Università degli Studi di Cagliari
Corso di Laurea in Ingegneria Elettronica

ELEMENTI DI INFORMATICA

http://agilegroup.eu

A.A. 2015/2016

Docente: **Michele Marchesi**

ALGORITMI ELEMENTARI

Sommario

- **Sequenze di elaborazioni**
- **Elaborazioni condizionali**
- **Cicli**
- **Collezioni e vettori di dati**
- **Cicli annidati**

Sequenza di elaborazioni

- **Il "cuore" di ogni algoritmo è una sequenza composta da una o più elaborazioni**
- **Le elaborazioni sono effettuate tramite:**
  - **valutazioni di *espressioni***
  - **assegnazioni a *variabili* (simbolo: ':=' "*prende*")**
- **In *input* si hanno delle variabili date**
- **Si possono definire e usare *variabili locali* usate per memorizzare risultati intermedi**
- **In *output*, sono calcolati uno o più valori**
  - **restituiti direttamente**
  - **assegnati a variabili di output**

Calcolo delle radici di un'equazione di 2 grado (reali): $ax^2+bx+c=0$

Soluzione: $$x_{1,2}=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

1. **Input: variabili *a*, *b* e *c* (numeri reali)**
2. **Calcolo del discriminante: $D := b^2 - 4ac$ (≥ 0!)**
3. **Calcolo della radice di *D*: $r := \sqrt{D}$**
4. **I radice: $x_1 := (-b + r) / 2a$**
5. **II radice: $x_2 := (-b - r) / 2a$**
6. **Output: x_1 e x_2**

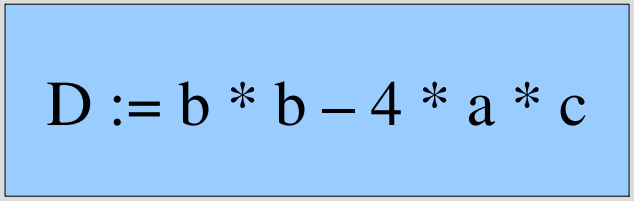
Variabili locali usate (reali): *D*, *r*, x_1 e x_2

Diagrammi di flusso (*flowchart*)

- **I diagrammi di flusso sono una notazione *grafica* per visualizzare algoritmi**
- **Aiutano a ragionare sull'algoritmo**
- **Sono di aiuto specialmente per imparare a scrivere algoritmi**
- **Oggigiorno, non sono più molto usati, ma esiste un diagramma di UML (Unified Modeling Language, lo standard per documentare sistemi software) molto simile: *il diagramma di attività* (Activity Diagram)**

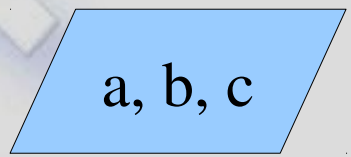
Diagrammi di flusso - simboli

- **Computazione o processo:**

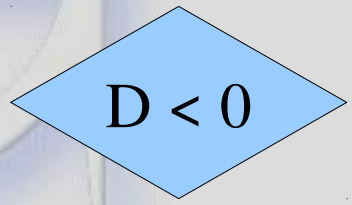


- **Flusso di operazioni:**

- **Input/output:**



- **Decisione:**

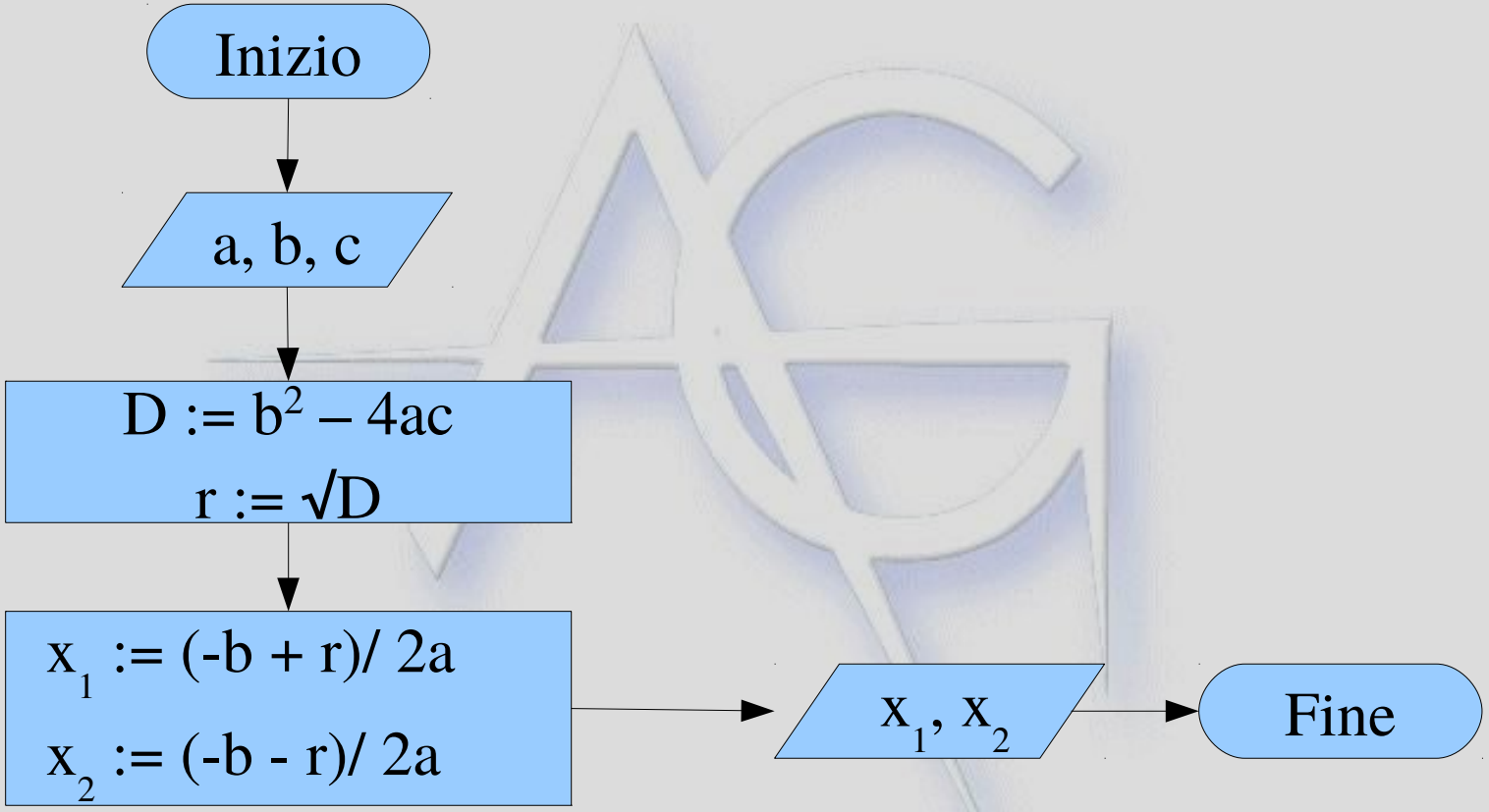


- **Inizio o fine**

Calcolo delle radici di un'equazione di 2 grado (reali):

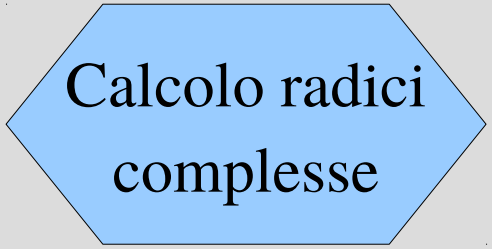
$$ax^2 + bx + c = 0$$

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$



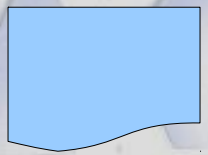
Diagrammi di flusso – altri simboli

- **Blocco (sottodiagramma):**

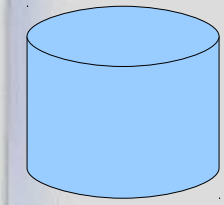


- **Display:**

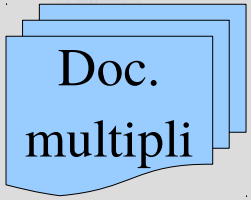
- **Documento:**

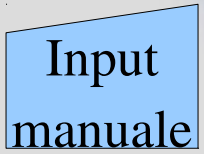


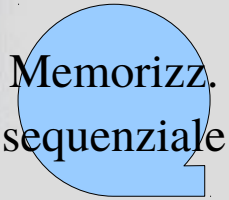
- **Memorizzazione permanente:**



- **Anche:**
- **e altri...**







Elaborazioni Condizionali

- **Con le sequenze di elaborazioni da sole non si va molto lontano...**
- **Molto spesso occorre *verificare delle condizioni* ed effettuare elaborazioni differenti, in conseguenza di tali condizioni**
- **A tal fine, si usano le elaborazioni condizionali:**
  - **se *condizione* allora: *esegui sequenza 1***
    altrimenti: *esegui sequenza 2*
- **Una condizione è un'espressione il cui valore è logico, e può essere *vero* o *falso***
- ***sequenza 1* e *2* possono anche essere vuote**

