega una generazione alla successiva è

$$y_i = 5y_{i-1}.$$

Questa è una formula ricorrente lineare del primo ordine, la cui soluzione per qualunque fissato valore di i può essere calcolata⁴ da:

Algoritmo – catena di S. Antonio

```
begin catena_S.A.
  var a,b,y_zero: real
  var frlcn_1: real function
  var i: integer
  begin
    a := 5.0
    b := 0.0
    y_zero := 1.0
    read i
    print frlcn_1(i,a,b,y_zero)
  end
end catena_S.A.
```

⁴ La soluzione può essere espressa in modo diretto come $y_i = 5^i y_0$. Se si è interessati a sapere quale cifra si guadagnerebbe partecipando al gioco, basta osservare che il denaro viene spedito solo al primo della lista e quindi dobbiamo conoscere il valore di y_i con i = generazione_in_cui_il_nostro_nome_è_al_primo_posto. Ciò accade per $i = 6$ e quindi il denaro che riceveremmo è $5^6 \cdot 10$ euro = 156.250 euro

ESEMPIO:

La velocità operativa dei processori sul mercato cresce approssimativamente del 50% all'anno. Allora detta y_i la velocità operativa della generazione di processori dell'anno i , l'evoluzione della velocità operativa è descritta da

$$y_i = (1 + 0.5)y_{i-1}$$

Questa è una formula ricorrente lineare del primo ordine di tipo Malthus.

Lo stato della velocità operativa dei processori al generico anno i , nell'ipotesi che il tempo iniziale sia l'anno 1996 e $y_0 = 200$ (Mhz), è calcolato dall'algoritmo :

Algoritmo - stima velocità media processori-ver. 1

```
begin velocita_processori
  var a,b,y_zero: real
  var frlcn_1: real function
  var i: integer
  begin
    a := 1.0+0.5
    b := 0.0
    y_zero := 2E2
    read i
    print frlcn_1(i-1996,a,b,y_zero)
  end
end velocita_processori
```

Il modello precedente può essere reso più realistico. Di seguito sono riportati alcuni dati relativi alla potenza “media” dei processori Intel dal 1985 al 2005.

1985	5 MHz	8088
1986	6 Mhz	80286
1987	13 Mhz	80286
1988	20 Mhz	80386
1989	25 Mhz	80386
1990	25 Mhz	80386
1991	33 Mhz	80486
1992	66 Mhz	80486
1993	66 Mhz	80486
1994	90 Mhz	80486
1995	120Mhz	pentium
1996	200Mhz	pentium
1997	300Mhz	Pentium
1998	400Mhz	Pentium 2
1999	500Mhz	Pentium 3
2000	1 GHz	Pentium 3
2001	1.3 GHz	Pentium 4
....
2005	3.8 GHz	Pentium X

Si noti che fino ai primi anni 90 la crescita è stata più lenta del 50% annuo. Infatti la rapidità di crescita è stata del 35% annuo fino al 93 e poi è diventata del 50%. Dal 2000/2001 in poi, inoltre, tale rapidità di crescita è diminuita ancora, diciamo è arrivata al 30% circa. Il seguente algoritmo realizza tale modello di crescita:

Algoritmo – stima velocità media processori-ver. 2

```
begin velocita_processori
  var a,y: real
  var i: integer
  begin
    y := 5.0
    i:= 1985
    print y
    a := 1.0+0.35
    for i := 1986 to 1993 do
      y := a*y
      print i, y
    endfor
    a := 1.0+0.5
    for i := 1994 to 2000 do
      y := a*y
      print i,y
    endfor
    a := 1.0+0.3
    for i := 2001 to 2005 do
      y := a*y
      print i,y
    endfor
  end
end velocita_processori
```

I risultati sono sintetizzati in tabella :

1985	5 Mhz
1986	7 Mhz
1987	9 Mhz
1988	12 Mhz
1989	17 Mhz
1990	24 Mhz
1991	30 Mhz
1992	41 Mhz
1993	55 Mhz
1994	83 Mhz
1995	124 MHz
1996	186 MHz
1997	279 MHz
1998	418 MHz
1999	628 MHz
2000	942 MHz
2001	1,2 GHz
.....
2005	3,5 GHz

ESEMPIO:

un modello (elementare) ampiamente accettato di crescita della concentrazione di anidride carbonica in atmosfera è il seguente: fino agli inizi della Rivoluzione industriale (circa 1870) la concentrazione aumenta approssimativamente dello 0.03% all'anno, mentre dopo tale data la rapidità di crescita non è più costante ma cresce anch'essa all'incirca del 2.5% all'anno.

Detta y_i la concentrazione di anidride carbonica all'anno i , si ha

$$y_i = (1 + c_0)y_{i-1} \quad , \text{ anno } < 1870, c_0 = 0.0003$$

mentre

$$c_i = (1 + 0.025)c_{i-1} \quad , \text{ anno } \geq 1870$$

$$y_i = (1 + c_i)y_{i-1} \quad , \text{ anno } \geq 1870$$

Algoritmo – concentrazione CO₂

```
begin concentrazione_C02
  var a,r: real
  var frlcn_1: real function
  var i,anno_iniziale: integer
  begin
    c := 0.0003
    % concentrazione iniziale 270 parti per milione (ppm)
    anno_iniziale := 1750
    y := 270.0
    for i := anno_iniziale to 1869
      y := (1+c)*y
      print y
    endfor
    r := 1.025
    for i := 1870 to 2030
      c := r*c
      y := (1+c)*y
      print y
    endfor
  end
end concentrazione_C02
```

ESEMPIO:

sviluppare un algoritmo per calcolare il saldo annuale y_i di un conto bancario, nell'ipotesi che il tasso di interesse annuo pagato dalla banca sia p e che ogni anno sul conto venga depositata una quantità fissa b di denaro.

La relazione che lega il saldo di un anno al saldo dell'anno precedente è:

$$y_i = (1 + p)y_{i-1} + b$$

Questa è una formula ricorrente lineare del primo ordine. La quantità b è detta *input* o *forzante*.

Il saldo dopo 5 anni, di un conto attivato con 5.000 euro, interesse del 2% e deposito annuale di 2.000 euro è calcolato dall'algoritmo:

Algoritmo – saldo annuale di un c/c bancario

```
begin saldo
  var a,b,y_zero: real
  var frlcn_1: real function
  var i: integer
  begin
    a := 1.0+0.02
    b := 2E3
    y_zero := 5E3
    i := 5
    print frlcn_1(i,a,b,y_zero)
  end
end saldo
```

ESEMPIO:

Il modello di Malthus descrive l'evoluzione di una popolazione quando non ci sono fattori che ne inibiscono la crescita. In molti casi l'evoluzione di una popolazione è frenata dall'effetto della sovrappopolazione, per esempio quando le risorse alimentari sono limitate. Un modello di evoluzione a tempo discreto che tiene conto degli effetti della sovrappopolazione fu proposto da Verhulst⁵ e prende il nome di formula *logistica discreta*.

$$y_i = y_{i-1} + a \cdot y_{i-1} \cdot \left(1 - \frac{y_{i-1}}{L}\right).$$

Questa è una formula ricorrente non lineare del primo ordine; a è il tasso relativo di crescita in assenza di fattori inibitori; L è la cosiddetta *popolazione limite* o *capacità portante*⁶.

La seguente function calcola il numero di individui al tempo discreto n di una popolazione che evolve secondo la formula logistica.

Algoritmo – formula logistica

```
function logistica(n,a,L,y_zero): real
var a,L,y_zero: real
var n: integer
begin
  logistica := y_zero
  for i:=1 to n do
    logistica := logistica+a*logistica*(1.0-logistica/L)
  endfor
end
```

ESEMPIO:

la somma cumulativa di un vettore x è un vettore y la cui i -sima componente è la somma delle prime i componenti di x .

Le componenti di y sono date dalla seguente formula ricorrente

$$\begin{aligned} y_0 &= x_0 \\ y_i &= y_{i-1} + x_i, i > 0 \end{aligned}$$

⁵ Pierre Verhulst (1804-1849), matematico belga autore di *Recherches mathématiques sur la loi d'accroissement de la population*.

⁶ La quantità $-ay_{i-1}^2 / L$ indica l'effetto inibitore (segno negativo) della sovrappopolazione, modellizzato dal numero degli incontri intraspecie che (per un principio noto come *azione di massa*) è proporzionale al quadrato del numero di individui.

In molte applicazioni, il vettore x è detto *segnale di ingresso*, il vettore y è il *segnale di uscita* e l'equazione alle differenze è chiamata *il filtro*.

Algoritmo – somma cumulativa

```
procedure somma_cumulativa(in:x,n; out:y)
var n: integer
var x,y: array(0..n) of real
begin
    y(0) := x(0)
    for i:=1 to n do
        y(i) := y(i-1)+x(i)
    endfor
end
```

ESEMPIO:

la *media mobile* di un vettore x è un vettore y la cui i -sima componente è la media di alcune componenti di x .

Nel caso della media mobile di ordine 4, le componenti di y sono date dalla seguente formula ricorrente:

$$y_1 = x_1$$

$$y_2 = \frac{1}{2}(x_2 + x_1)$$

$$y_3 = \frac{1}{3}(x_3 + x_2 + x_1)$$

$$y_i = \frac{1}{4}(x_i + x_{i-1} + x_{i-2} + x_{i-3}) \quad , \quad i > 3$$

La media mobile è un classico esempio di filtro, che a partire dal segnale di ingresso x genera il segnale di uscita y .

Algoritmo – media mobile

```
procedure media_mobile(in:x,n,ordine_media; out:y)
var n,ordine_media: integer
var somma: real
var x,y: array(1..n) of real
begin
    somma_cumulativa(x(1..ordine_media),ordine_media,...
                    y(1..ordine_media))
    somma := y(ordine_media)
    for i:=1 to ordine_media do
        y(i) := y(i)/float(i)
    endfor
    for i:=ordine_media+1 to n do
        somma := somma+x(i)-x(i-ordine_media)
        y(i) := somma/float(ordine_media)
    endfor
end
```

Una formula ricorrente può anche essere applicata *all'indietro*, cioè fissando un valore y_n e ricavando iterativamente tutti i valori fino a y_1 .

Introduzione al calcolo matriciale - parte 1

•Queste slides derivano da lezioni di vari corsi di Calcolo Numerico e Programmazione tenuti dal prof. A. Murli e, sotto diversa forma, parzialmente presenti in:

A.Murli, G.Giunta, G.Laccetti, M.Rizzardi
Laboratorio di Programmazione I

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

1

1

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

2

2

Introduzione al calcolo matriciale

- nuclei computazionali di base
- algoritmi di back e forward substitution

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

3

3

Queste slides derivano da lezioni di vari corsi di Calcolo Numerico e Programmazione tenuti dal prof. A. Murli e, sotto diversa forma, parzialmente presenti in:

A.Murli, G.Giunta, G.Laccetti, M.Rizzardi
Laboratorio di Programmazione I

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

4

4

Parte1 - Nuclei computazionali di base

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

5

5

• calcolo prodotto scalare tra 2 vettori

Il prodotto scalare tra 2 vettori x ed y di n elementi
ciascuno è il numero

$$s = x^T y = \sum_{i=1}^n x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

A diagram illustrating the dot product of two vectors x and y . On the left, a horizontal rectangle is labeled x^T . To its right is a vertical rectangle labeled y . An equals sign follows, and then a small square is labeled s .

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

6

6

Algoritmo per il calcolo del prodotto scalare tra 2 vettori

```
function prod_scal (in: x,y,n) : real
  var i, n: integer
  var x,y : array [1..n] of real
  begin
    prod_scal := 0.
    for i:= 1 to n do
      prod_scal := prod_scal +x(i)*y(i)
    endfor
  end
end prod_scal
```

17/11/2020

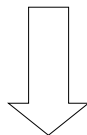
Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

7

7

Algoritmo per il calcolo del prodotto scalare tra 2 vettori

complessità di tempo: n A , n M



complessità di tempo del prodotto scalare tra due vettori

$$T(n) = O(n)$$

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

8

8

- calcolo di una saxpy

Una operazione saxpy è la somma di un multiplo di un vettore e di un altro vettore

$$z = \alpha x + y$$

Con x e y vettori di n componenti, α scalare

$$z = \alpha x + y = (\alpha x_1 + y_1, \alpha x_2 + y_2, \dots, \alpha x_n + y_n)$$



In genere il calcolo si effettua *in place*, cioè il risultato va nello stesso vettore y , cioè $y = \alpha x + y$

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

9

9

Algoritmo per il calcolo di una saxpy

```
procedure saxpy (in:  $\alpha, x, n$ ; in/out:  $y$ )  
  var  $i, n$  : integer  
  var  $\alpha$  : real  
  var  $x, y$  : array [1.. $n$ ] of real  
  begin  
    for  $i := 1$  to  $n$  do  
       $y(i) := y(i) + \alpha x(i)$   
    endfor  
  end  
end saxpy
```

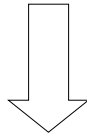
17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

10

10

Algoritmo per il calcolo di una saxpy
complessità di tempo: $n A, n M$



complessità di tempo dell'operazione saxpy
 $T(n) = O(n)$

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

11

11

calcolo della somma di 2 matrici

$C = A + B$; A e B matrici di dimensioni $m \times n$, (di conseguenza anche C avrà queste dimensioni)

$$c_{ij} = a_{ij} + b_{ij} \quad i = 1, \dots, m, \quad j = 1, \dots, n$$

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

12

12

Algoritmo per il calcolo della somma di 2 matrici

```
procedure somma_matrici (in:m,n,A,B; out:C)
var m,n,i,j : integer
var A,B,C : array [1..m, 1..n] of real
begin
  for i := 1 to m do
    for j := 1 to n do
      C(i,j) := A(i,j) + B(i,j)
    endfor
  endfor
end
end somma_matrici
```

17/11/2020

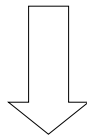
Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

13

13

Algoritmo per il calcolo della somma di 2 matrici

complessità di tempo: $n \times m$



complessità di tempo della somma di 2 matrici supposte
entrambi di dimensioni $n \times n$

$$T(n) = O(n^2)$$

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

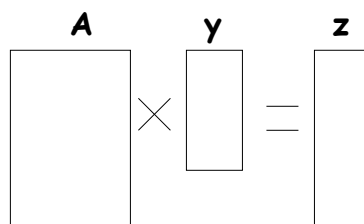
14

14

calcolo del prodotto di una matrice per un vettore

**$Ay = z$ con A matrice di dimensioni $m \times n$ e
 y vettore di n componenti, z vettore di m componenti**

$$z_i = \sum_{j=1}^n a_{ij} y_j \quad i = 1, \dots, m$$



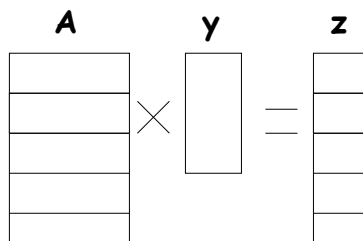
17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

15

15

**Il prodotto è definito in termini di prodotti scalari;
infatti z_i è il prodotto scalare della i -ma riga di A ed il
vettore y**



17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

16

16

Algoritmo per il calcolo del prodotto matrice-vettore

```
procedure prod_mat_vet (in:m,n,A,y; out:z)
var m,n,i,j : integer
var A : array [1..m, 1..n] of real
var y: array [1..n] of real
var z: array [1..m] of real

begin
  for i := 1 to m do
    z(i) := 0
    for j := 1 to n do
      z(i) := z(i) + A(i,j) * y(j)
    endfor
  endfor
end
end prod_mat_vet
```

17/11/2020

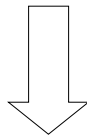
Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

17

17

Algoritmo per il calcolo del prodotto matrice-vettore

complessità di tempo: $n \times m$ A, $n \times m$ M



complessità di tempo del prodotto matrice-vettore,
supponendo la matrice di dimensioni $n \times n$ ed il vettore di n
componenti

$$T(n) = O(n^2)$$

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

18

18

- calcolo di una gaxpy

Una operazione gaxpy è una generalizzazione della saxpy, e cioè la somma di una matrice per un vettore e di un altro vettore

$$z = Ax + y$$

Con A matrice $m \times n$, x vettore di n componenti, y e z vettori di m componenti



In genere il calcolo si effettua *in place*, cioè il risultato va nello stesso vettore y, cioè $y = Ax + y$

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

19

19

- Come esercizio, realizzare in P-like l'algoritmo per il calcolo di una gaxpy, e studiarne la complessità

17/11/2020

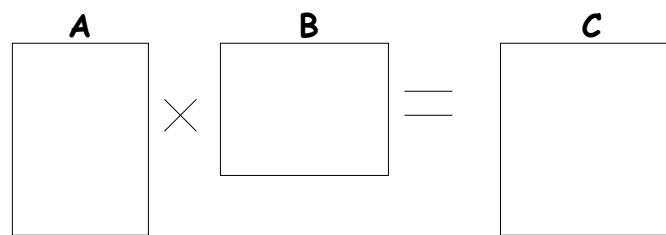
Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

20

20

calcolo del prodotto di 2 matrici

$C = AB$, con A matrice di dimensioni $m \times n$, B matrice di dimensioni $n \times p$, C ovviamente allora matrice di dimensioni $m \times p$



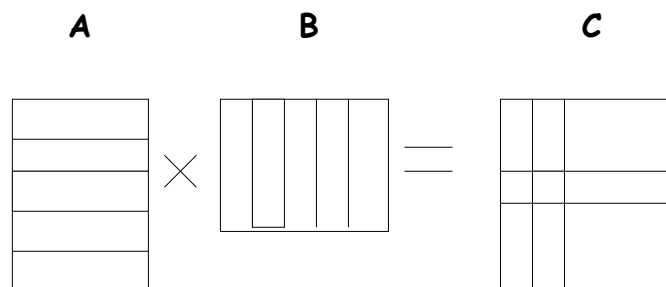
17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo matriciale - parte 1

21

21

Il prodotto è definito in termini di prodotti scalari; infatti c_{ij} è il prodotto scalare della i -ma riga di A e della j -ma colonna di B



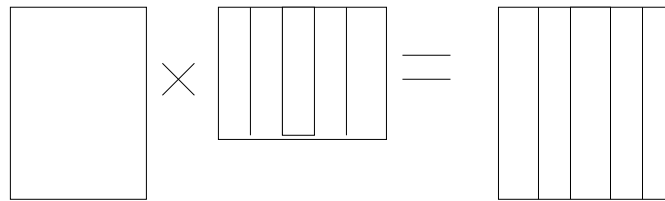
17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo matriciale - parte 1

22

22

Il prodotto può essere anche visto in termini di prodotti matrice-vettore, cioè la j-ma colonna di C è data dal prodotto della matrice A per la j-ma colonna di B



17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo matriciale - parte 1

23

23

Algoritmo per il calcolo del prodotto matrice-matrice

```

procedure prod_mat_mat (in:m,n,p,A,B; out:C)
var m,n,p,i,j,k : integer;  var A : array [1..m, 1..n] of real;
var B: array [1..n, 1..p] of real; var C: array [1..m, 1..p] of real ;
begin
  for i := 1 to m do
    for j := 1 to p do
      C(i,j) := 0.
      for k:= 1 to n do
        C(i,j) := C(i,j) + A(i,k)*B(k,j)
      endfor
    endfor
  endfor
end
end prod_mat_mat

```

17/11/2020

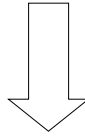
Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo matriciale - parte 1

24

24

Algoritmo per il calcolo del prodotto matrice-matrice

complessità di tempo: $m \cdot p \cdot n$ A, $m \cdot p \cdot n$ M



complessità di tempo del prodotto matrice-matrice,
supponendo le matrici di dimensioni $n \times n$

$$T(n) = O(n^3)$$

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

25

25

Fine Introduzione al calcolo matriciale- parte 1

17/11/2020

Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021- Introduzione al calcolo
matriciale - parte 1

26

26

Introduzione al calcolo matriciale - parte 2

•Queste slides derivano da lezioni di vari corsi di Calcolo Numerico e Programmazione tenuti dal prof. A. Murli e, sotto diversa forma, parzialmente presenti in:

A.Murli, G.Giunta, G.Laccetti, M.Rizzardi
Laboratorio di Programmazione I

1

2

Introduzione al calcolo matriciale - parte 2

•Queste slides derivano da lezioni di vari corsi di Calcolo Numerico e Programmazione tenuti dal prof. A. Murli e, sotto diversa forma, parzialmente presenti in:

A.Murli, G.Giunta, G.Laccetti, M.Rizzardi
Laboratorio di Programmazione I

3

Introduzione al calcolo matriciale

- nuclei computazionali di base
- algoritmi di back e forward substitution

4

Queste slides derivano da lezioni di vari corsi di Calcolo Numerico e Programmazione tenuti dal prof. A. Murli e, sotto diversa forma, parzialmente presenti in:

A.Murli, G.Giunta, G.Laccetti, M.Rizzardi
Laboratorio di Programmazione I

Parte 2 – Algoritmi di back e forward substitution

Esempio: risolviamo il seguente sistema

$$\begin{cases} 10x_1 + x_2 - 5x_3 = 1 \\ -20x_1 + 3x_2 + 20x_3 = 2 \\ 5x_1 + 3x_2 + 5x_3 = 6 \end{cases}$$

con il metodo di Cramer

• **passo 1: calcolo di** $\begin{vmatrix} 10 & 1 & -5 \\ -20 & 3 & 20 \\ 5 & 3 & 5 \end{vmatrix} = 10 \times \begin{vmatrix} 3 & 20 \\ 3 & 5 \end{vmatrix} + 20 \times \begin{vmatrix} 1 & -5 \\ 3 & 5 \end{vmatrix} + 5 \times \begin{vmatrix} 1 & -5 \\ 3 & 20 \end{vmatrix}$

$\Leftrightarrow = 10 \times (3 \times 5 - 20 \times 3) + 20 \times (1 \times 5 + 5 \times 3) + 5 \times (1 \times 20 + 5 \times 3) = 125$

• **passo 2: calcolo di**

$$x_1 = \frac{\begin{vmatrix} 1 & 1 & -5 \\ 2 & 3 & 20 \\ 6 & 3 & 5 \end{vmatrix}}{125} = 1$$

$$x_2 = \frac{\begin{vmatrix} 10 & 1 & -5 \\ -20 & 2 & 20 \\ 5 & 6 & 5 \end{vmatrix}}{125} = -2$$

$$x_3 = \frac{\begin{vmatrix} 10 & 1 & 1 \\ -20 & 3 & 2 \\ 5 & 3 & 6 \end{vmatrix}}{125} = 1.4$$

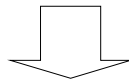
Numero di operazioni floating point effettuate:

• **passo 1: calcolo di 1 determinante di ordine 3:**

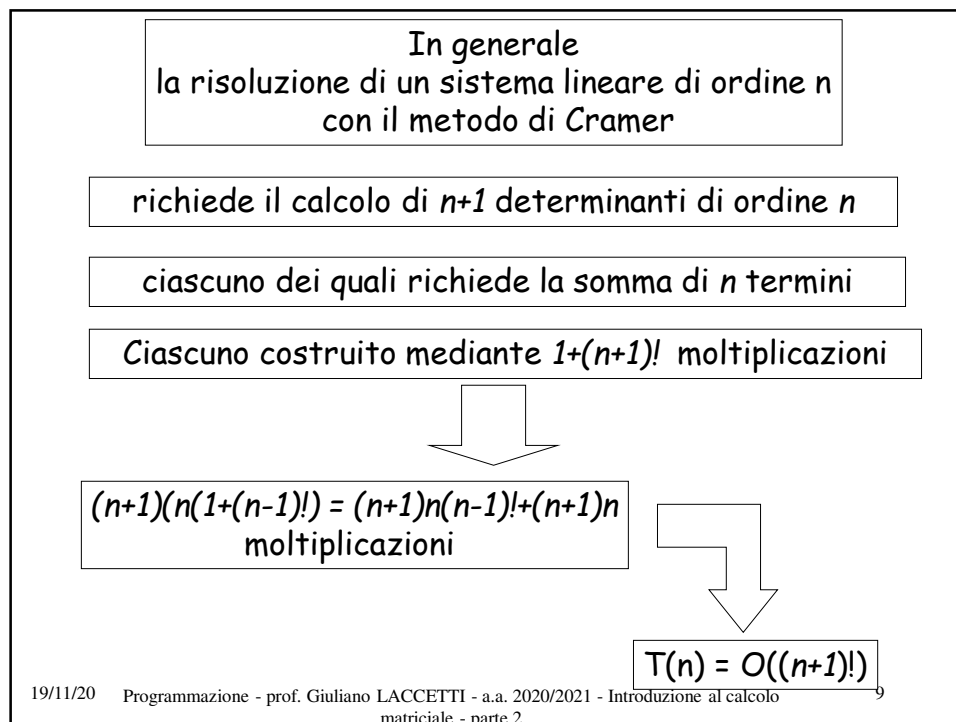
$$9M + 5A$$

• **passo 2: calcolo di 3 determinanti di ordine 2 più 3 divisioni:**

$$(3 \times 9 + 3)M + (3 \times 5)A$$



$$\text{TOTALE} = 39M + 20A$$



9

**Tempo necessario alla risoluzione di un
sistema di ordine n con il metodo di Cramer**

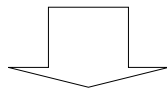
n	PC	Summit
10	4×10^{-4} secondi	4×10^{-9} secondi
20	7.7 anni	24 secondi
30	8×10^{14} anni	84 anni

velocità operativa:
PC = 10^{10} operazioni al secondo
Summit= 10^{17} operazioni al secondo

19/11/20 Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021 - Introduzione al calcolo
matriciale - parte 2

10

Metodo di Cramer :
costruttivo
ma
non effettivamente utilizzabile



**Necessità di metodi utilizzabili
(praticabili)**

Come costruire un
metodo numerico efficiente ?

Esempio: Risoluzione di un sistema diagonale

$$\begin{cases} 7x_1 = 3 \\ 6.5x_2 = 2 \\ -8x_3 = 1.4 \end{cases}$$

La forma del sistema suggerisce in modo naturale l'algoritmo risolutivo

$$\begin{cases} 7x_1 = 3 \\ 6.5x_2 = 2 \\ -8x_3 = 1.4 \end{cases} \Rightarrow$$

$$\begin{cases} x_1 = \frac{3}{7} \\ x_2 = \frac{2}{6.5} \\ x_3 = \frac{1.4}{-8} \end{cases}$$

Numero di operazioni: 3 Moltiplicazioni/divisioni

19/11/20 Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021 - Introduzione al calcolo matriciale - parte 2

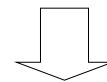
13

13

In generale:

$$\begin{cases} d_{1,1}x_1 = b_1 \\ d_{2,2}x_2 = b_2 \\ \dots \\ \dots \\ d_{n,n}x_n = b_n \end{cases}$$

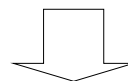
Sistema diagonale



Algoritmo risolutivo:

$$\begin{cases} x_i = \frac{b_i}{d_{i,i}} \quad i = 1, n (d_{i,i} \neq 0) \end{cases}$$

nM operazioni fl.p.



