Validazione attraverso il TESTING

Un programma funziona su un insieme di input se la sua esecuzione su ciascun elemento dell'insieme dà il risultato voluto

se l'insieme di input possibili è piccolo è possibile eseguire il programma su tutti gli inputs e controllare il risultato: testing esaustivo

Ma per la maggior parte dei programmi l'insieme dei casi possibili è così grande che un testing esaustivo è impossibile

un insieme ben scelto di casi di test può accrescere la nostra fiducia che il programma funziona come specificato o rilevare la maggior parte degli errori

Per un buon testing sono fondamentali

- **la scelta dei casi di test**
- **l'organizzazione del processo di testing**

TESTING

Lo scopo del testing è rilevare la presenza di errori

Il testing non indica dove sono localizzati gli errori
questa informazione si ottiene con il debugging

Nel testing si controlla la relazione tra gli inputs e gli outputs
nel debugging per cercare l'errore si controllano anche agli stati intermedi della computazione

la chiave per il successo del testing è la scelta di dati di test

TESTING

Si deve trovare un insieme ragionevolmente piccolo di casi di test che consenta di approssimare l'informazione che avremmo ottenuto con il testing esaustivo. Ci sono due approcci fondamentali per selezionare i dati di test:

1. **testing a scatola nera (black-box testing)**

2. **testing basato sul codice (glass-box testing)**

Nel testing a scatola nera i casi di test sono generati considerando la sola specifica, senza considerare il codice del programma sotto test

TESTING A SCATOLA NERA

I **vantaggi di questo approccio sono**

1. **il testing non è influenzato dall'implementazione del programma**

   esempio

   il programmatore ha erroneamente ed implicitamente assunto che il programma non sarebbe stato mai chiamato su un certo insieme di valori di input e quindi non ha incluso il codice per trattare tale insieme di valori. Se i dati di test fossero generati guardando l'implementazione, non si genererebbero mai dati di quell'insieme

2. **robustezza rispetto a cambiamenti dell'implementazione**

   infatti i dati non devono essere cambiati anche se sono stati fatti cambiamenti al programma sotto test

3. **i risultati di un test possono essere interpretati da persone che non conoscono i dettagli interni dei programmi**

TESTING A SCATOLA NERA: scelta dei dati

1. Dati ottenuti esaminando la specificazione della funzione.

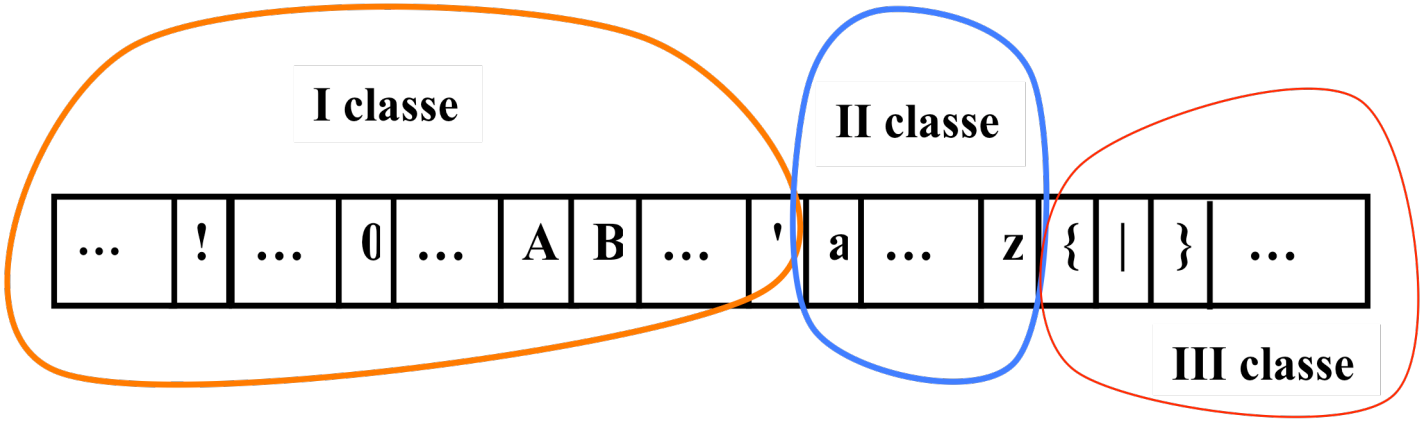
Analizziamo un esempio: la funzione toupper

char toupper(char c)

postc: restituisce il carattere maiuscolo corrispondente a quello in input se si tratta di un carattere alfabetico minuscolo, restituisce il carattere inalterato altrimenti