Elaborazioni Condizionali

- **Esempi di condizioni:**
  - $a \geq b$; *k* **è dispari ;** *anno* **è bisestile;**
- **I linguaggi di programmazione hanno espressioni il cui valore è logico:**
  - `a >= b;` `isOdd(k);`
- **Istruzioni condizionali in C e Smalltalk:**

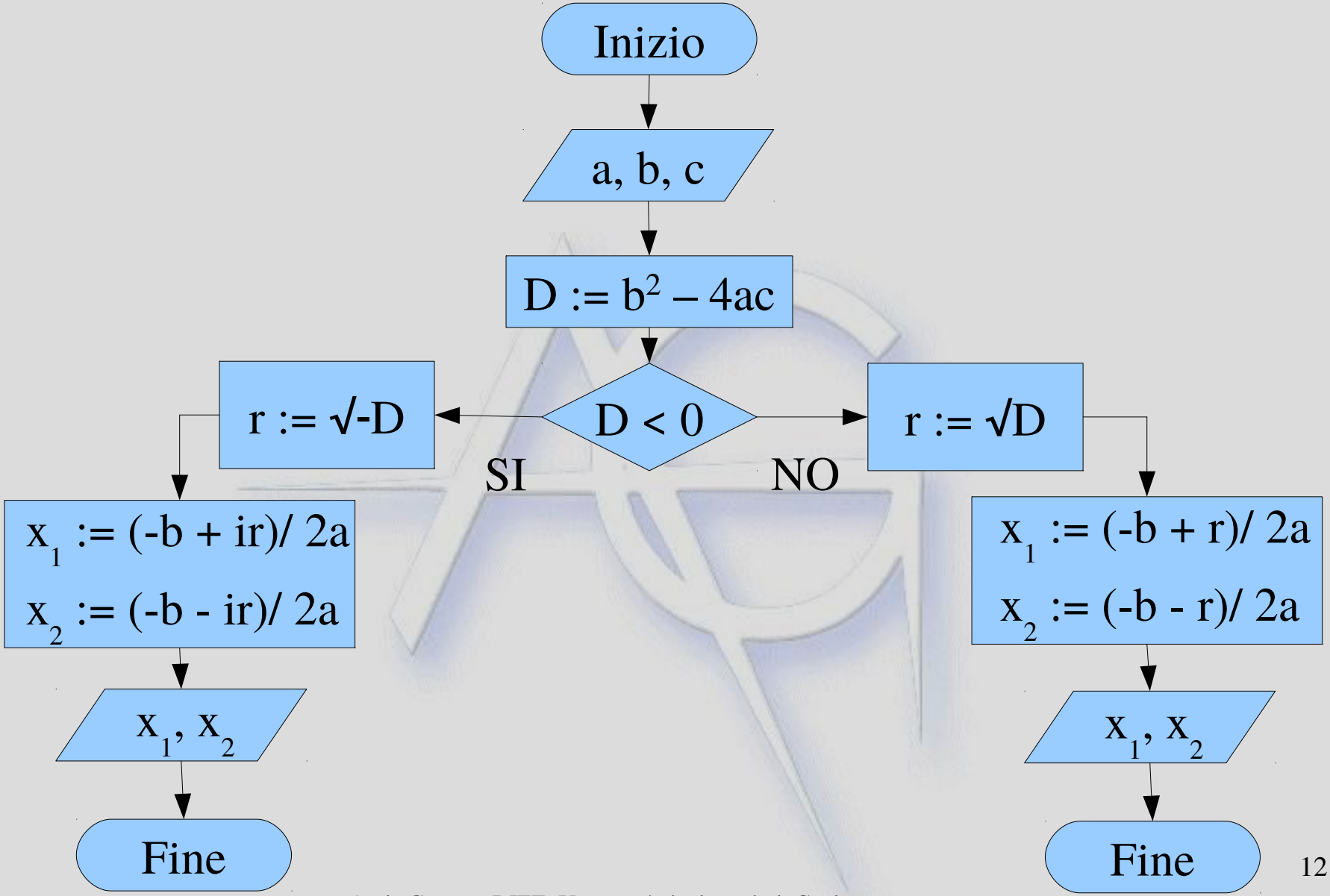
```
C:  if (condizione) {sequenza1};
    if (condizione) {sequenza1}
       else {sequenza2};
St:  condizione ifTrue: [sequenza1]
                ifFalse: [sequenza2].
```

Calcolo delle radici di un'equazione di 2 grado (tutte):