$T(n) = O(n)$

19/11/20 Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021 - Introduzione al calcolo matriciale - parte 2

14

14

Algoritmo per la risoluzione di un sistema diagonale: $Dx=b$

```

.....
for i:= 1 to n do
    x(i):=b(i)/D(i,i)
endfor
.....

```

Esempio: Risoluzione di un sistema triangolare superiore

$$\begin{cases} 2x_1 + 2x_2 + 4x_3 = 5 \\ 7x_2 + 11x_3 = 8 \\ 2x_3 = 2 \end{cases}$$

La forma del sistema
suggerisce
in modo naturale
l'algoritmo risolutivo

$$\begin{cases} 2x_1 + 2x_2 + 4x_3 = 5 \\ 7x_2 + 11x_3 = 8 \\ 2x_3 = 2 \end{cases}$$



$$x_1 = \frac{5 - 2x_2 - 4x_3}{2} = \frac{13}{14}$$



$$x_2 = \frac{8 - 11x_3}{7} = \frac{8 - 11}{7} = -\frac{3}{7}$$



$$x_3 = \frac{2}{2} = 1$$

Numero di operazioni: $3A+6M$

Esempio: Risoluzione di un sistema triangolare inferiore

$$\begin{cases} 2x_1 &= 4 \\ 3x_1 + 2x_2 &= 5 \\ x_1 + 2x_2 - 3x_3 &= 1 \end{cases}$$

La forma del sistema suggerisce in modo naturale l'algoritmo risolutivo

$$\begin{cases} 2x_1 &= 4 \\ 3x_1 + 2x_2 &= 5 \\ x_1 + 2x_2 - 3x_3 &= 1 \end{cases}$$



$$x_1 = \frac{4}{2} = 2$$



$$x_2 = \frac{5 - 3x_1}{2} = -\frac{1}{2}$$



$$x_3 = \frac{2}{2} = 1$$

Numero di operazioni: 3A+6M

19/11/20 Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021 - Introduzione al calcolo matriciale - parte 2

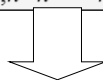
17

17

In generale:

$$\begin{cases} u_{1,1}x_1 + u_{1,2}x_2 + u_{1,3}x_3 + \dots + u_{1,n}x_n = b_1 \\ u_{2,2}x_2 + u_{2,3}x_3 + \dots + u_{2,n}x_n = b_2 \\ \dots \\ u_{n,n}x_n = b_n \end{cases}$$

Sistema triangolare superiore



Algoritmo di back substitution

$$\begin{aligned} x_n &= b_n / u_{n,n} \\ x_i &= (b_i - u_{i,i+1}x_{i+1} - u_{i,i+2}x_{i+2} - \dots - u_{i,n}x_n) / u_{i,i} = \\ &= (b_i - \sum_{k=i+1}^n u_{i,k}x_k) / u_{i,i} \quad i = n-1, n-2, \dots, 1 \end{aligned}$$

19/11/20 Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021 - Introduzione al calcolo matriciale - parte 2

18

18

In generale:

$$\left\{ \begin{array}{l} l_{1,1}x_1 = b_1 \\ l_{2,1}x_1 + l_{2,2}x_2 = b_2 \\ \dots\dots\dots \\ l_{n,1}x_1 + l_{n,2}x_2 + l_{n,3}x_3 + \dots\dots + l_{n,n}x_n = b_n \end{array} \right.$$

Sistema triangolare inferiore

Algoritmo di forward substitution

$$\begin{aligned} x_1 &= b_1/l_{1,1} \\ x_i &= (b_i - l_{i,1}x_1 - l_{i,2}x_2 - \dots - l_{i,i-1}x_{i-1})/l_{i,i} = \\ &= (b_i - \sum_{k=1}^{i-1} l_{i,k}x_k)/l_{i,i} \quad i = 2, 3, \dots, n \end{aligned}$$

19/11/20 Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021 - Introduzione al calcolo matriciale - parte 2

19

19

Algoritmo di back substitution: $Ux=b$

```

.....
x(n) := b(n)/U(n,n)
for i:= n-1 to 1 step -1 do
    x(i):=b(i)
    for j:=i+1 to n do
        x(i):= x(i)-U(i,j)*x(j)
    endfor
    x(i):=x(i)/U(i,i)
endfor

```

19/11/20 Programmazione - prof. Giuliano LACCETTI - a.a. 2020/2021 - Introduzione al calcolo matriciale - parte 2

20

20

Algoritmo di forward substitution: $Lx=b$

```

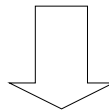
.....
x(1) := b(1)/L(1,1)
for i:= 2 to n do
    x(i):=b(i)
    for j:=1 to i-1 do
        x(i):= x(i)-L(i,j)*x(j)
    endfor
    x(i):=x(i)/L(i,i)
endfor
.....

```

21

Complessità di tempo

• 1 M	per $x(n)$
• 2A + 2M	per $x(n-1)$
• $(n-i)A + (n-i+1)M$	per $x(i) \ i=n-3, \dots, 1$



$$(1+2+3+\dots+(n-1)+n)M = n(n+1)/2 \ M$$

$$(1+2+3+\dots+(n-2)+(n-1))M = n(n-1)/2 \ A$$



$$T_{Back}(n) = O\left(\frac{n^2}{2}\right)$$

$$T_{Forw}(n) = O\left(\frac{n^2}{2}\right)$$

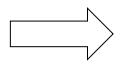
22

In generale è necessario risolvere un sistema lineare del tipo $Ax = b$ con matrice dei coefficienti **né diagonale né triangolare**



IDEA

Trasformare il sistema in uno equivalente con matrice dei coefficienti *"almeno triangolare"*



Algoritmo di Gauss



Corso di calcolo numerico

23

Tempo necessario (all'incirca) alla risoluzione di un sistema di ordine n con il metodo di Gauss $O(n^3)$ (prestazioni/velocità operative massime teoriche)

n	PC	Summit
10	10^{-7} secondi	10^{-14} secondi
20	0.8×10^{-6} secondi	0.8×10^{-13} secondi
30	0.27×10^{-5} secondi	0.27×10^{-12} secondi
100000	1.16 giorni	10^{-2} secondi

velocità operativa:

PC = 10^{10} operazioni al secondo

Summit = 10^{17} operazioni al secondo

24

Fine Introduzione al calcolo matriciale - parte 2

LA PROGETTAZIONE DEGLI ALGORITMI: COMPONENTI DI BASE E METODOLOGIE DI SVILUPPO

- **variabili e costanti**
- **strutture di controllo**
- **variabili strutturate**
- **procedure**

Variabili e costanti

Esempio 1:

Calcolo della circonferenza di un cerchio

$$\text{raggio} = 1$$

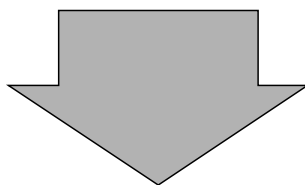
$$\text{circonferenza} = 2 \cdot \pi \cdot 1$$

$$\text{raggio} = 1.5$$

$$\text{circonferenza} = 2 \cdot \pi \cdot 1.5$$

$$\text{raggio} = 3$$

$$\text{circonferenza} = 2 \cdot \pi \cdot 3$$

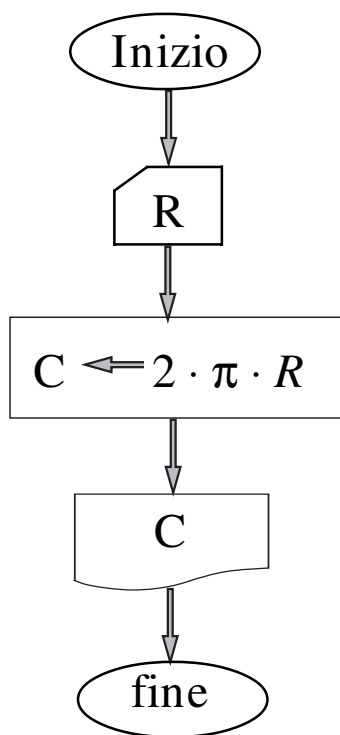


$2 \cdot \pi$ uguale per tutti i cerchi

raggio, circonferenza variano al variare del cerchio

ALGORITMO (prima versione)

FLOWCHART



PASCAL-LIKE

begin circonfer

read R

$C = 2 \cdot \pi \cdot R$

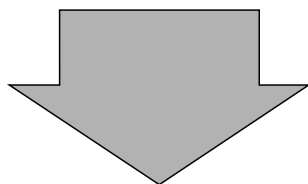
print C

end circonfer

L'algoritmo specifica
un procedimento generale per il calcolo
della lunghezza di una circonferenza

$2 \cdot \pi$ è un valore costante;

il valore del raggio R non è specificato;
al suo posto si utilizza un
nome che denota un oggetto variabile
(analogamente per C)



$2 \cdot \pi = \text{COSTANTE}$
 $R, C = \text{VARIABILI}$

una *variabile* è un **nome** a cui si associa un valore appartenente ad un insieme prefissato

In un linguaggio di programmazione
una variabile è il nome simbolico
dell'indirizzo di una
locazione di memoria

Esempio:

variabile

locazione di memoria

indirizzo

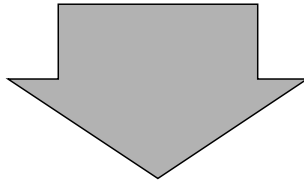
R

--	--	--	--	--	--	--	--

0011

Nell'algoritmo precedente
la **variabile** R indica un
generico numero reale

Il tipo di una variabile è l'insieme
dei valori che essa può assumere
(es. numeri reali, numeri interi, ...)



R = variabile di **tipo reale**

DICHIARAZIONE DI UNA VARIABILE

La dichiarazione di una variabile è la
specifica del suo tipo
(reale, intero, alfanumerico, logico, ...)

Esempio:

var: R:	real	(R: variabile di tipo reale)
var: I:	real	(I: variabile di tipo reale)
var: NOME:	character	(NOME: variabile di tipo alfanumerico)
var: L:	logical	(L: variabile di tipo logico)

Esempio:

R

0	1	0	1	0	0	0	0
---	---	---	---	---	---	---	---

R: integer

R = +80

rappr. binaria del numero

intero 80 (primo bit = segno)

R: character

R = P

rappresentazione del dato

alfanumerico P (codice ASCII)

R: real

R = 0.625

rappr. f.p. norm. in base 2

del numero reale 0.625

(bit 1 = segno mant.,

bit 2-5 = cifre mant.,

bit 6 = segno espon.,

bit 7-8 = cifre espon.)

La dichiarazione di una variabile
 consente di interpretare il contenuto
 della corrispondente locazione di
 memoria in base al tipo della variabile

Algoritmo per il calcolo della circonferenza (seconda versione)

begin circonfer

var: R, C: real

read R

$C = 2 \cdot \pi \cdot R$

print C

end circonfer



*dichiarazione
delle variabili R e C
di tipo reale*

Nell'algoritmo precedente
il dato $2 \cdot \pi$ è costante

**La costante identifica la locazione
di memoria a cui essa è associata,
mediante l'indicazione esplicita
del suo contenuto**

RAPPRESENTAZIONE DELLE COSTANTI

**Una costante denota esplicitamente
un dato di un certo tipo**

COSTANTI ALFANUMERICHE

una sequenza di caratteri alfanumerici racchiusa tra
apici

‘Napoli’
‘telefono’
‘12345’

COSTANTI LOGICHE

.TRUE. (VERO)

.FALSE. (FALSO)

COSTANTI INTERE

una sequenza di cifre decimali precedute eventualmente dal segno

1234

+27

−99012

COSTANTI REALI

una sequenza di cifre decimali preceduta eventualmente dal segno e contenente necessariamente il punto decimale

4.25

6.

+7.1

−2349.333

oppure

notazione esponenziale (floating-point), con le convenzioni precedenti per la mantissa

95.6e4

−0.13e2


4.e−3

Algoritmo per il calcolo della circonferenza (versione finale)

begin circonfer

var: R, C: real

read R

$C = 2 \times 3.141592 \times R$  *dichiarazione
esplicita del
valore di π*

print C

end circonfer

DEFINIZIONE DI UNA VARIABILE

La definizione di una variabile è
l'assegnazione di un valore
alla variabile

Esempio:

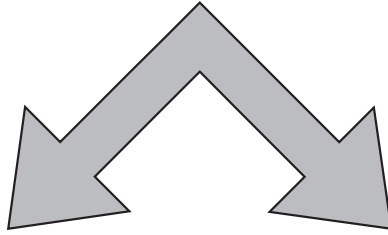
```
begin area_triangolo  
  var: altezza, base: real
```

```
    read altezza  
    base = 1.3
```

```
    .....
```

```
end area_triangolo
```


L'assegnazione di un valore ad una
variabile dichiarata di un certo tipo
può essere fatta mediante
le operazioni di:



assegnazione

lettura

base = 1.3

read altezza

Esempio:

```
begin area_triangolo  
    var: altezza, base, area: real  
  
    read altezza  
    base = 5.  
  
    area = base × altezza/2.  
  
    print area  
end area_triangolo
```

In una **istruzione di assegnazione** **prima** si valuta l'espressione al secondo membro e **poi** si assegna il **valore** alla variabile al primo membro.

La **valutazione dell'espressione** consiste **prima** nella **sostituzione** a ciascuna variabile del proprio valore e **poi** nella **esecuzione delle operazioni** specificate.

Esempio:

```
begin area_triangolo  
    var: altezza, base, area: real  
  
    read altezza  
    base = 5.  
    area = base × altezza  
  
    area = area/2.  
  
    print area  
end area_triangolo
```

L'ultima assegnazione ha senso perché si valuta **pri-**
ma il secondo membro, e **poi** si assegna il risultato
alla variabile area.

L'ultima assegnazione fa perdere il valore precedente della variabile **area** (l'operazione di assegnazione è *distruttiva*)

Esempio:

```
begin perimetro_triangolo  
    var: lato1, lato2, lato3, somma: real  
  
    read lato1  
    lato2 = 5.  
  
    somma = lato1+lato2+lato3  
  
end perimetro_triangolo
```

ERRATO !!!!

la variabile **lato3** non è stata definita

tutte le variabili
devono essere definite
prima di essere utilizzate

Esempio:

```
begin area_rettangolo  
    var: area: logical  
    var: base, altezza: real  
  
    altezza = 2.5  
    base = 5.1  
  
    area = base × altezza  
  
end area_triangolo
```

ERRATO !!!!!

Il valore dell'espressione deve essere dello stesso tipo di quello della variabile a cui è assegnato

L'assegnazione di un valore ad una variabile è **ERRATA** se:

- l'espressione non può essere valutata perché vi compaiono variabili indefinite
- il valore dell'espressione è di tipo diverso da quello della variabile a cui è assegnato.

Nel linguaggio della Computer Science

il termine variabile indica un oggetto il cui valore, se non viene effettuata alcuna operazione su tale oggetto, rimane costante nel tempo.

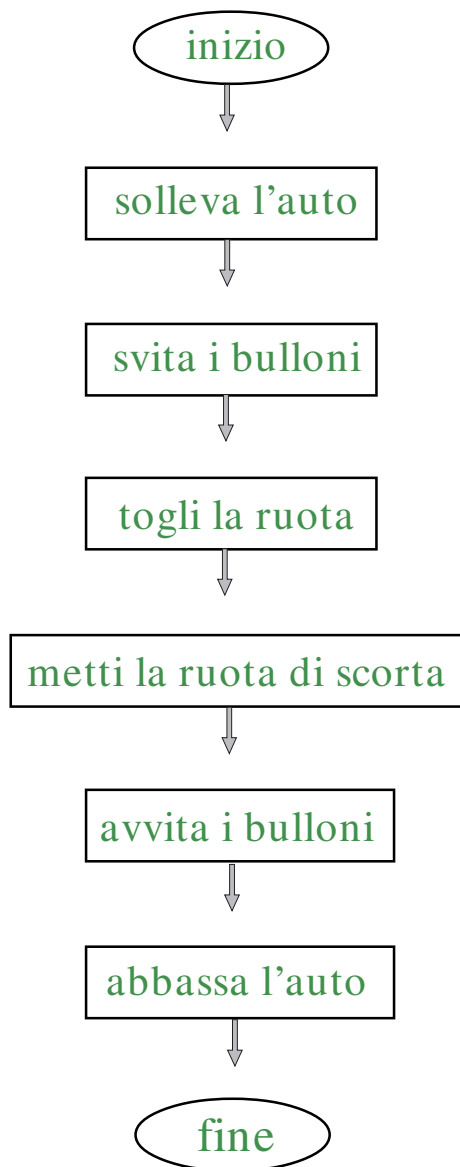
Nel linguaggio della Matematica

una variabile non rappresenta uno specifico valore costante nel tempo ma il generico elemento di un certo insieme

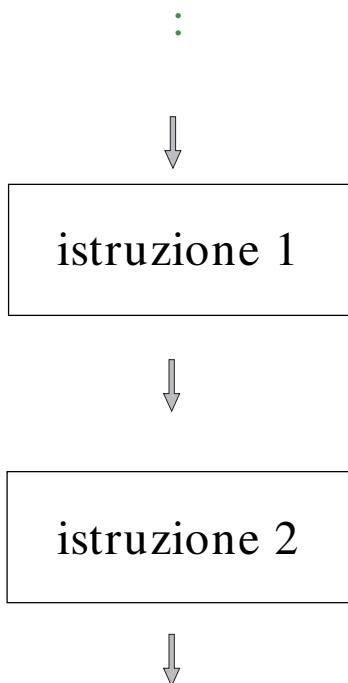
(ES.: $\forall x \in \mathcal{R}, x^2 \geq 0$ indica un generico numero reale.)

Strutture di controllo

Flow chart dell'algoritmo per il cambio della ruota



le istruzioni del flow chart



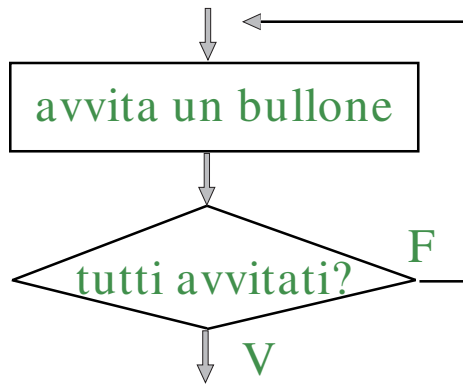
costituiscono una

**SEQUENZA
DI ISTRUZIONI**

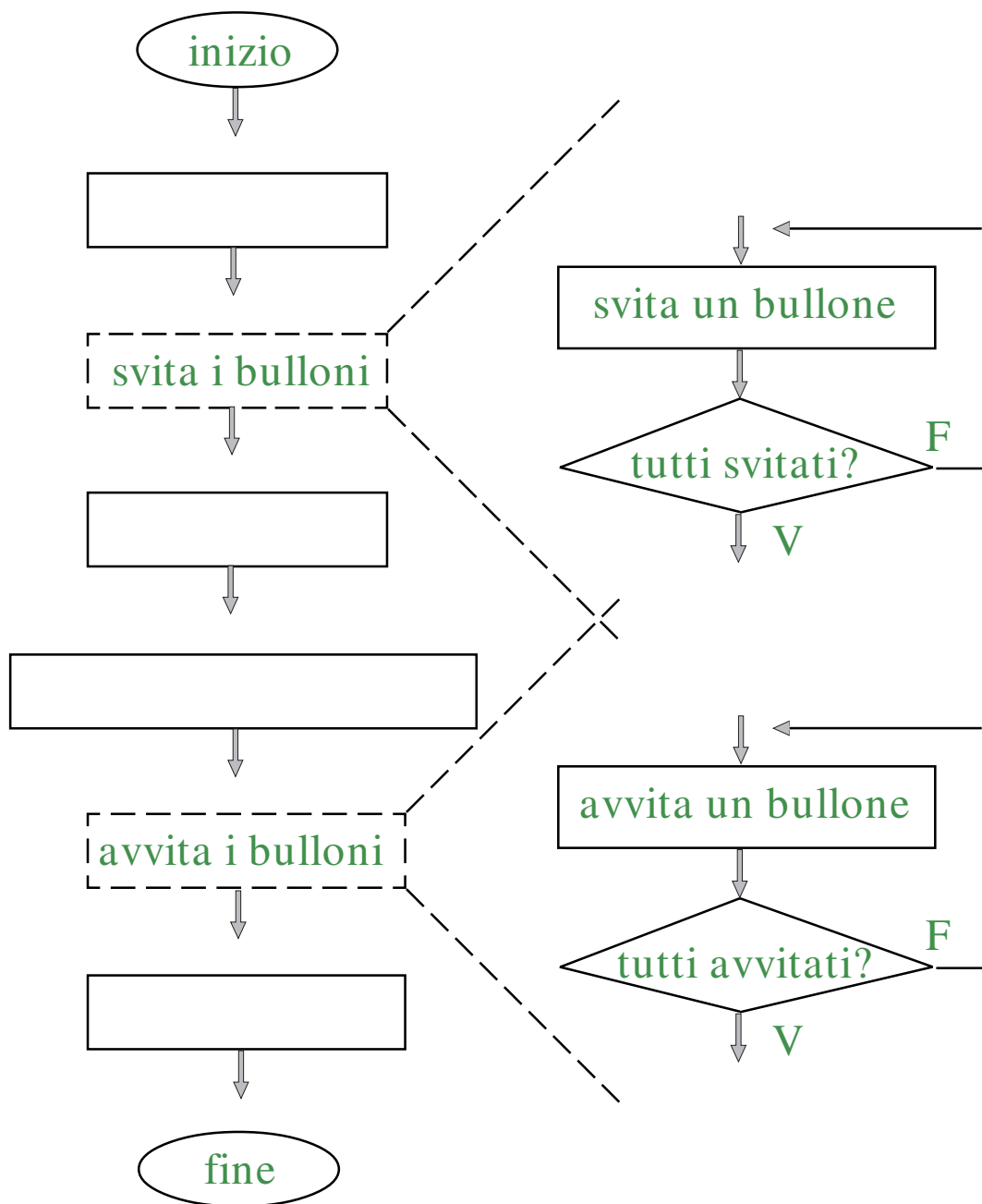
l'istruzione

avvita i bulloni

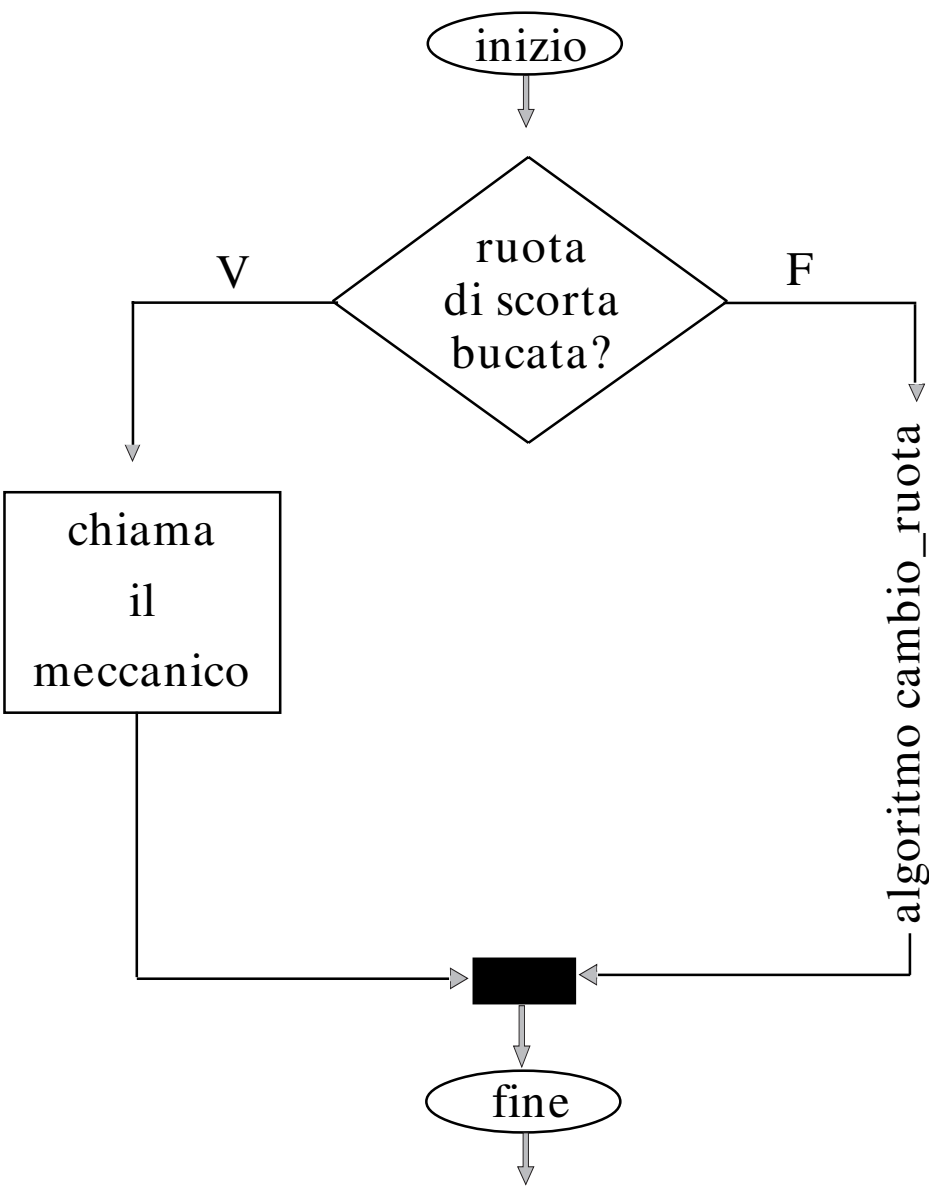
può essere sostituita da



**STRUTTURA
DI ITERAZIONE**



Che succede se la ruota di scorta è bucata?



