

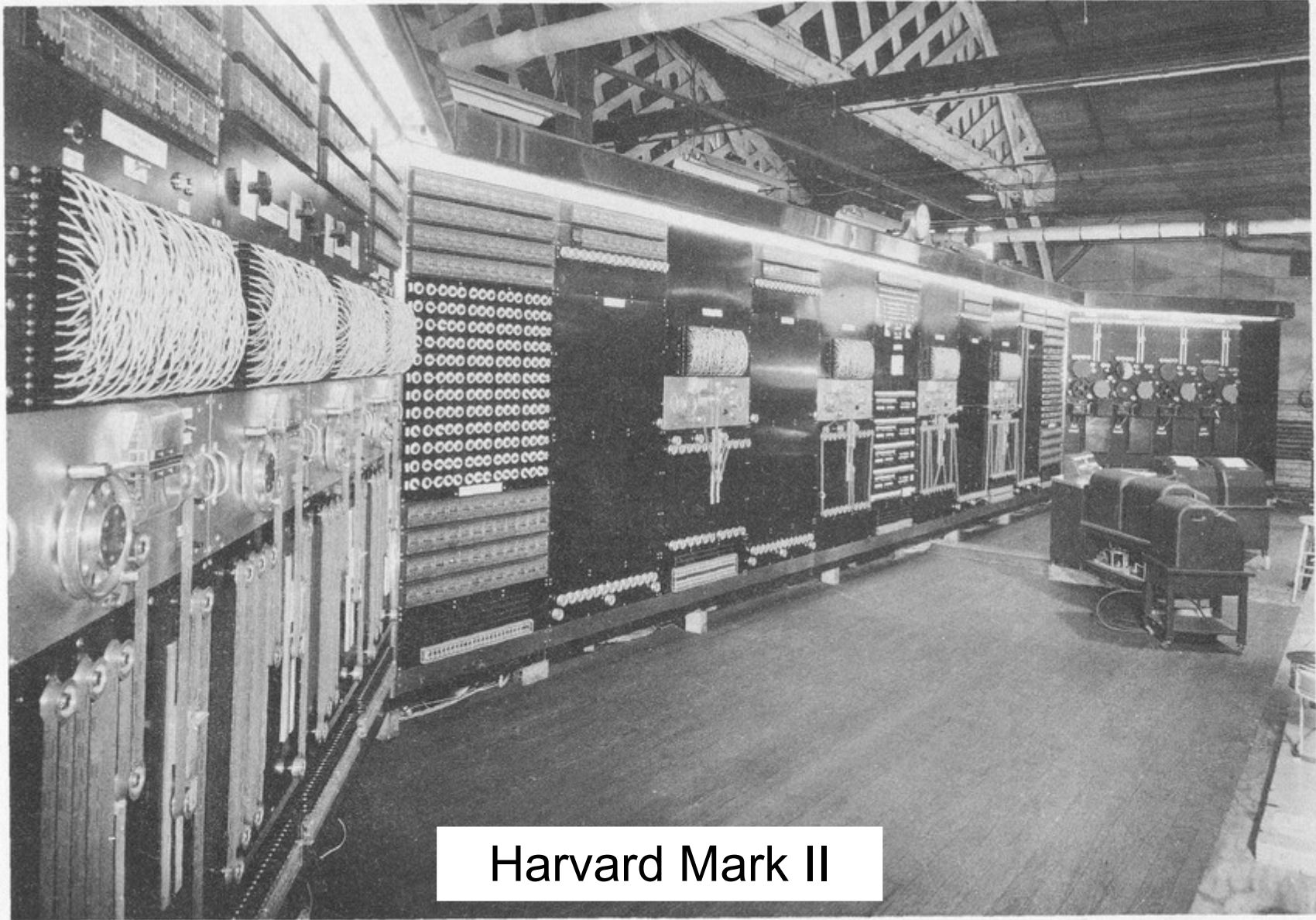
Assertzioni e invarianti

Moreno Marzolla
Dipartimento di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna
<https://www.moreno.marzolla.name/>

Copyright © 2018–2023 Moreno Marzolla, Università di Bologna, Italy
<https://www.moreno.marzolla.name/teaching/LabASD/>



This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0) License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/4.0/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.