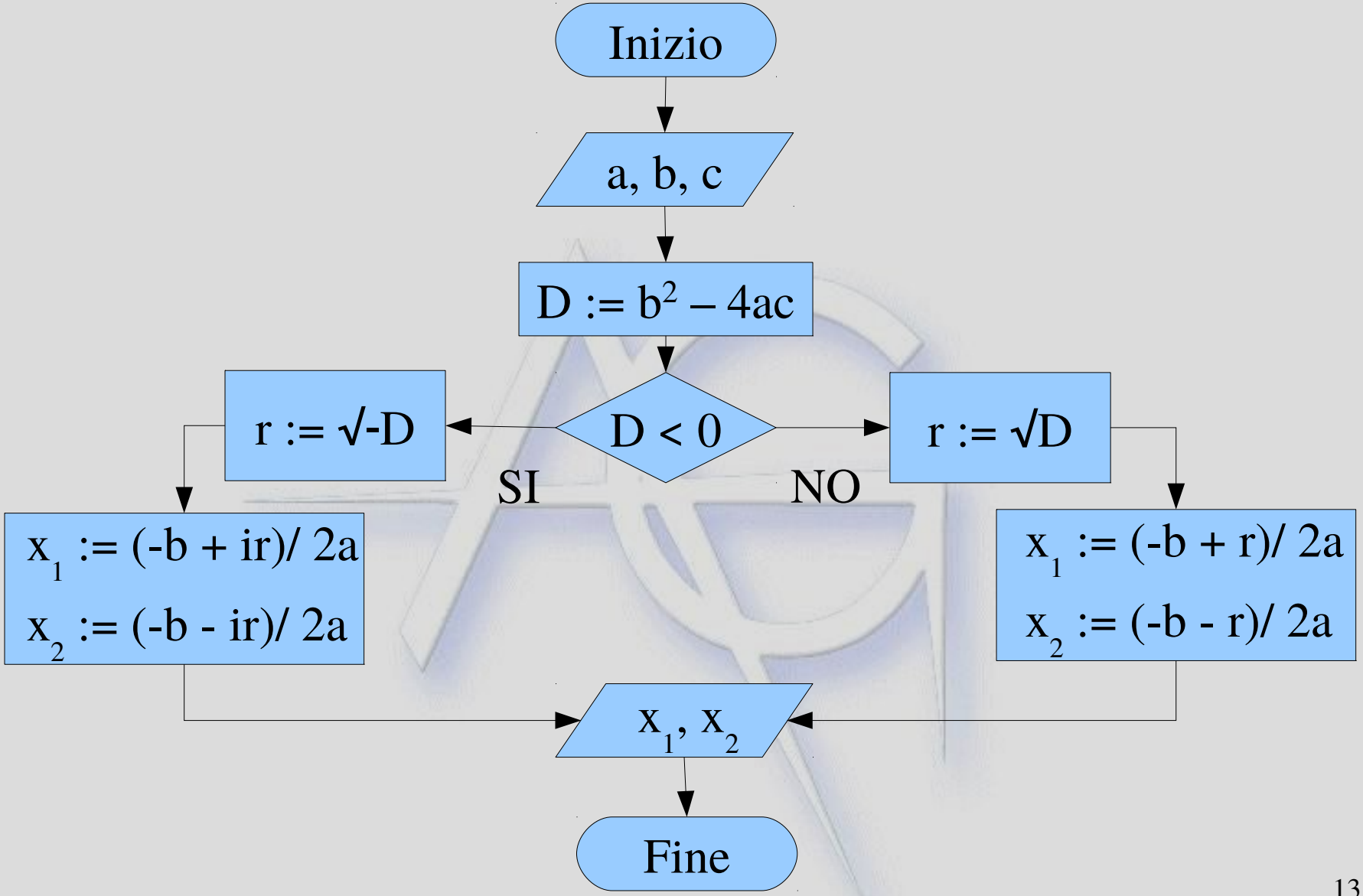
$$ax^2 + bx + c = 0$$

1. Input: variabili *a*, *b* e *c*
2. Calcolo del discriminante: $D := b^2 - 4ac$
3. Se $D < 0$ (radici complesse coniugate)
   - 3.1 $r := \sqrt{-D}$
   - 3.2 $x_1 := (-b + i\,r) / 2a,\ x_2 := (-b - i\,r) / 2a.$
4. Altrimenti (radici reali):
   - 4.1 $r := \sqrt{D}$
   - 4.2 $x_1 := (-b + r) / 2a,\ x_2 := (-b - r) / 2a.$
5. Output di x_1 e x_2

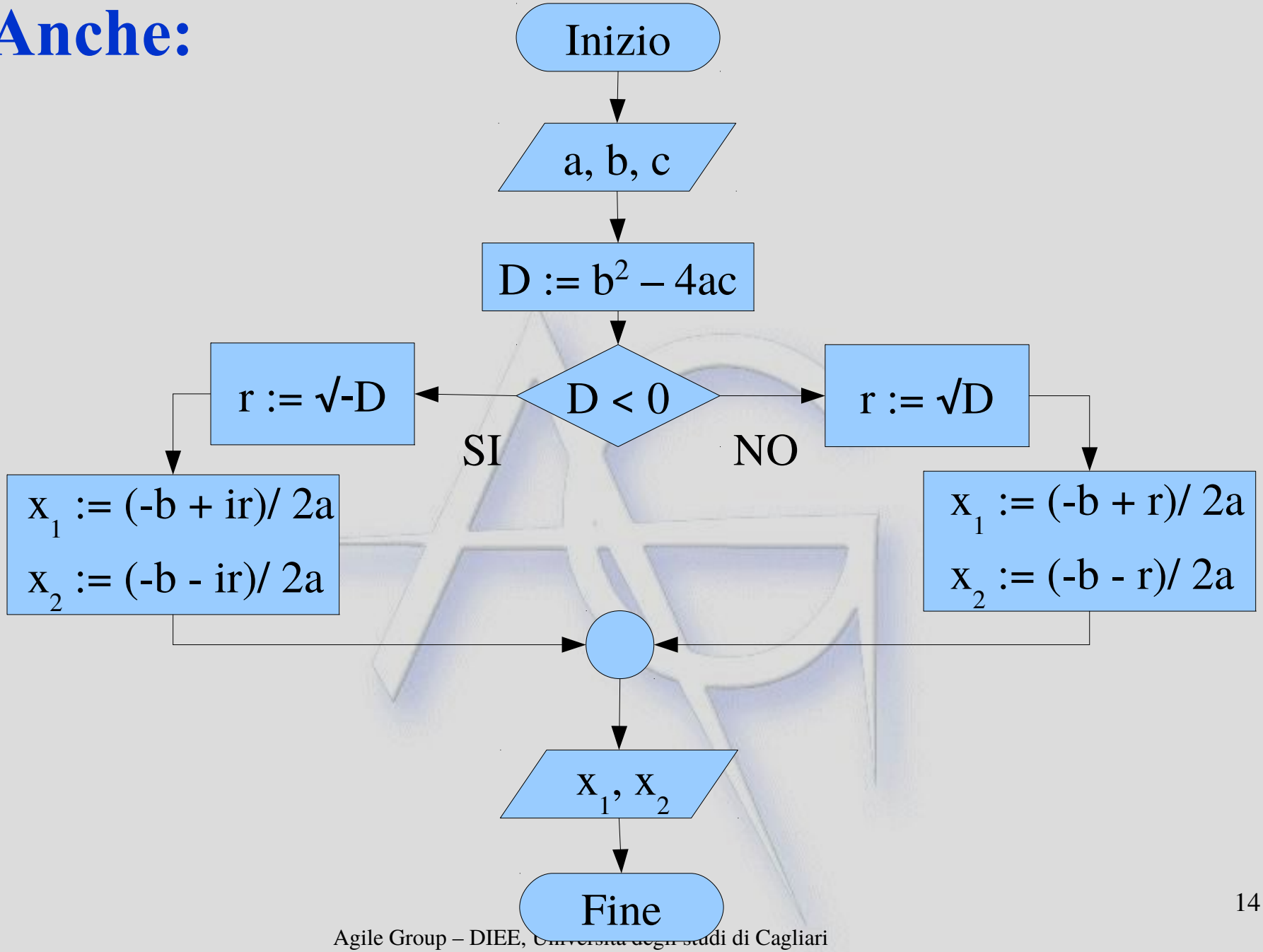
Radici di un'equazione di 2 grado (tutte):



Anche:


Inizio
a, b, c
D := b2 – 4ac
D < 0
SI
NO
r := √-D
r := √D
x1 := (-b + ir)/ 2a
x2 := (-b - ir)/ 2a
x1 := (-b + r)/ 2a
x2 := (-b - r)/ 2a
x1, x2
Fine

Anche:


Inizio
a, b, c
D := b2 – 4ac
D < 0
SI
NO
r := √-D
r := √D
x1 := (-b + ir)/ 2a
x2 := (-b - ir)/ 2a
x1 := (-b + r)/ 2a
x2 := (-b - r)/ 2a
x1, x2
Fine

Calcolo del MCD di due interi *i* e *k*

- **Idea:**
  - **se il più grande tra *i* e *k* è multiplo dell'altro, allora il più piccolo è il MCD**
  - **altrimenti, divido il più grande per il più piccolo e calcolo il resto *r* > 0**
    - **se il resto *r* è sottomultiplo del più piccolo, allora il MCD è *r*. Es: 18, 12 → *r* = 6, 12 è divisibile per 6 → 6 è l'MCD!**
    - **altrimenti, continuo a dividere il resto precedente per quello successivo**
      - **se arrivo a un resto = 1 → l'MCD = 1**
      - **se no, è il sottomultiplo > 1 trovato**