**STRUTTURA
DI SELEZIONE**

Le strutture di controllo (o costrutti di controllo)

determinano l'ordine con cui
devono essere
eseguite le istruzioni

- **sono indipendenti** dalla natura delle istruzioni
- sono strumenti logici universali **utilizzabili in qualunque problema**

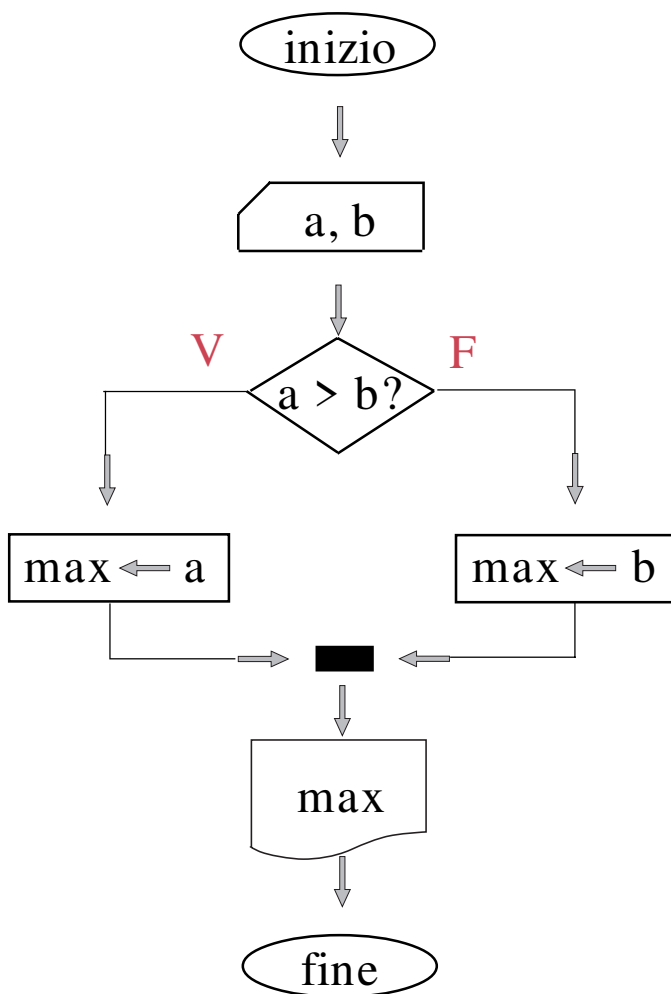
Analisi delle strutture di controllo

Esempio:

massimo tra 2 numeri

dati di input: i due numeri (a,b)

dati di output: il massimo (max)



P A S C A L L I K E

begin massimo

var: a, b, max: real

read a, b

if (a > b) then

max = a

else

max = b

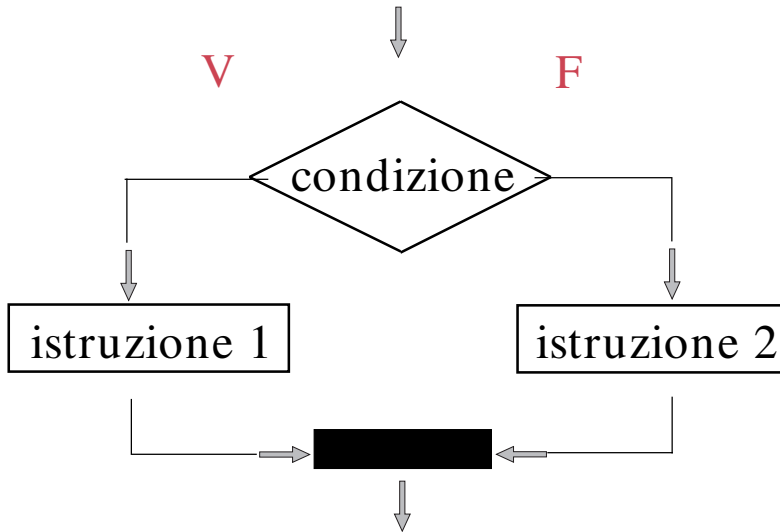
endif

print max

end massimo

}
struttura
di
selezione

La struttura di selezione del linguaggio del **flow chart**:



nel linguaggio Pascal-like
si traduce con

if (condizione) then

istruzione 1

else

istruzione 2

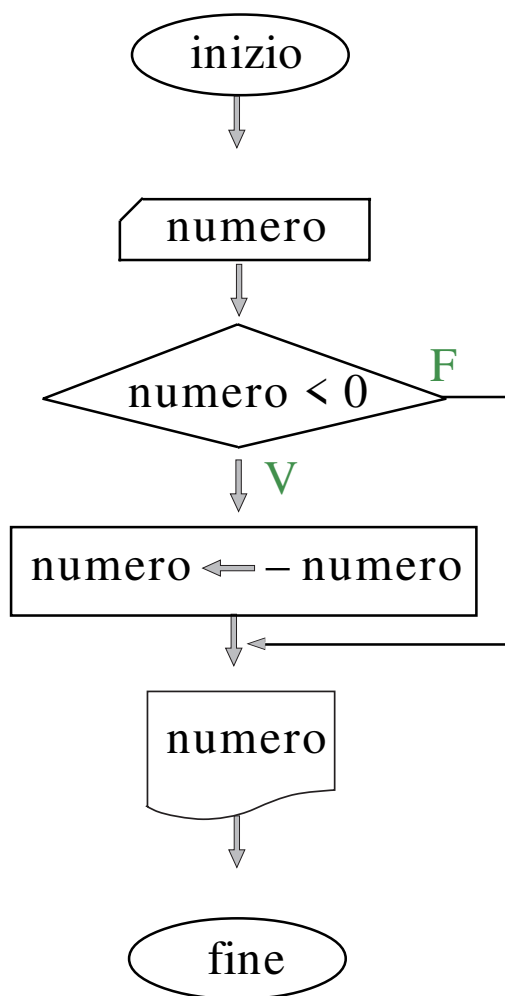
endif

Esempio:

valore assoluto di un numero reale

dati di input: numero

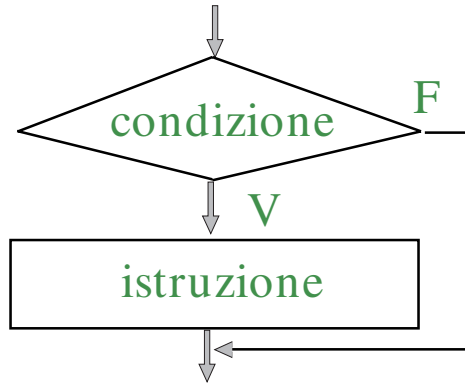
dati di output: valore assoluto del numero



PASCAL-LIKE

```
begin valore_assoluto  
  var: num: real  
  
  read num  
  
  if (num < 0) then  
    num = – num  
  
  endif  
  
  print num  
  
end valore_assoluto
```

la struttura di selezione
del linguaggio del flow chart:



nel linguaggio Pascal-like
si traduce con

if (condizione) then

istruzione

endif

Esempio:

somma di N numeri

dati di input: valore di N,
N numeri

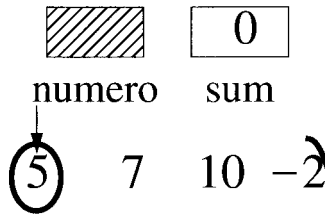
dati di output: somma degli N
numeri

dati di input: 4
5 7 10 -2

dati di output: 20

inizio:

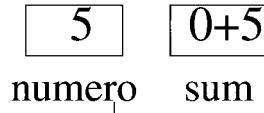
sum = 0



passo 1:

read numero

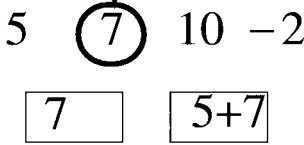
sum = sum + numero



passo 2:

read numero

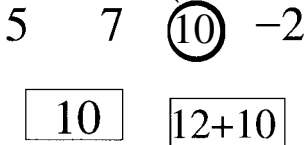
sum = sum + numero



passo 3:

read numero

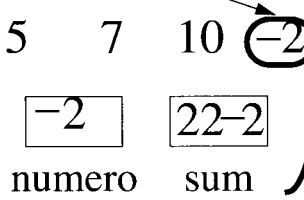
sum = sum + numero



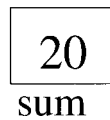
passo 4:

read numero

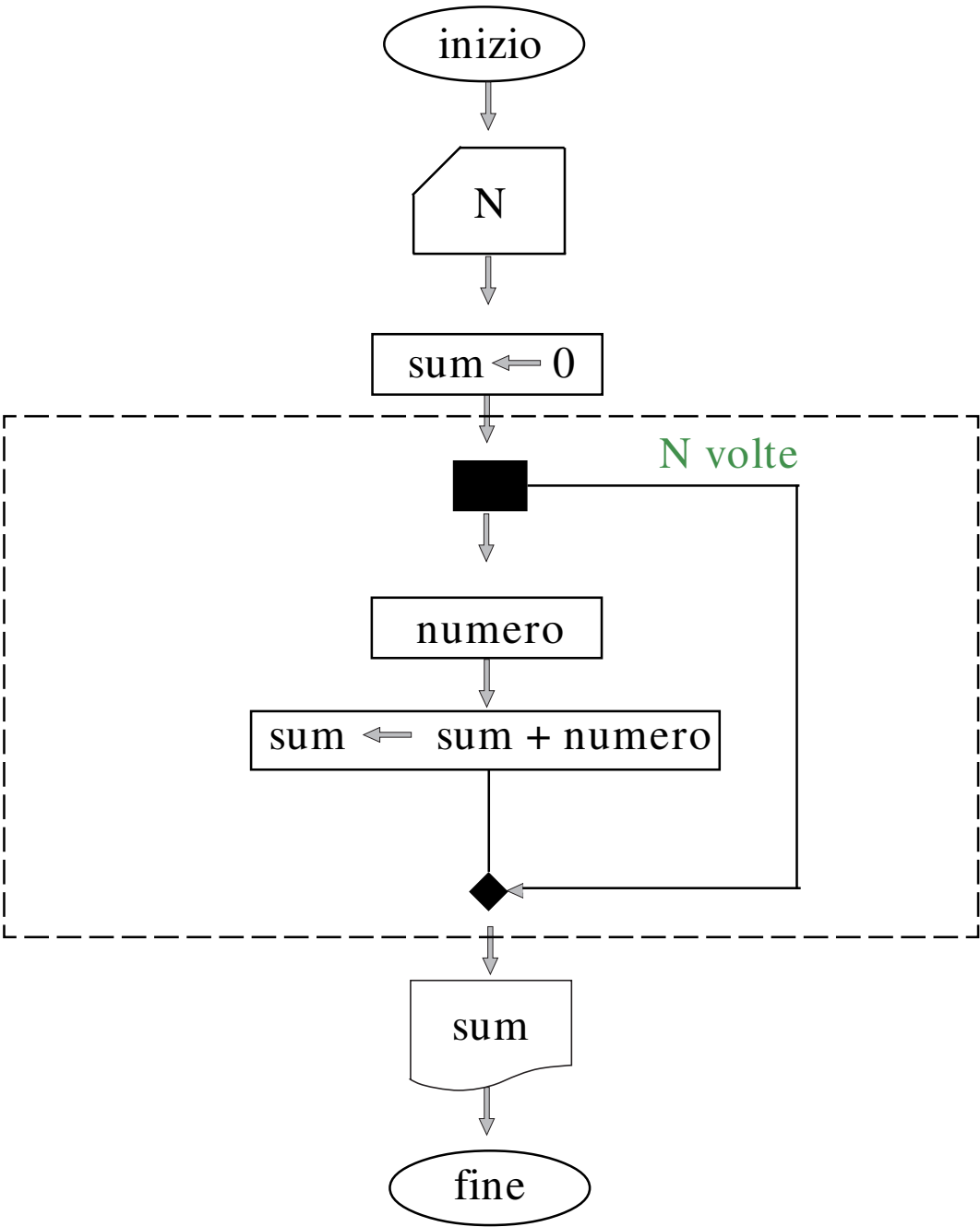
sum = sum + numero



4 volte
(# dati)



FLOW CHART



P A S C A L L I K E

begin somma

var: N, i: integer

var: sum, numero: real

read N

sum = 0.

for i = 1, N do

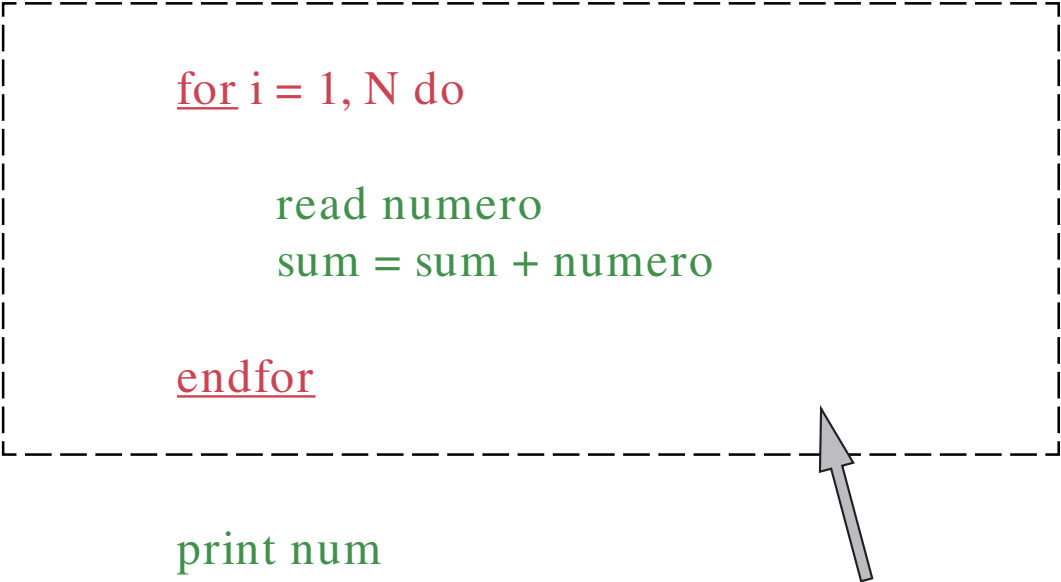
read numero

sum = sum + numero

endfor

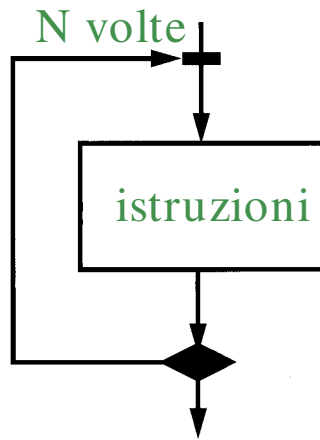
print num

end somma



struttura
di iterazione

la struttura di iterazione del
linguaggio del flow chart:



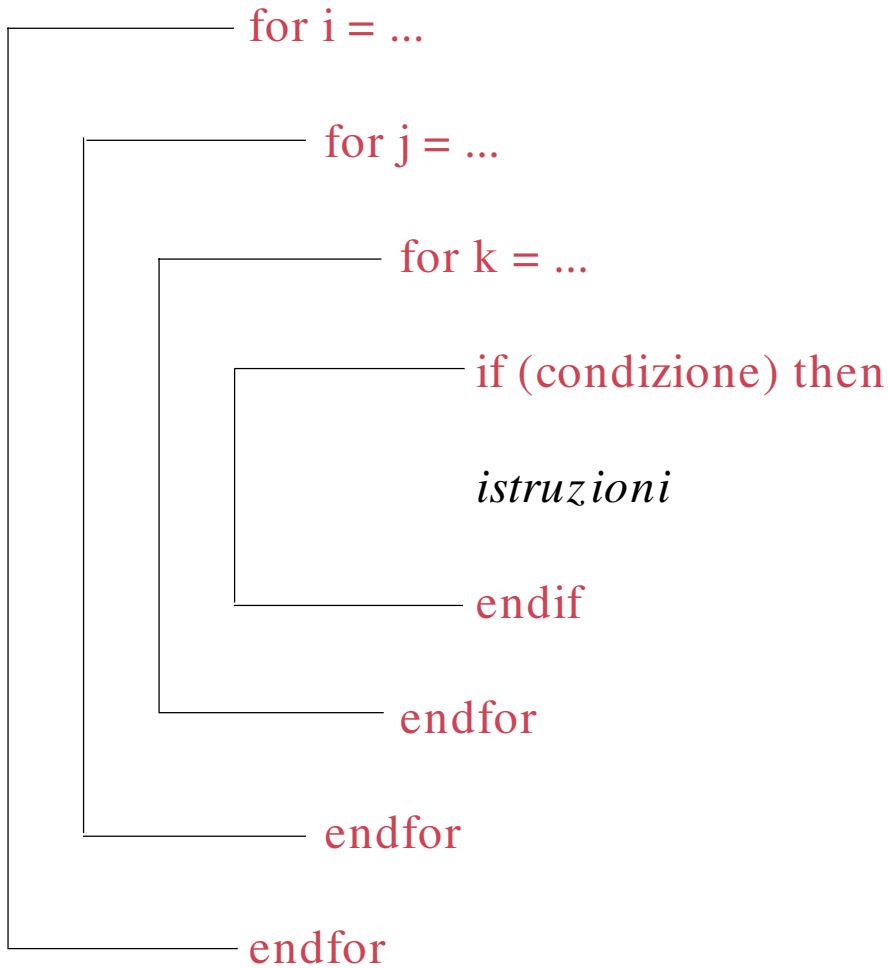
si traduce nel linguaggio
Pascal-like con

for i = 1, N do

istruzioni

endfor

Le strutture di controllo
possono essere
innestate l'una nell'altra



Esempio:

calcolare il massimo
di N numeri

dati di input: N , N numeri

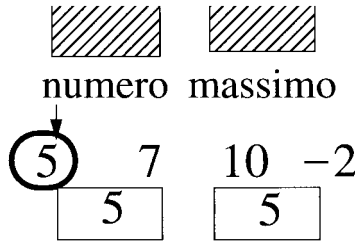
dati di output: massimo degli N
numeri

dati di input: 4
5 7 10 -2

dati di output: 10

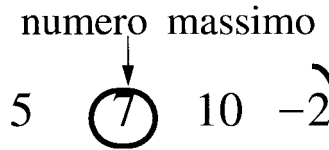
inizio:

read numero
massimo:=numero



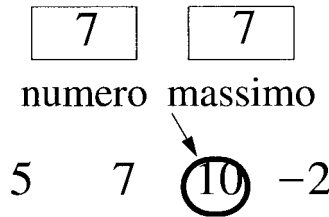
passo 1:

read numero
if (massimo<numero)
massimo:=numero



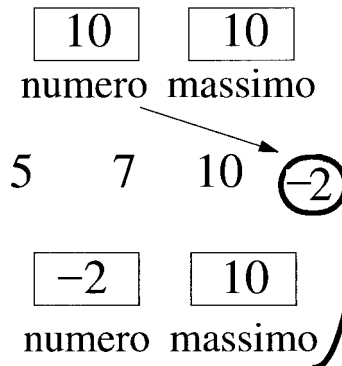
passo 2:

read numero
if (massimo<numero)
massimo:=numero



passo 3:

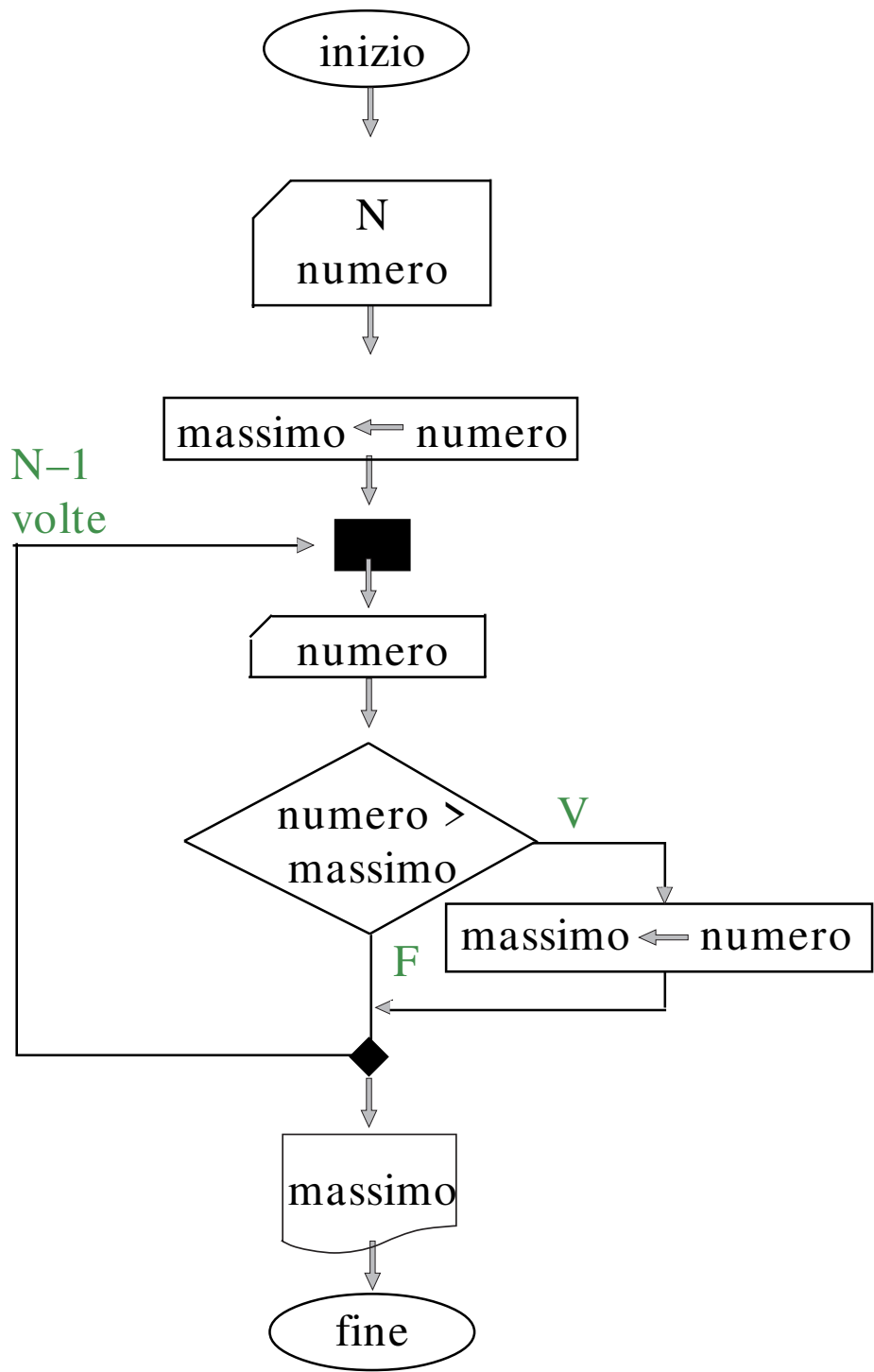
read numero
if (massimo<numero)
massimo:=numero



4 volte
(# dati)

10
massimo

FLOW CHART



P A S C A L L I K E

```
begin max_n_numeri
    var: numero, massimo: real
    var: i, N: integer

    read N, numero
    massimo = numero

    for i = 1, N-1 do

        read numero

        if (numero > massimo) then
            massimo = numero
        endif

    endfor

    print massimo

end max_n_numeri
```

Esempio:

Calcolo del prodotto
di un insieme di N numeri

dati di input: N , N numeri

dati di output: prodotto degli N
numeri

dati di input: 4
5 2 0 1

dati di output: 0

inizio: ⑤ 2 0 1

read numero

prodotto:=numero



numero prodotto



numero prodotto

passo 1: 5 ② 0 1

read numero

prodotto:=prodotto*numero



numero prodotto

passo 2: 5 2 ① 1

read numero

prodotto:=prodotto*numero



numero prodotto

passo 3: 5 2 ① 1

read numero

prodotto:=prodotto*numero

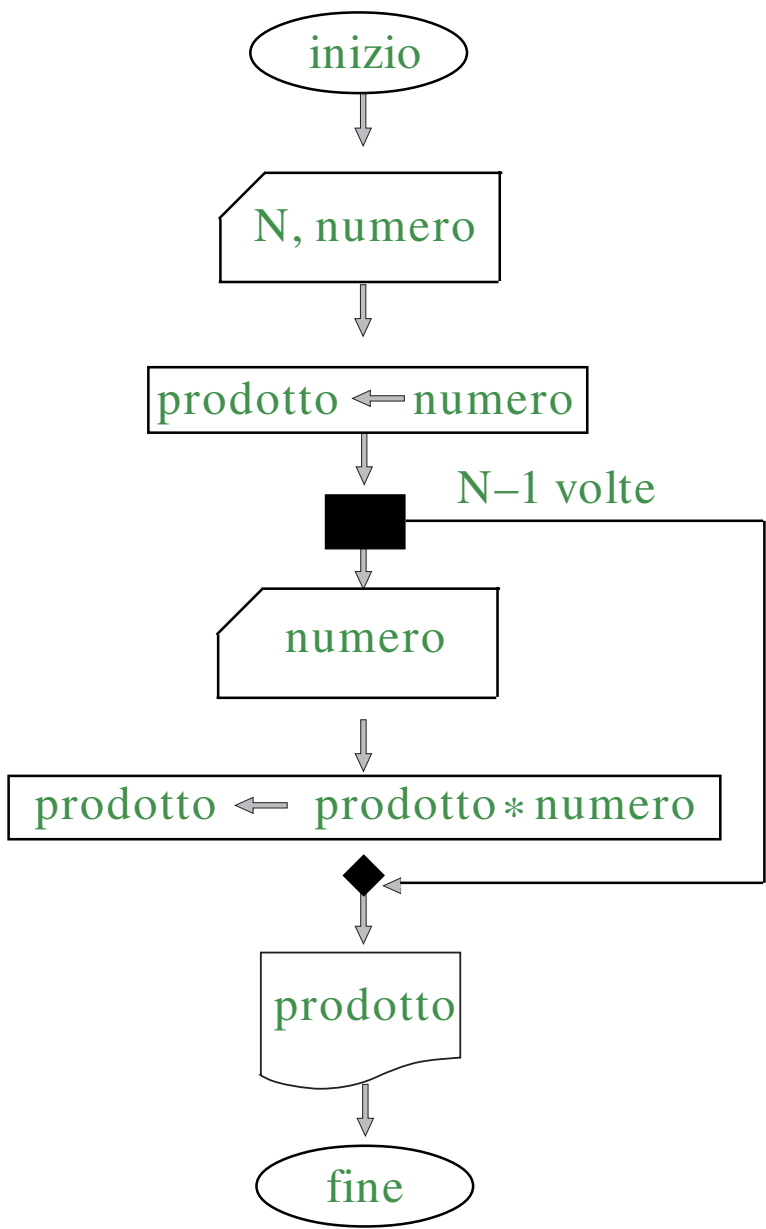


numero prodotto



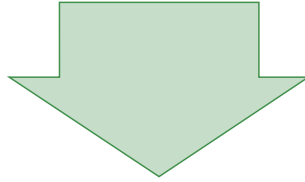
prodotto

FLOW CHART

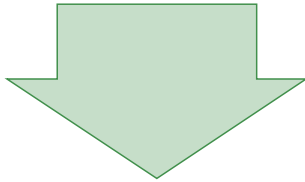


PROBLEMA

Se un numero è nullo,
il prodotto è nullo



Prevedere il caso in cui
un dato sia nullo



Si arresta il calcolo quando
sono stati moltiplicati tutti i numeri
oppure
quando un dato è nullo

		
i	numero	prodotto

inizio : ⑤ 2 0 1

read numero
prodotto=numero
i=1

1	5	5
i	numero	prodotto

passo 1 : 5 ② 0 1

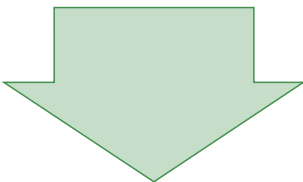
read numero
prodotto=prodotto*numero
i=i+1

