# Fortran 90[/95]

M. Bianco

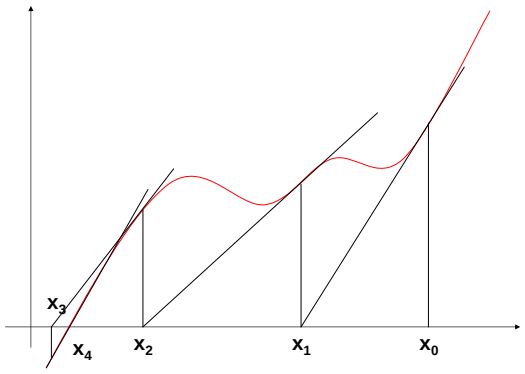
## Metodo di Newton-Raphson

- Metodo per trovare uno zero di una funzione generica $f(x) \in C^1(\mathbb{R})$
- Metodo molto robusto anche se esistono casi in cui non funziona (tipico dei metodi generali)
- Partendo da un punto iniziale $x_0$ si calcolano la tangente di f in $x_0$ e l'intercetta di questa sull'asse delle ascisse ($x_1$)

## Metodo di Newton-Raphson

- Si procede **iterativamente** fino a che non si raggiunge un punto $x_n$ che possiamo considerare uno zero di f(x) ($f(x_n) \approx 0$)
- Quindi: dato $x_k$ si calcola il punto $x_{k+1}$ in cui la tangente di f in $x_k$ intercetta l'asse delle ascisse

## Metodo di Newton-Raphson




## Progetto:

- La tangente in $x_k$ è la retta seguente:

$$y=f'(x_k)(x-x_k)+f(x_k)$$

- Calcolare il valore della funzione in un punto
- Calcolare la derivata della funzione in un punto
- Calcolare il punto in cui la tangente intercetta l'asse delle ascisse:

$$x_{k+1}= x_k-f(x_k)/f'(x_k)$$

## Progetto

- Il derivata può essere approssimata come

$$\mathbf{f'(x_k)=( f(x_k+delta) - f(x_k) ) / delta}$$

- Dove '**delta**' è un valore sufficientemente piccolo per garantire la precisione del calcolo… ma non troppo…
- Non è l'unico modo in cui si può fare.

## Ecco l'algoritmo

1. Fisso $x_0$
2. x ? $x_0$
3. Fino a che abs(f(x)) > precisione
   a. Calcola f(x)
   b. Calcola f(x+delta)
   c. Calcola f'(x) ? (f(x+delta) -f(x))/delta
   d. Calcola x ? x-f(x)/f'(x)
   e. Torna a 3.
4. Stampa x

## Requisiti

- In ogni ciclo dobbiamo calcolare 3 volte il valore di f in tre punti distinti
- f può essere arbitrariamente complessa da calcolare
- Scrivere tre volte il codice per il calcolo all'interno del programma ha due controindicazioni
  - **Cambiare la funzione risulta difficile e *error prone***
  - **Il programma diventa illeggibile**

## Sottoprogrammi e Funzioni

- Un sottoprogramma (**subroutine**) è una procedura che accetta un certo numero fissato di **argomenti** che può elaborare
- Consente di isolare compiti specifici che possono essere **sviluppati e testati singolarmente**
- Questi compiti possono essere progettati in modo tale da poter essere eseguiti su dati diversi e quindi di **riutilizzare il codice**
- Gli argomenti limitano il numero di variabili che la routine può accedere e modificare, limitando così alcuni effetti indesiderati (**side effects**)

## Subroutine e Funzioni

- Le subroutine vengono ***chiamate*** con una istruzione **call**

```
CALL MYSUB(i,x)
```

- Le funzioni, come le subroutine, accettano un lista di argomenti (anch'essi modificabili)
- Producono un valore dello stesso tipo del tipo di funzione (una funzione intera produce un valore intero, una reale un reale, etc…)
- Si chiamano come le funzioni intrinseche

```
a=miafun(x)+miafun(y)
```