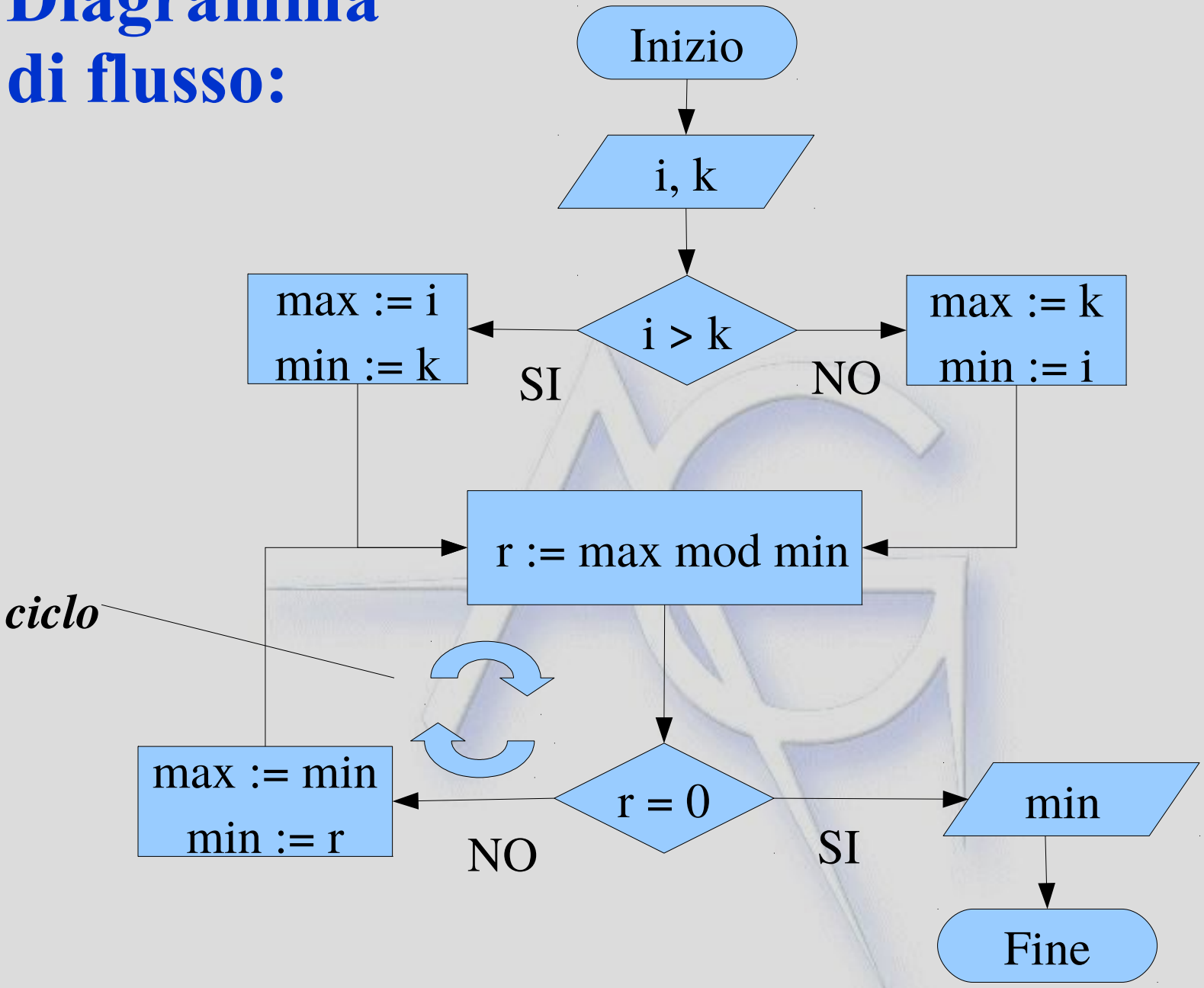
Calcolo del MCD: algoritmo

1. **Input: variabili *i* e *k* (i due numeri naturali)**
2. **Se *i* > *k* *max* := *i* ; *min* := *k* ;**
   altrimenti: *max* := *k* ; *min* := *i* ;
3. ***r* := *max* mod *min* (resto della divisione *max/min*)**
4. **Se *r* = 0 Il risultato è *min* (output); FINE**
5. **Altrimenti: *max* := *min* ; *min* := *r* ;**
6. **Va al passo 3**

Variabili locali usate (intere): *max*, *min*, *r*

***r* diminuisce a ogni passo, quindi l'algoritmo termina sempre, al più con risultato pari a 1 (*i*, *k* primi fra loro)**

Diagramma di flusso:



Cicli

- **Il calcolo del MCD presenta un *ciclo***
- **Un ciclo è una ripetizione di una sequenza di istruzioni zero, una o più volte, senza che tale numero sia in genere noto a priori**
- **Nei cicli, le stesse variabili assumono valori diversi nelle varie iterazioni del ciclo**
- **Nell'algoritmo, il ciclo è implementato con un'istruzione condizionale al passo 4 e con un "salto" al passo 6 ("GOTO")**
- **Esiste il costrutto "*while*" (finché) per realizzare questo tipo di cicli senza usare il GOTO**

Calcolo del MCD di due interi *i* e *k*

1. **Input: variabili *i* e *k***
2. **Se *i* > *k* *max* := *i* ; *min* := *k* ;**
   altrimenti: *max* := *k* ; *min* := *i* ;
3. ***r* := *max* mod *min***
4. **Finché *r* ≠ 0**
   - ***max* := *min* ; *min* := *r* ;**
   - ***r* := *max* mod *min***
5. **Il risultato è *min* (output);**

} ciclo: eseguito finché la condizione *r* ≠ 0 è vera

Esempi di calcolo di MCD

1. **Input: variabili *i* e *k***
2. **Se *i* > *k* *max* := *i* ; *min* := *k* ;**
   altrimenti: *max* := *k* ; *min* := *i* ;
3. ***r* := *max* mod *min***
4. **Finché *r* ≠ 0**
   - ***max* := *min* ; *min* := *r* ;**
   - ***r* := *max* mod *min***
5. **Il risultato è *min* (output);**

Input: 18 e 48

ciclo	max	min	r
1	48	18	12
2	18	12	6
3	12	6	0

Input: 48 e 35

ciclo	max	min	r
1	48	35	13
2	35	13	9
3	13	9	4
4	9	4	1
5	4	1	0

In linguaggio C

```
int MCD(int i, int k) {  // funzione MCD
 int max, min, r;  // var. locali
 if(i > k) {max = i; min = k;}
    else {max = k; min = i;}
  r = max % min;  // operatore mod in C
  while(r != 0) { // "!=" è "diverso"
    max = min;
    min = r;
    r = max % min;
  }
  return min; //valore di ritorno (output)
}
```

Collezioni di dati

- **Quasi tutti gli algoritmi informatici utilizzano *collezioni* di dati**
- **Una collezione è un insieme di dati di tipo simile, organizzati in vari modi possibili**
- **La collezioni più semplice è il *vettore* (*array*):**
  - ***N* dati dello stesso tipo reperibili tramite un indice intero *i***
  - **Es. (in C)**

```
float vx[100];
int vet[10];
vet[0] = -7; // I elemento
vet[9] = vet[0]*4; // ult. elem.
```

  - **In C gli elementi variano da 0 a *N*-1 !**

Cicli di iterazioni su vettori

- **Si esegue un ciclo su tutti gli elementi del vettore, dal primo all'ultimo (o viceversa)**
- **Serve a:**
  - **trovare un elemento particolare (il min, il max...)**
  - **eseguire un'operazione cumulativa (somma, prodotto, somma dei moduli...)**
  - **selezionare degli elementi (per l'output, per inserirli in un altra collezione, …)**
  - **elaborare ogni elemento, per inserirlo in un altro vettore (ad es., i quadrati, l'intero superiore,...)**
- **Occorre un *indice corrente* sul vettore**
- **Attenti agli errori "*off-by-one*" relativi ai valori estremi assegnati all'indice corrente**