2	2	10
i	numero	prodotto

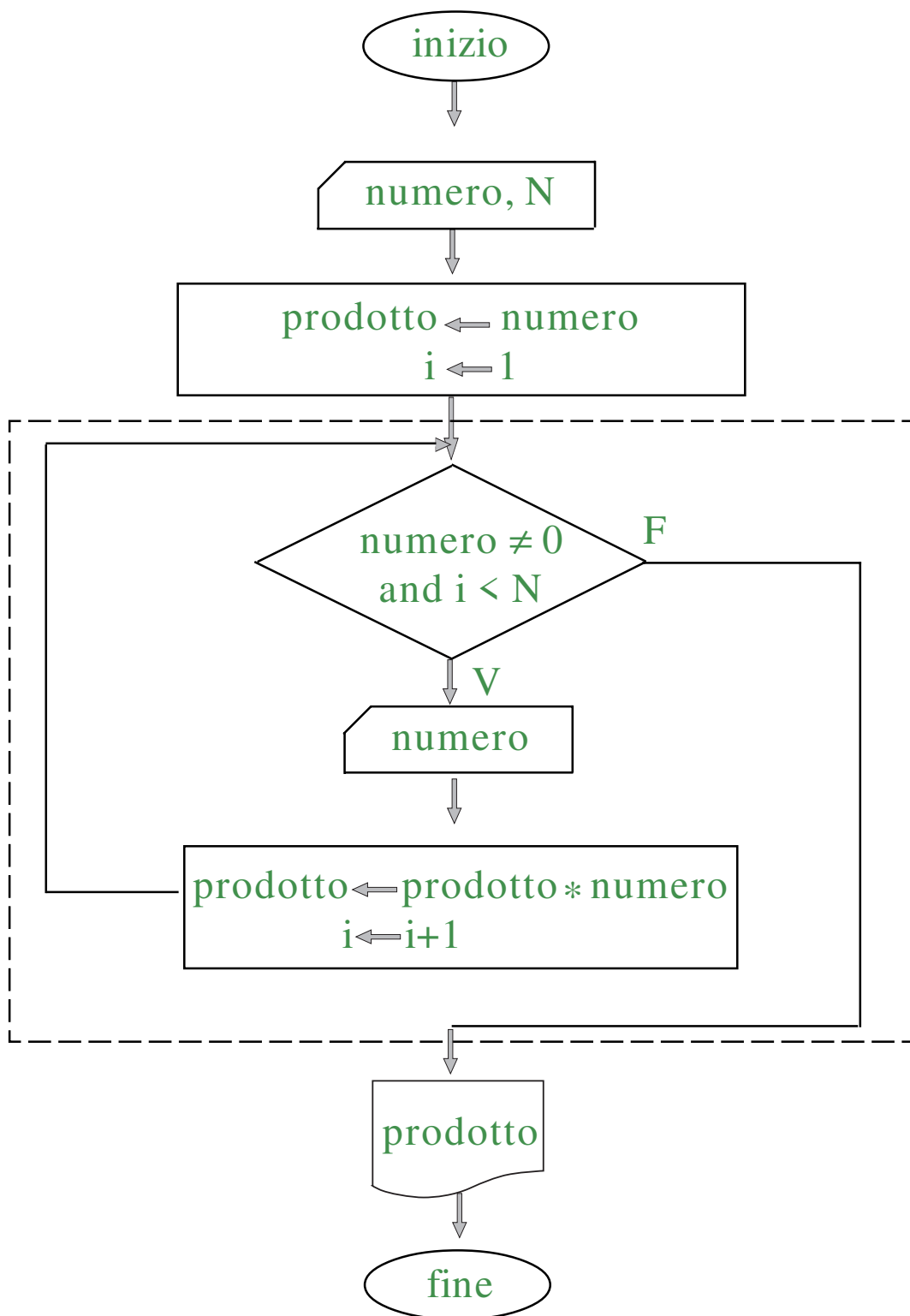
passo 2 : 5 2 ① 1

read numero
prodotto=prodotto*numero
i=i+1

3	0	0
i	numero	prodotto



STOP



PASCAL-LIKE

begin prod_numeri

var: N, i: real

var: numero, prodotto: real

read numero, N

prodotto := numero

i := 1

while (numero \neq 0 and $i < N$) do

read numero

prodotto := prodotto*numero

i := i+1

enwhile

print prodotto

end prod_numeri

Esempio:

ricerca di un elemento dato
in un insieme di N numeri

dati di input: N, N numeri
numero da cercare

dati di output: informazione che indica
se il num. è stato trovato


“posizione” del numero

dati di input: 4
2 1 10 5
10

dati di output: .true., 3

inizio:

.....

0		10	F
---	---	----	---

i numero elem. trovato

passo 1 ② 1 10 5
read numero
numero = elemento ?
i=i+1

1	2	10	F
---	---	----	---

i numero elem. trovato

passo 2: 2 ① 10 5
read numero
numero = elemento ?
i=i+1

2	1	10	F
---	---	----	---

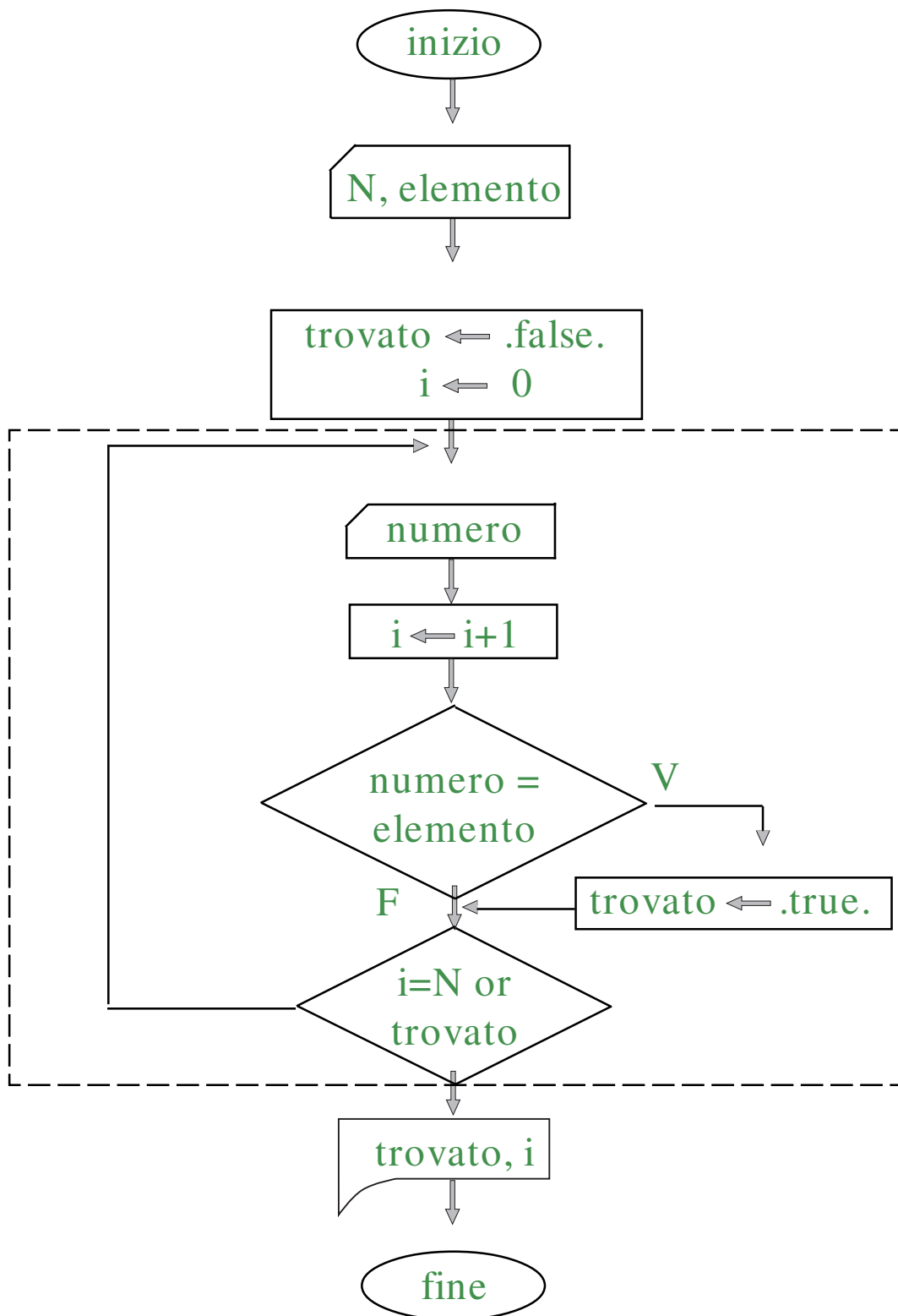
i numero elem. trovato

passo 3: 2 1 ⑩ 5
read numero
numero = elemento ?
i=i+1

3	10	10	V
---	----	----	---

i numero elem. trovato

STOP !



PASCAL LIKE

begin ricerca

var: N, numero, elemento, i: integer

var: trovato: logical

read N, elemento

trovato: = .false.

i: = 0

repeat

read numero

i: = i+1

if (elemento = numero) then

trovato: = .true.

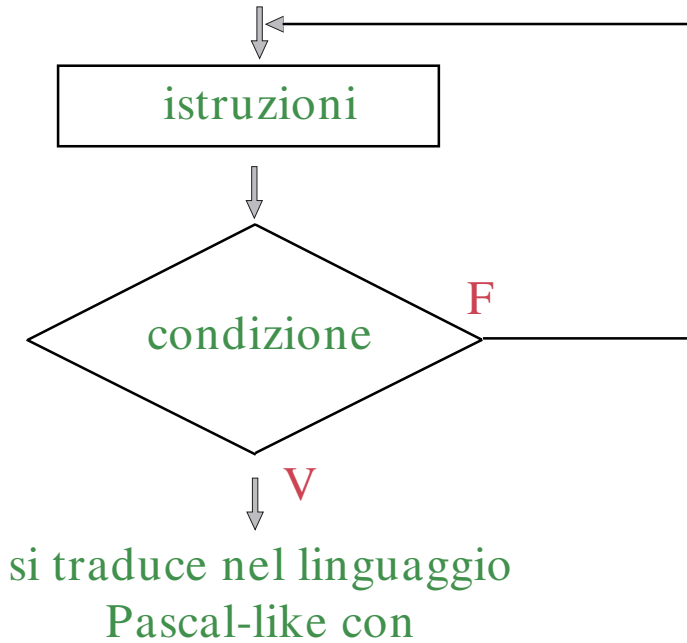
endif

until (i=N or trovato)

print trovato, i

end ricerca

la struttura di iterazione del linguaggio del flow chart:

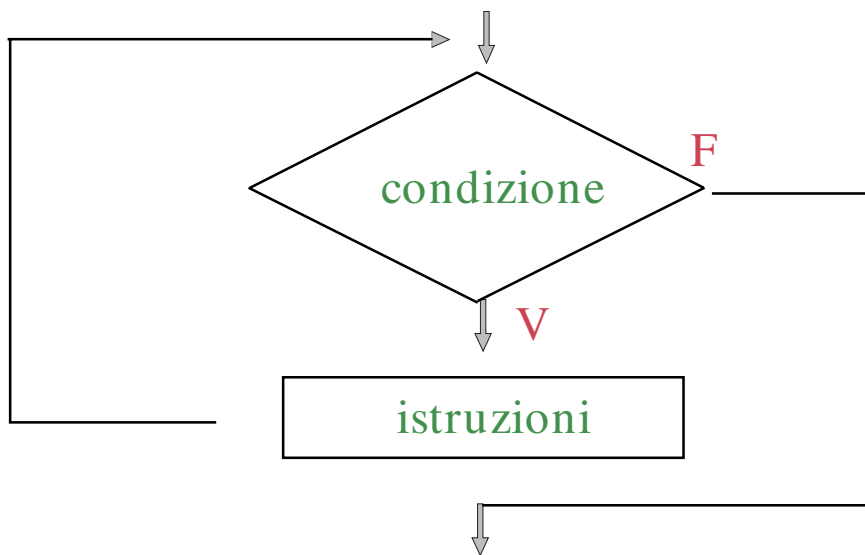


repeat

istruzioni

until (condizione)

la struttura di iterazione del
linguaggio del flow chart:



si traduce nel linguaggio
Pascal-like con

while (condizione) do

istruzioni

endwhile

Differenze fra le tre strutture di iterazione

- Il **for** richiede che sia **noto a priori il numero di iterazioni** da effettuare
- **while** e **repeat** non richiedono tale informazione.
- nel **while** se la condizione è falsa non si esegue nessuna istruzione del ciclo
- nel **repeat** le istruzioni del ciclo sono eseguite almeno una volta

Algoritmo di sequential search

30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di sequential search

1

1

30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di sequential search

2

2

Algoritmo di sequential search

30/10/2020

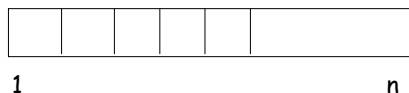
Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di sequential search

3

3

- dato un array di n interi, cercare un elemento dato
(ricerca sequenziale, *sequential search*)

ad ogni passo del procedimento si confronta un elemento dell'array con l'elemento dato; si procede fino a quando si trova l'elemento oppure si è controllato l'intero array



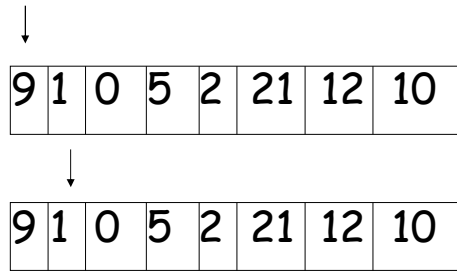
30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di sequential search

4

4

Dato $x=2$ verificare se tale valore è presente nell'array



30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di sequential search

5

5

Algoritmo di sequential search - 1

```
... ..  
i:= 0  
repeat  
    i:= i+1  
until ( x=A(i).OR.i>n )  
if (i>n)then ...  
else ...  
endif  
... ..
```

30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di sequential search

6

6

Algoritmo di sequential search - 2

```
... ..  
i:= 1  
while (A(i) ≠ x .AND. i<=n) do  
    i:= i+1  
endwhile  
  
if (i>n)then ...  
else ...  
endif  
... ..
```

30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di sequential search

7

7

Algoritmo di *sequential search* sotto forma di logical function (con while..do)

```
logical function sequential_search (in: n, A, x )  
    var i, n, x : integer  
    var A : array [1..n] of integer  
    begin  
        i:= 1  
        while (A(i) ≠ x .AND. i<=n) do  
            i:= i+1  
        endwhile  
        if (i>n)then sequential_search:= .FALSE.  
        else  
            sequential_search:= .TRUE.  
        endif  
    end  
end sequential_search
```

30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di sequential search

8

8

```

.....
var i, n, x : integer
var A : array [1..100] of integer
logical function sequential_search ( .. )
oppure
var: sequential_search (..): logical function
begin
    read n, x
    for i:= 1 to n do
        read A(i)
    endfor
    if (sequential_search(n, A, x)) then
        print ('il numero', x, 'è presente nell'array')
    else
        print ('il numero', x, 'non è presente nell'array')
    end
end

```

30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
 Algoritmo di sequential search

9

9

Algoritmo di sequential search: complessità di tempo

- Operazione significativa: operazione di confronto;
- caso peggiore: n confronti
- caso migliore: 1 confronto
- caso medio: $n/2$ confronti

Complessità asintotica $O(n)$

30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
 Algoritmo di sequential search

10

10

- Progettare un algoritmo in P-like sotto forma di procedure

```
procedure sequential_search_con_pos (n, A, x, pos)
```

che, dato un elemento x ed un array A di dimensione n , verifichi se x è presente nell'array A , e restituisca in pos la posizione di x in A
- progettare anche un possibile chiamante

(sugg.: pos può valere -1 in caso di assenza di x in A)

30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
 Algoritmo di sequential search

11

11

```
procedure sequential_search_con_pos(n,A,x,pos)
  var i, x, n, pos : integer
  var A : array [1..n] of integer
  begin
    i:= 1
    while (A(i) ≠ x .AND. i<=n) do
      i:= i+1
    endwhile
    if (i>n)then pos:= -1
    else
      pos:= i
    endif
  end
end sequential_search_con_pos
```

30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
 Algoritmo di sequential search

12

12

```

... ..
var i, x, n, pos : integer
var A : array [1..100] of integer
begin
  read n
  read x
  for i:=1 to n do
    read A(i)
  endfor
  sequential_search_con_pos (n, A, x, pos)
  if pos=-1 then print '...'
  else
    print '..' pos ...
  endif
end

```

30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
 Algoritmo di sequential search

13

13

Progettare la function
logical function sequential_search (in: n, A, x)
 e la procedure
procedure sequential_search_con_pos (n, A, x, pos)
 e successiva variante utilizzando il costrutto
repeat .. until invece di
while ..do

30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
 Algoritmo di sequential search

14

14

Fine sequential search

30/10/2020

Programmazione 9CFU - prof. Giuliano LACCETTI - a.a. 2020/2021
Algoritmo di sequential search

15

Strutture dati dinamiche

parte 1

27/11/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

1

1

27/11/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

2

2

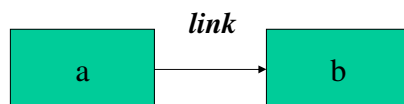
L'organizzazione *a tabella* (e i tipi di dato **array** e **record**) non esaurisce le possibili **relazioni strutturali** dei dati

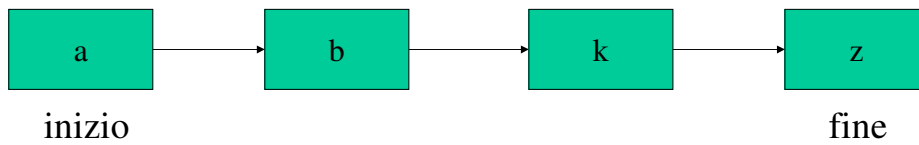
I dati possono essere organizzati in strutture in cui il **numero di componenti varia nel tempo**;

Le strutture possono denotare relazioni più complesse di quella tabellare (ad es. **gerarchica**, ...)

Strutture lineari: stack, coda, lista

In una struttura lineare ogni componente è *“collegata”* al più a 2 componenti:
predecessore e successore





In una struttura dinamica lineare, l'ordine è determinato dal collegamento di un oggetto al suo successore.



Per accedere ad una specifica componente è **necessario accedere a tutte le componenti che la precedono.**

27/11/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

5

5

Le operazioni fondamentali sulle strutture dati dinamiche sono:

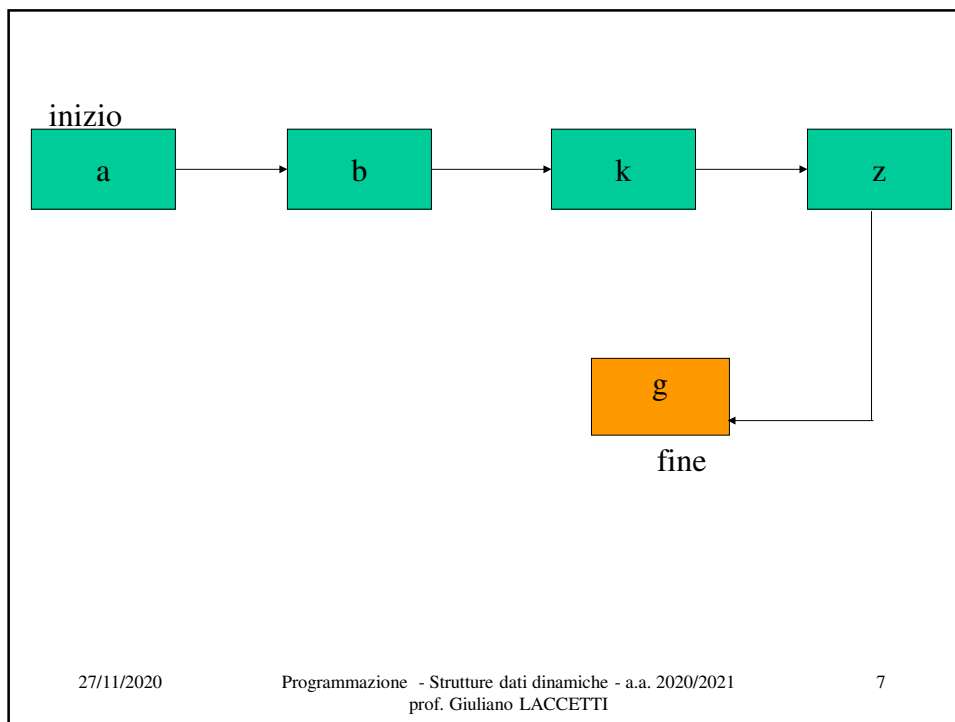
inserimento
eliminazione (o estrazione)
ricerca

27/11/2020

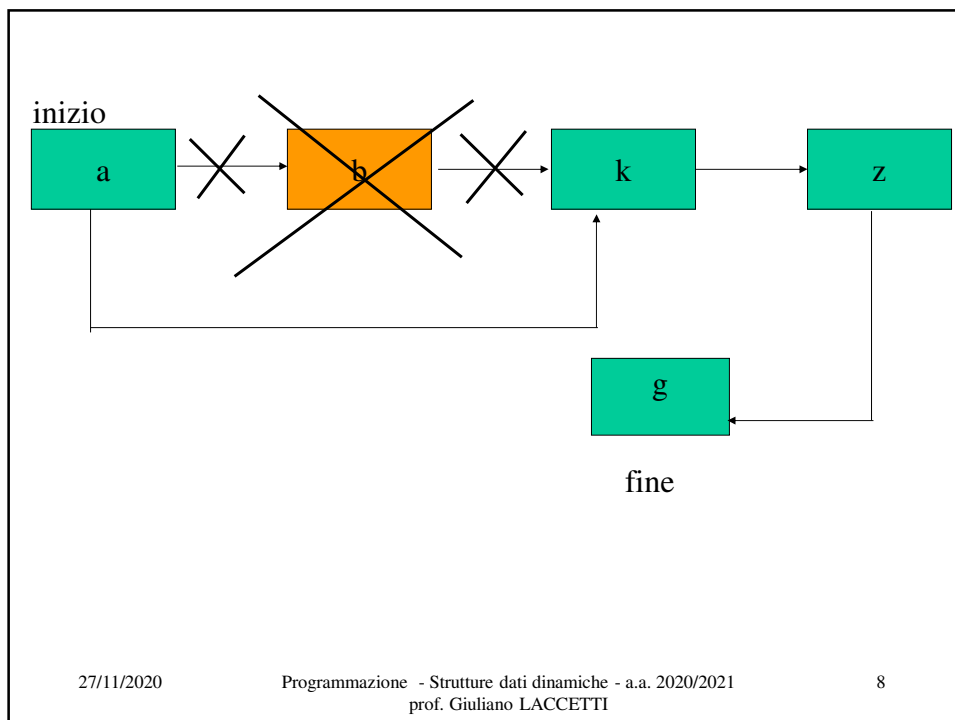
Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

6

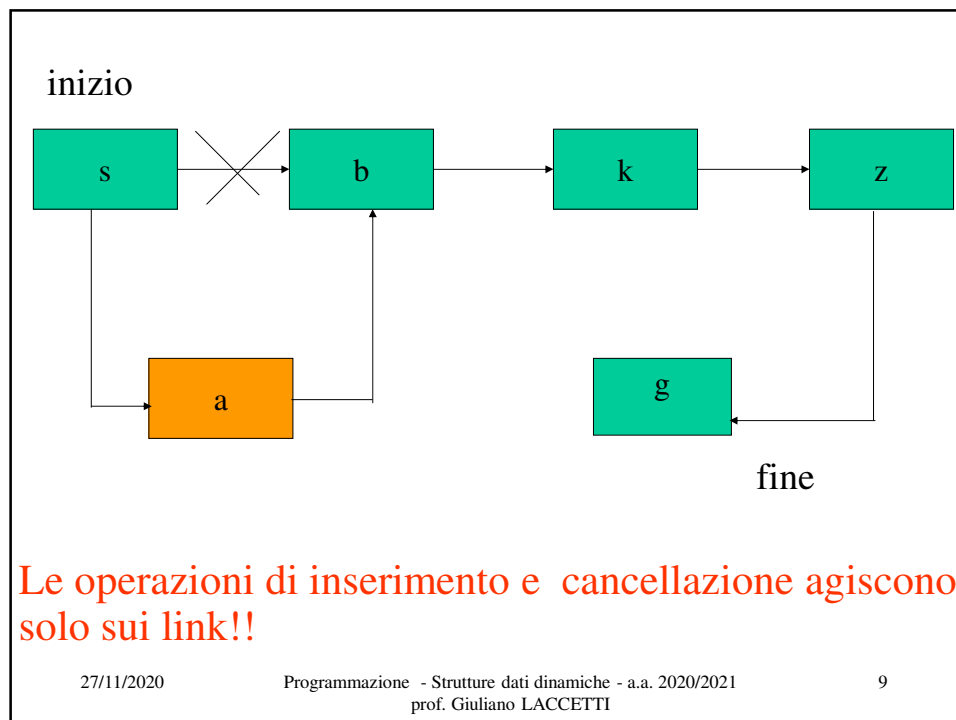
6



7



8

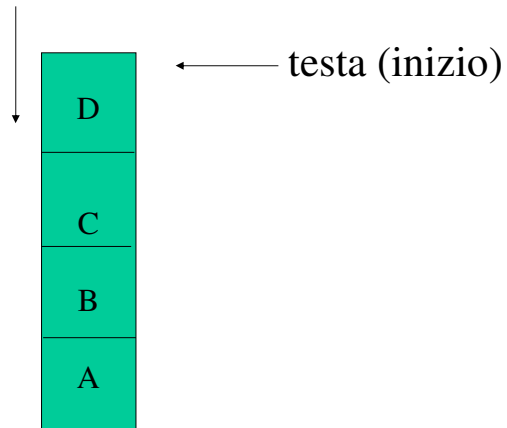


9



10

Inseriamo nello stack A, B, C, D (operazione di *push*)



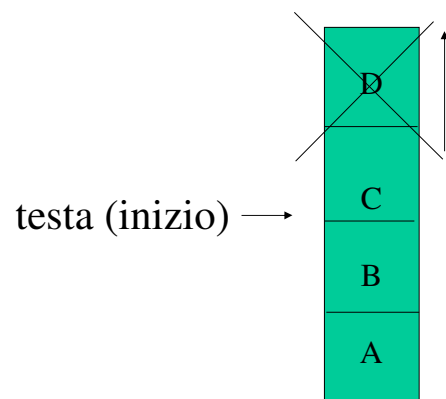
27/11/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

11

11

Estrazione di un elemento (operazione di *pop*)



27/11/2020

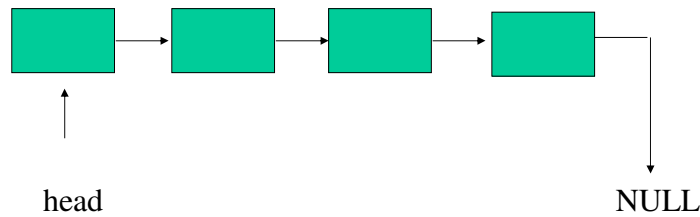
Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