I possibili input di questa funzione sono i 256 caratteri, codificati con 8 bits.

TESTING A SCATOLA NERA: scelta dei dati



Dalle postcondizioni ricaviamo che l'insieme dei dati di input andrebbe suddiviso in due sottoinsiemi: i caratteri alfabetici e i non alfabetici. Poiché i dati sono ordinati però, conviene considerare i non alfabetici che precedono gli alfabetici e quelli che li seguono separatamente.
Consideriamo quindi la divisione dei dati in tre insiemi che contengono elementi che possono essere considerati EQUIVALENTI rispetto al testing:

- **i caratteri minori di 'a',**
- **quelli tra 'a' e 'z'**
- **quelli maggiori di 'z'.**

Scegliamo un elemento per ogni insieme, per esempio '0', 'f' e '}'

TESTING A SCATOLA NERA: scelta dei dati

2. Dati ottenuti esaminando considerando i casi al "confine" tra le classi.

In questo caso notiamo che la funzione dovrebbe essere controllata sui caratteri 'a' e 'z', che sono il primo e l'ultimo dell'insieme dei dati su cui il risultato è diverso dagli altri, ma anche l'ultimo del primo insieme '\' e il primo del terzo '{'.

Quindi un insieme di dati di test ragionevole per questa funzione è: '0', 'f' , '}', 'a', 'z' e '{'.

TESTING A SCATOLA NERA: scelta dei dati

La verifica dei casi suggerita dalle postcondizioni consente di controllare i casi eccezionali: non trattarli è altrettanto grave quanto non dare il risultato giusto con un input normale

La verifica dei casi al confine è fondamentale per il controllo di due tipi di errori

1. **errori di tipo logico**

   Per esempio è stato trascurato un caso speciale
2. **mancato controllo di condizioni che possono causare il sollevamento di errore o da parte del linguaggio o da parte del sistema (per esempio overflow aritmetico)**

TESTING A SCATOLA NERA: scelta dei dati

Consideramo un altro esempio

```
int palindrome (char *s);
/*prec: s != NULL
postc: restituisce 1 se s è palindroma, 0 altrimenti */
```

1. In base alle postcondizioni dividiamo l'insieme di tutte le possibili stringhe in input in quelle palindrome e quelle no, e prendiamo una stringa in ciascun insieme: "osso" e "mare".
2. I casi al confine: la stringa vuota " ",
3. Quindi l'insieme dei dati di test che ricaviamo è

1. "osso", "mare"," "

TESTING A SCATOLA NERA: scelta dei dati

```
int cerca (int[] a, int n, int x)
/* restituisce una posizione di x nel vettore  a se vi
      occorre, -1 altrimenti
prtec: a != NULL*/
```

1. dalle postcondizioni consideriamo i seguenti sottoinsiemi, da cui trarre i dati di test per ogni vettore di n elementi:
   1. elementi che occorrono nel vettore: x = a[i], per 0<=i<n
   2. elementi che non occorrono nel vettore: x≠ a[i] per 0<=i<n

2. I casi al confine per a (a[0],a[n-1]) sono già considerati, per n prendiamo n=1 e n= 2.

TESTING A SCATOLA NERA: scelta dei dati

In conclusione per ogni input

1. **Si deve dividere l'insieme dei valori che può assumere in sottoinsiemi disgiunti, ciascuno contenenti elementi equivalenti rispetto al testing**

2. **Scegliere un valore a caso come rappresentante di ogni classe**

3. **prendere valori al confine di ogni classe.**

TESTING: scelta dei dati

L'approccio a scatola nera è un buon punto di partenza per il testing ma raramente è sufficiente.