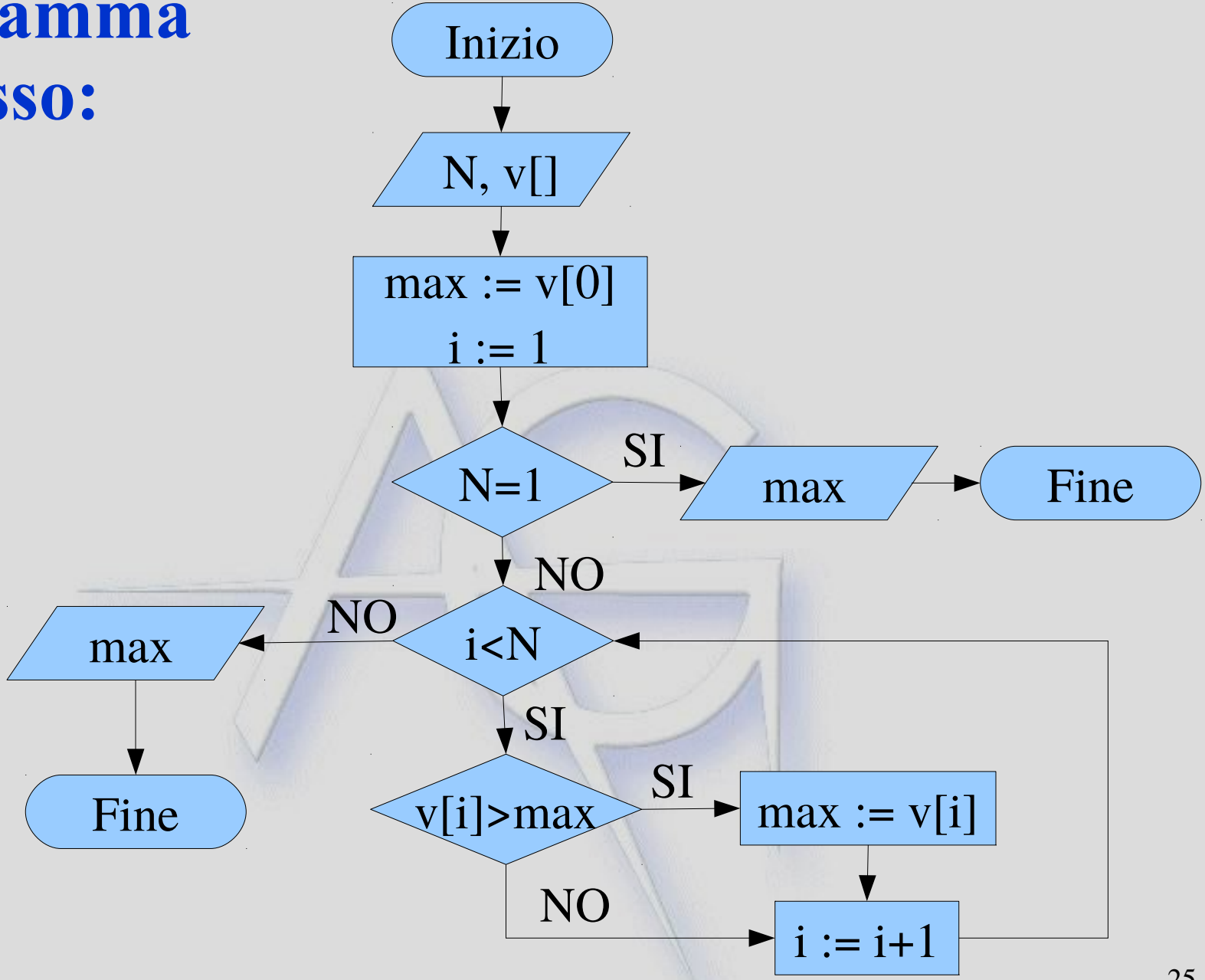
Calcolo del massimo

- **Sia un vettore di *N* interi *v* da 0 a *N*-1 (input)**
- **Cerchiamo il valore massimo *max***

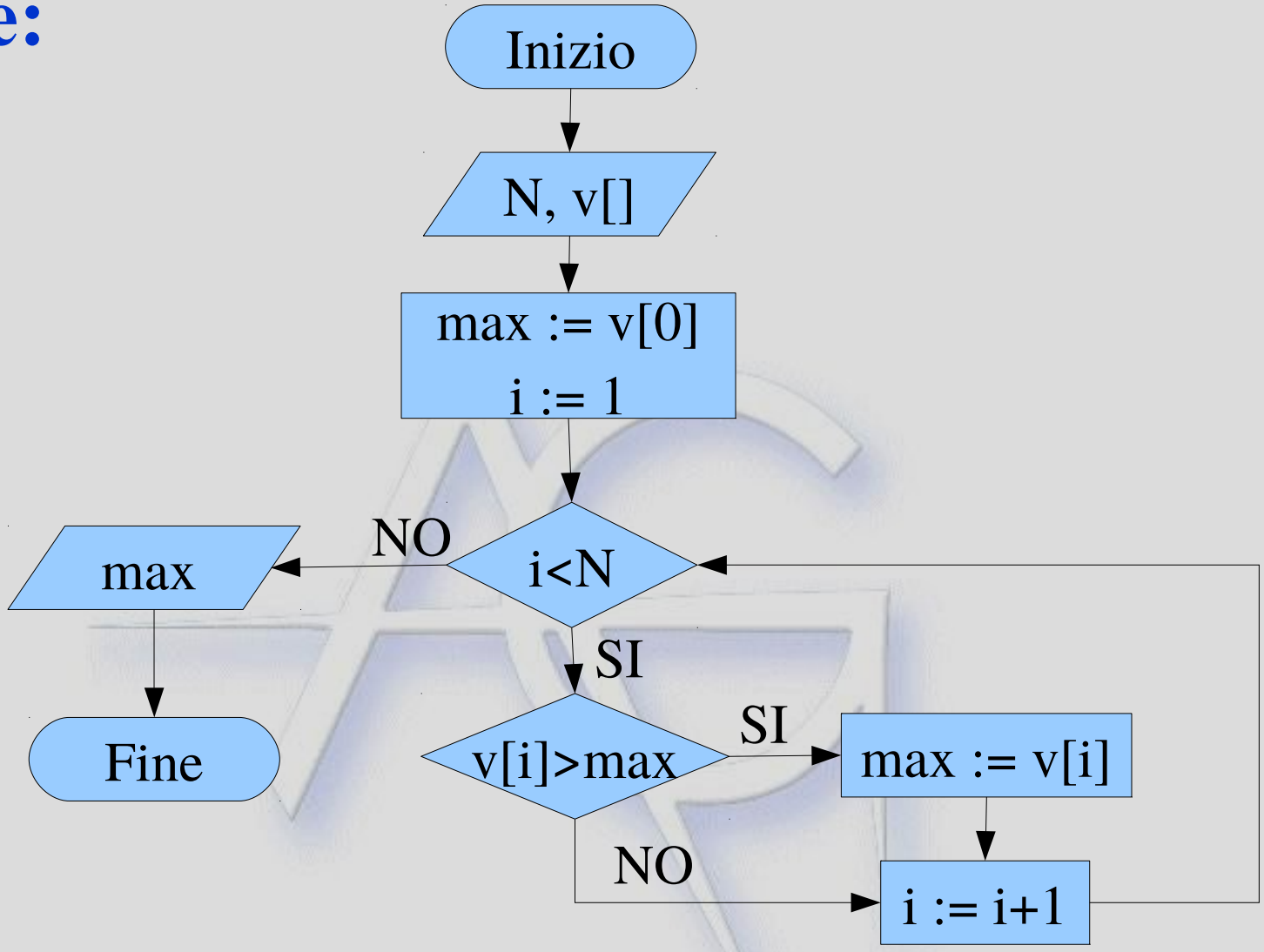
1. ***max* := *v*[0] ;**
2. **Se *N* = 1 output: *max* FINE**
3. ***i* := 1 ;**
4. **Finché *i* < *N***
   - **Se *max* < *v*[*i*] *max* := *v*[*i*] ;**
   - ***i* := *i* + 1 ;**
5. **output: *max* FINE**

L'indice corrente è *i*

Diagramma di flusso:



Anche:



Calcolo della somma

- Sia un vettore di *N* interi *v* da 0 a *N*-1 (input)
- Calcoliamo la somma dei suoi valori: *sum*