12

12

```
Type stack_pointer : *nodo_stack  
  nodo_stack : record  
    info: ... ..  
    link: stack_pointer  
  end
```

```
var head: stack_pointer
```



27/11/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

13

13

`stack_create`

- dichiarazione di head
- assegnazione head := NULL

`end stack_create`

27/11/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

14

14

`stack_insert (head, elemento) = push (..)`

- allocazione spazio per un nuovo nodo (di tipo `stack_pointer`)
- assegnazione valori ai campi `info` e `link` del nuovo nodo (`new_node.info:=elemento; new_node.link:=head`)
- aggiornamento di `head` (`head:= new_node`, questo è il puntatore al nuovo nodo)

`end stack_insert`

`stack_delete (head, elemento) = pop (..)`

- verifica se lo stack è vuoto (`head == NULL`;
questa può essere una function logica
`is_stack_empty (head))`
- *estrazione* del campo `info` di `head` (`elemento:= head.info`)
- salvataggio di `head` (`temp := head`)
- deallocazione spazio *puntato* da `head`
- aggiornamento di `head` (`head:= temp.link`)

`end stack_delete`

Strutture dati dinamiche - parte 1

fine

27/11/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

17

Strutture dati dinamiche

parte 2

1/12/2020

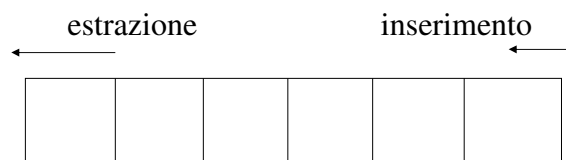
Programmazione II modA - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

1

1

La struttura coda (queue)

La **coda** è una struttura lineare (aperta) in cui è possibile inserire ad un estremo (*fine* della coda) ed estrarre dall'altro (*testa* della coda)



Struttura FIFO

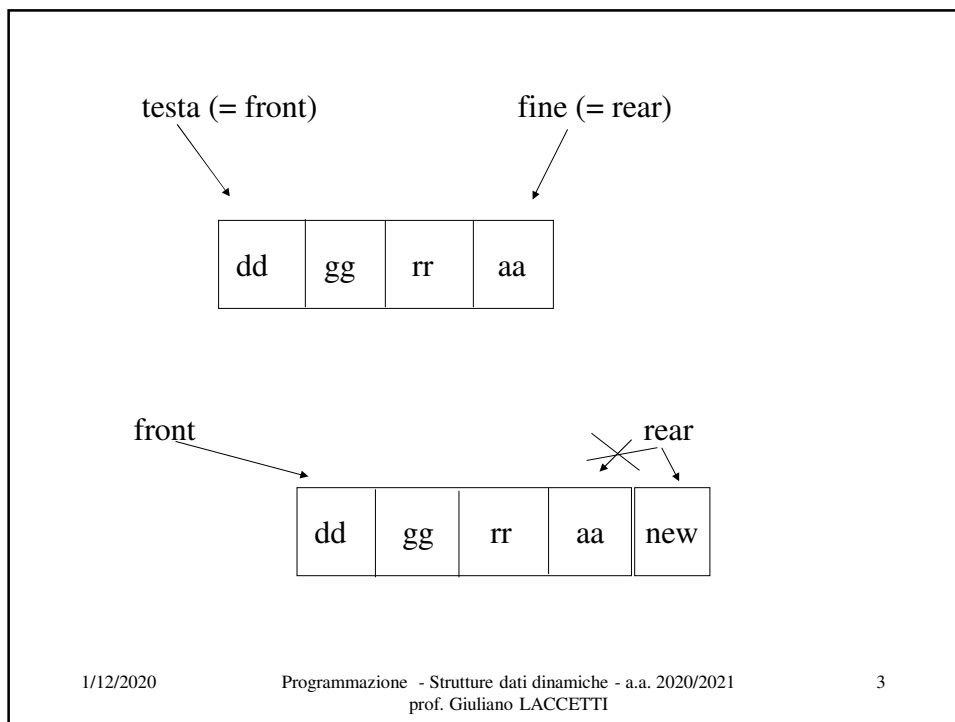
First In First Out

1/12/2020

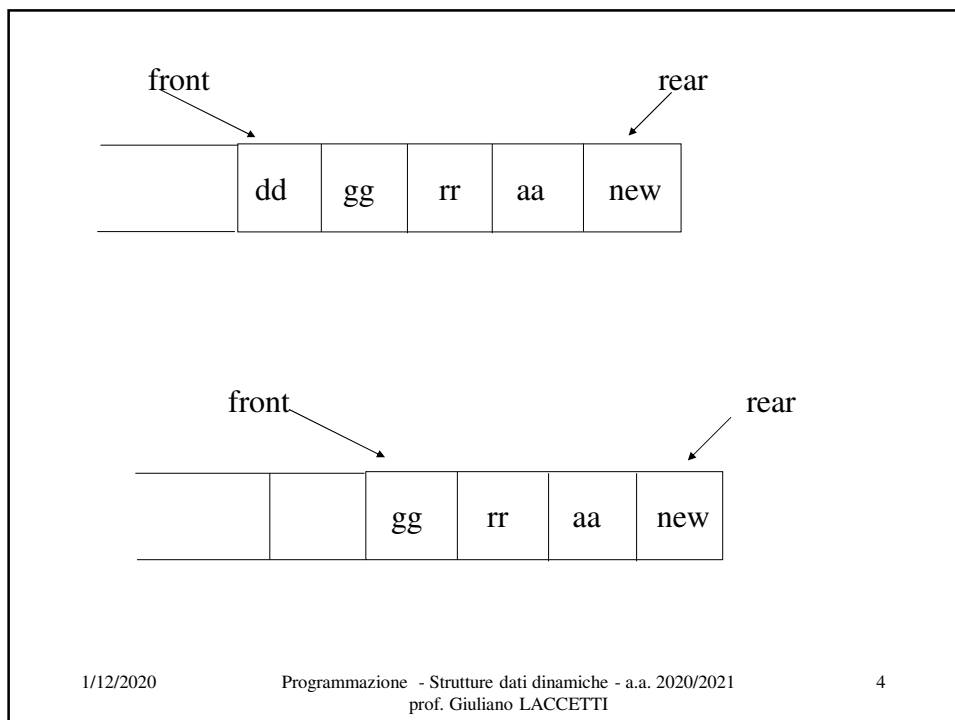
Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

2

2

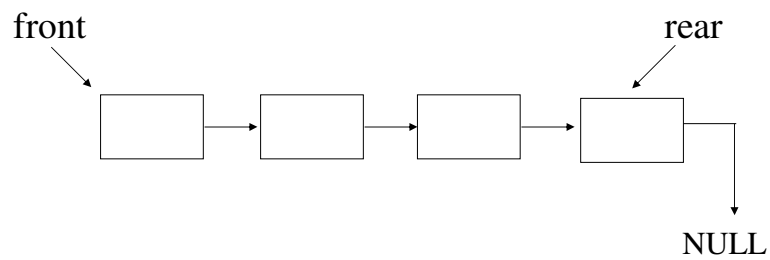


3



4

coda come linked list

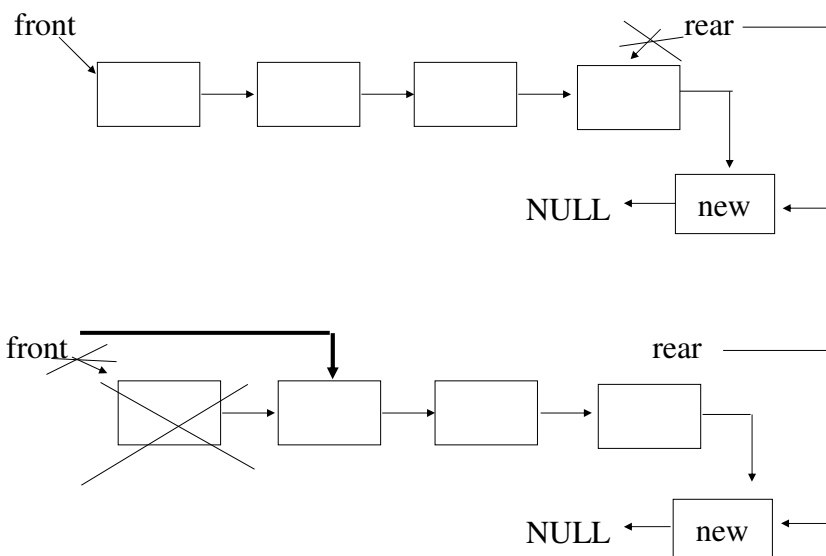


1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

5

5



1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

6

6

```
type queue_pointer : *nodo_queue
  nodo_queue : record
    info: ... ..
    link: queue_pointer
  end

var front, rear : queue_pointer
```

```
queue_create (front, rear)

dichiarazione di front, rear
allocazione spazio per front e rear
assegnazione front=rear := NULL

end queue_create
```


queue_insert (front, rear, elemento)

- * allocazione spazio per un nuovo nodo (new_node di tipo queue_pointer)
- * assegnazione valori ai campi info e link del nuovo nodo (new_node.info := elemento ; new_node.link := NULL)
- * verifica se la coda è vuota (front == rear == NULL; questa può essere una function logica is_queue_empty (front,rear))
- *.1 coda vuota
aggiornamento di front e rear (front=rear := new_node)
- *.2 coda non vuota
 - ° aggiornamento del campo link del precedente rear (rear.link := new_node adesso il “vecchio” rear non è più l’ultimo e “punta” al “nuovo” ultimo, cioè new_node)
 - ° aggiornamento di rear (rear := new_node, questo è il puntatore al nuovo “ultimo” nodo)

end queue_insert

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

9

9

queue_delete (front, rear, elemento)

- * verifica se la coda è vuota (front == rear == NULL;)
- * coda non vuota
- * salvataggio di front (temp := front)
- *.1 un solo elemento (front == rear)
 - * aggiornamento di front e rear (rear = front := NULL)
- *.2 due o più elementi
 - * aggiornamento di front (front := temp.link)
- * estrazione dell’elemento in testa (elemento:= temp.info)
- * deallocazione spazio *puntato* da temp

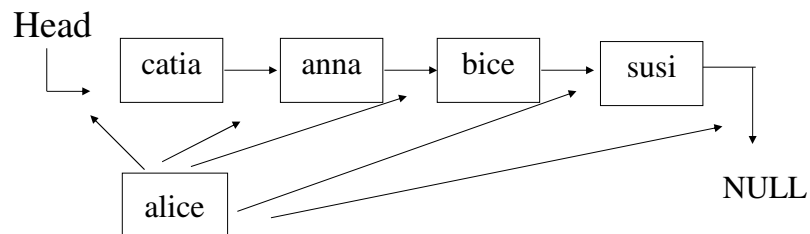
end queue_delete

1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

10

10



Il nodo alice si può inserire in qualunque posto !!

(caratteristica di una linked list “generica”)

“definizione” nodo di una linked list

```

type list_pointer : *nodo_lista
  nodo_lista : record
    info: ... ..
    link: list_pointer
  end

```

```

var head : list_pointer

```

linked_list_create (head)

dichiarazione di head

assegnazione head := NULL

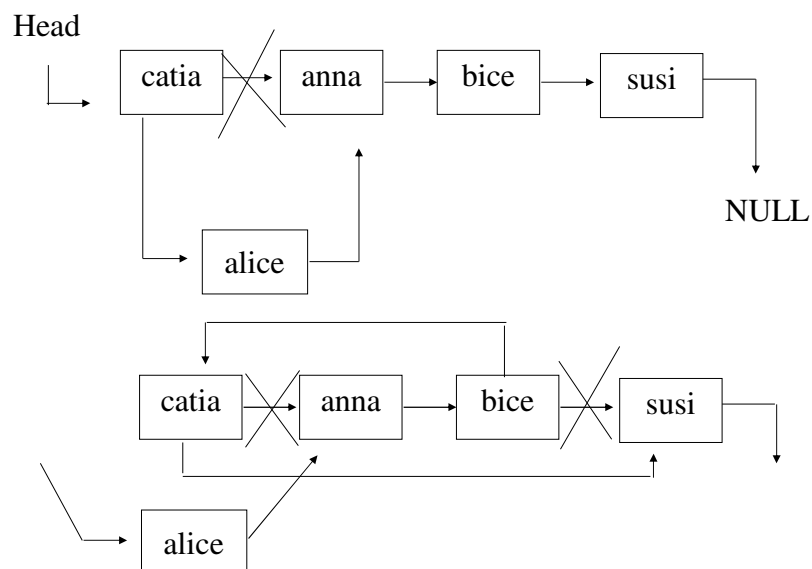
end linked_list_create

1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

13

13



1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

14

14

list_insert (head, nodo_dopo_cui, elemento)

```
* allocazione spazio per un nuovo nodo (new_node di tipo
list_pointer)
* assegnazione di valori a info di new_node
      (new_node.info := elemento)
*.1 nodo_dopo_cui == NULL -> inserimento in testa
      (new_node.link:=head; head:= new_node)
*.2 nodo_dopo_cui != NULL
      aggiornamento di new_node.link e nodo_dopo_cui.link
      (new_node.link:=nodo_dopo_cui.link;
      nodo_dopo_cui.link:=new_node)
```

end list_insert

1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

15

15

list_visualize (head)

```
* visualizza i campi info di tutti i nodi in sequenza, a
partire da head
```

```
first:=head
```

```
while first!=NULL
```

```
      visualizzazione del campo info di first
```

```
      first:=first.link
```

```
endwhile
```

end list_visualize

1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

16

16

I problemi che implicano lo "scorrimento" di una lista, si risolvono elegantemente con la ricorsione .

```
list_visualize_ricorsiva (head_list)

.....
if (head_list != NULL) then
    visualizzazione del campo info di head_list
    list_visualize_ricorsiva(head_list.link)
else
    ... (fine lista)
endif

end list_visualize_ricorsiva
```

1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

17

17

list_search (head, elemento, nodo_in_cui, trovato)

•Ricerca il nodo contenente un campo info assegnato, restituendo il pointer a tale nodo, e l'informazione trovato oppure no

```
trovato:=false
temp:=head
while (not trovato .AND. temp !=NULL)
    if (temp.info=elemento)then
        trovato:=true
        nodo_in_cui:=temp
    endif
    temp:=temp.link
endwhile
```

.....

1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

18

18

list_ordered_insert (head, elemento)

```
** Parametri di input:  head, elemento
** Parametri di output: head
* Allocazione spazio per un nuovo nodo (new_node di tipo list_pointer)
* Assegnazione di valori a info di new_node (new_node.info := elemento)
* Ricerca posizione in cui inserire il nuovo nodo
  previous:= NULL
  if head != NULL then
    temp:= head
    while ( (NOT is_list_empty(temp)) AND (temp.info<elemento) )
      previous:= temp
      temp:=temp.link
    endwhile
  endif
* Inserimento nodo
  list_insert(head, previous, elemento)
  .....
```

end list_ordered_insert

1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

19

19

- Data una linked list ordinata secondo il campo key, progettare un algoritmo ricorsivo per l'inserimento di un nuovo nodo nella lista, mantenendo l'ordine.

```
procedure recursive_ordered_insert (head, new_nodo)
  if (head=NULL) then
    new_nodo.link:= head
    head:= new_nodo
  else
    if (new_nodo.key < head.key) then
      new_nodo.link:= head
      head:= new_nodo
    else
      recursive_ordered_insert(head.link, new_nodo)
    endif
  endif
```

1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

20

20

ESERCIZI

1. Data una lista con campo info di tipo intero scalare, scrivere una procedura ricorsiva che restituisca il valore max presente nei campi info degli elementi della lista. (hint: pensare che i parametri della procedura/function possono essere il puntatore alla testa della lista e il max corrente ...)
2. stessa traccia, versione iterativa.

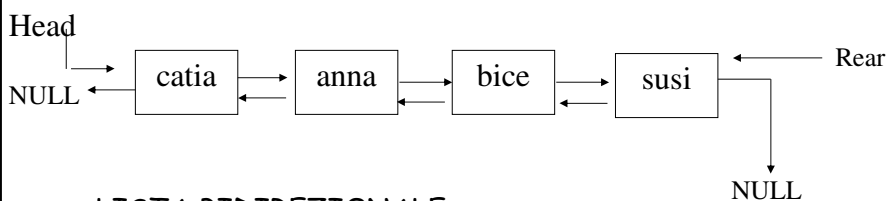
1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

21

21

ESERCIZI



LISTA BIDIREZIONALE

1. Scrivere una procedura per la creazione di una lista bidirezionale (definizione tipo di dato astratto **bidirectional_list**)
2. Scrivere una procedura per inserire e estrarre un elemento in una lista bidirezionale
3. Scrivere una procedura per inserire un elemento in una lista bidirezionale ordinata

1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

22

Strutture dati dinamiche

parte 2 - fine

1/12/2020

Programmazione - Strutture dati dinamiche - a.a. 2020/2021
prof. Giuliano LACCETTI

23

Sia dato un array 2D di tipo integer, di dimensioni $M \times N$, progettare in p like un algoritmo sottoforma di function di tipo logical (function `duerig_duecol`) che restituisca TRUE se vi sono due righe e due colonne consecutive di elementi uguali tra loro, FALSE altrimenti.

```
function duerig_duecol(A, M, N): logical
var: A[M,N]:array of integer
var: i, k, m, n: integer
var: righe, colonne: logical
begin
i:=1
righe:= FALSE
while (i<=M-1 .and. righe= FALSE) do
    righe:= .TRUE.
    k:= 1
    while (k<=N .and. righe:= TRUE) do
        if (A[i,k] != A[i+1,k]) then
            righe:= FALSE
        endif
        k:= k+1
    endwhile
    i:= i+1
endwhile
if (righe = TRUE) then
    k:=1
    colonne:= .FALSE.
    while (k<=N-1 .and. colonne:= FALSE) do
        colonne:= TRUE
        i:=1
        while (i<=M .and. colonne:= TRUE) do
            if (A[i,k] != A[i,k+1]) then
                colonne:= FALSE
            endif
            i:= i+1
        endwhile
        k:=k+1
    endwhile
    if (colonne:= TRUE) then
        duerig_duecol:= TRUE
    else
        duerig_duecol:= FALSE
    end if
else
    duerig_duecol:= .FALSE.
endif
end function
```

Sia dato un array 2D A di dimensioni MxN di tipo integer. Progettare un algoritmo in P-like sottoforma di procedure (procedure elementi_non_di_bordo) che restituisca in output un array 2D COORDINATE in cui per ogni riga ci siano le coordinate (i,j) degli elementi non di bordo dell'array A che hanno come elementi adiacenti (nord, est, sud, ovest) elementi di valore minore.

```
procedure elementi_non_di_bordo(in: a, m, n; out: coordinate)
var: i, j, k, m, n: integer
var: a[m,n], coordinate[k,2]: array of integer
var: nord, sud, est, ovest: integer
k:=1
for i:= 2 to m-1 do
    for j:= 2 to n-1 do
        nord:= a(i-1,j)
        sud:= a(i+1,j)
        est:= a(i,j+1)
        ovest:= a(i,j-1)
        if(a(i,j)>nord .AND. a(i,j)>sud .AND. a(i,j)>est .AND. a(i,j)>ovest) then
            coordinate(k,1):=i
            coordinate(k,2):=j
            k:=k+1
        endif
    endfor
endfor
endfor
end
```

Sia dato un array A 2D di tipo character di dimensioni NxM. Progettare in P-like un algoritmo sottoforma di function (logical function carattere_in_riga (A, N,M) che restituisca in output TRUE se nessuno dei caratteri presenti in una riga è presente nella riga successiva, FALSE altrimenti.

```
function carattere_in_riga(A, N, M): logical
var: i, j, k, n, m: integer
var: A[n, m]: array of integer
var: controllo: logical
begin
  controllo:= .TRUE.
  i:= 1
  while (i<=N-1 .AND. controllo:= .TRUE.) do
    j:= 1
    while k<=M .AND. controllo:= .TRUE. do
      k:= 1
      while k<=M .AND. controllo:=.TRUE. do
        if A[i,j]= A[i+1,k] then
          controllo:= .FALSE.
        endif
        k:= k+1
      endwhile
      j:= j+1
    endwhile
    i:= i+1
  endwhile
  carattere_in_riga:= controllo
end function
```

Si consideri la successione $a_1=0, a_2=1, a_3=2 \dots$ con $n>3$. Progettare in p like, sottoforma di function (logical function controllo_numeri_successione (N1,N2,N3)) che, dati 3 numeri interi $N_1 < N_2 < N_3$, con $N_1 > a_3$, restituisca TRUE se il numero di elementi della successione compresi tra N_1 e N_2 (cioè $N_1 \leq a_3 \leq N_2$) è uguale al numero di elementi maggiori di N_2 e minori o uguali di N_3 , FALSE altrimenti. Progettare anche una chiamante.

```
function controllo_numeri_successione (N1, N2, N3): logical
var: temp: integer
var: a1, a2, a3, cont1, cont2: integer
begin
cont1:= 0
cont2:= 0
a1:= 0
a2:= 1
a3:= 2
temp:= a1+2*a2+a3
while (temp <= N3) do
    if (temp >= N1 .AND. temp <= N2) then
        cont1:= cont1+1
    endif
    if (temp > N2 .AND. temp <= N3) then
        cont2:= cont2+1
    endif
    a1:= a2
    a2:= a3
    a3:= temp
    temp:= a1+2*a2+a3
endwhile
if (cont1:= cont2) then
    controllo_numeri_successione:= .TRUE.
else
    controllo_numeri_successione:= .FALSE.
endif
End function
```

Chiamante:

```
var: n1, n2, n3: integer
var: controllo_numeri_successione: logical function
begin
read n1, n2, n3
controllo_numeri_successione(n1, n2, n3)
if (controllo_numeri_successione(n1, n2, n3)) then
    print "vero"
else
    print "falso"
endif
end
```

Dato un array 2D A di tipo intero, di dimensione ALPHAxBETA, con ALPHA pari, BETA>5, progettare in P-like un algoritmo, sotto forma di function logica, (quattro_elementi_uguali_somma_altri (A, ALPHA, BETA)) che restituisca TRUE se la somma del primo, terzo, quarto, e penultimo elemento di tutte le righe pari è uguale alla somma dei restanti elementi della medesima riga, FALSE altrimenti.

```
Function quattro_elementi_uguali_somma_altri (A, ALPHA, BETA): logical
Var: i, j, alpha, beta, somma1, somma2: integer
Var: A[alpha, beta]: array of integer
Var: controllo: logical
Begin
Somma1:= 0, Somma2:= 0
i:= 2
Controllo:= false
while (i<alpha .and. controllo= false) do
    somma1:= A[i,1] + A[i, 3] + A[i, 4] + A[i, beta - 1]
    somma2:= A[i, 2] + A[i, beta]
    for j:=5 to beta-2 do
        somma2:= somma2 + A[i,j]
    end for
    if (somma1 = somma2) then
        controllo:= true
    else
        i:= i+2
    endif
end while
quattro_elementi_uguali_somma_altri:= controllo
end function
```

Sia dato un array 2D A di tipo integer, di dimensioni HxK. Progettare in P-like un algoritmo sotto forma di function di tipo logical (function sommaRIGA_uguale_elemento) che restituisca TRUE se la somma degli opposti degli elementi di una riga è uguale ad uno degli elementi della riga successiva, FALSE altrimenti.

```
function sommaRIGA_uguale_elemento(A, h, k): logical
var: h, k, somma, i, j, s: integer
var: a[h, k]: array of integer
var: controllo: logical
begin
  controllo:= false
  i:=1
  while (i<h-1 .and. controllo= false) do
    for j:=1 to k do
      somma= somma + (-A[i,j])
    end for
    s:= 1
    while (s<=k and controllo= false) do
      if (somma = A[i+1, s]) then
        controllo:= true
      end if
      s:= s+1
    end while
    i:= i+1
  end while
  sommaRIGA_uguale_elemento:= controllo
end function
```

Data una lista $head=L1 \rightarrow L2 \rightarrow L3 \rightarrow \dots \rightarrow L_n \rightarrow NULL$, il suo prefisso i -esimo consiste della sotto lista $head(i)=L1 \rightarrow L2 \rightarrow L3 \rightarrow \dots \rightarrow L_n$ (?) con $i \leq n$. Il prefisso i -esimo di $head$, vale a dire, è costituito dai suoi primi i elementi. Il peso di una lista $head$ (con il campo `info` di tipo `integer`) è la somma dei valori dei campi `info`. Data una lista $head$ con il campo `info` di tipo `integer` ed un intero `PESO`, supponendo che i valori dei campi `info` siano tutti positivi, si scriva una function ricorsiva di tipo `logical` (function `esiste_prefisso`) per verificare se esiste un prefisso di $head$ di peso `PESO`. Progettare in `p like` anche un algoritmo che richiami la function

```
function esiste_prefisso(head, peso, somma): logical
var: peso, somma: integer
var: head: list_pointer
Begin
if(head=NULL) then
    esiste_prefisso:=FALSE
else
    somma:= somma+head.info
    if(somma<peso) then
        esiste_prefisso:= esiste_prefisso(head.link, peso, somma)
    else if(somma>peso) then
        esiste_prefisso:=FALSE
    else if(somma=peso) then
        esiste_prefisso:= true
    endif
endif
end if
end
```

Chiamante :

```
var: head: list_pointer
var: peso, somma, ele, n, i : integer
var: esiste_prefisso: logical function
var: create_list, push: list_pointer function
begin
create_list(head)
read peso, n
somma:= 0
for i:=1 to n
    read ele
    push(head, ele)
endfor
esiste_prefisso(head,peso,somma)
if(esiste_prefisso(head,peso,somma))
    print "vero"
else
    print "falso"
endif end
```


Data una linked list head con il campo info di tipo integer, progettare un algoritmo in p like sottoforma di procedure (procedure list_sort) che, utilizzando il metodo exchange sort, e che restituisca la lista head ordinata in senso crescente secondo i valori del campo info. Ad esempio, se head= 3->5->6->4->3, l'output sarà head=2->3->4->5->6

```
Procedure list_sort(head, n)
var: i, j, n: integer
var: head, succ, temp: list_pointer
var: sorted: logical
begin
sorted:= FALSE
succ:= head.link
i:=1
while (i<=n AND sorted= false) do
    sorted:=TRUE
    for j:=1 to n-i do
        if (head.info > succ.info) then
            temp:= head.info
            head.info:= succ.info
            succ.info:= temp
            sorted:= FALSE
        endif
        head:= head.link
        succ:= head.link
    endfor
    i:=i+1
endwhile
end
```

Data una linked list list, con il campo info di tipo integer, progettare in P like un algoritmo sottoforma di function ricorsiva, function elimina_da_lista(list,elem) che restituisca la lista privata degli elementi con i campi info uguali a elem

```
function elimina_da_lista (head, elem): list_pointer
var: head, temp: list_pointer
var: elem: integer
begin
if head:= NULL then
    elimina_da_lista:= head
else
    temp:= head.link
    if head.info = elem then
        head:= temp
        elimina_da_lista:= elimina_da_lista(head, elem)
    else if temp.info = elem then
        head.link:= temp.link
        elimina_da_lista:= elimina_da_lista(head.link, elem)
    else
        elimina_da_lista:= elimina_da_lista(head.link, elem)
    end if
end if
end if
end
```