Harvard Mark II

Plate I Main Control Board and Wings

<http://www.cbi.umn.edu/hostedpublications/Tomash/Images%20web%20site/Image%20files/H%20Images/pages/Harvard.Vol%2024.1949.view%20of%20Mark%20II.htm>

Il primo "bug" software

9 settembre 1947

9/9

0800 Antan started
1000 " stopped - antan ✓

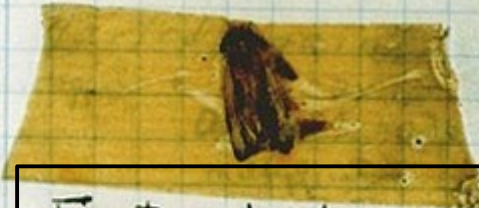
13⁰⁰ (032) MP - MC $\left\{ \begin{array}{l} 1.2700 \quad 9.037847025 \\ 1.982647000 \quad 9.037846995 \text{ correct} \\ 2.130476415 \quad 4.615925059(-2) \end{array} \right.$

(033) PRO 2 2.130476415
correct 2.130676415

Relays 6-2 in 033 failed special speed test
in relay " 10.000 test.

Relays changed

1100 Started Cosine Tape (Sine check)
1525 Started Multi-Adder Test.

1545  Relay #70 Panel F
(moth) in relay.

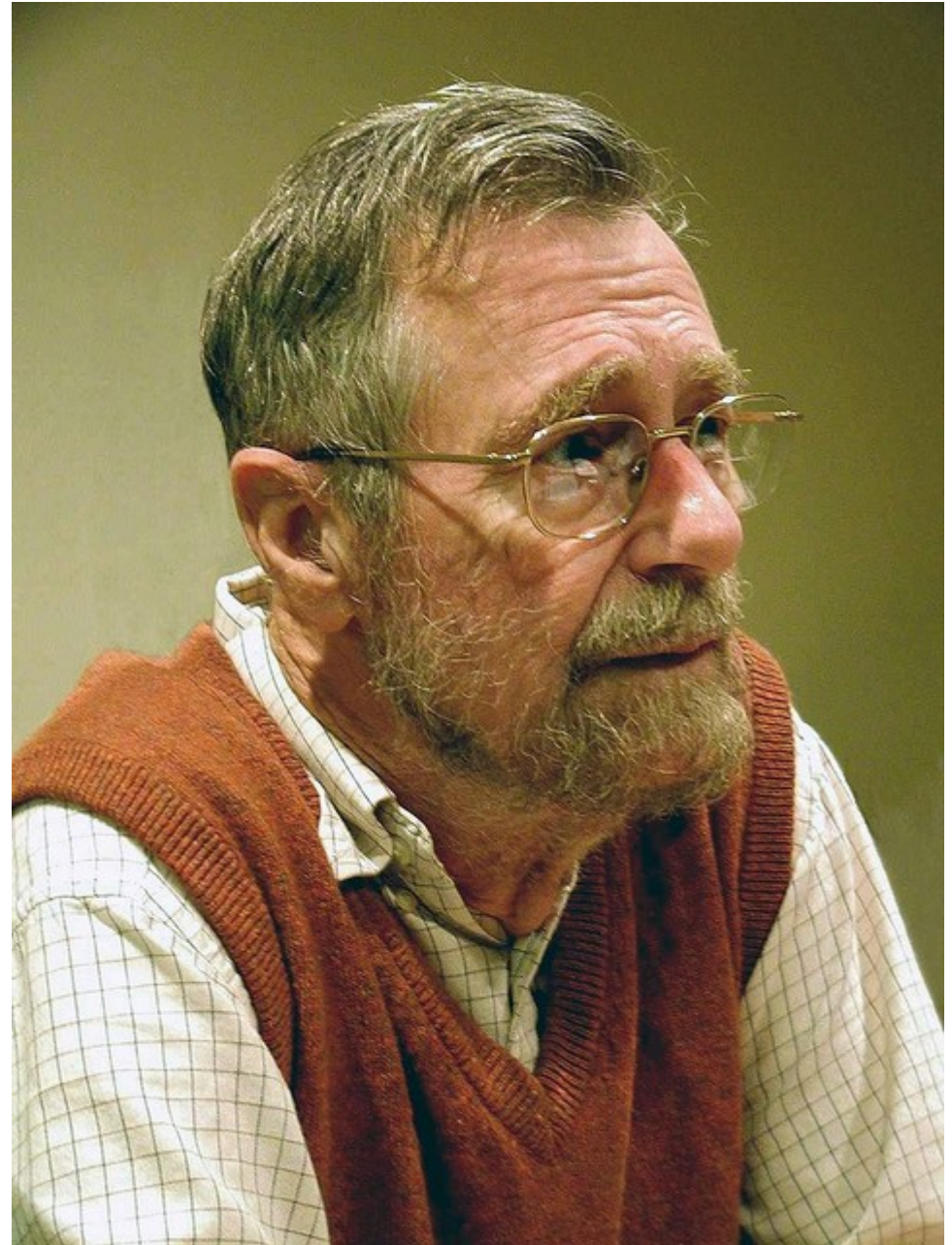
1700 First actual case of bug being found.
"First actual case of bug being found"