## Esempio Subroutine

```
PROGRAM primes
  IMPLICIT NONE
  INTEGER :: i ! Non e` la stessa variabile
               ! che usiamo in primetest
  INTEGER :: n ! Input
  LOGICAL :: isprime

  write (*,*) "Inserire n"
  read (*,*) n

  DO i=2,n
    CALL prime(i, isprime) ! Chk se i e` primo
    IF (isprime) WRITE (*,*) i, "e` primo"
  END DO
END PROGRAM primes
```

## Esempio Subroutine

```
SUBROUTINE prime(n, eprime)
  IMPLICIT NONE ! ANCHE QUI!
  INTEGER, INTENT(IN) :: n ! Non e` l'input!
  LOGICAL, INTENT(OUT) :: eprime
  INTEGER :: i ! Variabile locale!

  eprime=.true.
  DO i=2,n/2+1
    IF ((n/i)*i==n) THEN
      eprime=.false.
      EXIT
    END IF
  END DO
  RETURN ! Opzionale alla fine della routine
END SUBROUTINE prime
```

## Esempio Funzione

```
PROGRAM primes
  IMPLICIT NONE
  INTEGER :: i ! Non e` la stessa variabile
               ! che usiamo in isprime()
  INTEGER :: n ! Input
  LOGICAL, EXTERNAL :: isprime

  write (*,*) "Inserire n"
  read (*,*) n

  DO i=2,n
    IF (isprime(i)) WRITE (*,*) i, "e` primo"
  END DO
END PROGRAM primes
```

## Esempio Funzione

```
LOGICAL FUNCTION isprime(n)
  IMPLICIT NONE ! ANCHE QUI!
  INTEGER, INTENT(IN) :: n ! Non e` l'input!
  INTEGER :: i ! Variabile locale!

  isprime=.true.
  DO i=2,n/2+1
    IF ((n/i)*i==n) THEN
      isprime=.false.
      EXIT
    END IF
  END DO
 ! Il valore di isprime e` il valore calcolato
END FUNCTION isprime
```

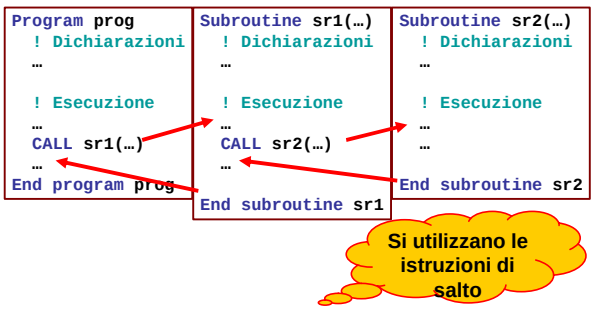
## Esempio Funzione

```
LOGICAL FUNCTION isprime(n) RESULT(p)
  IMPLICIT NONE ! ANCHE QUI!
  INTEGER, INTENT(IN) :: n ! Non e` l'input!
  INTEGER :: i ! Variabile locale!

  p=.true.
  DO i=2,n/2+1
    IF ((n/i)*i==n) THEN
      p=.false.
      EXIT
    END IF
  END DO
  ! Il valore di p e` il valore calcolato
END FUNCTION isprime
```

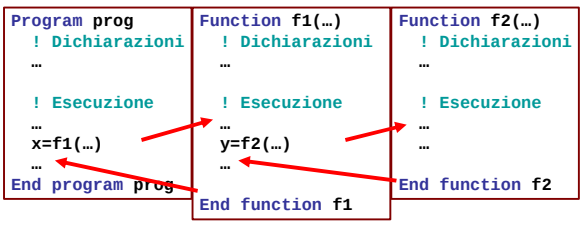
## Struttura di subroutine e funzioni

- Come un normale programma le subroutine e le funzioni sono composte da
  - Una sezione dichiarativa
  - Una sezione esecutiva
  - Una sezione conclusiva

## Schema di chiamata




## Schema di chiamata



## Chiamate innestate

- Una subroutine o una funzione può chiamare qualsiasi altra subroutine o funzione
- Una subroutine o una funzione non può chiamate se stessa a meno che non sia dichiarata **RECURSIVE**