Bisogna generalmente affiancare alla scelta di dati basata sull'approccio a scatola nera quelli che nascono

dall' analisi del codice, in cui si tiene conto del codice del programma sotto test

Uno dei criteri di scelta dei dati basati sul codice comporta la scelta di dati con i quali vengono eseguiti cammini diversi nel programma.

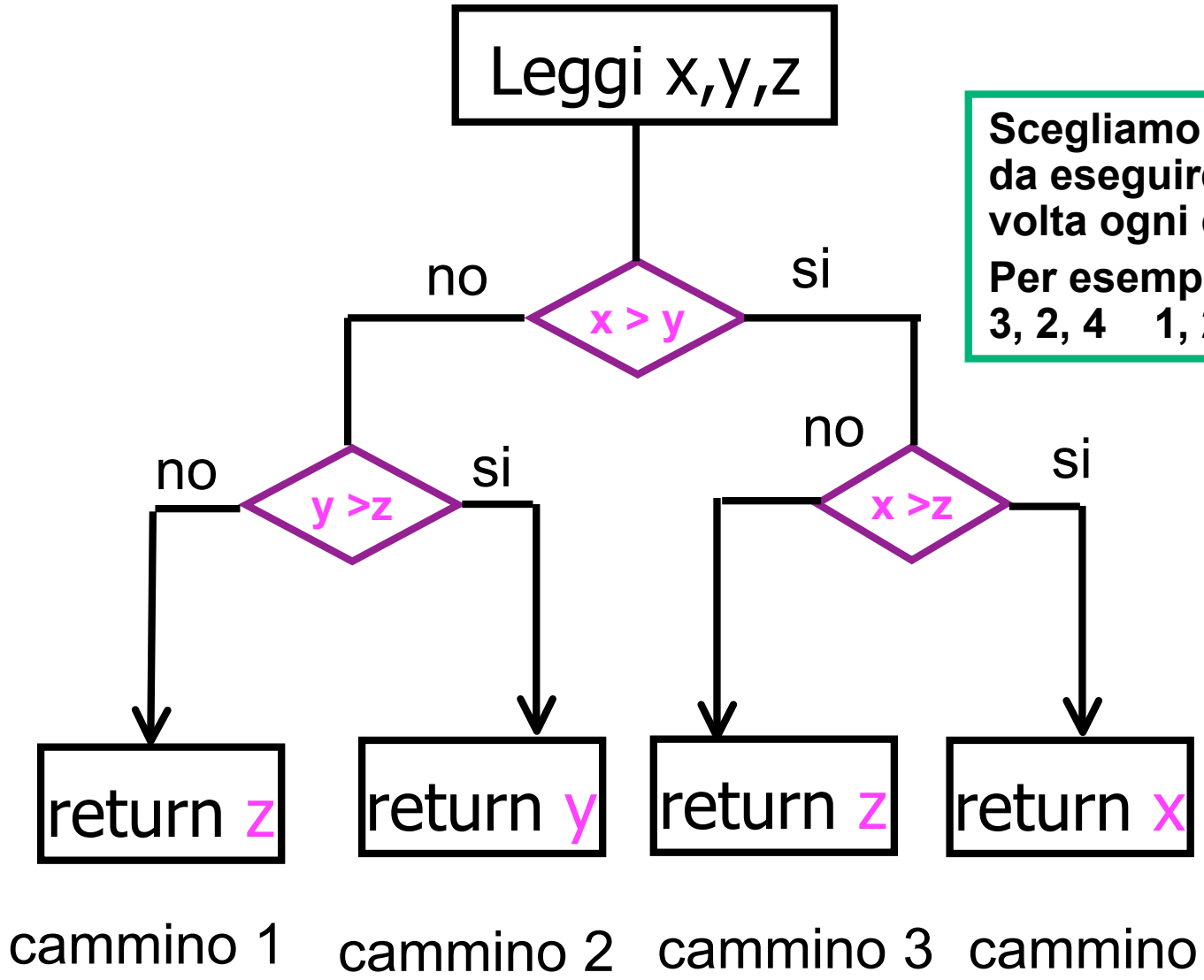
Per ogni cammino del programma, ci dovrebbe essere un dato nell'insieme di test (criterio di minima copertura)