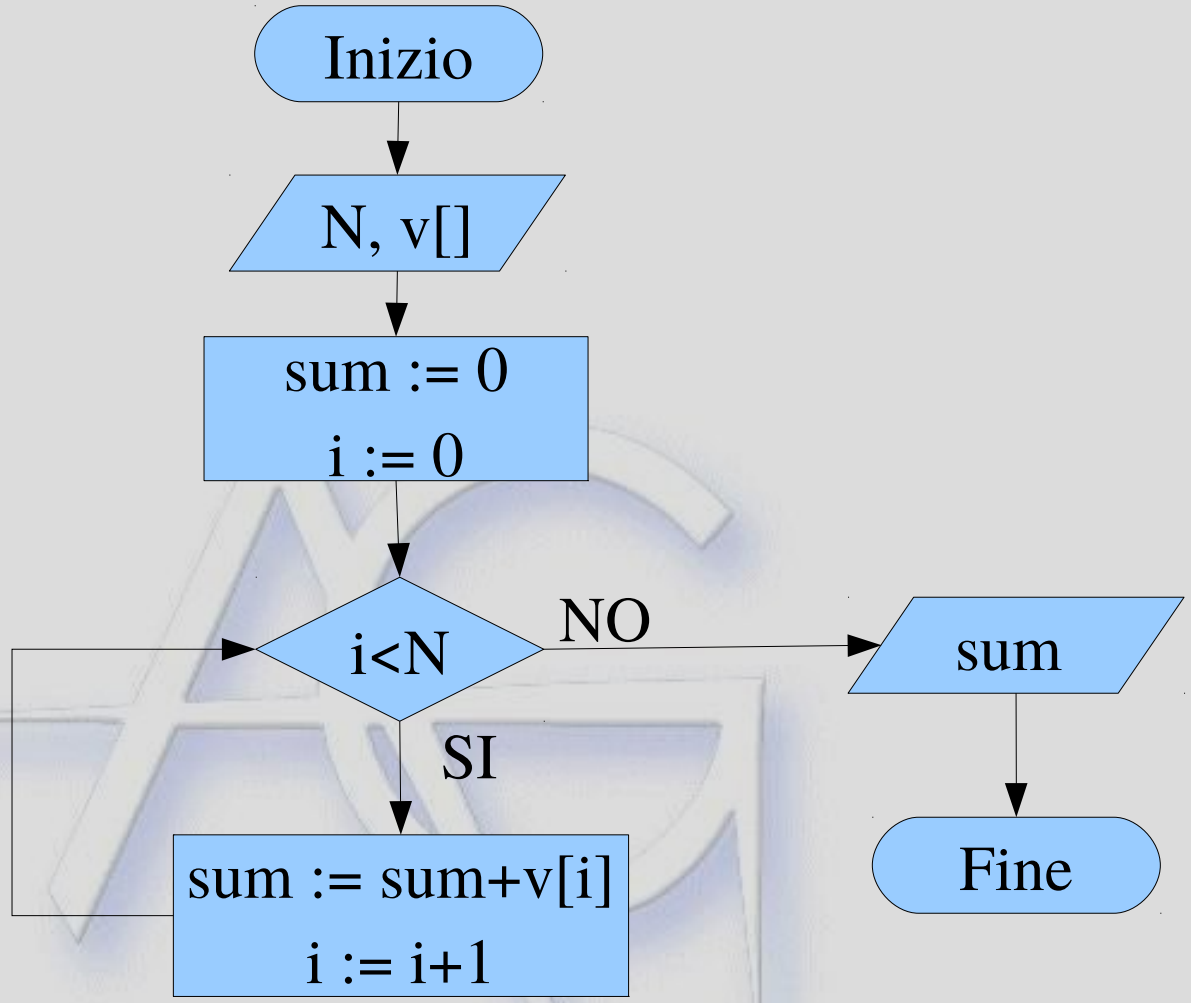
1. *sum* := 0 ;
2. *i* := 0 ;
3. Finché *i* < *N*
   - *sum* := *sum* + *v*[*i*] ;
   - *i* := *i* + 1 ;
4. output *sum*

in linguaggio C:

```
int N, v[MAX_DIM];
// … v e N sono riempiti

int sum = 0;
int i = 0;
while(i < N) {
  sum = sum + v[i];
  i = i + 1; // o  i++;
}
// sum contiene la somma
```

Calcolo della somma:



Ciclo “for”

- **Un ciclo si può anche esprimere col costrutto: “*per ogni*” (for), del tipo:**
- **Per ogni *i* che va da 0 a *N*-1, con passo di 1**
  - ***sequenza di elaborazioni***
- **Occorre fornire:**
  - **Valore iniziale della variabile/i del ciclo; es. $i := 0$**
  - **Condizione per continuare; es. $i < N$**
  - **Incremento alla fine di ogni ciclo; es. $i := i+1$**

Calcolo della somma col for

- **Sia un vettore di *N* interi *v* da 0 a *N*-1 (input)**
- **Calcoliamo la somma dei suoi valori: *sum***

1. ***sum* := 0 ;**
2. **Per ogni *i*, con:**
   - **$i := 0$; all'inizio**
   - **finché $i < N$;**
   - **3.1 $sum := sum + v[i]$;**
   - **($i := i + 1$;)**
4. **output *sum***

in linguaggio C:

```
int N, v[MAX_DIM];
// … v e N sono riempiti

int sum = 0;
int i;
for(i=0; i < N; i++) {
   sum = sum + v[i];
   }
// sum contiene la somma
```

Annidamenti

- **Negli algoritmi, elaborazioni condizionali e cicli si possono annidare l'uno entro l'altro, anche a molti livelli:**
  - **cicli entro cicli, sia di tipo *while* che *for***
  - **elaborazioni condizionali entro cicli (già visto nel calcolo del massimo valore di un vettore)**
  - **cicli eseguiti solo a certe condizioni**
- **Molto importanti sono i cicli annidati, ad es.**
  - **per ordinare un vettore**
  - **per elaborare dati bidimensionali (tabelle, immagini,...)**

Esempio: ordinamento di un vettore

1. Per ogni *i*, con: <- ciclo esterno, inserisce il minimo
- ***i* := 0 ; all'inizio** in posizione *i*
- **finché *i* < *N* - 1 ;**

1.1 *posmin* := *i*;

1.2 Per ogni *j*, con: <- ciclo interno, trova la posiz.
- ***j* := *i* + 1; all'inizio** del minimo tra *i* e *N* - 1
- **finché *j* < *N* ;**
- **1.2.1 se *v*[*j*] < *v*[*posmin*] *posmin* := *j* ;**
- **(*j* := *j* + 1;)**

1.3 *aux* := *v*[*i*] ; *v* [*i*] := *v*[*posmin*] ;

1.4 *v*[*posmin*] := *aux* ;

- **(*i* := *i* + 1;)**

Esempio: $v[] = \{7,2,5,3,4\}$, $N = 5$

1. Per ogni *i*, con:
   - *i* := 0 ; all'inizio
   - finché *i* < *N* - 1 ;

   1.1 *posmin* := *i*;

   1.2 Per ogni *j*, con:
   - *j* := *i* + 1; all'inizio
   - finché *j* < *N* ;

   1.2.1 se *v*[*j*] < *v*[*posmin*]
   posmin := *j* ;

   - (*j* := *j* + 1;)

   1.3 *aux* := *v*[*i*] ;

   1.4 *v* [*i*] := *v*[*posmin*] ;

   1.5 *v*[*posmin*] := *aux* ;
- (*i* := *i* + 1;)

				v[]				
i	*j*	*posmin*	*aux*	0	1	2	3	4
0		0		7	2	5	3	4
0	1	1						
0	2	1						
0	3	1						
0	4	1						
0		1	7	2	7	5	3	4
1		1						
1	2	2						
1	3	3						
1	4	3						
1		3	7	2	3	5	7	4
2		2						
2	3	2						
2	4	4						
2		4	5	2	3	4	7	5
3		3						
3	4	4						
3		4	7	2	3	4	5	7
				0	1	2	3	4

Sotto-algoritmi e funzioni

- **Negli algoritmi, spesso occorre eseguire altri algoritmi di livello inferiore, anche più volte.**
- **A tal fine esiste il concetto di *funzione* (in senso informatico, non matematico)**
- **Una funzione (detta anche *procedura* o *subroutine*) è un'elaborazione (un algoritmo) con un nome univoco, che si può richiamare passandogli zero o più parametri di input**
- **La funzione *può:***
  - **modificare lo stato del sistema**
  - **restituire un valore di ritorno singolo**
  - **restituire anche altri valori, oltre al ritorno**

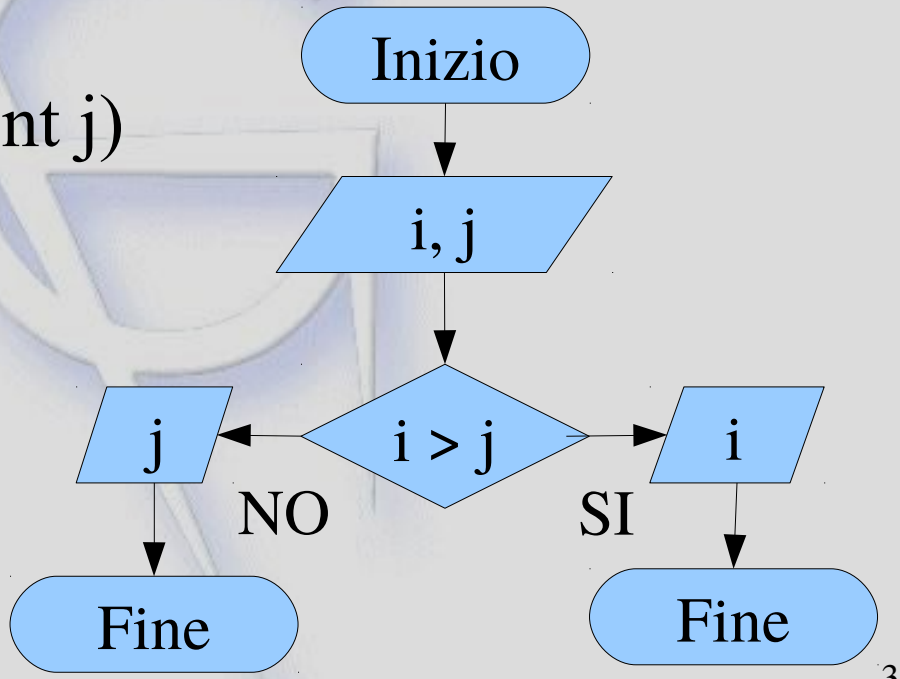
La funzione max

- Sia la funzione di nome *max*
- Scopo: calcolare il massimo tra due interi
- Ha due parametri di input (di tipo int)
- Ha un valore di ritorno, che è il massimo tra i due:

Funzione: int max(int i, int j)

se i > j ***return*** i

altrimenti ***return*** j



Esempio: calcolo codifica cognome per codice fiscale

- Algoritmo per il calcolo del cognome (3 caratteri), assumendo che sia di almeno 3 caratteri
  - scrivi di seguito le consonanti del cognome
  - se sono meno di 3, scrivi le vocali, in ordine
  - ignora spazi e apici, come Di Chiara, de' Magistris
- Input: un vettore di caratteri col cognome (tutti maiuscoli) e la sua dimensione
- Output, un vettore di 3 caratteri, con la parte di CF
- Per ogni carattere del cognome occorre sapere se è
  - vocale
  - consonante
  - altro

Calcolo codifica cognome

- Input: char cognome[], int dimCognome
- Output char cognomeCF[3]
- Var. locali:
  - char c : carattere corrente
  - int dimCF : nr. di caratteri scritti in cognomeCF[]
- Idea: scandisco cognome[] selezionando le consonanti e aggiungendole in cognomeCF[]; alla 3a restituisco il CF
- Se ho meno di 3 consonanti, scandisco di nuovo cognome[] e cumulo le vocali; al 3 carattere restituisco il CF
- I due casi sono eseguiti uno dopo l'altro
- Se ho meno di 3 lettere nel cognome, dà errore
- *Nello pseudo-codice seguente i vettori hanno indice da 1 a dim, e non da 0 a dim-1 come in C*

Esempio: calcolo codifica cognome (1)

- Mi serve una funzione int tipoChar(char c) che renda:
  - -1 se non è lettera maiuscola
  - 0 se è vocale maiuscola
  - 1 se è consonante maiuscola
- Input: char cognome[], int dimCognome
- char cognomeCF[3]; int i; int tipo;
  int dimCF := 0;
- per ogni i = 1 .. dimCognome ← *ciclo da 1 a dimCognome*
  - tipo := tipoChar(cognome[i]) ← *chiamata funzione*
  - se tipo = 1 (*consonante*) dimCF := dimCF + 1,
    cognomeCF[dimCF] := cognome[i],
    se dimCF = 3 return cognomeCF[]

Esempio: calcolo codifica cognome (2)

- *Se siamo arrivati qui, ci sono meno di 3 consonanti*
- per ogni i = 1 .. dimCognome
  - tipo := tipoChar(cognome[i]) ← *chiamata funzione*
  - se tipo = 0 (*vocale*) dimCF := dimCF + 1,
    cognomeCF[dimCF] := cognome[i],
    se dimCF = 3 return cognomeCF[]
- *Se siamo arrivati qui, ci sono meno di 3 consonanti e vocali: ERRORE!*

Diagramma di flusso (1)

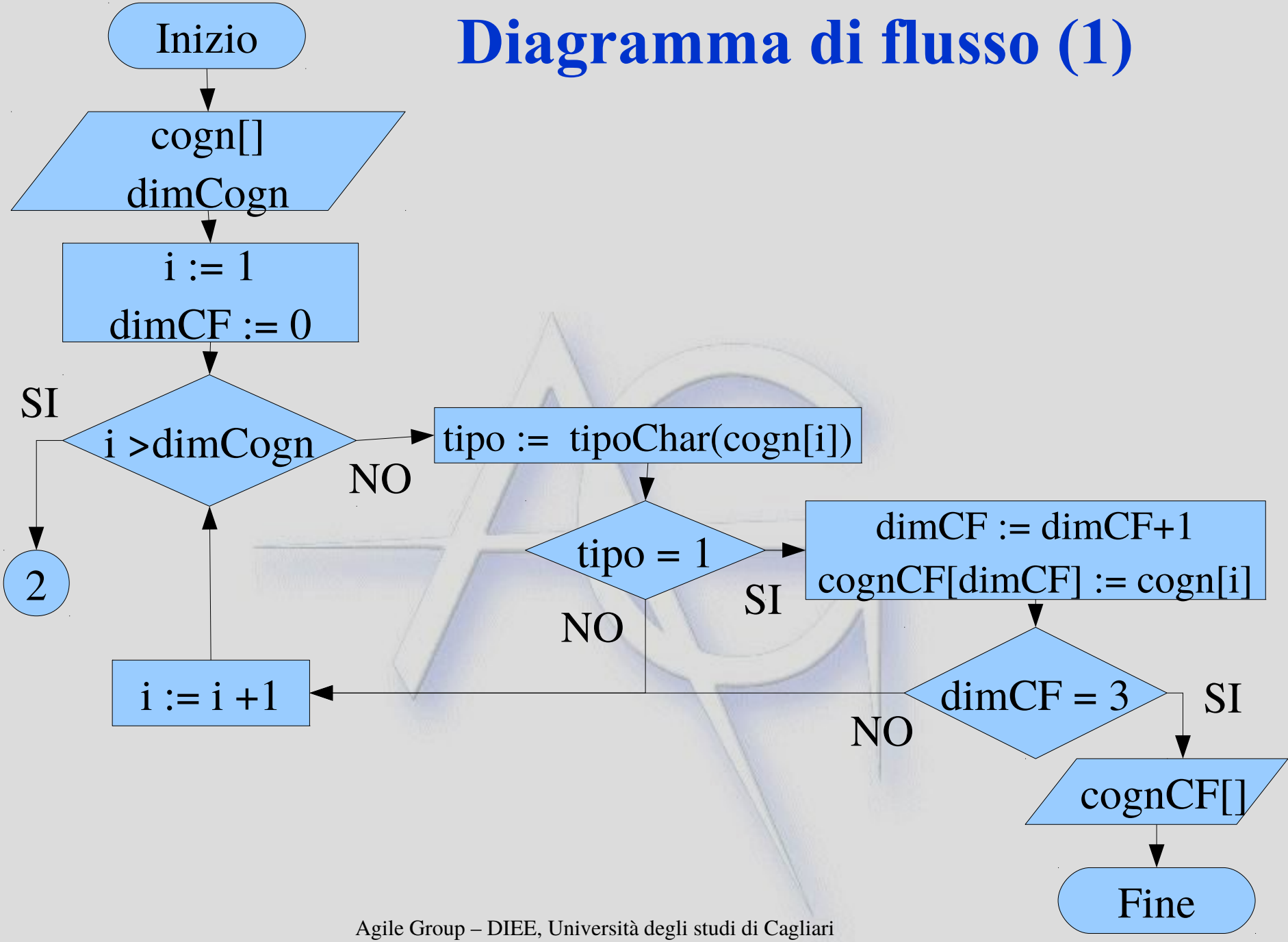
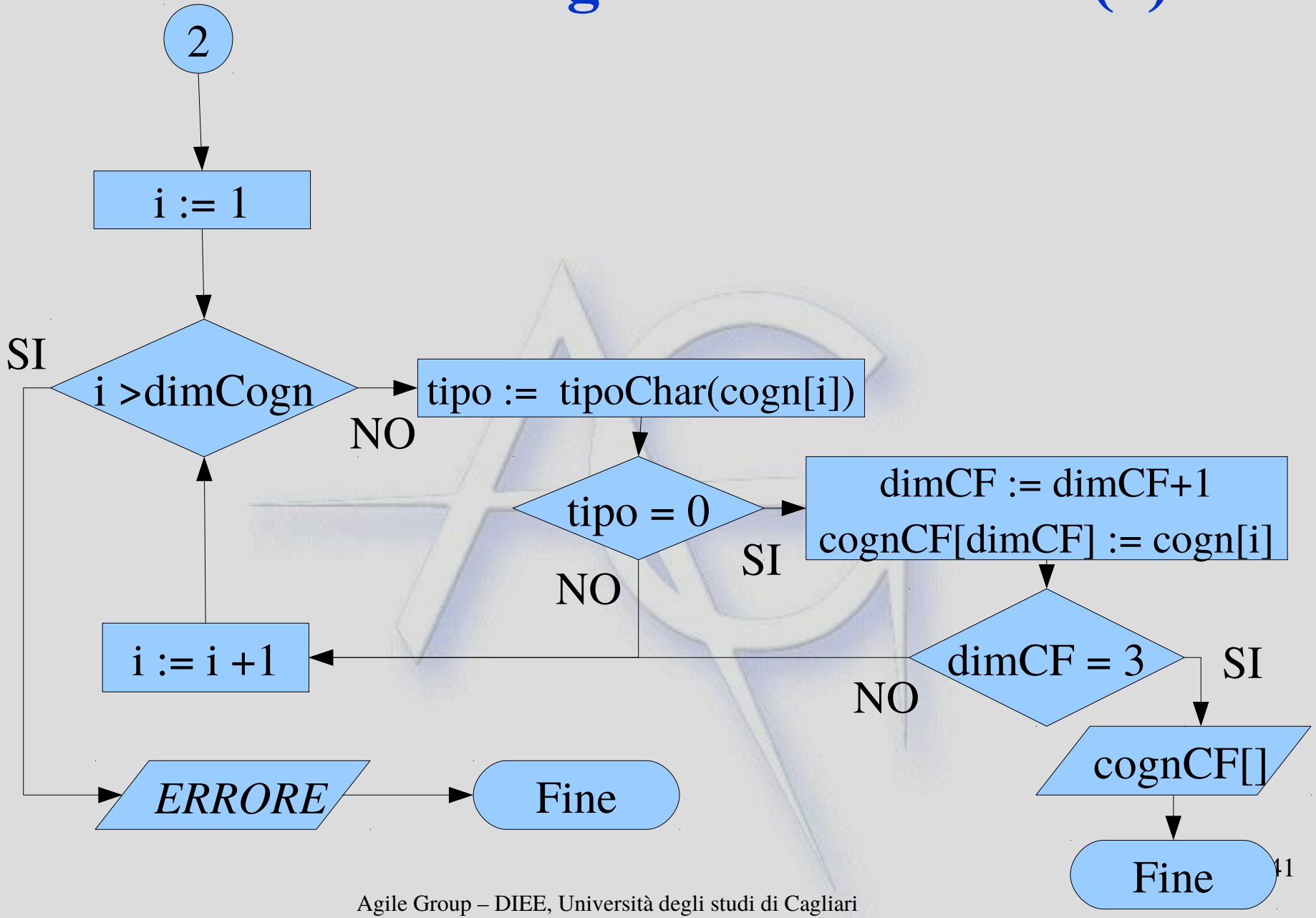