Dato un array A di tipo integer, di dimensione n, un sottoarray di A è formato di elementi contigui di A, ed il suo peso è la somma dei valori di tali elementi. Se $A=(1,2,3,4,5,6,7,8,9)$, il sottoarray $A(2...5)$ è uguale a $(2,3,4,5)$ e il suo peso è 14, mentre $A[4...7]$ è uguale a $(4,5,6,7)$ ed il suo peso è 22. Progettare in p like una function ricorsiva (logical function exist_sottoarray_didatopeso) che, dato un intero PESO ed un array A di dimensione N, di tipo integer, restituisca TRUE se esiste un sottoarray di A di peso PESO, FALSE altrimenti. Progettare anche una versione iterativa di tale funzione

iterativo:

```
function exist_sottoarray_didatopeso (a, n, peso): logical
var: i, j, n, ps, peso: integer
var: a[n]: array of integer
var: controllo: logical
begin
  controllo:= FALSE
  i:=1, j:=i
  while (i<=n AND controllo =FALSE) do
    ps:=0
    while (ps < peso AND j<=n) do
      ps:= ps+a(j)
      j:=j+1
    endwhile
    j:=1
    if (ps=peso) then
      controllo:=TRUE
    else if (ps > peso) then
      i:=i+1, j:=i
    endif
  endwhile
end
```

ricorsivo:

```
function exist_sottoarray_didatopeso (a, i, j, n, peso, ps): logical
var: i, j, n, ps, peso: integer
var: a[n]: array of integer
var: controllo: logical
begin
  if (i=n AND ps != peso) then
    controllo:= FALSE
  else
    if (ps=peso) then
      controllo:= TRUE
    else if (i<n) then
      if (ps < peso AND j<n) then
        ps:= ps+a(j)
        exist_sottoarray_didatopeso (a,i,j+1,n,peso,ps)
      else if (ps > peso)
        exist_sottoarray_didatopeso (a,i+1,i+1,n,peso,0)
      endif
    endif endif endif end
```

Data una linked list head con il campo info di tipo intero progettare un algoritmo ricorsivo in p like che elimini dalla lista tutti gli elementi che nel campo info contengono il valore dato, ELEM. Si organizzino in 2 procedure, entrambe ricorsive, una che elimina ELEM dalla testa del_testa(head,ELEM), una che elimina ELEM dal mezzo della lista del_mezzo(head,ELEM), ed una terza procedura, delete_elem (head, ELEM) che utilizza le altre due.

```
procedure del_testa (head, elem)
var: elem: integer
var: head, temp: list_pointer
begin
if (head!=NULL AND head.info=elem) then
    temp:= head
    head:= temp.link
    del_testa:= del_testa (head, elem)
endif
end
```

```
procedure del_mezzo (head, elem)
var: elem: integer
var: head,temp: list_pointer
if (head!=NULL)
    if(head.info=elem)
        temp:=head
        head:=temp.link
        del_mezzo:= del_mezzo (head, elem)
    else
        del_mezzo:= del_mezzo (head.link, elem)
    endif
endif
end
```

```
procedure delete_elem(head, elem)
var: elem: integer
var:head: list_pointer
var: del_mezzo, del_testa: procedure
begin
del_testa (head, elem)
del_mezzo (head.link, elem)
end
```

Sia A una matrice di dimensione $n \times n$, contenente numeri interi. Sviluppare in P-like un algoritmo ricorsivo, sottoforma di function di tipo logical (function diagonale (A,riga)), con A array 2D di tipo integer, di dimensioni $n \times n$, implementazione della matrice A, che restituisca TRUE se tutti gli elementi della diagonale principale sono nulli, FALSE altrimenti. La chiamata main è del tipo:
matrice_diagonale:= diagonale (A,n)

```
function diagonale (a,riga): logical
var: riga: integer
var: a[riga,riga]: array of integer
Begin
If (a(1,1) = 0) then
    diagonale:= true
else
    diagonale:= false
endif
if (a (riga,riga) = 0)
    diagonale (a, riga-1)
else
    diagonale:= false
endif
```

Una rotazione di vettore x è il vettore che si ottiene spostando ogni componente del vettore un certo numero di posti verso sinistra, assumendo che il vettore sia circolare.

Ad esempio, ruotando di 3 posti il vettore $x=(0;1;2;3;4;5;6;7;8;9)$ si ottiene il vettore $xrot=(3;4;5;6;7;8;9;0;1;2)$. Progettare in P-like un algoritmo sottoforma di procedure (procedure rotazione_array) che, dati due array X e Y , di dimensione n , restituisce in output una variabile rotazione di tipo character che vale 's' se uno è la rotazione dell'altro, 'n' altrimenti.

```
procedure rotazione_array (x,y,n)
var: i, j, k, n: integer
var: x[n], y[n]: array of integer
var: flag, rot: logical
begin
  rot:= FALSE
  i:=1 j:=1 k:=1
  while (j<=n AND rot=FALSE) do
    if (x(i) != y(j)) then
      k:=k+1
      i:=k
    else
      if(x(i) = y(j))
        j:=j+1
        i:=i+1
        if(i>n AND k>1)
          i:=1
          flag:=TRUE
          while(i<=k AND j<=n AND flag=TRUE)
            if(x(i) = y(j))
              flag:=TRUE
            else
              flag:=FALSE
            endif
            j:=j+1
            i:=i+1
          endwhile
          if (flag=TRUE) then
            rot:=TRUE
          else
            j=n+1
            rot:=FALSE
          endif
        endif
      endif
    endif
  endwhile
endwhile
```

Progettare un algoritmo in P-like, sottoforma di function ricorsiva (function campoinfo_uguale) per verificare se i valori dei campi info, di tipo intero, degli elementi di una data linked list head, contengono valori uguali; la function restituisce TRUE se ciò è verificato, FALSE altrimenti.

```
Function campoinfo_uguale(head): logical
Var: head: list_pointer
Var: controllo: logical
Begin
If (head = null) then
    Controllo:= true
else
    Succ:= head.link
    If (head.info = succ.info) then
        Campoinfo_uguale(head.link)
    Else
        Controllo:= false
    Endif
Endif
Campoinfo_uguale:= controllo
end
```

Dati 2 stack head1 e head2 “ordinati”, progettare in P like un algoritmo per la costituzione di uno stack head3, anch’esso “ordinato”, merge tra head1 e head2. Utilizzare la procedure pop (head,elem) e push (head,elem).

```
Procedure merge (in: Head1, Head2 out: Head3)
Var: elem: integer
Var: head1, head2, head3: stack_pointer
begin
while Head1 != NULL .AND. Head2 != NULL do
    if Head1.info <= Head2.info then
        pop(head1, elem)
        push(head3, elem)
    else if Head2.info <= Head1.info then
        pop(head2, elem)
        push(head3, elem)
    endif
endif
endwhile
end
```

Sia head una linked-list i cui campi info possono essere 0 e 1 in modo che head rappresenti sia un numero binario. La lista head è implementata con puntatori doppi (ogni elemento punta al precedente e al successivo). Progettare in P like un algoritmo sottoforma di procedure, procedure incrementa (head, r) che incrementi di 1 il valore rappresentato da head. La chiamata del main è del tipo $r:=1/r$ è il riporto/u, incrementa (head, r)

```

procedure incrementa (head, r)
var: head, newnode: list_pointer
var: r: integer
Begin
While (head != NULL) do
    if (head.next != NULL) then
        if (head.info=1 AND r=1)
            head.info:= 0
            if(head.next.info=0)
                head.next.info:=1
                r:=0
            else
                head.next.info:=0
                r:=1
            endif
        else
            if(head.info=0 AND r=1)
                head.info:=1
                r:=0
            else
                if(head.info=1 AND r=0)
                    head.info:=1
                    r:=0
                else
                    if(head.info=1 AND r=1) then
                        head.info:=0
                        newnode.info:=1
                        newnode.prev:=head
                        newnode.next:=NULL
                        head:=newnode
                        r:= 0
                    else
                        if(head.info=0 AND r=1)
                            head.info:=1
                            r:=0
                        else
                            if(head.info=1 AND r=0)
                                head.info:=1
                                r:=0
                            endif
                        endif
                    endif
                endif
            endif
        endif
    endif
    head:=head.next endwhile
head:=head.next endwhile

```


Dato un array A di dimensione N , di tipo **integer**, la finestra $[i,j]$ (si supponga $i < j$) è un sottoarray di A comprendente le posizioni di i a j incluse. Il peso di una finestra è dato dalla somma dei suoi elementi. Se un intero k è strettamente compreso tra i e j si dice che k taglia la finestra. Progettare in **p like** un algoritmo, sottoforma di procedure (procedure peso_massimo ($A, N, k, \text{max_weight}, \text{flag}$)) con parametri di output max_weight e flag , che restituisca in max_weight il peso massimo tra tutte le finestre tagliate da k , $\text{flag}=1$ segnala qualche anomalia, $\text{flag}=0$ indica “tutto bene”, cioè il valore in max_weight è proprio il risultato che si attende (prevedere, nell’algoritmo, possibili situazioni anomale per cui assegnare 1 a flag)+ un possibile chiamante, prima di visualizzare il contenuto di max_weight , cosa deve prevedere? Stesso esercizio, supponendo che i valori di A siano tutti positivi.

```

procedure Peso_massimo (in: A, k, N; ou: Max_weight, Flag)
var : A [ N ] : array of integer
var : k, N, Max_weight := 0, Flag := 0, i := 1, j := N, Sum, x : integer
begin
if (k <= 1 .OR. k >= N) then
    Flag := 1
else
    while (i <= k) do
        while (j >= k) do
            Sum := 0
            for x := i to j do
                Sum := Sum + A(x)
            endfor
            if (Max_weight < Sum) then
                Max_weight := Sum
            endif
            j := j-1
        endwhile
        i := i+1
        j := N
    endwhile
endif
end

```

```

procedure Peso_massimo (in: A, k, N ; out: Max_weight, Flag)
var : A [N] : array of integer
var : k, N, Max_weight := 0, Flag := 0, i := 1 : integer
begin
if (k <= 1 .OR. k >= N) then
    Flag := 1
else
    for i := 1 to N do
        Max_weight := Max_weight + A[i]
    endfor
endif
end

```

Progettare in p like un algoritmo sottoforma di function (logical function precedente_maggiore_sommasuccessivi) che data una linked list head (con campo info di tipo integer), restituisca TRUE se ciascun valore nei campo info è maggiore della somma di tutti i valori dei campi info degli elementi successivi, FALSE altrimenti. Utilizzare una function ricorsiva (function somma_valori_campinfosuccessivi), pure da progettare, in p like, che, data una linked list, restituisca la somma dei valori dei campi info (di tipo integer) dei suoi elementi. Progettare anche in p like, un algoritmo che richiami la function precedente_maggiore_sommasuccessivi

```
function somma_valori_campinfosuccessivi (head): integer
var: somma: integer
var: head: list_pointer
begin
if (head = NULL)
    somma_valori_campinfosuccessivi:= somma
else
    somma:= somma+head.info
    somma_valori_campinfosuccessivi:= somma_valori_campinfosuccessivi (head.link)
endif
end
```

```
function precedente_maggiore_sommasuccessivi (head): logical
var: head: list_pointer
var: somma: integer
var: controllo: logical
var: somma_valori_campinfosuccessivi: integer function
begin
controllo:= TRUE
while (head != NULL AND controllo= TRUE)
    if (head.info<= somma_valori_campinfosuccessivi (head.link))
        controllo:=FALSE
    endif
    head:= head.link
endwhile
precedente_maggiore_sommasuccessivi:= controllo
end
```

```
chiamante:
var: head: list_pointer
var: n, ele, i: integer
var: precedente_maggiore_sommasuccessivi: logical function
begin
read n
create_list(head)
for i:=1 to n do
    read ele
    push(head,ele)
endfor
precedente_maggiore_sommasuccessivi(head)
if(precedente_maggiore_sommasuccessivi(head))
    print "vero"
else
    print "falso"
endif end
```

Siano date due linked list, head1 e head2, ordinate secondo il campo info di tipo integer, progettare in p like una procedura ricorsiva (procedure merge_ordered_list) che restituisca una linked list head3, ordinata secondo il campo info, risultato del merge tra head1 e head2. Progettare in p like anche un algoritmo che richiami tale procedura.

```

procedure merge_ordered_list (head1, head2, head3)
var: head1, head2, head3, newnode: list_pointer
Begin
newnode:= NULL
if (head1!=NULL AND head2!=NULL)
    if (head1.info <= head2.info)
        head3.info:= head1.info
        head3.link:= newnode
        merge_ordered_list (head1.link,head2,head3.link)
    else
        if (head1.info > head2.info)
            head3.info:= head2.info
            head3.link:= newnode
            merge_ordered_list (head1, head2.link, head3.link)
        endif
    endif
endif
if (head1=NULL AND head2!=NULL)
    head3.info:= head2.info
    head3.link:= newnode
    merge_ordered_list (head1,head2.link,head3.link)
else
    if (head1!=NULL AND head2=NULL)
        head3.info:= head1.info
        head3.link:= newnode
        merge_ordered_list (head1.link,head2,head3.link)
    endif
endif
endif
end

```

```

chiamante:
begin
var: head1, head2, head3: list_pointer
var: ele1, ele2, n1, n2, i: integer
var: merge_ordered_list: procedure
var: create, push: list_pointer function
head1:=list_create(head1), head2:=list_create(head2), head3:=list_create(head3)
read n1, n2
for i:=1 to n1
    read ele1
    push(head1,ele1)
endfor
for i:=1 to n2 do
    read ele2
    push(head2,ele2)
endfor
merge_ordered_list (in:head1,head2;in/out:head3)
end

```

Un pettine è una struttura dato che consiste in una lista di liste, vale a dire che una linked list comb in cui, ciascun elemento è costituito da altri 2 campi info di tipo link, link al successivo elemento di tipo pettine, link ad una linked list

Un pettine si dice di alta moda se ogni linked list è più lunga di un solo elemento della procedure.

Supponendo che un dato pettine possa essere definito nel seguente modo, progettare in p like sottoforma di function ricorsiva (integer function lenght(list)) che restituisca la larghezza (=numero di elementi di una linked list) e progettare in p like un algoritmo sottoforma di function ricorsiva (logical function is_highfashion_comb(comb)) che restituisca TRUE se un pettine comb è un pettine di alta moda, FALSE altrimenti.

```
function lenght(list): integer
Var: list: list_pointer
begin
  If(list!=NULL) then
    Lenght:=lenght(list.link)+1
  Endif
End
```

```
function is_highfashion_comb (comb): logical
Var: comb: comb_pointer
Is_highfashion_comb:= .TRUE.
If(comb.linkC!=NULL) then
  If(lenght(comb.linkL)<lenght((comb.linkC).linkL)) then
    Is_highfashion_comb:= is_highfashion_comb(comb.linkC)
  Else
    Is_highfashion_comb:= .FALSE.
  Endif
Endif
End
```

Date 3 linked list list1,list2,list3, progettare in p like un algoritmo sottoforma di function ricorsiva(logical function somma_lista(head1,head2,head3) che restituisca TRUE se head3 è la somma di head1 e head2, elemento per elemento. FALSE altrimenti. Scrivere anche con possibile chiamante per tale funzione.

```
logical function somma_lista (head1,head2,head3)
var: head1, head2, head3: list_pointer
Begin
If (head1!=NULL AND head2!=NULL AND head3!=NULL)
    if(head3.info = head1.info+head2.info)
        somma_lista:= somma_lista (head1.link,head2.link,head3.link)
    else
        somma_lista:=FALSE
    endif
endif
if (head1=NULL AND head2!=NULL AND head3!=NULL)
    if (head3.info = head2.info)
        somma_lista:= somma_lista (head1,head2.link,head3.link)
    else
        somma_lista:=FALSE
    endif
else
    if(head1!=NULL AND head2=NULL AND head3!=NULL)
        if(head3.info = head1.info)
            somma_lista:= somma_lista (head1.link,head2,head3.link)
        else
            somma_lista:=FALSE
        endif
    endif
endif
if(head1=NULL AND head2=NULL AND head3=NULL)
    somma_lista:=TRUE
endif
end
```

Progettare un algoritmo in p like, sottoforma di function ricorsiva (function c) che, data una linked list head con il campo info di tipo intero, ed un elemento x, di tipo intero, restituisca il puntatore dell'elemento della linked list il cui campo info è uguale a x, se esso è presente, NULL altrimenti.

```
function punt_di_nodo (head,x): list_pointer
var: head: list_pointer
var: x: integer
begin
if(head = NULL)
    punt_di_nodo:= NULL
else
    if(head != NULL)
        if(head.info = x)
            punt_di_nodo= head
        else
            punt_di_nodo (head.link,x)
        endif
    endif
endif end
```

Un polinomio può essere rappresentato sottoforma di linked list, in cui ci sono tanti elementi quanti termini dei polinomi, ed ogni elemento contiene un campo coef ed un campo esp, il coefficiente e la potenza della variabile; per esempio il polinomio $P1(x)=7x^3-3x^2+4$ (rappresentazione). Assegnati due polinomi rappresentati da due liste P1 e P2, progettare un algoritmo in p like per determinare il polinomio P3(x) dato dalla somma di P1(x) e P2(x), che restituisca cioè una linked list P3 che rappresenti P3(x).

```

procedure polinomi(p1,p2,p3)
var: p1, p2, p3, newnode: list_pointer
begin
if (p1!=NULL AND p2!=NULL)
    newnode:=NULL
    if(p1.esp > p2.esp)
        p3.esp:=p1.esp
        p3.coef:=p1.coef
        p3.link:=newnode
        polinomi(p1.link,p2,p3.link)
    else
        if(p1.esp < p2.esp)
            p3.esp:=p2.esp
            p3.coef:=p2.coef
            p3.link:=newnode
            polinomi(p1,p2.link,p3.link)
        else
            if(p1.esp = p2.esp)
                p3.esp:=p1.esp
                p3.coef:=p1.coef+p2.coef
                p3.link:=newnode
                polinomi(p1.link,p2.link,p3.link)
            endif
        endif
    endif
endif
end if
if(p1!=NULL AND p2=NULL)
    newnode:=NULL
    p3.esp:=p1.esp
    p3.coef:=p1.coef
    p3.link:=newnode
    polinomi(p1.link,p2,p3.link)
endif
if(p2!=NULL AND p1=NULL)
    newnode:=NULL
    p3.esp:=p2.esp
    p3.coef:=p2.coef
    p3.link:=newnode
    polinomi(p1,p2.link,p3.link)
endif
end

```

Sia una head una linked-list i cui campi info possono essere 0 e 1 in modo che head rappresenti un numero binario. La lista head è implementata con puntatori doppi (ogni elemento punta al precedente e al successivo). Progettare in p like un algoritmo sottoforma di procedure, procedure incrementa(head,r) che incrementi di 1 il valore rappresentato da head La chiamata del main è del tipo r:=1, incrementa (head,r)

```
procedure incrementa(in: head, r)
var: temp, new_node: list_pointer
begin
temp:= head
while temp.next != NULL do
    temp := temp.next
endwhile
while temp != NULL do
    if temp.info:= 1 .AND. r:= 1 then
        temp.info := 0
        if temp.prev:= NULL then
            new_node.info:= 1
            new_node.link:= head
            new_node.prev:= head.prev
            head:= new_node
        endif
    else
        if temp.info:= 0 .AND. r:= 1 then
            temp.info:= 1
            r:= 0
        endif
        temp := temp.prev
    endif
endwhile
end procedure
```

Progettare in P-like un algoritmo sottoforma di procedure (procedure diff_elem_successione) che, dati 2 interi N1 e N2, con $N1 < N2$ e $N1 > an$, restituisca in output l'intero (N1 o N2) che ha una minore differenza rispetto ad un elemento della successione, ed il valore di tale differenza. Progettare in P-like anche un esempio di algoritmo che utilizza la procedura

```

procedure diff_elem_successione (in/out:n1,n2;out:diff)
var: a1, a2, a3, an: integer
var: n1, n2, diff, diff1, diff2: integer begin
begin
a1=0 a2=1 a3=2 an=a1+2*a2+a3
while(an<n1)
    a1:=a2 a2:=a3 a3:=an
    an:=a1+2*a2+a3
endwhile
if(abs(n1-an) < abs(n1-a3))
    diff1:=abs(n1-an)
else
    diff1:=abs(n1-a3)
endif
while(an<n2)
    a1:=a2 a2:=a3 a3:=an
    an:=a1+2*a2+a3
endwhile
if(abs(n2-an) < abs(n2-a3))
    diff2:=abs(n2-an)
else
    diff2:=abs(n2-a3)
endif
if(diff1 < diff2)
    diff:=diff1
else
    diff:=diff2
endif

```

Chiamante:

```

var: n1, n2, diff : integer
var: diff_elem_successione: procedure
begin
read n1, n2
diff_elem_successione(in: n1, n2; out: diff)
print "differenza: diff_elem_successione(n1, n2, diff) "
end

```


Siano dati due array 2D, `ipod_negozio`, di dimensioni $K \times 2$, e `ipod_deposito`, di dimensioni $Y \times 2$, entrambi di tipo intero. In questi array sono memorizzati, per ciascuna riga, nella colonna 1 il codice di un ipod, e nella colonna 2 quanti ipod di quel tipo sono presenti, rispettivamente, in negozio ed in deposito. Le righe di entrambi gli array sono ordinate per codice del ipod. Progettare in P-like un algoritmo, sotto forma di procedure (procedure `Quantita_ipod`), che fornisca come risultato un array 2D `Totale_ipod` che, sempre rispettando l'ordine secondo il codice-ipod, nella colonna 2 indichi quanti ipod, per ciascun tipo, sono disponibili in totale (tra negozio e deposito).

```

procedure Quantita_ipod (in:neg,dep,k,y;out:ipod)
var: i, j, h, k, y, s: integer
var: neg[k,2], dep[y,2], ipod[h,2]: array of integer
var: flag: logical
Begin
h:=k+y
i:=1 j:=1 s:=1 flag:=FALSE
while(s<=h AND flag=FALSE) do
    while(i<=k AND j<=y) do
        if(neg(i,1) > dep(j,1)) then
            ipod(s,1):=dep(j,1)
            ipod(s,2):=dep(j,2)
            j:=j+1
            s:=s+1
        else
            if(neg(i,1) < dep(j,1))
                ipod(s,1):=neg(i,1)
                ipod(s,2):=neg(i,2)
                i:=i+1
                s:=s+1
            else
                ipod(s,1):=neg(i,1)
                ipod(s,2):=neg(i,2)+dep(j,2)
                i:=i+1
                j:=j+1
                s:=s+1
            endif
        endif
    endwhile
    if(i>k AND j>y)
        flag:=TRUE
    else
        if(i>k)
            while(j<=y)
                ipod(s,1):=dep(j,1)
                ipod(s,2):=dep(j,2)
                j:=j+1
                s:=s+1
            endwhile
        else
            if(j>y)
                while(i<=k)
                    ipod(s,1):=neg(i,1)
                    ipod(s,2):=neg(i,2)
                    i:=i+1
                    s:=s+1
                endwhile
            endif
        endif
    endif
endwhile
end

```

Progettare un algoritmo in P-like, sottoforma di procedure (procedure el_successione) che, dati 2 numeri interi positivi $N1 < N2$, con $N1 \geq 0$, ed un intero $K > 0$, restituisca in output un array N_EL che contiene i primi K elementi della successione compresi tra $N1$ e $N2$ (cioè $N1 \leq \text{aqualcosa} \leq N2$). Progettare in p-like anche un possibile chiamante per questa function che, comunque, visualizzi correttamente, visualizzi correttamente l'array N_EL

```
procedure el_successione (in:n1,n2,k;out:ar)
var: a1, a2, a3, anew: integer
var: i, dim, n1, n2, k: integer
var: ar[dim]: array of integer
Begin
a1:=0 a2:=1 a3:=2 anew:=a3+2*a2+a1
i:=1
dim:=0
while(anew<=n2 AND i<=k)
    if(anew>=n1 AND anew<=n2)
        ar(i):=anew
        i:=i+1
        dim:=dim+1
        a1:=a2 a2:=a3 a3:=anew anew:=a3+2*a2+a1
    else
        a1:=a2
        a2:=a3
        a3:=anew
        anew:=a3+2*a2+a1
    endif
endwhile
end
```

```
chiamante:
var : n1, n2 ,k: integer
var n_el[k]: array of integer
var : i : integer
var: el_susccessione: procedure
begin
read n1, n2, k
el_successione(in : n1,n2,k out : n_el)
for i:=1 to k do
    print n_el[i]
endfor
end
```

Sia dato un array 1D A di tipo alfanumerico di dimensione N ed un array 1D B di tipo intero di dimensione N. L'array A contiene M sequenze di caratteri uguali di differente lunghezza. Progettare un algoritmo in p like sottoforma di procedure (procedure lunghezza_sequenze) che restituisca in output una variabile OK di tipo logico che vale TRUE se i primi M sequenze di caratteri uguali in A e gli altri (N-M) elementi di B sono 0, FALSE altrimenti.

```

procedure lunghezza_sequenze(a, b, n, m)
var: i, j, n, m, cont: integer
var: a[n]: array of character
var: b[n]: array of integer
var: seq, flag: logical
Begin
seq:=FALSE
flag:=TRUE
i:=1 j:=1
while(j<=n AND flag=TRUE)
    while(i<=m AND flag=TRUE)
        if(b(i) = 1)
            if(a(j) != a(j+1))
                j:=j+1
                flag:=TRUE
                j:=j+1
            else
                flag:= false
            endif
        else
            if(b(i) > 1)
                cont:=1
                while(a(j) = a(j+1) AND cont<b(i))
                    cont:=cont+1
                    j:=j+1
                endwhile
                if(cont = b(i))
                    flag:=TRUE
                    j:= j+1
                    i:= i+1
                else
                    flag:=FALSE
                endif
            endif
        endif
    endwhile
endif
endwhile
if(flag=TRUE)
    while(i<=n AND b(i) = 0)
        i:=i+1
    endwhile
    if(i>n)
        seq:= true
    else
        seq:= false
    endif
endif end

```

Sia dato un array 2D A di dimensione NxN di tipo intero, contenente solo numeri positivi; progettare in p like una procedura sottoforma di function di tipo logico (function dispari_in_riga(A,N)) che restituisce TRUE se in ciascuna riga di A c'è almeno un elemento dispari, FALSE altrimenti (supporre di avere a disposizione una logical function is_odd(num), CHE NON SI DEVE PROGETTARE, che restituisce TRUE se num è dispari, FALSE altrimenti). Progettare in p like anche un possibile chiamante per la function dispari_in_riga

```
function dispari_in_riga (a,n): logical
var: i, j, n, cont: integer
var: a[n,n]: array of integer
var: is_odd: logical function
begin
dispari:=TRUE
i:=1
while(i<=n AND dispari=TRUE)
    cont:=0
    for j:=1 to n do
        if(is_odd(a(i,j))
            cont:=cont+1
        endif
    endfor
    if(cont=0)
        dispari:=FALSE
    endif
    i:=i+1
endwhile
end
chiamante:
var : a[n,n] : array of integer
var : i, j, n : integer
var : dispari_in_riga(a,n): logical function
begin
read n
for i:=1 to n do
    for j:=1 to n do
        read a(i,j)
    endfor
endfor
dispari_in_riga(a,n)
if(dispari_in_riga(a,n))
    print "vero"
else
    print "falso"
endif
end
```