Esempio: Massimo tra tre numeri

```
int maxDiTre (int x, int y, int z)
{if (x > y)
        if (x > z) return x; else return z;
 if (y > z) return y; else return z;
 }
```

Qui ci sono n^3 diversi inputs, dove n è il numero degli interi consentito dal linguaggio di programmazione

Esempio: Massimo tra tre numeri

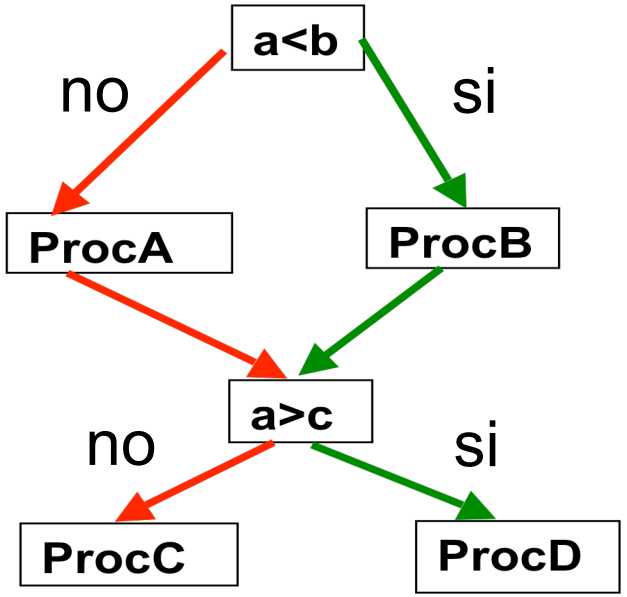


Scegliamo i dati in modo da eseguire almeno una volta ogni cammino.

Per esempio: 3, 2, 1
3, 2, 4 1, 2, 1 1, 2, 3

TESTING basato sul codice: scelta dei dati

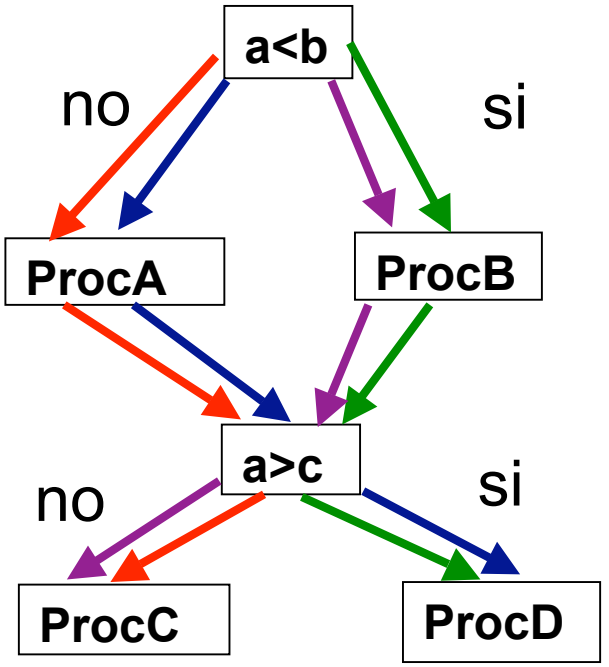
```
int f( int a ) {
  if( a < b ) {
    a = procA( a );
    }
  else {
    a = procB( a, b );
    }
  if( a > c ) {
    procC( a, c );
    }
  else {
    procD( a, b );
    }
  return a;
  }
```



I cammini non sono 2!!

TESTING basato sul codice: scelta dei dati

```
int f( int a ) {
  if( a < b ) {
    a = procA( a );
    }
  else {
    a = procB( a, b );
    }
  if( a > c ) {
    procC( a, c );
    }
  else {
    procD( a, b );
    }
  return a;
  }
```



I cammini sono 4.