## Esempio: numeri di Fibonacci

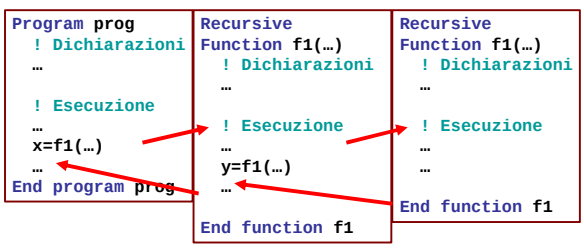
- Definizione ricorsiva:
  f(1)=1, f(2)=1
  f(n)=f(n-1)+f(n-2) se n>2
- I primi numeri di Fibonacci

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233,…

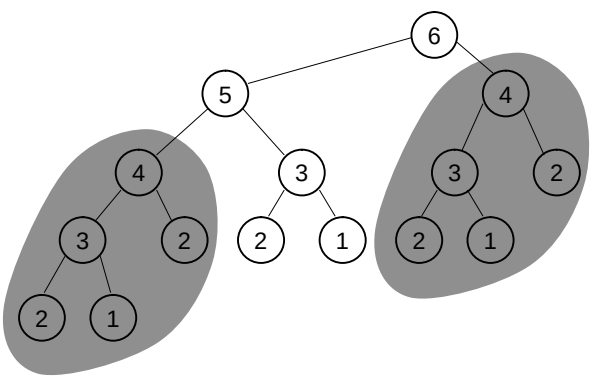
```
program progfibonacci
  integer :: n
  integer, external :: fibonacci

  write (*,*) "Inserire n"
  read (*,*) n
  write (*,*) fibonacci(n)
end program progfibonacci

integer recursive function &
  fibonacci(n) result(fib)
  integer, intent(in) :: n
  if (n<3) then
     fib=1
  else
     fib=fibonacci(n-1)+fibonacci(n-2)
  end if
end function fibonacci
```

Nel caso di funzioni ricorsive deve essere specificato

## Schema di chiamata




## Albero delle chiamate




```
integer function fibonacci(n) result(fib)
  implicit none
  integer, intent(in) :: n
  integer :: i
  integer :: fibmeno1, fibmeno2
  if (n<3) then
     fib=1
  else
    fibmeno1=1 ! Numero del passo precedente
    fibmeno2=1 ! Numero di due passi prec.
    do i=3,n
       fib=fibmeno1+fibmeno2
       fibmeno2=fibmeno1 ! Aggiorno i numeri
       fibmeno1=fib
    end do
  end if
end function fibonacci
```

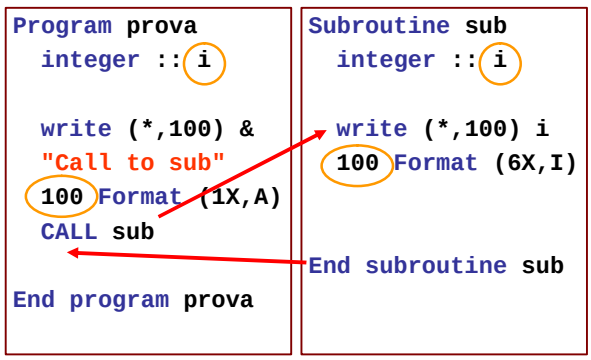
In questo caso è opzionale!

```
integer function fibonacci(n)
  implicit none
  integer, intent(in) :: n
  integer :: i
  real, parameter :: sq5=sqrt(5.)
  real, parameter:: fi=(1.+sqrt(5.))/2.

  fibonacci=nint((fi**n-fi**(-n))/sq5)

end function fibonacci
```

## Compilazione

- Le funzioni e le subroutine vengono compilate separatamente e possono quindi contenere etichette e nomi di variabili uguali!
- Questo consente lo sviluppo indipendente delle singole unità di programma!!