Relay 2145
Relay 3370

Come verificare che un programma sia corretto?

- Mediante **test (collaudo)**
 - Si preparano dei casi di test
 - Per ognuno di essi, si verifica che il programma fornisca il risultato atteso
- Mediante **dimostrazioni di correttezza**
 - Si dimostra formalmente (matematicamente) che il programma calcola il risultato corretto

“Il testing verifica la presenza di errori, non la loro assenza”



Edsger W. Dijkstra (1930–2002)

Dimostrazioni di correttezza

- Dimostrare che un dato frammento di codice calcola il risultato atteso
- Come? Specificando dopo ogni istruzione quali proprietà valgono in quel punto (**asserzioni**)
 - cioè di quali proprietà godono i valori delle variabili in quel punto
- Una asserzione è una **espressione logica** che deve essere **sempre vera** in quel punto
 - si possono usare gli operatori logici *and*, *or*, *not* ...

Esempio

- Due variabili x , y di tipo numerico
 - Es., int, float, double...
- Supponiamo che all'inizio si abbia $x = x_0$,
 $y = y_0$
- Cosa possiamo dire dei valori di x e y alla fine?

```
/* x = x0 ^ y = y0 */  
x = x - y;  
  
y = x + y;  
  
x = y - x;
```


Esempio

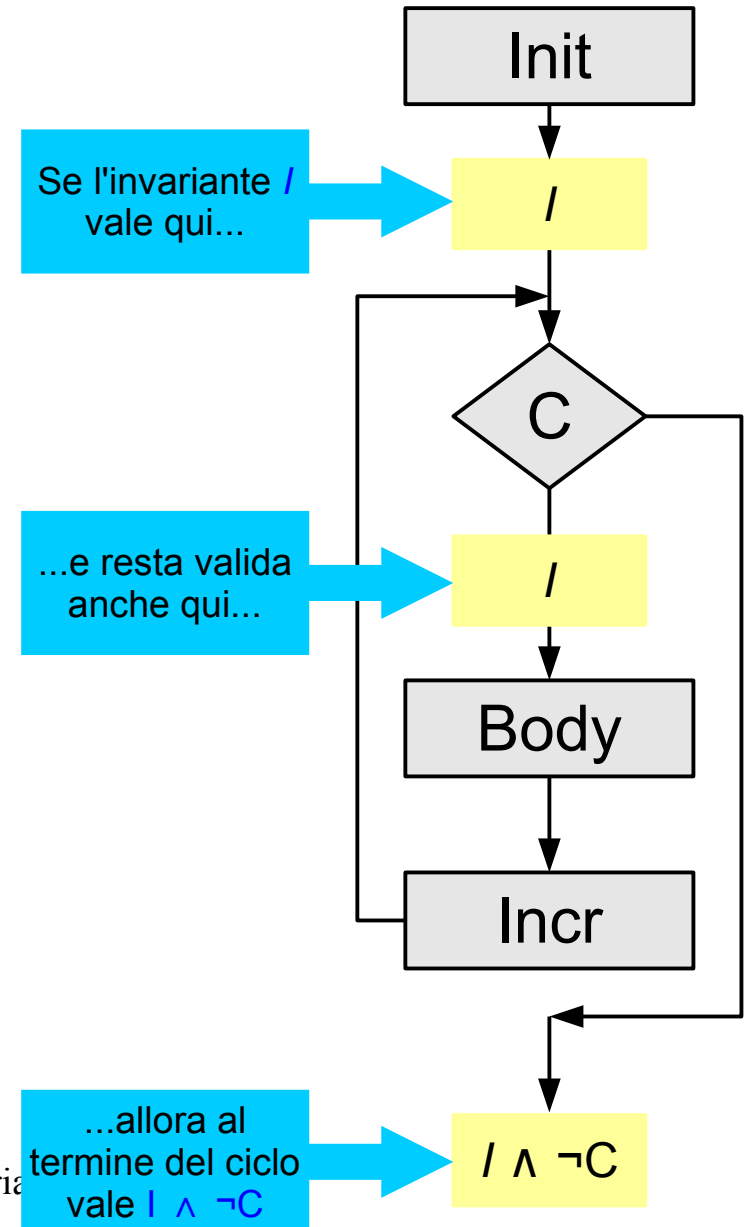
- Due variabili x , y di tipo numerico
 - Es., int, float, double...
- Supponiamo che all'inizio si abbia $x = x_0$,
 $y = y_0$
- Cosa possiamo dire dei valori di x e y alla fine?

```
/* x = x0 ∧ y = y0 */  
x = x - y;  
/* x = x0 - y0 ∧ y = y0 */  
y = x + y;  
/* x = x0 - y0 ∧ y = x0 */  
x = y - x;  
/* x = y0 ∧ y = x0 */
```