TESTING basato sul codice: scelta dei dati

Esistono strumenti automatici (Test Coverage Tools e profiler) per analizzare un programma dal punto di vista del numero di esecuzioni di una singola istruzione

Possono essere utilizzati per individuare istruzioni mai eseguite. Si aggiungono nuovi dati di test fino a ottenere il risultato di minima copertura delle istruzioni

TESTING a scatola nera o trasparente?

```
int maxDiTre (int x, int y, int z)
{ return x; }
```

Qui c'è un solo cammino e quindi un insieme di dati come x= 2, y=1, z=1 rispetta il criterio di copertura minima. Poichè il programma eseguito su questa scelta non dà errore potremmo concludere che il programma è corretto.

Il problema è che guardando solo l'implementazione NON si vedono i cammini che mancano

E questo è un tipico errore nel passaggio da specifica a implementazione

Conclusione: è sempre necessario prevedere una selezione di dati basata sulla specifica: testing a scatola nera

TESTING a scatola trasparente

```
char toupper( char c )
```

postc: restituisce il carattere maiuscolo corrispondente a quello in input se si tratta di un carattere alfabetico minuscolo, restituisce il carattere inalterato altrimenti

```
{ if ( (c>='a') || (c=<'z') )
     return c - 'a' + 'A';
  else    return c;  }
```

Qui ci sono due cammini quindi l'esame del codice non suggerisce ulteriori dati di test rispetto a quelli individuati a scatola nera.

TESTING a scatola trasparente

```
int palindrome (char *s);
/*prec: s != NULL
postc: restituisce 1 se s è palindroma, 0 altrimenti */
{int in = 0;
 int fine=strlen(s)-1;
 while (fine >in) {
   if (s[in] != s[fine]) return 0;
     in++; fine--; }
  return 1; }
```

Qui abbiamo un ciclo e quindi il numero di cammini di esecuzione dipende dalla lunghezza della stringa in input

Spesso è impossibile verificare tutti i cammini. Una regola consiste nel considerare

- almeno due iterazioni attraverso un ciclo
- almeno due chiamate ricorsive

TESTING a scatola trasparente

```
int palindrome (char *s);
/*prec: s != NULL
postc: restituisce 1 se s è palindroma, 0 altrimenti */
{int in = 0;
 int fine=strlen(s)-1;
 while (fine >in) {
   if (s[in] != s[fine]) return 0;
         in++; fine--; }
  return 1; }
```

casi da controllare

non esecuzione del ciclo, s= " ", c'è già

restituzione di falso dopo una sola iterazione, s= "mare", c'è già

restituzione di vero dopo una sola iterazione, aggiungiamo s= "a"

restituzione di falso dopo la seconda iterazione, aggiungiamo "osio"

restituzione di vero dopo la seconda iterazione, s= "osso" c'è già

poichè c'è solo una stringa di lunghezza dispari, e pari a 1, consideriamo anche "ama"

In conclusione una possibile scelta di dati di test è:

" " , " a ", " ama ","osso ", "osio " ,"mare ".

TESTING a scatola trasparente

Regole generali:

1. **includiamo sempre casi di test per ciascun ramo di un condizionale**

2. **approssimiamo il criterio di copertura minima dei cammini per cicli e ricorsione: per cicli con un numero fissato di iterazioni usiamo due iterazioni**
   1. **scegliamo di percorrere il ciclo due volte e non una sola perché sono possibili errori dovuti a mancata riinizializzazione dopo la prima iterazione**
   2. **dobbiamo anche includere nei test tutti i possibili modi di terminare il ciclo**

3. **per cicli con un numero di iterazioni variabile**
   1. **includiamo nel test zero, una, due iterazioni**
   2. **includiamo casi di test per tutti i possibili modi di terminare il ciclo**

4. **per le procedure ricorsive includiamo casi di test**
   1. **dati che non danno origine a una chiamata ricorsiva e**
   2. **che provocano esattamente una chiamata ricorsiva**

TESTING : scelta dei dati

Riassumendo i dati di test dovrebbero esssere scelti in modo da contenere

1. **un rappresentante per ogni classe di equivalenza**
2. **valori di confine per ogni classe**
3. **valori particolari, come 0,1, o altri (dipende dall'applicazione)**
4. **valori che soddisfino il criterio di minima copertura dei cammini**
5. **casi particolamente importanti o comuni**

Osserviamo che non è un caso che la scelta indicata come ultima riguardi proprio i casi comuni, sono infatti quelli su cui è più facile che il programma sia corretto, perchè sono quelli che il programmatorre ha in mente quando scrive la funzione.