## Etichette delle istruzioni




## INTENT

- L'attributo INTENT specifica l'uso che si intende fare dei parametri della subroutine o della funzione
  - IN: ingresso, il parametro non può essere modificato dentro la subroutine o funzione
  - OUT: uscita, al termine della subroutine o funzione il valore del parametro potrà essere stato modificato, il valore in ingresso non sarà utilizzato (come se fosse una variabile non inizializzata)
  - INOUT: ingresso e uscita: il parametro ha informazioni in ingresso ma potrà essere modificato (DEFAULT)

## Esempio

```
integer function sd(n) result(s)
intger, intent(in) :: n

s=0
do i=1,n
  s=s+2*i-1
  ! 2*i-1 i-esimo numero dispari
end do
end function sd
```

## Esempio

```
integer function sd(n) result(s)
integer, intent(in) :: n

s=0
n=2*n-1 ! ERRORE DI COMPILAZIONE!!!
do
  s=s+n
  n=n-2
  if (n==-1) exit
end do
end function sd
```

## Esempio

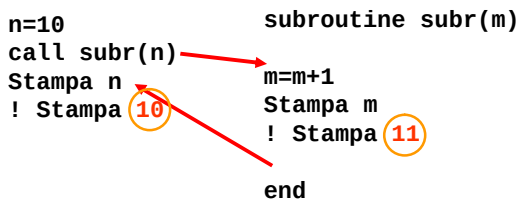
```
integer function sd(n) result(s)
integer, intent(in) :: n
integer :: tmp=n
s=0
tmp=2*tmp-1 ! OK!!
do
  s=s+tmp
  tmp=tmp-2
  if (tmp==-1) exit
end do
end function sd
```

## Passaggio dei parametri

- Esistono fondamentalmente due modi per passare gli argomenti a una funzione
  - Per valore
  - Per riferimento
- Fortran utilizza solo il passaggio per riferimento

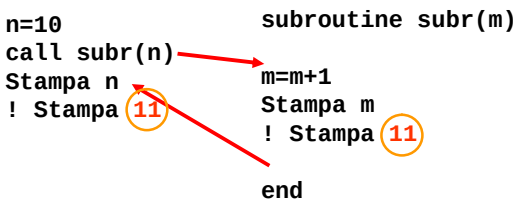
## Passaggio per valore

- Il **valore** dell'argomento viene **copiato** nell'argomento della funzione
- Qualsiasi modifica all'argomento **non si riflette** sulla variabile passata alla funzione




## Passaggio per riferimento

- L'argomento diventa un **secondo nome** per il parametro (**alias**)
- Qualsiasi modifica all'argomento **si riflette** sulla variabile passata alla funzione




## Passaggio per riferimento

- **Fortran usa esclusivamente questo!**
- Uno dei pochi dettagli che differenziano Fortran dagli altri linguaggi di programmazione
- Gli attributi INTENT non cambiano il modo di passare gli argomenti ma inducono il **compilatore** a effettuare dei controlli per garantire che l'uso degli argomenti sia conforme alle specifiche

## Ricordate che…

- Gli errori di compilazione sono errori che sono rilevati dal compilatore, quindi possono essere rilevati dalla struttura del codice sorgente!! Sono derivabili dall'**analisi sintattica e grammaticale** del programma
- Gli errori al run-time non sono direttamente derivabili dall'analisi grammaticale. Il programma in quel caso ha una grammatica corretta ma una **semantica** errate: non calcola quello che dovrebbe calcolare

## Newton-Raphson in Fortran

- Ci serve una funzione che calcoli f(x)
- Una che calcoli la derivata
- Una che calcoli l'intercetta della tangente con l'asse x
- Quest'ultima, non esistendo particolari difficoltà nella sua realizzazione non la mettiamo in una funzione ma la calcoliamo direttamente a programma

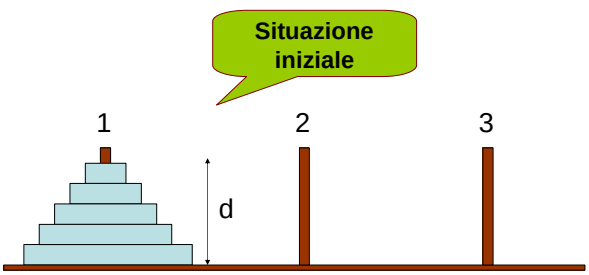
## Newton-Raphson in Fortran

- Il calcolo di f conviene che sia in una funzione a sé in quanto usiamo quella chiamata più volte e se vogliamo cambiare funzione la cambiamo in un solo punto
- Il calcolo della derivata può portare a complicazioni per funzioni particolari, quindi è meglio calcolarla in una funzione a parte e poi cambiare il calcolo della derivata senza toccare il corpo del programma
- L'intercetta si calcola in un solo modo…