Diagramma di flusso (2)



Funzione tipoChar()

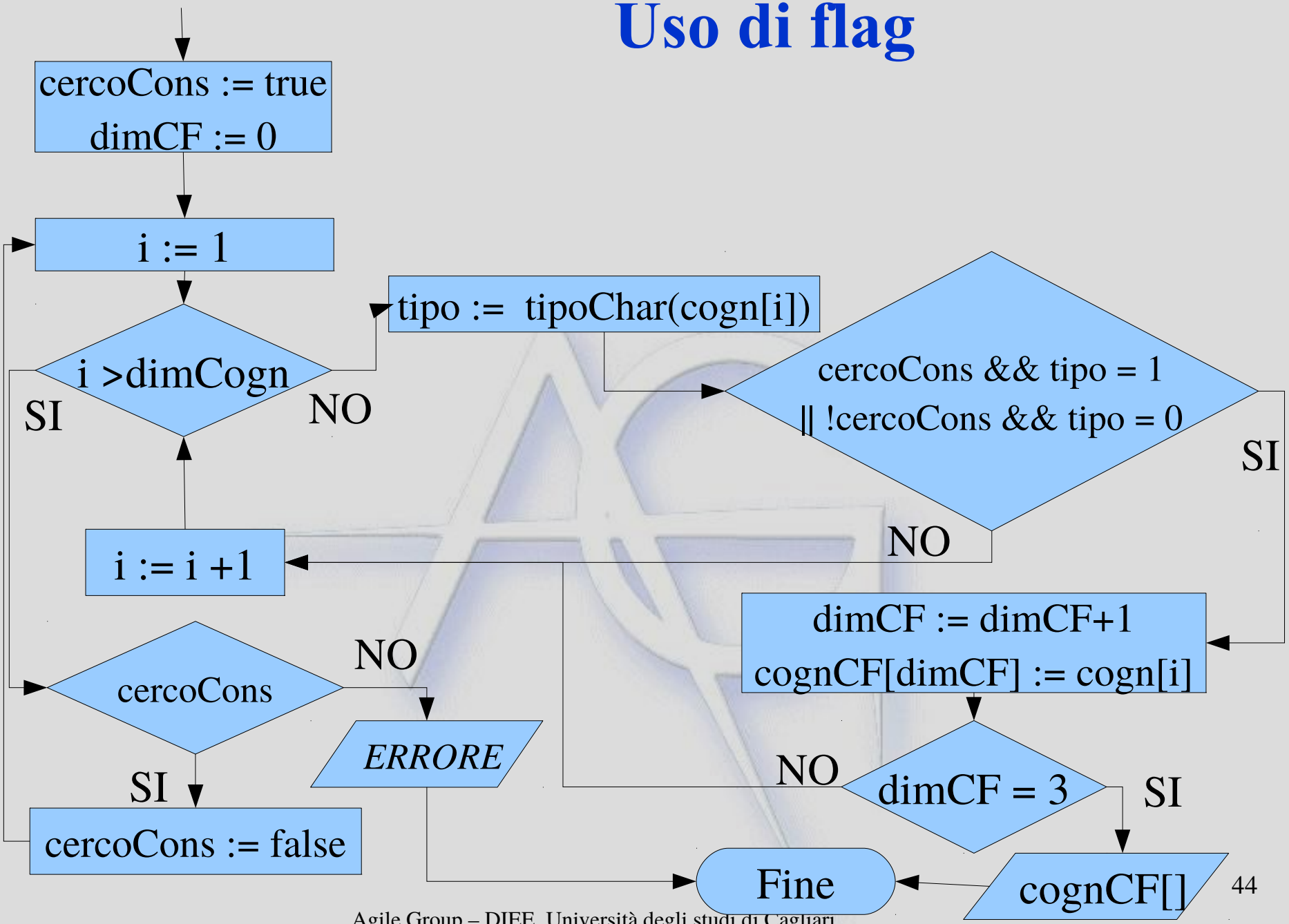
- Input: char c
- Output int
- Var. locali:
  - char vocali[5] := {'A', 'E', 'I', 'O', 'U'}
  - char consonanti [20] := {'B', 'C', 'D', 'F', … , 'Z'}
  - int i (indice)
- per ogni i = 1 .. 5
  - se c = vocali[i] ***return*** 0
- per ogni i = 1 .. 20
  - se c = consonanti[i] ***return*** 1
- ***return*** -1

Uso di *flag*

- Una flag (bandierina) è una variabile boolean (può valere *true* o *false*)
- Viene usata per differenziare un algoritmo, a seconda che la condizione segnalata dalla flag sia vera o falsa
- Ad esempio, la ricerca di consonanti o vocali poteva essere fatta in un solo ciclo, con una flag che discrimina se stiamo cercando una consonante o una vocale:

```
boolean cercoCons
```

Uso di flag


cercoCons := true
dimCF := 0
i := 1
i >dimCogn
SI
NO
tipo := tipoChar(cogn[i])
cercoCons && tipo = 1
|| !cercoCons && tipo = 0
SI
NO
i := i +1
dimCF := dimCF+1
cognCF[dimCF] := cogn[i]
cercoCons
NO
ERRORE
SI
cercoCons := false
NO
dimCF = 3
SI
Fine
cognCF[]