

Correttezza dei programmi

Gli algoritmi risolvono problemi computazionali
Lo con specifici input e specifici output

Requisiti:

- 1 → dato un input l'algoritmo deve produrre il giusto output
- 2 → l'algoritmo deve essere efficiente dal punto di vista delle risorse

L'algoritmo può essere diviso in 3 parti:

- 1 - l'algoritmo in sé
- 2 - la dimostrazione della sua correttezza
- 3 - la derivazione della sua esecuzione

Dimostrazione della correttezza:

Esempio: int betterLinearSearch (A, n, x) {

```
    for (int i=0; i<n; i++) {
        if (A[i]==x) return i;
    }
    return -1;
}
```

INPUT:

A → array

n → dimensione dell'array

x → valore che cerco all'interno dell'array

OUTPUT:

- i → indice se $A[i] = x$
- not found (-1) se x non è presente in A[0:n]

Invariante del ciclo

Affermazione che rimane vera ogni qualvolta il ciclo inizia

formato da 3 parti:

1 - Inizializzazione:

L'invariante è vera prima che il ciclo inizi

2 - Sussistenza (conservazione)

L'invariante rimane vera prima dell'inizio di una nuova iterazione

3 - Terminazione:

Quando il ciclo finisce l'invariante e la causa per il quale il ciclo è terminato ci forniscono una proprietà per discutere la correttezza del codice.

Invariante per $\textcircled{*}$: Se x è presente in A allora è presente nel suo sottodarray $\{A[i], A[n-1]\} \rightarrow A[i:n]$

n escluso

Prima del ciclo:

x (se presente) in A è presente in $A[0:n]$

Durante il ciclo:

inizio 1^a iterazione: x presente in $A[0:n] \rightarrow$ se non ritorno nulla significa che $A[0] \neq x$ quindi:

inizio 2^a iterazione: x presente in $A[1:n] \rightarrow$ se non ritorno nulla significa che $A[1] \neq x$ quindi:

inizio 3^a iterazione: x presente in $A[2:n] \rightarrow$ se non ritorno nulla significa che $A[2] \neq x$ quindi:

... =

Terminazione:

Se $A[i]=x \rightarrow$ apposto

Se x non è presente in $A[0:n] \rightarrow x$ non è presente in A

↳ se $i > n-1 \rightarrow A[i:n] \rightarrow$ un sottodarray vuoto $\rightarrow x$ non è presente nel sottodarray vuoto $\rightarrow x$ non è in A

↳ ritorno -1

Altra possibile invariante per $\textcircled{*}$: x non è presente in $A[0:i]$

Initializzazione:

x non presente in $A[0:0]$ (sottoarray vuoto)

Durante il ciclo:

All'inizio di ogni iterazione x non presente in $A[0:i]$ $\xrightarrow{\text{escluso}}$ se non ritorno nulla $\rightarrow x$ non presente in $A[0:i+1]$ $\rightarrow i+1 \rightarrow$ nuova iterazione

Terminazione

Se $A[i] = x$ ✓

Se x non presente in $A[0:i]$ $\xrightarrow{i=u}$ $\rightarrow x$ non presente in A

Esempio 2: int linearsearch(A, u, x) {

```
int aus = -1;  
for (int i=0; i<u; i++) {  
    if (A[i] == x) aus = i;  
}  
return aus;
```

INPUT

$A \rightarrow$ array

$u \rightarrow$ dimensione array

$x \rightarrow$ elemento da trovare

OUTPUT

• $i =$ indice di $A[i] == x$

• $-1 =$ not found

Invariante: Se $aus \neq -1 \rightarrow A[aus] == x$

altrimenti $A[0:i]$ non contiene x

Initializzazione

$aus = -1 \rightarrow A[0:i]$ (sottoarray vuoto) non contiene x

Durante il ciclo

Se $aus \neq -1 \rightarrow A[aus] == x$

altrimenti $x \notin A[0:i]$

Terminazione

Se $aus \neq -1 \rightarrow A[aus] == x$

altrimenti $x \notin A[0:i]$ $\xrightarrow{i=u}$ $\rightarrow x$ non è presente nell'array

Caso base:

$i = j$ ($i > j$) [min index > max index]

Ipotesi induttiva:

Vero per tutti gli $i \leq n$ ($i \in \{0, n\}$)

Passo induttivo:

Dimostrare che è vero per n

Funzione potenza:

```
int pow(int base, int esp){  
    if (esp == 0)  
        return 1;  
    return base * pow(base, esp - 1);  
}
```

Input:

- base \rightarrow numero intero
- esp \rightarrow numero intero ≥ 0

Output

- $(base)^{\text{esp}} \rightarrow$ numero intero

Caso base:

$\text{esp} = 0 \rightarrow$ ritorno 1 ✓

Ipotesi induttiva:

$P(b, e-1)$ vera $\rightarrow b * P(b, e-2)$

$P(b, e)$ vera $\rightarrow b * P(b, e-1) \rightarrow b * (b * P(b, e-2)) \dots * P(b, e-e) \rightarrow$ vera
caso base = 1