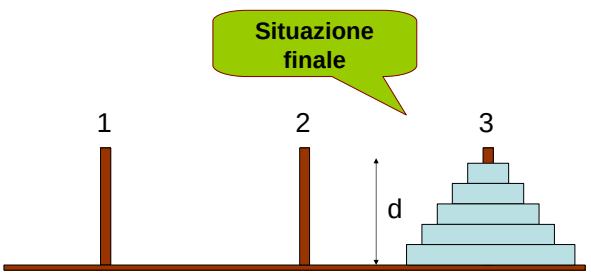
```
program NR
  real :: xi=0. ! Punto iniziale
  real :: dx=1.e-3 ! Per f'
  real, parameter :: prec=1.e-3
  ! prec = precisione da raggiungere
  real, external :: f, df
  ! f() calcola f, df calcola f'()

  do while (abs(f(xi))>=prec)
     xi= xi-f(xi)/df(xi,dx)
  end do

  write (*,*) "Zero di f trovato in",xi
end program NR
```

```
real function f(x)
  ! Valuta f in x
  real, intent(IN) :: x
  f=(0.1*x**3-3*x**2+6*x+20)
  !f=(x+2)*sin(.5*x-cos(x))
end function f

real function df(x,delta)
  ! Derivata di f in x, delta=dx
  real, intent(in) :: x,delta
  real, external :: f

  df=(f(x+delta)-f(x))/delta

end function df
```

## Torre di Hanoi



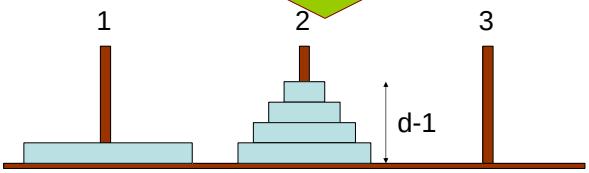

## Torre di Hanoi




**Si deve spostare un solo disco alla volta da un paletto all'altro e non si può mettere un disco sopra uno più grande**

## Torre di Hanoi

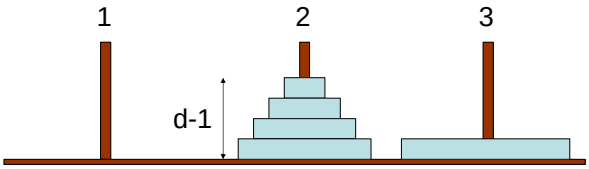
**Risolvo il problema con d-1 dischi dal paletto iniziale a quello intermedio**




**Lo posso fare perché il disco che rimane nel paletto 1 è il più grande di tutti e quindi non è possibile sovrapporci un disco ancora più grande**