Invariante di ciclo

- Serve a dimostrare la correttezza di un ciclo
- Proprietà dell'invariante I
 - Deve valere **prima** del ciclo
 - Se è vera all'inizio di una iterazione, deve **rimanere vera** all'inizio dell'iterazione successiva
 - Al termine del ciclo fornisce informazioni utili su ciò che il ciclo ha calcolato

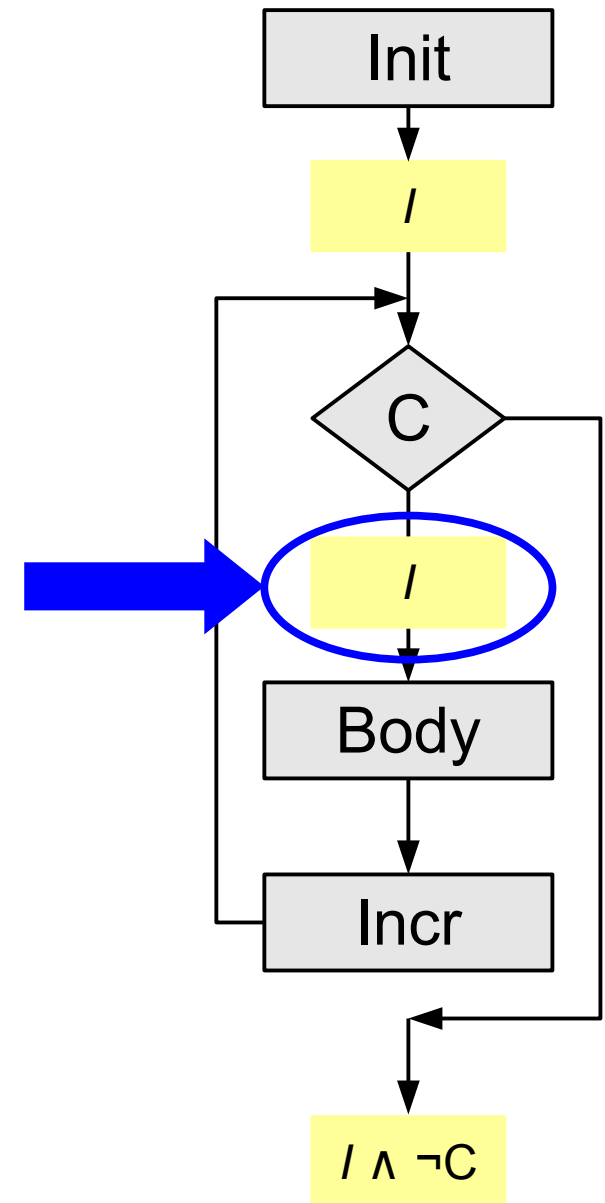
```
for (init; C; incr) {  
    Body  
}
```