Siano dati un array 1D A di dimensione N, di tipo integer, e due variabili di tipo integer p1,p2. Progettare in pascal una procedura (procedure partizione_in_tre) che partizioni A in tre zone contigue: nella prima zona si trovano gli elementi minori o uguali a p1, nella seconda quelli maggiori di p1 e minori o uguali a p2 e nella terza quelli maggiori di p2.

```
procedure partizione_in_tre (a,n,p1,p2)
```

```
var: i,j,k,n,p1,p2,temp: integer
```

```
var: a[n]: array of integer
```

```
Begin
```

```
i:=1 j:=n
```

```
k:=1
```

```
while(i<=j)
```

```
    if(a(i) <= p1)
```

```
        i:=i+1
```

```
        k:=k+1
```

```
    else
```

```
        if(a(j) > p1)
```

```
            j:=j-1
```

```
        else
```

```
            temp:=a(i)
```

```
            a(i):=a(j)
```

```
            a(j):=temp
```

```
            i:=i+1
```

```
            j:=j+1
```

```
        endif
```

```
    endif
```

```
endwhile
```

```
i:=k j:=n
```

```
while(i<=j)
```

```
    if(a(i) <= p2)
```

```
        i:=i+1
```

```
    else
```

```
        if(a(j) > p2)
```

```
            j:=j-1
```

```
        else
```

```
            temp:=a(i)
```

```
            a(i):=a(j)
```

```
            a(j):=temp
```

```
            i:=i+1
```

```
            j:=j-1
```

```
        endif
```

```
    endif
```

```
endwhile
```

```
end
```

Si consideri la successione, per $a > 2$. Progettare in p like un algoritmo sottoforma di function di tipo intero function `conta_elementi_successione_data(K)` che, dato un intero, $K > 1$, restituisca il numero degli elementi della successione $\leq K$, escludendo i multipli di 3 e di 7. Si supponga di avere a disposizione 2 functions di tipo logico, `is_multiple_of_3(num)` e `is_multiple_of_7(num)`, che restituiscono TRUE se num è multiplo di 3 o di 7, rispettivamente.

```
function conta_elementi_successione_data (k): integer
var: a1,a2,a3,an,k: integer
begin
a1:=0 a2:=1 a3:=2 an:=a1+2*a2+a3
conta_elementi_successione_data:=1
while(an<=k)
    if ( ! is_multiple_of_3 (an) AND ! is_multiple_of_7 (an))
        conta_elementi_successione_data:= conta_elementi_successione_data +1
        a1:=a2
        a2:=a3
        a3:=an an:=a1+2*a2+a3
    else
        a1:=a2
        a2:=a3
        a3:=an an:=a1+2*a2+a3
    endif
endwhile
```

Un array A (di tipo integer) si dice palindromo se letto da sinistra verso destra si ottiene la stessa sequenza di numeri che si otterrebbe leggendolo all'incontrario.
Progettare un algoritmo, sottoforma di function ricorsiva (logical function `is_array_palindromo`) per verificare se un array è un palindromo

```
function is_array_palindromo (a,n,i,j): logical
var: i, j, n: integer
var: a[n]: array of integer
Begin
if(i=n AND j=1)
    if(a(i)=a(j))
        is_array_palindromo:=TRUE
    else
        is_array_palindromo:=FALSE
    endif
endif
if(a(i)=a(j))
    is_array_palindromo:= is_array_palindromo (a,n,i+1,j-1)
else
    is_array_palindromo:=FALSE
endif
end
```

Data una matrice A di dimensione $N \times N$, di tipo intero, progettare un algoritmo sotto forma di function (logical function controllo_righe_colonne) che restituisca TRUE se la somma degli elementi di ogni riga è minore di quella della riga precedente e la somma degli elementi di ogni colonna è maggiore o uguale della somma della colonna precedente, FALSE altrimenti.

```
function controllo_righe_colonne (a,n): logical
var: i, j, k, h, n: integer
var: sommar, sommapre, sommac, sommasucc: integer
Begin
controllo:=TRUE
i:=n
while(i>=1 AND controllo_righe_colonne =TRUE)
    sommar:=0
    sommapre:=0
    for j:=1 to n
        sommar:=sommar+a(i,j)
    endfor
    for k:=1 to n
        sommapre:=sommapre+a(i-1,k)
    endfor
    if(sommar >= sommapre)
        controllo_righe_colonne:=FALSE
    endif
    i:=i-1
endwhile
j:=n
while(j>=1 AND controllo_righe_colonne =TRUE)
    sommac:=0
    sommasucc:=0
    for i:=1 to n
        sommac:=sommac+a(i,j)
    endfor
    for h:=1 to n
        sommasucc:=sommasucc+a(h,j-1)
    endfor
    if(sommac < sommasucc)
        controllo_righe_colonne:=FALSE
    endif
    j:=j-1
endwhile
end
```

Date due linked list head1 ed head2 con il campo info di tipo character si dice che head2 è una sottosequenza di head1 se i caratteri di head2 compaiono tutti all'interno di head1 nello stesso ordine eventualmente intervallati da altri caratteri. Progettare in P-like una function ricorsiva (logical function sottosequenza) che, date le due linked list head1 e head2, restituisca TRUE se head2 è sottosequenza di head1, FALSE altrimenti.

```
function sottosequenza (head1,head2): logical
var: head1, head2: list_pointer
begin
if(head2=NULL)
    sottosequenza:=TRUE
else
    if(head1=NULL AND head2!=NULL)
        sottosequenza:=FALSE
    endif
    if(head1!=NULL AND head2!=NULL)
        if(head2.info = head1.info)
            sottosequenza:= sottosequenza (head1.link,head2.link)
        else
            sottosequenza:= sottosequenza (head1.link,head2)
        endif
    endif
endif
end
```

Progettare in P-like un algoritmo ricorsivo sotto forma di function(function costrList_ric) di tipo puntatore a nodo_lista, che costruisca una linked list con i campi info di tipo intero, contenenti i numeri da 1 a N, in ordine crescente. Indicare anche un esempio di chiamata da un main con i valori dei parametri attuali. Stesso esercizio, versione iterativa.

```
function costrList_ric (head,i,n): list_pointer
var: i, n: integer
var: head, newnode: list_pointer
newnode:=NULL
Begin
if(i=n)
    head.info:=i
    head.link:=newnode
    costrList_ric:=head
endif
if(i<n)
    head.info:=i
    head.link:=newnode
    costr:=costr(head.link,i+1,n)
endif end
```

```
iterativo:
list_pointer function costr(head,n)
var: i,n: integer
var: head,newnode: list_pointer
i:=1
while(i<=n)
    newnode:=NULL
    head.info:=i
    head.link:=newnode
    head:=head.link
    i:=i+1
endwhile end
```


Utilizzando solo le seguenti procedure che implementano particolari operazioni sull'ADT linked list, sviluppare un algoritmo in P-like, sotto forma di procedure(procedure NoZeri) che, data una linked list head1, con il campo info di tipo intero, restituisca in output: la linked list priva di zeri, il numero degli elementi cancellati. Function ennesimolista(head,n):integer, procedure eliminaennesimo(head,n)

```

procedure nozeri(head,n)
var: i, n, cont: integer
var: head, temp: list_pointer
Begin
i:=1
cont:=0
while(head!=NULL)
    if(ennesimolista(head,i) = 0)
        cont:=cont+1
        temp:=head
        eliminaennesimo(head,i)
        head:=temp.link
        i:=i+1
    else
        head:=head.link
    endif
endwhile

```

Si consideri la successione: $a_1=0$ $a_2=1$ $a_3=2$, $a_1+2*a_2+a_3$ per $n>3$. Progettare un algoritmo in P-like, sotto forma di procedure (procedure somma_elementi_uguale_numero(in:k,NUMERO,out:somma,valore,posizione)) che controlli se, sommando a mano a mano gli elementi di posto pari della successione(fino ad un massimo di k) si trovi che tale somma sia uguale a NUMERO (si supponga $NUMERO>2$), restituendo in tale caso attraverso la variabile logica somma, il valore TRUE, attraverso la variabile valore, NUMERO, e attraverso la variabile posizione la posizione che l'ultimo elemento sommato occupa nella successione; altrimenti restituisca FALSE, ed indichi inoltre, in valore e posizione, valore e posizione dell'ultimo elemento sommato che più fa avvicinare la somma a NUMERO.

```

procedure somma_elementi_uguale_numero (in:k,numero;out:somma,valore,pos)
var: a1, a2, a3, anew: integer
var: pos, k, numero, valore: integer
var: somma: logical
begin
a1:=0 a2:=1 a3:=2
anew:=a1+2*a2+a3
pos:=4
somma:=FALSE
while(anew<numero AND pos<=k)
    a1:=a2 a2:=a3 a3:=anew anew:=a1+2*a2+a3
    pos:=pos+2
endwhile
if(anew = numero)
    somma:=TRUE
    valore:=numero
else
    somma:=FALSE
    valore:=anew
endif

```

- 1) Sia dato un array 2D di tipo integer, di dimensioni $M \times N$, progettare in p like un algoritmo sottoforma di function di tipo logical (function `duerig_duecol`) che restituisca TRUE se vi sono due righe e due colonne consecutive di elementi uguali tra loro, FALSE altrimenti. Progetta in p like anche un esempio di algoritmo che utilizza la function.

```
logical function consecutive(a,m,n)
var: i,j,m,n: integer
var: a[m,n]:array of integer
var: flag: logical
begin
consecutive:=FALSE
i:=1
while(i<=m-1 AND consecutive=FALSE)
    j:=1
    flag:=FALSE
    while(j<=n AND flag=FALSE)
        if(a(i,j) = a(i+1,j))
            if(j=n)
                consecutive:=TRUE
            endif
            j:=j+1
        else
            flag:=TRUE
        endif
    endwhile
    i:=i+1
endwhile
j:=1 if(consecutive=TRUE)
flag:=FALSE consecutive:=FALSE
while(j<=n-1 AND consecutive=FALSE)
    i:=1
    flag:=FALSE
    while(i<=m AND flag=FALSE)
```

```

        if(a(i,j) = a(i,j+1)
            if(i=m)
                consecutive:=TRUE
            endif
        i:=i+1      else
            flag:=TRUE
        endif
    endwhile
j:=j+1
endwhile
endif

```

seguito dell esercizio :

chiamante:

begin

var: i,j,m,n: integer var: a[m,n]: array of

integer var: consecutive(a,m,n): logical

function read m,n

for i:=1 to m

for j:=1 to n

read a(i,j)

endfor

endfor

consecutive(a,m,n) print

consecutive

end

- 2) Sia dato un array 2D A di dimensioni MxN di tipo integer. Progettare un algoritmo in P-like sottoforma di procedure (procedure elementi_non_di_bordo) che restituisca in output un array 2D COORDINATE in cui per ogni riga ci siano le coordinate (i,j) degli elementi non di bordo dell'array A che hanno come elementi adiacenti (nord, est, sud, ovest) elementi di valore minore.

```
procedure bordo(a,m,n,coordinate)
var: i,j,k,m,n: integer
var: a[m,n], coordinate[k,2]: array of integer
var: nord,sud,est,ovest: integer k:=1 for i:=2
to m-1
    for j:=2 to n-1
        nord:=a(i-1,j)
        sud:=a(i+1,j)      est:=a(i,j+1)
        ovest:=a(i,j-1)
        if(a(i,j)>nord AND a(i,j)>sud AND a(i,j)>est AND a(i,j)>ovest)
            coordinate(k,1):=i
            coordinate(k,2):=j
            k:=k+1
        endif
    endfor
endfor
```

- 3) Sia dato un array 2D A di dimensioni MxN, con M pari, A[i,j] rappresenta il risultato dello studente j ottenuto alla prova i. L'array 1D P (di dimensione M) contiene per ciascuna prova, il punteggio minimo per il superamento della prova stessa (P(i) è il punteggio minimo per superare la prova i). Progettare in p like, un algoritmo che restituisca sottoforma di procedure l'array RIS(di dimensione ovviamente N) che fornisca il punteggio totale ottenuto nelle prove superate se lo studente ha superato almeno metà della prove, altrimenti restituisca 0.

```
procedure risultato(in:a,p,m,n;out:ris)
var: i,j,m,n: integer var: a[m,n], p[m],
ris[n]: array of integer var: tot,sup:
integer for j:=1 to n
    tot:=0
sup:=0
```

```

        for i:=1 to m
if(a(i,j) >= p(i))
                tot:=tot+a(i,j)
                sup:=sup+1
        endif
    endfor
    if(sup >= m/2)
        ris(j):=tot
    else
        ris(j):=0
    endif
endfor

```

- 4) Sia dato un array A 2D di tipo character di dimensioni NxM. Progettare in P-like un algoritmo sottoforma di function (logical function carattere_in_riga (A, N, M) che restituisca in output TRUE se nessuno dei caratteri presenti in una riga è presente nella riga successiva, FALSE altrimenti.

```

logical function carattere(a,n,m)
var: i,j,k,n,m: integer var:
a[n,m]: array of character
carattere:=TRUE
i:=1 j:=1
while(i<=n-1 AND carattere=TRUE)
    while(j<=m AND carattere=TRUE)
        for k:=1 to m
            if(a(i,j) = a(i+1,k))
                carattere:=FALSE
            endif
        endfor
    endwhile
    j:=j+1
endwhile
i:=i+1
endwhile

```

- 5) Si consideri la successione $a_1=0, a_2=1, a_3=2, a_n=a_1+2*a_2+a_3$ con $n>3$. Progettare in p like, sottoforma di function (logical function controllo_numeri_successione (N1,N2,N3)) che, dati 3 numeri interi $N_1<N_2<N_3$, con $N_1>a_3$, restituisca TRUE se il numero di elementi della successione compresi tra N_1 e N_2 (cioè $N_1 \leq a_3 \leq N_2$) è uguale al numero di elementi maggiori di N_2 e minori o uguali di N_3 , FALSE altrimenti. Progettare anche un chiamante

```
logical function controllo(in:n1,n2,n3)
```

```
var: a1,a2,a3,anew: integer var:
```

```
cont,cont1,n1,n2,n3: integer
```

```
a1=0 a2=1 a3=2
```

```
anew=a1+2*a2+a3
```

```
cont:=0 cont1:=0 controllo:=FALSE
```

```
while(anew<=n2)      if(anew<=n2  
AND anew>=n1)
```

```
    cont=cont+1
```

```
    a1=a2
```

```
    a2=a3
```

```
    a3=anew
```

```
    anew=a1+2*a2+a3
```

```
else
```

```
    a1=a2
```

```
    a2=a3
```

```
    a3=anew
```

```
    anew=a1+2*a2+a3
```

```
endif
```

```
endwhile while(anew<=n3)
```

```
if(anew>n2 AND anew<=n3)
```

```
    cont1=cont1+1
```

```
    a1=a2
```

```
a2=a3    a3=anew
```

```
    anew=a1+2*a2+a3
```

```
endif
```

```
endwhile
```

```
if(cont=cont1)
```

```
    controllo:=TRUE
```

endif

Chiamante :

var : n1,n2,n3 : integer

var : controllo_numeri_succeSSIONE(...) : logical
function

begin

read n1,n2,n3

controllo_numeri_succeSSIONE(n1,n2,n3)

if(controllo_numeri_succeSSIONE(n1,n2,n3)) then

print "vero"

else

print "falso"

endif

end

6) Dato un array 2D A di tipo intero, di dimensione ALPHAxBETA, con ALPHA pari, BETA>5, progettare in P-like un algoritmo, sotto forma di function logica, (quattro_elementi_uguali_somma_altri (A, ALPHA, BETA)) che restituisca TRUE se la somma del primo, terzo, quarto, e penultimo elemento di tutte le righe pari è uguale alla somma dei restanti elementi della medesima riga, FALSE altrimenti.

logical function sommadiv(a,alpha,beta)

var: i,j,alpha,beta: integer

var: a[alpha,beta]: array of integer

var: somma4,sommarest: integer

Begin

sommadiv:=TRUE

i:=2

while(i<=alpha AND sommadiv=TRUE)

 somma4:=0

 sommarest:=0

 for j:=1 to beta

 sommarest:=sommarest+a(i,j)

```

        endfor

        somma4:=a(i,1)+a(i,3)+a(i,4)+a(i,beta-1)

        sommarest=sommarest-somma4

    if(somma4!=sommarest)

        sommadiv:=FALSE

    else

        i:=i+2

    endif

endfor

```

7) Sia dato un array 2D A di tipo integer, di dimensioni HxK. Progettare in P-like un algoritmo sotto forma di function di tipo logical (function sommaRIGA_uguale_elemento) che restituisca TRUE se la somma degli opposti degli elementi di una riga è uguale ad uno degli elementi della riga successiva, FALSE altrimenti.

```

logical function sommariga(a,h,k)

var: i,j,s,h,k: integer

var: a[h,k]: array of integer

var: somma: integer

var: flag: logical

begin

    sommariga:=TRUE

    i:=1

    while(i<=h-1 AND sommariga=TRUE)

        somma:=0

        for j:=1 to k

            somma:=somma+(-a(i,j))

        endfor

        flag:=FALSE

        s:=1

        while(s<=k AND flag=FALSE)

            if(somma = a(i+1,s))

                flag:=TRUE

            endif

            s:=s+1

        endfor

    endwhile

endfunction

```



```

        endwhile
        if(flag=FALSE)
            sommariga:=FALSE
        endif    i:=i+1
    endwhile

```

8) Data una lista $head=L1 \rightarrow L2 \rightarrow L3 \rightarrow \dots \rightarrow L_n \rightarrow NULL$, il suo prefisso i -esimo consiste della sotto lista $head(i)=L1 \rightarrow L2 \rightarrow L3 \rightarrow \dots \rightarrow L_n$ (?) con $i \leq n$. Il prefisso i -esimo di $head$, vale a dire, è costituito dai suoi primi i elementi. Il peso di una lista $head$ (con il campo `info` di tipo `integer`) è la somma dei valori dei campi `info`. Data una lista $head$ con il campo `info` di tipo `integer` ed un intero `PESO`, supponendo che i valori dei campi `info` siano tutti positivi, si scriva una function ricorsiva di tipo `logical` (function `esiste_prefisso`) per verificare se esiste un prefisso di $head$ di peso `PESO`. Progettare in `p like` anche un algoritmo che richiami la function `logical function esiste(head,peso,somma)`

```

    var: peso,somma: integer
    var: head: list_pointer
    Begin
        if(head=NULL)
            esiste:=FALSE
        endif if(head!=NULL)
            somma:=somma+head.info
        if(somma<peso)
            esiste:=esiste(head.link,peso,somma)
        else
            if(somma>peso)
                esiste:=FALSE
            else
                if(somma==peso)
                    esiste:=TRUE
                endif
            endif
        endif
    endif

```

Chiamante :

var : head: list_pointer

var : peso,somma,ele,n,i : integer

var : esiste_prefisso(...) : logical

function

begin

head:=create_list(head)

read peso,n

somma:=0

for i:=1 to n

read ele

push(head,ele)

endfor

esiste_prefisso(head,peso,somma)

if(esiste_prefisso(head,peso,som
ma))

print "vero"

else

print "falso"

endif

end

9) Data una linked list head con il campo info di tipo integer, progettare un algoritmo in p like sottoforma di procedure (procedure list_sort) che, utilizzando il metodo exchange sort, e che restituisca la lista head ordinata in senso crescente secondo i valori del campo info. Ad esempio, se head= 3->5->6->4->2, l'output sarà head=2->3->4->5->6

procedure sort(head,n)

var: i,j,n: integer

var: head,curr,temp: list_pointer

sort:=FALSE curr:=head.link i:=1

while(i<=n AND sort=FALSE)

sort:=TRUE

for j:=1 to n-i do

if(head.info > curr.info)

```

                temp:=head.info
                head.info:=curr.info
            curr.info:=temp
            sort:=FALSE
        endif
        head:=head.link
        curr:=head.link
    endfor

    i:=i+1
endwhile

```

10) Data una linked list list, con il campo info di tipo integer, progettare in P like un algoritmo sottoforma di function ricorsiva, function elimina_da_lista(list,elem) che restituisca la lista privata degli elementi con i campi info uguali a elem

```

function elimina(list,ele)
var: ele: integer
var: list,temp: list_pointer
begin
    if(list = NULL)
        elimina:=list
    endif
    if(list!=NULL)
        if(list.info = ele)
            temp:=list
            dealloca list
            list:=temp.link
            elimina:=elimina(list,ele)
        else
            elimina:=elimina(list.link,ele)
        endif
    endif
endif

```

11) Dato un array A di tipo integer, di dimensione n, un sottoarray di A è formato di elementi contigui di A, ed il suo peso è la somma dei valori di tali elementi. Se A=(1,2,3,4,5,6,7,8,9), il sottoarray A(2...5)

è uguale a (2,3,4,5) e il suo peso è 14, mentre A[4...7] è uguale a (4,5,6,7) ed il suo peso è 22. Progettare in p like una function ricorsiva (logical function exist_sottoarray_didatopeso) che, dato un intero PESO ed un array A di dimensione N, di tipo integer, restituisca TRUE se esiste un sottoarray di A di peso PESO, FALSE altrimenti. Progettare anche una versione iterativa di tale funzione.

iterativo:

logical function esiste(a,n,peso)

var: i,j,n,ps,peso: integer

var: a[n]: array of integer

begin

esiste:=FALSE

i:=1

j:=i

while(i<=n AND esiste=FALSE)

ps:=0

while(ps < peso AND j<=n)

ps:=ps+a(j)

j:=j+1

endwhile

j:=1

if(ps=peso)

esiste:=TRUE

else if(ps > peso)

i:=i+1

j:=i

endif

endwhile

ricorsivo:

logical function esiste(a,i,j,n,peso,ps)

var: i,j,n,ps,peso: integer var: a[n]:

array of integer if(i=n AND ps!=peso)

esiste:=FALSE

```

else if(ps=peso)
    esiste:=TRUE
endif if(i<n)
    if(ps < peso AND j<n)
        ps:=ps+a(j)
        esiste:=esiste(a,i,j+1,n,peso,ps)
    else if(ps > peso)
        esiste:=esiste(a,i+1,i+1,n,peso,0)
    endif
endif

```

12) Data una linked list head con il campo info di tipo intero progettare un algoritmo ricorsivo in p like che elimini dalla lista tutti gli elementi che nel campo info contengono il valore dato, ELEM. Si organizzino in 2 procedure, entrambe ricorsive, una che elimina ELEM dalla testa del_testa(head,ELEM), una che elimina ELEM dal mezzo della lista del_mezzo(head,ELEM), ed una terza procedura, delete_elem (head, ELEM) che utilizza le altre due.

```

procedure testa(head,elem)
var: elem: integer var: head,temp:
list_pointer if(head!=NULL AND
head.info=elem)
    temp:=head
    dealloca head
head:=temp.link
testa:=testa(head,elem)
endif end

```

```

procedure mezzo(head,elem)
var: elem: integer var:
head,temp: list_pointer
if(head!=NULL)
    if(head.info=elem)
        temp:=head
        dealloca head
        head:=temp.link
    endif
endif

```

```

        mezzo:=mezzo(head,elem)

    else

        head:=head.link
        mezzo:=mezzo(head,elem)
    endif
endif
end

```

procedure delete(head,elem)

var: elem: integer var:

head: list_pointer

testa(head,elem)

mezzo(head.link,elem) end

13) Sia A una matrice di dimensione $n \times n$, contenente numeri interi. Sviluppare in P-like un algoritmo ricorsivo, sottoforma di function di tipo logical (function diagonale (A,riga)), con A array 2D di tipo integer, di dimensioni $n \times n$, implementazione della matrice A, che restituisca TRUE se tutti gli elementi della diagonale principale sono nulli, FALSE altrimenti. La chiamata main è del tipo:
matrice_diagonale:= diagonale (A,n)

logical function diag(a,riga)

var: riga: integer

var: a[n,n]: array of integer

Begin

if(a(1,1) = 0)

diag:=true

else

diag:=false

endif if(a(riga,riga)

= 0)

diag(a,riga-1)

else

diag:=false

endif

Chiamante :

var : a[n,n] : array of integer

var : i,j,n: integer

var : diagonale(...): logical function

begin

read n

for i:=1 to n do

 for j:=1 to n do

 read a(i,j)

 endfor

endfor

diagonale(a,n)

if(diagonale(a,n))

 print "vero"

else

 print "falso"

endif

end

14) Siano dati 2 array 1D di tipo character, A di dimensione N,B di dimensione M, con $N < M$. Progettare in p like un algoritmo, sottoforma di function (function `occorrenze_A_in_B`) che restituisca in output il numero di occorrenze di A in B. Ad esempio se $A=aa$ e $B=ccdaaaabaa$ allora la function restituisce 4. Progettare anche in p like un algoritmo che usi la function

integer function `occorrenze(a,b,n,m)`

var: i,j,k,n,m: integer var:

a[n],b[m]: array of character i:=1

j:=1 k:=1

while(k<=m)

 if(a(i) = b(j))

 if(i=n)

 occorrenze:=occorrenze+1

```

                i:=1
            k:=k+1
        j:=k
            endif
        i:=i+1        j:=j+1
    else
        i:=1
        k:=k+1
        j:=k
            endif
    endwhile
End

```

Chiamante :

var : a[n] , b[m] : array of
character

var : n,m,i,j :integer

var occorrenze_in_a_in_b(...):
integer function

begin

read n,m

for i:=1 to n do

 read a(i)

endfor

for j:=1 to m do

 read b(j)

endfor

occorrenze_in_a_in_b(a,b,n,m)

print "occorrenze:

occorrenze_in_a_in_b(a,b,n,m)"

end

- 15) Una rotazione di vettore x è il vettore che si ottiene spostando ogni componente del vettore un certo numero di posti verso sinistra, assumendo che il vettore sia circolare. Ad esempio, ruotando di 3 posti il vettore $x=(0;1;2;3;4;5;6;7;8;9)$ si ottiene i , vettore $xrot=(3;4;5;6;7;8;9;0;1;2)$. Progettare in P like un algoritmo sottoforma di procedure (procedure rotazione_array) che, dati due array X e Y , di dimensione n , restituisce in output una variabile rotazione di tipo character che vale 's' se uno è la rotazione dell'altro, 'n' altrimenti.

```
procedure rotazione(x,y,n)
var: i,j,k,n: integer
var: x[n],y[n]: array of integer
var: flag,rot: logical
begin
rot:=FALSE
i:=1 j:=1 k:=1 while(j<=n
AND rot=FALSE)
if(x(i) != y(j))
k:=k+1
i:=k
else if(x(i) = y(j))
i:=i+1
j:=j+1
if(i>n AND k>1)
i:=1
flag:=TRUE
while(i<=k AND j<=n AND flag=TRUE)
if(x(i) = y(j))
flag:=TRUE
else
flag:=FALSE
endif
i:=i+1
j:=j+1
endwhile
```

```

        if(flag=TRUE)
            rot:=TRUE
        else
            j=n+1
            rot:=FALSE
        endif
    endif
endif
endwhile