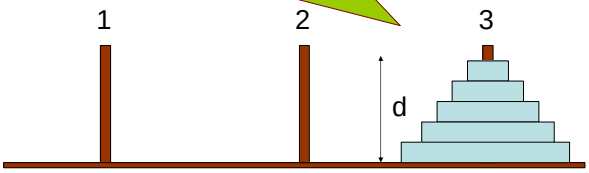
## Torre di Hanoi

**Sposto il disco più grande sul paletto finale**



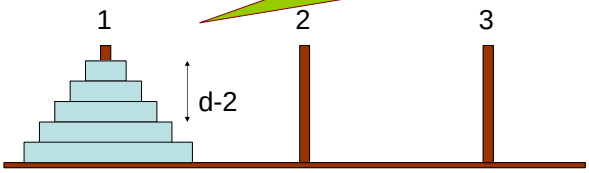

## Torre di Hanoi

**Risolvo il problema con d-1 dischi dal paletto intermedio a quello finale**



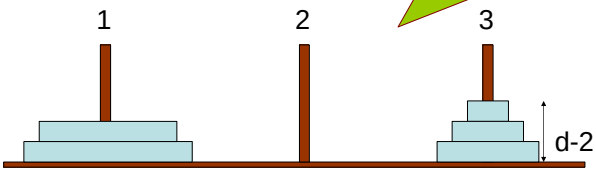

## Torre di Hanoi (un passo in più)

**Il problema con d-1 dischi lo risolvo risolvendo quello con d-2 sul paletto 3**



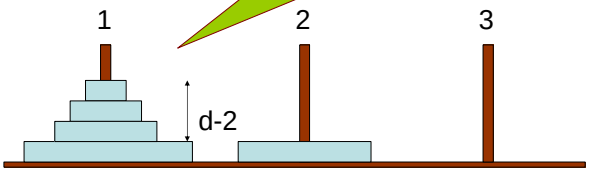

## Torre di Hanoi (un passo in più)

**...Spostando il disco da 1 a 2**

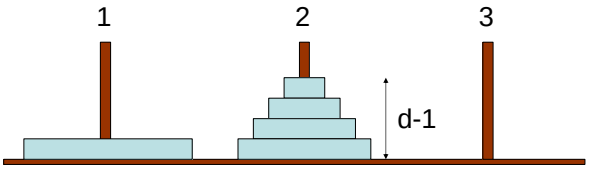



## Torre di Hanoi (un passo in più)

**Adesso risolvo il problema con d-2 da 1 a 2**



## Torre di Hanoi (un passo in più)




## Torre di Hanoi

Se i, j e k sono tre numeri distinti da 1 a 3:

Hanoi(d, i, j) è il problema di Hanoi con d dischi dal paletto i al paletto j

- Se d>1
  1. Hanoi(d, i, j) Si risolve così:
     1. Hanoi(d-1, i, k) Risolvi Hanoi con d-1 dischi da i a k
     2. m(i,j) Sposta il disco dal palo i a quello j
     3. Hanoi(d-1, k, j) Risolvi Hanoi con d-1 dischi da k a j
- Se d=1
  1. Hanoi(1, i, j)=m(i, j)

## Passare degli array

- Gli array possono essere passati alle subroutine e alle funzioni
- Le dimensioni degli array possono essere impostate all'interno delle subroutine e delle funzioni usando i valori forniti da altri argomenti in ingresso
- Questo è possibile grazie al fatto che il passaggio è per riferimento e la taglia degli array dentro una subroutine deve essere impostata solo per il calcolo degli indici
- Impostate sempre le taglie degli array dentro le subroutine e le funzioni

## Trova minimo

```
integer function findmin(a,n) result(pos)
! Restituisce l'indice del valore minimo
! dell'array a(1:n)
implicit none
integer, intent(in), dimension(n) :: a
integer, intent(in) :: n
integer :: i, min
min=a(1)
pos=1
do i=2,n
  if (a(i)<min) then
    min=a(i) ! Memorizzo il valore minimo
    pos=i ! Posizione del minimo 'locale'
  end if
end do
end function findmin
```

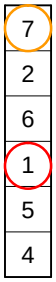
## Sorting

- Data una sequenza di *n* numeri in un array ordinare l'array in ordine crescente
- Input **a=(/ 5, 2, 8, 4, 1, 7, 2 /)**
- Output **a=(/ 1, 2, 2, 4, 5, 7, 8 /)**
- Algoritmo di swap: scambia i contenuti di due variabili:

```
tmp=a ! tmp è una variabile
a=b   ! temporanea il suo
b=tmp ! contenuto alla fine
      ! non ci interessa
```

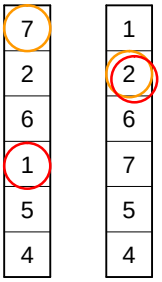
```
subroutine sort(dati, n) ! Insertion sort
integer, intent(in) :: n
integer, intent(inout), dimension(n) :: dati
integer :: i, j, tmp, min, pos

do i=1,n-1
  min=dati(i)
  pos=i
  do j=i+1,n ! Cerco il minimo
    if (dati(j)<min) then
      min=dati(j) ! Aggiorno in minimo
      pos=j       ! Aggiorno la posizione
    end if
  end do
  tmp=dati(i) ! Algoritmo di SWAP: scambio di dati
  dati(i)=dati(pos)
  dati(pos)=tmp
end do

end subroutine sort
```