Inoltre bisogna controllare che le precondizioni o più in generale i valori controllati con l'assert siano trattati come atteso.

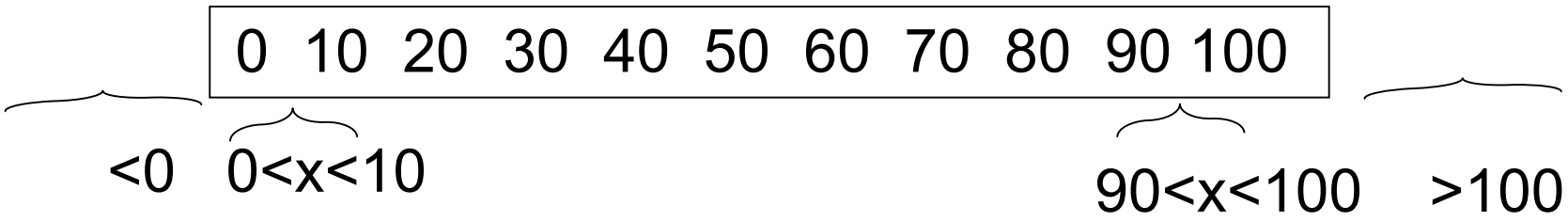
TESTING: un esempio di organizzazione

```
#define ricerca(v,x,n)   binarySearch(v,x,n)

int binarySearch(int *v, int x, unsigned int n)
/* restituisce la posizione di x in un vettore ordinato v se presente, -1
     altrimenti
prec v!=NULL e v[i] <= v[i+1],per ogni 0<=i<n*/
{ int i,m,f;
i=0;
f = n-1;
while (i<=f)
     { m= (i+f)/2;
     if ( v[m] <x) i = m+1;
     else if (v[m] == x) return m;
              else /* v[m] > x*/
          f = m-1;
      }
return -1;}
```

TESTING: ancora un esempio

Qui possiamo basare la scelta dei dati di test sull'algoritmo, piuttosto che sull'esame dei cammini,raggiungendo un risultato analogo ma più semplicemente: osserviamo che un elemento x può trovarsi nella posizione i se è presente e se non lo è va a cadere in uno degli n+1 intervalli tra elementi presenti.



Dati vettori del tipo di quello disegnato, verifichiamo il risultato per x = 0,10,20,..., 10*(n-1) e per
x = -5,5,15,25,...,10*(n-1) -5, 10*n

TESTING: organizzazione

```
int main()
{int *a,numTest,j,i ;
printf("inserisci il numero  massimo di elementi di un vettore su cui testare la
     ricerca binaria\n");
scanf("%d",&numTest);
for (i=1;i<numTest;i++)
     {a = malloc(i*sizeof(double));
      for (j=0;j<i;j++)  a[j] = 10*j; /* il caso di elementi tutti diversi*/
      for (j=0;j<i;j++)
      {assert(ricerca(a,10*j,i)==j); /* la ricerca con successo*/
       assert(ricerca(a,10*j-5,i) == -1); /* quella con insuccesso*/
       }
     assert(ricerca(a,10*i-5,i)== -1); /* penultimo intervallo*/
     assert(ricerca(a,10*i,i)== -1); /* ultimo intervallo*/
     assert(ricerca(a,10,0) == -1); /* dimensione 0 del vettore*/
     for (j=0;j<i;j++)  a[j] = 10;  /* il caso di elementi tutti uguali*/
    assert(ricerca(a,10,i)>=0 && ricerca(a,10,i)<i);
    assert(ricerca(a,5,i) == -1);
     assert(ricerca(a,15,i) == -1);
}
```