```

16) Progettare un algoritmo in P-like, sottoforma di function ricorsiva (function campoinfo_uguale) per verificare se i valori dei campi info, di tipo intero, degli elementi di una data linked list head, contengono valori uguali; la function restituisce TRUE se ciò è verificato, FALSE altrimenti.

```

logical function uguale(head)
var: head,curr: list_pointer
curr=head.link if(head=NULL)

    uguale:=TRUE
endif if(head!=NULL)
if(head.info = curr.info)
    uguale:=uguale(head.link)
else
    uguale:=FALSE endif
endif

```

17) Dati 2 stack head1 e head2 “ordinati”, progettare in P like un algoritmo per la costituzione di uno stack head3, anch’esso “ordinato”, merge tra head1 e head2. Utilizzare la procedure pop (head,elem) e push (head,elem).

```

procedure merge(head1,head2,head3)
var: head1,head2,head3: list_pointer
begin
    while(head1!=NULL AND head2!=NULL)
        if(head1.info <= head2.info)
            pop(head1,ele)
            push(head3,ele)
        endif
    endwhile

```

```

        else if(head1.info > head2.info)
            pop(head2,ele)
        push(head3,ele)      endif
    endwhile while(head1!=NULL AND
head2=NULL)
        pop(head1,ele)
    push(head3,ele)
    endwhile while(head2!=NULL AND
head1=NULL)
        pop(head2,ele)
    push(head3,ele)
endwhile

```

18) Sia head una linked-list i cui campi info possono essere 0 e 1 in modo che head rappresenti sia un numero binario. La lista head è implementata con puntatori doppi (ogni elemento punta al precedente e al successivo). Progettare in P like un algoritmo sottoforma di procedure, procedure incrementa (head, r) che incrementi di 1 il valore rappresentato da head. La chiamata del main è del tipo $r:=1 / r$ è il riporto /u, incrementa (head, r).

```

procedure incrementa(head,r)
var: head,newnode: list_pointer
var: r: integer
Begin
while(head!=NULL)
if(head.next!=NULL)
if(head.info=1 AND r=1)
            head.info:=0
            if(head.next.info=0)
                head.next.info:=1
                r:=0
            else
                head.next.info:=0
                r:=1
            endif
        endif
    endif
endwhile

```

```

        else if(head.info=0 AND r=1)
            head.info:=1
            r:=0
        else if(head.info=1 AND r=0)
            head.info:=1
            r:=0
        endif
    else
        if(head.info=1 AND r=1)
            head.info:=0
        newnode.info:=1
        newnode.prev:=head
        newnode.next:=NULL

        head:=newnode
        r:=0
        else if(head.info=0 AND r=1)
            head.info:=1
            r:=0
        else if(head.info=1 AND r=0)
            head.info:=1
            r:=0
        endif
    endif
    head:=head.next
endwhile

```

19) Progettare in p like un algoritmo sottoforma di function (logical function precedente_maggiore_sommasuccessivi) che data una linked list head (con campo info di tipo integer), restituisca TRUE se ciascun valore dei campi info è maggiore della somma di tutti i valori dei campi info degli elementi successivi, FALSE altrimenti. Utilizzare una function ricorsiva (function somma_valori_campinfosuccessivi), pure da progettare, in p like, che, data una linked list, restituisca la somma dei valori dei campi info (di tipo integer) dei suoi elementi. Progettare anche in p like, un algoritmo che richiami la function precedente_maggiore_sommasuccessivi

```

logical function somma(head)
var: head: list_pointer
begin
somma:=TRUE while(head!=NULL AND
somma=TRUE) if(head.info <=
sommasucc(head.link))
            somma:=FALSE
endif
            head:=head.link
endwhile end

```

```

function sommasucc(head)
var: somma: integer var:
head: list_pointer if(head =
NULL)
            sommasucc:=somma
endif if(head!=
NULL)
            somma:=somma+head.info
sommasucc:=sommasucc(head.link)
endif
end

```

Chiamante :

```

var : head : list_pointer
var : n,ele,i : integer
var:
precedente_maggiore_sommasuc
cessivi(...): logical function
begin
read n
head:=create_list(head)

```

```

for i:=1 to n do
    read ele
    push(head,ele)
endfor

precedente_maggiore_sommasuccessivi(head)

if(precedente_maggiore_sommasuccessivi(head))
    print "vero"
else
    print "falso"
endif
end

```

20) Siano date due linked list, head1 e head2, ordinate secondo il campo info di tipo integer, progettare in p like una procedura ricorsiva (procedure merge_ordered_list) che restituisca una linked list head3, ordinata secondo il campo info, risultato del merge tra head1 e head2. Progettare in p like anche un algoritmo che richiami tale procedura.

```

procedure merge(head1,head2,head3)
var: head1,head2,head3,newnode: list_pointer
Begin
newnode:=NULL
if(head1!=NULLAND head2!=NULL)
if(head1.info <= head2.info)
    head3.info:= head1.info    head3.link:= newnode
merge(head1.link,head2,head3.link)
else if(head1.info > head2.info)
    head3.info:= head2.info    head3.link:= newnode
    merge(head1,head2.link,head3.link)
endif
endif

```

```

endif if(head1=NULL AND
head2!=NULL)
head3.info:= head2.info    head3.link:=
newnode
merge(head1,head2.link,head3.link)
else if(head1!=NULL AND head2=NULL)
head3.info:= head1.info    head3.link:=
newnode
merge(head1.link,head2,head3.link)
endif
chiamante:
begin var: head1,head2,head3:
list_pointer var: ele1,ele2,n1,n2,i:
integer head1:=list_create(head1)
head2:=list_create(head2)
head3:=list_create(head3) read
n1,n2
for i:=1 to n1
    read ele1
    push(head1,ele1)
endfor
for
i:=1 to n2
    read ele2
    push(head2,ele2)
endfor merge(in:head1,head2;in/out:head3)
end
Chiamante :
var : head1,head2,head3 :
list_pointer
var : i,j : integer
var : n1,n2,ele1,ele2 : integer

```

```

begin
read n1,n2
head1:=create_list(head1)
head2:=create_list(head2)
head3:=create_list(head3)
for i:=1 to n1 do
    read ele1
    push(head1,ele1)
endfor
for j:=1 to n2 do
    read ele2
    push2(head2,ele2)
endfor
merge( in : head1,head2, in/out :
head3)
end

```

21) Date 3 linked list list1,list2,list3, progettare in p like un algoritmo sottoforma di function ricorsiva(logical function somma_lista(head1,head2,head3) che restituisca TRUE se head3 è la somma di head1 e head2, elemento per elemento. FALSE altrimenti. Si supponga di avere il numero di elementi di head3 al massimo numero di elementi tra head1 e head2.

```

logical function somma(head1,head2,head3)
var: head1,head2,head3: list_pointer
Begin
if(head1!=NULL AND head2!=NULL AND head3!=NULL)
if(head3.info = head1.info+head2.info)
somma:=somma(head1.link,head2.link,head3.link)
else
somma:=FALSE      endif
endif
if(head1=NULL AND head2!=NULL AND head3!=NULL)

```



```

        if(head3.info = head2.info)
somma:=somma(head1,head2.link,head3.link)
        else
somma:=FALSE      endif
else if(head1!=NULL AND head2=NULL AND head3!=NULL)
        if(head3.info = head1.info)
                somma:=somma(head1.link,head2,head3.link)
        else
somma:=FALSE      endif
endif
if(head1=NULL AND head2=NULL AND head3=NULL)
        somma:=TRUE
endif

```

22) Progettare un algoritmo in p like, sottoforma di function ricorsiva (function punt_di_nodo) che, data una linked list head con il campo info di tipo intero, ed un elemento x, di tipo intero, restituisca il puntatore dell'elemento della linked list il cui campo info è uguale a x, se esso è presente, NULL altrimenti.

```

function puntatore(head,x)
var: head: list_pointer
var: x: integer
begin
if(head = NULL)
        puntatore:=NULL
endif if(head !=
NULL)
        if(head.info = x)
                puntatore=head
        else
                puntatore(head.link,x)
        endif
endif

```

23) Un polinomio può essere rappresentato sottoforma di linked list, in cui ci sono tanti elementi quanti termini dei polinomi, ed ogni elemento contiene un campo coef ed un campo esp, il coefficiente e la potenza della variabile; per esempio il polinomio $P1(x)=7x^3-3x^2+4$ (rappresentazione). Assegnati due polinomi rappresentati da due liste P1 e P2, progettare un algoritmo in p like per determinare il polinomio P3(x) dato dalla somma di P1(x) e P2(x), che restituisca cioè una linked list P3 che rappresenti P3(x).

```
procedure polinomi(p1,p2,p3)
var:p1,p2,p3,newnode: list_pointer
begin
  if(p1!=NULL AND p2!=NULL)
    newnode:=NULL
    if(p1.esp > p2.esp)
      p3.esp:=p1.esp
    p3.coef:=p1.coef
    p3.link:=newnode
    polinomi(p1.link,p2,p3.link)
    else if(p1.esp < p2.esp)
      p3.esp:=p2.esp
    p3.coef:=p2.coef
    p3.link:=newnode
    polinomi(p1,p2.link,p3.link)
  else if(p1.esp = p2.esp)
    p3.esp:=p1.esp
    p3.coef:=p1.coef+p2.coef
    p3.link:=newnode
    polinomi(p1.link,p2.link,p3.link)
  endif
endif
if(p1!=NULL AND p2=NULL)
  newnode:=NULL
  p3.esp:=p1.esp
  p3.coef:=p1.coef
  p3.link:=newnode
```

```

        polinomi(p1.link,p2,p3.link)
endif if(p2!=NULL AND
p1=NULL)
        newnode:=NULL
p3.esp:=p2.esp
p3.coef:=p2.coef
p3.link:=newnode
        polinomi(p1,p2.link,p3.link)
endif

```

24) Sia data la successione: $a_1:=0$ $a_2:=1$ $a_3:=2$, $a_n:=a_1+2*a_2+a_3$ per $n>3$. Progettare in P-like un algoritmo sottoforma di procedure (procedure diff_elem_successione) che, dati 2 interi N_1 e N_2 , con $N_1<N_2$ e $N_1>a_3$, restituisca in output l'intero (N_1 o N_2) che ha una minore differenza rispetto ad un elemento della successione, ed il valore di tale differenza. Progettare in P-like anche un esempio di algoritmo che utilizza la procedura.

```

procedure diffe(in/out:n1,n2;out:diff)
var: a1,a2,a3,an: integer
var: n1,n2,diff,diff1,diff2: integer
begin
a1=0 a2=1 a3=2 an=a1+2*a2+a3
while(an<n1)
        a1:=a2
a2:=a3 a3:=an
        an:=a1+2*a2+a3
endwhile if(abs(n1-an) <
abs(n1-a3))
        diff1:=abs(n1-an)
else
        diff1:=abs(n1-a3)
endif while(an<n2)
        a1:=a2
a2:=a3
a3:=an

```

```

        an:=a1+2*a2+a3
    endwhile
    if(abs(n2-an) < abs(n2-a3))
        diff2:=abs(n2-an)
    else
        diff2:=abs(n2-a3)
    endif
    if(diff1 < diff2)
        diff:=diff1
    else
        diff:=diff2
    endif

```

Chiamante :

```

var : n1,n2,diff : integer

var: diff_elem_successione(...):
integer function

begin

read n1,n2

diff_elem_successione(in:n1,n2;ou
t:diff)

print "differenza:
diff_elem_successione(n1,n2,diff)
"

end

```

25) Siano dati due array 2D, `Ipod_negozio`, di dimensioni $K \times 2$, e `ipod_deposito`, di dimensioni $Y \times 2$, entrambi di tipo intero. In questi array sono memorizzati, per ciascuna riga, nella colonna 1 il codice di un ipod, e nella colonna 2 quanti ipod di quel tipo sono presenti, rispettivamente, in negozio ed in deposito. Le righe di entrambi gli array sono ordinate per codice del ipod. Progettare in P-like un algoritmo, sotto forma di procedure (procedure `Quantita_ipod`), che fornisca come risultato un array 2D `Totale_ipod` che, sempre rispettando l'ordine secondo il codice-ipod, nella colonna 2 indichi quanti ipod, per ciascun tipo, sono disponibili in totale (tra negozio e deposito).

```

procedure ipodtot(in:neg,dep,k,y;out:ipod)

var: i,j,h,k,y,s: integer

```

var: neg[k,2],dep[y,2],ipod[h,2]: array of integer

var: flag: logical

Begin

h:=k+y

i:=1 j:=1 s:=1 flag:=FALSE while(s<=h

AND flag=FALSE) while(i<=k

AND j<=y) if(neg(i,1) >

dep(j,1))

 ipod(s,1):=dep(j,1)

 ipod(s,2):=dep(j,2)

 j:=j+1

 s:=s+1

 else if(neg(i,1) < dep(j,1))

 ipod(s,1):=neg(i,1)

 ipod(s,2):=neg(i,2)

 i:=i+1

 s:=s+1

 else

 ipod(s,1):=neg(i,1)

 ipod(s,2):=neg(i,2)+dep(j,2)

 i:=i+1

 j:=j+1

 s:=s+1

 endif

 endwhile

 if(i>k AND j>y)

 flag:=TRUE

 else if(i>k)

 while(j<=y)

ipod(s,1):=dep(j,1)

ipod(s,2):=dep(j,2)

```

                j:=j+1
s:=s+1
        endwhile
    else if(j>y)
        while(i<=k)
ipod(s,1):=neg(i,1)
ipod(s,2):=neg(i,2)
                i:=i+1
s:=s+1
        endwhile
    endif endwhile

```

26) Si consideri la successione: $a_1=0$ $a_2=1$ $a_3=2$, $a_{n+3}=a_{n+2}+2*a_{n+1}+a_n$ per $n \geq 3$. Progettare un algoritmo in P-like, sottoforma di procedure (procedure `ele_successione`) che, dati 2 numeri interi positivi $N_1 < N_2$, con $N_1 \geq 0$, ed un intero $K > 0$, restituisca in output un array `N_EL` che contiene i primi K elementi della successione compresi tra N_1 e N_2 (cioè $N_1 \leq \text{aqualcosa} \leq N_2$). Progettare in p-like anche un possibile chiamante per questa function che, comunque, visualizzi correttamente, visualizzi correttamente l'array `N_EL`

```

procedure ele_succ(in:n1,n2,k;out:ar)
var: a1,a2,a3,anew: integer
var: i,dim,n1,n2,k: integer
var: ar[dim]: array of integer
Begin
a1=0 a2=1 a3=2
anew=a3+2*a2+a1
i:=1
dim:=0 while(anew<=n2
AND i<=k)
    if(anew>=n1 AND anew<=n2)
        ar(i):=anew
        i:=i+1
        dim:=dim+1
        a1:=a2
        a2:=a3
    endif
    anew=a3+2*a2+a1
    i:=i+1
endwhile

```

```

        a3:=anew
        anew:=a3+2*a2+a1

    else

        a1:=a2
        a2:=a3
        a3:=anew
        anew:=a3+2*a2+a1
    endif
endwhile

```

Chiamante :

```

var : n1,n2,k : integer
var n_el[k]: array of integer
var : i : integer
begin
read n1,n2,k
el_succezione(in : n1,n2,k out :
n_el)
for i:=1 to k do
    print n_el[i]
endfor
end

```

27) Sia dato un array 1D A di tipo alfanumerico di dimensione N ed un array 1D B di tipo intero di dimensione N. L'array A contiene M sequenze di caratteri uguali di differente lunghezza. Progettare un algoritmo in p like sottoforma di procedure (procedure lunghezza_sequenze) che restituisca in output una variabile OK di tipo logico che vale TRUE se i primi M elementi di B indicano la lunghezza di ciascuna delle M sequenze di caratteri uguali in A e gli altri (N-M) elementi di B sono 0, FALSE altrimenti.

```

procedure sequenze(a,b,n,m)
var: i,j,n,m,cont: integer

```

var: a[n]: array of character

var: b[n]: array of integer

var: seq,flag: logical

Begin

seq:=FALSE flag:=TRUE

i:=1 j:=1

while(j<=n AND flag=TRUE)

 while(i<=m AND flag=TRUE)

 if(b(i) = 1)

 if(a(j) != a(j+1))

 flag:=TRUE

 j:=j+1

 i:=i+1

 else

 flag:=FALSE

 endif

 else if(b(i) > 1)

 cont:=1

 while(a(j) = a(j+1) AND cont<b(i))

 cont:=cont+1

 j:=j+1

 endwhile

 if(cont = b(i))

 flag:=TRUE

 i:=i+1

 j:=j+1

 else

 flag:=FALSE

 endif

 endif

endwhile


```

endwhile if(flag=TRUE)
    while(i<=n AND b(i) = 0)
        i:=i+1
    endwhile
if(i>n)
    seq:=TRUE
else
    seq:=FALSE
endif
endif

```

28) Sia dato un array 2D A di dimensione NxN di tipo intero, contenente solo numeri positivi; progettare in p like una procedura sottoforma di function di tipo logico (function `dispari_in_riga(A,N)`) che restituisce TRUE se in ciascuna riga di A c'è almeno un elemento dispari, FALSE altrimenti (supporre di avere a disposizione una logical function `is_odd(num)`, CHE NON SI DEVE PROGETTARE, che restituisce TRUE se num è dispari, FALSE altrimenti). Progettare in p like anche un possibile chiamante per la function `dispari_in_riga`

```

logical function dispari(a,n)
var: i,j,n,cont: integer
var: a[n,n]: array of integer
begin
    dispari:=TRUE
    i:=1
    while(i<=n AND dispari=TRUE) cont:=0

        for j:=1 to n do
            if(odd(a(i,j)))
                cont:=cont+1
            endif
        endfor

        if(cont=0)
            dispari:=FALSE
        endif
        i:=i+1
    endwhile

```

Chiamante :

var : a[n,n] : array of integer

var : i,j,n : integer

var : dispari_in_riga(a,n): logical
function

begin

read n

for i:=1 to n do

 for j:=1 to n do

 read a(i,j)

 endfor

endfor

dispari_in_riga(a,n)

if(dispari_in_riga(a,n))

 print "vero"

else

 print "falso"

endif

end

29) Siano dati un array 1D A di dimensione N, di tipo integer, e due variabili di tipo integer p1,p2.

Progettare in plike una procedura (procedure partizione_in_tre) che partizioni A in tre zone contigue: nella prima zona si trovano gli elementi minori o uguali a p1, nella seconda quelli maggiori di p1 e minori o uguali a p2 e nella terza quelli maggiori di p2.

procedure partizione(a,n,p1,p2)

var: i,j,k,n,p1,p2,temp: integer

var: a[n]: array of integer

Begin

i:=1

j:=n

k:=1

while(i<=j)

```

        if(a(i) <= p1)
            i:=i+1
k:=k+1
        else if(a(j) > p1)
            j:=j-1
        else
temp:=a(i)
a(i):=a(j)
a(j):=temp
            i:=i+1
j:=j-1
        endif
    endwhile
i:=k j:=n
while(i<=j)
    if(a(i) <= p2)
        i:=i+1
    else if(a(j) > p2)
        j:=j-1
    else
        temp:=a(i)
        a(i):=a(j)
        a(j):=temp
        i:=i+1
        j:=j-1
    endif
endwhile

```

30) Si consideri la successione: $a_1:=0$ $a_2:=1$ $a_3:=2$, $a_{n+3}:=a_n+2*a_{n+1}+a_{n+2}$ per $n>3$. Progettare in p
like un algoritmo sottoforma di function di tipo intero function
conta_elementi_successione_data(K) che, dato un intero, $K>1$, restituisca il numero degli elementi
della successione $\leq K$, escludendo i multipli di 3 e di 7. Si supponga di avere a disposizione 2
functions di tipo logico, is_multiple_of_3(num) e is_multiple_of_7(num), che restituiscono TRUE se
num è multiplo di 3 o di 7, rispettivamente.

```

integer function conta(k)
var: a1,a2,a3,an,k: integer

begin

a1:=0 a2:=1 a3:=2

an:=a1+2*a2+a3 conta:=1

while(an<=k)

    if(!m3(an) AND !m7(an))

        conta:=conta+1

        a1:=a2

        a2:=a3

        a3:=an

        an:=a1+2*a2+a3

    else

        a1:=a2

        a2:=a3

        a3:=an

        an:=a1+2*a2+a3

    endif

endwhile

```

31) Un array A (di tipo integer) si dice palindromo se letto da sinistra verso destra si ottiene la stessa sequenza di numeri che si otterrebbe leggendolo all'incontrario. Ad esempio A={0,1,2,3,4,3,2,1,0} è palindromo. Progettare un algoritmo, sottoforma di function ricorsiva (logical function is_array_palindromo) per verificare se un array è un palindromo

```

logical function palindromo(a,n,i,j)
var: i,j,n: integer
var: a[n]: array of integer

Begin

if(i=n AND j=1)

    if(a(i)=a(j))

        palidromo:=TRUE

    else

        palindromo:=FALSE

    endif

endif

```

```

endif if(a(i)=a(j))
    palindromo:=palidromo(a,n,i+1,j-1)
else
    palindromo:=FALSE
endif

```

32) Data una matrice A di dimensione NxN, di tipo intero, progettare un algoritmo sotto forma di function (logical function controllo_righe_colonne) che restituisca TRUE se la somma degli elementi di ogni riga è minore di quella della riga precedente e la somma degli elementi di ogni colonna è maggiore o uguale della somma della colonna precedente, FALSE altrimenti.

```

logical function controllo(a,n)
var: i,j,k,h,n: integer
var: sommar,sommapre,sommac,sommasucc: integer
Begin
controllo:=TRUE
i:=n
while(i>=1 AND controllo=TRUE)
    sommar:=0
    sommapre:=0
    for j:=1 to n
        sommar:=sommar+a(i,j)    endfor
    for k:=1 to n
        sommapre:=sommapre:=a(i-1,k)
    endfor
    if(sommar >= sommapre)
        controllo:=FALSE
    endif
    i:=i-1
endwhile
j:=n
while(j>=1 AND controllo=TRUE)
    sommac:=0
    sommasucc:=0

```

```

        for i:=1 to n
            sommac:=sommac+a(i,j)
        endfor
        for h:=1 to n
            sommasucc:=sommasucc+a(h,j-1)      endfor
            if(sommac < sommasucc)
                controllo:=FALSE
            endif
            j:=j-1
        endwhile

```

33) Date due linked list head1 ed head2 con il campo info di tipo character si dice che head2 è una sottosequenza di head1 se i caratteri di head2 compaiono tutti all'interno di head1 nello stesso ordine eventualmente intervallati da altri caratteri. Progettare in P-like una function ricorsiva (logical function sottosequenza) che, date le due linked list head1 e head2, restituisca TRUE se head2 è sottosequenza di head1, FALSE altrimenti.

```

logical function sotto(head1,head2)
var: head1,head2: list_pointer
begin
    if(head2=NULL)
        sotto:=TRUE
    else if(head1=NULL AND head2!=NULL)
        sotto:=FALSE
    endif

    if(head1!=NULL AND head2!=NULL)
        if(head2.info = head1.info)
            sotto:=sotto(head1.link,head2.link)
        else
            sotto:=sotto(head1.link,head2)
        endif
    endif
endif

```

34) Progettare in P-like un algoritmo ricorsivo sotto forma di function(function costrList_ric) di tipo puntatore a nodo_lista, che costruisca una linked list con i campi info di tipo intero, contenenti i numeri da 1 a N, in ordine crescente. Indicare anche un esempio di chiamata da un main con i valori dei parametri attuali. Stesso esercizio, versione iterativa.

ricorsivo:

```
list_pointer function costr(head,i,n)
var: i,n: integer
var: head,newnode: list_pointer newnode:=NULL
Begin
if(i=n)
    head.info:=i
    head.link:=newnode
costr:=head
endif if(i<n)
    head.info:=i
    head.link:=newnode
    costr:=costr(head.link,i+1,n)
endif end
```

iterativo:

```
list_pointer function costr(head,n)
var: i,n: integer var:
head,newnode: list_pointer
i:=1 while(i<=n)
    newnode:=NULL
    head.info:=i
    head.link:=newnode
    head:=head.link    i:=i+1
endwhile end
```

35) Utilizzando solo le seguenti procedure che implementano particolari operazioni sull'ADT linked list, sviluppare un algoritmo in P-like, sotto forma di procedure(procedure NoZeri) che, data una linked list head1, con il campo info di tipo intero, restituisca in output: la linked list priva di zeri, il numero degli elementi cancellati. Function ennesimolista(head,n):integer, procedure eliminaennesimo(head,n) procedure nozeri(head,n)

```

var: i,n,cont: integer
var: head,temp:list_pointer

Begin

i:=1

cont:=0

while(head!=NULL)

if(ennesimolista(head,i) = 0)

        cont:=cont+1

temp:=head

        eliminaennesimo(head,i)

head:=temp.link

        i:=i+1

    else

head:=head.link        endif

endwhile

```

36) Si consideri la successione: $a_1=0$ $a_2=1$ $a_3=2$, $a_{n+2}=a_n+2*a_{n+1}+a_{n+2}$ per $n>3$. Progettare un algoritmo in P-like, sotto forma di procedure(proceduresomma_elementi_uguale_numero(in:k,NUMERO,out:somma,valore,posizione)) che controlli se, sommando a mano a mano gli elementi di posto pari della successione(fino ad un massimo di k) si trovi che tale somma sia uguale a NUMERO (si supponga NUMERO>2), restituendo in tale caso attraverso la variabile logica somma, il valore TRUE, attraverso la variabile valore, NUMERO, e attraverso la variabile posizione la posizione che l'ultimo elemento sommato occupa nella successione; altrimenti restituisca FALSE, ed indichi inoltre, in valore e posizione, valore e posizione dell'ultimo elemento sommato che più fa avvicinare la somma a NUMERO.

```

procedure sommauguale(in:k,numero;out:somma,valore,pos)

var: a1,a2,a3,anew: integer

var: pos,k,numero,valore: integer

var: somma: logical a1:=0 a2:=1 a3:=2

begin

anew:=a1+2*a2+a3 pos:=4

somma:=FALSE while(anew<numero

AND pos<=k)

        a1:=a2

a2:=a3 a3:=anew

```



```
    anew:=a1+2*a2+a3
```

```
    a1:=a2  a2:=a3
```

```
  a3:=anew
```

```
  anew:=a1+2*a2+a3
```

```
  pos:=pos+2
```

```
endwhile if(anew
```

```
= numero)
```

```
    somma:=TRUE
```

```
  valore:=numero
```

```
else
```

```
    somma:=FALSE
```

```
  valore:=anew
```

```
endif
```