## Esempio



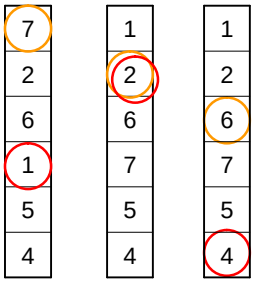

i=1
pos=4
min=1

## Esempio



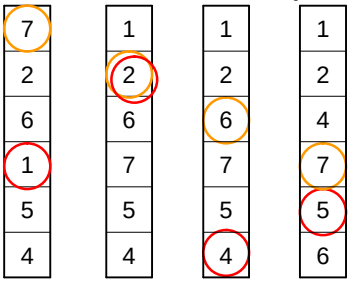

i=1 i=2
pos=4 pos=2
min=1 min=2

## Esempio



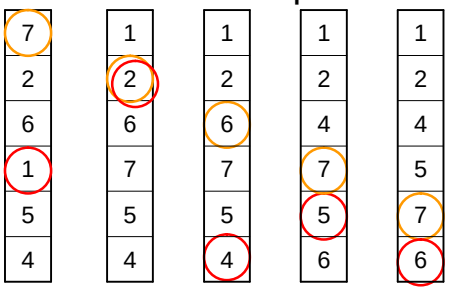

i=1 i=2 i=3
pos=4 pos=2 pos=6
min=1 min=2 min=4

## Esempio



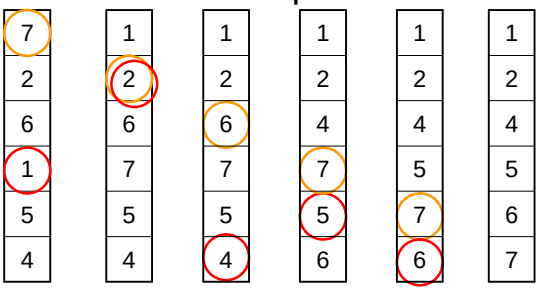

i=1 i=2 i=3 i=4
pos=4 pos=2 pos=6 pos=5
min=1 min=2 min=4 min=5

## Esempio




i=1 i=2 i=3 i=4 i=5
pos=4 pos=2 pos=6 pos=5 pos=6
min=1 min=2 min=4 min=5 min=6

## Esempio




i=1 i=2 i=3 i=4 i=5
pos=4 pos=2 pos=6 pos=5 pos=6
min=1 min=2 min=4 min=5 min=6

```
subroutine sort(dati, n) ! Insertion sort
integer, intent(in) :: n
integer, intent(inout), dimension(n) :: dati
integer :: i, tmp
integer, dimension(1) :: pos

do i=1,n-1
  pos=minloc(dati(i:n))+i-1
  ! Trova il minimo nella porzione restante
  tmp=dati(i) ! Algorimo di SWAP
  dati(i)=dati(pos(1))
  dati(pos(1))=tmp
end do

end subroutine sort
```

```
subroutine sort(dati, n) ! Bubble sort
integer, intent(in) :: n
integer, intent(inout),dimension(n) :: dati
integer :: i, j, tmp

do i=1,n-1 ! Esegue n-1 scansioni
  do j=1,n-1 ! Esegue una scansione
    if (dati(j)>dati(j+1)) then
      tmp=dati(j+1) ! SWAP
      dati(j+1)=dati(j)
      dati(j)=tmp
    end if
  end do
end do

end subroutine sort
```

## Esempio

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 7 | 6 | 6 | 6 | 6 | 6 | 4 | 4 | 4 |
| 6 | 6 | 7 | 4 | 4 | 4 | 4 | 6 | 6 | 6 |
| 4 | 1 | 4 | 7 | 5 | 5 | 5 | 5 | 5 | 1 |
| 5 | 5 | 5 | 5 | 7 | 1 | 1 | 1 | 1 | 5 |
| 1 | 1 | 1 | 1 | 1 | 7 | 7 | 7 | 7 | 7 |
| i=1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 |
| j=1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 |

## Esempio

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 4 | 4 | 4 | 4 | 4 | 4 | 4 | 1 | 1 | 1 |
| 6 | 6 | 6 | 1 | 1 | 1 | 1 | 4 | 4 | 4 |
| 1 | 1 | 1 | 6 | 5 | 5 | 5 | 5 | 5 | 5 |
| 5 | 5 | 5 | 5 | 6 | 6 | 6 | 6 | 6 | 6 |
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| i=3 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 4 |
| j=1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 |

## Esempio

| | | | | |
|---|---|---|---|---|
| 2 | 1 | 1 | 1 | 1 |
| 1 | 2 | 2 | 2 | 2 |
| 4 | 4 | 4 | 4 | 4 |
| 5 | 5 | 5 | 5 | 5 |
| 6 | 6 | 6 | 6 | 6 |
| 7 | 7 | 7 | 7 | 7 |
| i=5 | 5 | 5 | 5 | 5 |
| j=1 | 2 | 3 | 4 | 5 |

## Un po' di analisi

- Insertion sort fa due cicli:
  - i=1,n-1 (fa n-1 iterazioni)
  - j=i+1,n (fa n-i iterazioni)

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

- Bubble sort fa due cicli
- i=1,n-1
- j=1,n-1

$$(n-1)(n-1)=n(n-1)-(n-1)$$

iterazioni

## Moduli

- Spesso è conveniente che alcune subroutine o funzioni condividano delle variabili
- Si può fare *usando* i **moduli**
- Un modulo è una unità di programma compilata separatamente (come le subroutine e le funzioni) dove vengono dichiarate le variabili (le subroutine e le funzioni) **di** quel modulo
- Un modulo inizia con l'istruzione **`module`** e contiene solo la sezione dichiarativa (e le subroutine e le funzioni di modulo se ci sono)

## Esempio

```
module dati
  implicit none
  integer :: contatore
  integer, parameter :: max=100
end module dati
```

## Esempio

```
program provamodulo
  use dati ! Prima di implicit none
  implicit none
  integer :: n

  contatore=0 ! Variabile di modulo *init*
  n=rand(10023)*max ! Numero tra 0 e max-1
  do i=1,n
    call pariodispari(int(rand(0)*max))
  end do
  write (*,*) &
    "Media degli argomenti",&
    real(contatore)/real(n)
end program provamodulo
```

## Esempio

```
subroutine pariodispari(n)
  use dati ! Prima di implicit none
  implicit none
  integer, intent(in), n

  contatore=contatore+n
  if (mod(n,2)==1) then
    write (*,*) n, "e` dispari"
  else
    write (*,*) n, "e` pari"
  end if
end subroutine pariodispari
```

provarandom.f90

```
MODULE ran001
IMPLICIT NONE
INTEGER :: n = 9876
END MODULE ran001

SUBROUTINE random0 ( ran )
USE ran001
IMPLICIT NONE
REAL, INTENT(OUT) :: ran
n = MOD (8121 * n + 28411, 134456 )
ran = REAL(n) / (134456.)
END SUBROUTINE random0

SUBROUTINE seed ( iseed )
USE ran001
IMPLICIT NONE
INTEGER, INTENT(IN) :: iseed
n = ABS( iseed )
END SUBROUTINE seed
```

programrandom.f90

```
program provarandom
  implicit none
  real :: n
  integer :: i

  call seed(1213443)

  do i=1,1000
     call random0(n)
     write (*,*) n
  end do
end program provarandom
```

## Compilazione in due passate

- Se il programma è in due file separati li si può compilare con un unico comando
- gfortran provarandom.f90 programrandon.f90 -o pr

- Oppure si possono fare *due passate*:
- gfortran provarandom.f90 -c
- gfortran programrandon.f90 -c
- gfortran provarandom.o programrandom.o -o pr
- I primi due comandi producono il file oggetto il terzo esegue il link!!