Invariante di ciclo

- Possono esistere più invarianti
- Ma di solito solo una funziona per dimostrare la correttezza
- Per trovarla suggerisco di chiedersi: *che proprietà deve valere all'inizio di ogni iterazione?*
 - Attenzione ai casi particolari: per essere una invariante, la proprietà deve valere anche se il ciclo non viene mai eseguito

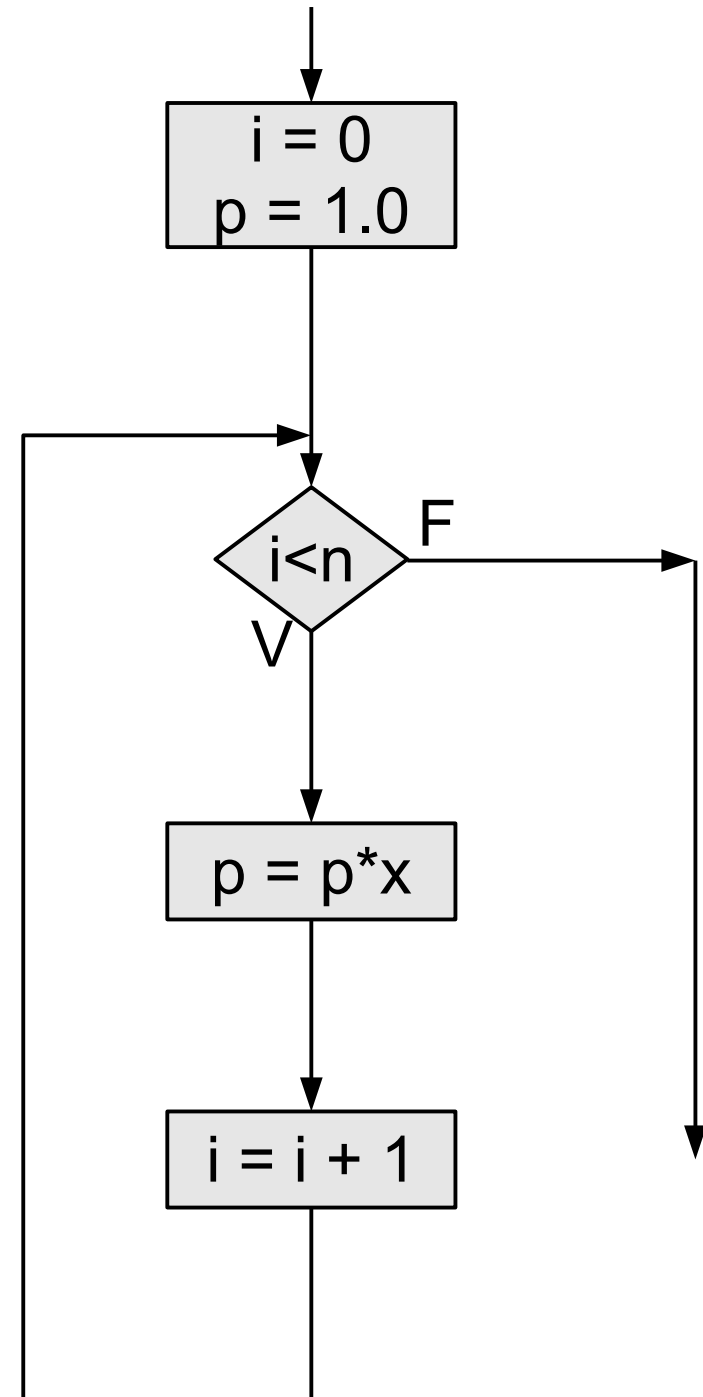
```
for (init; C; incr) {  
    Body  
}
```



Es.: elevamento a potenza

- Dato x e un intero $n \geq 0$, calcolare x^n

```
/* preconditione: n≥0 */  
int n = ...;  
int i;  
double x = ... , p = 1.0;  
for (i=0; i < n; i++) {  
    p = p * x;  
}
```



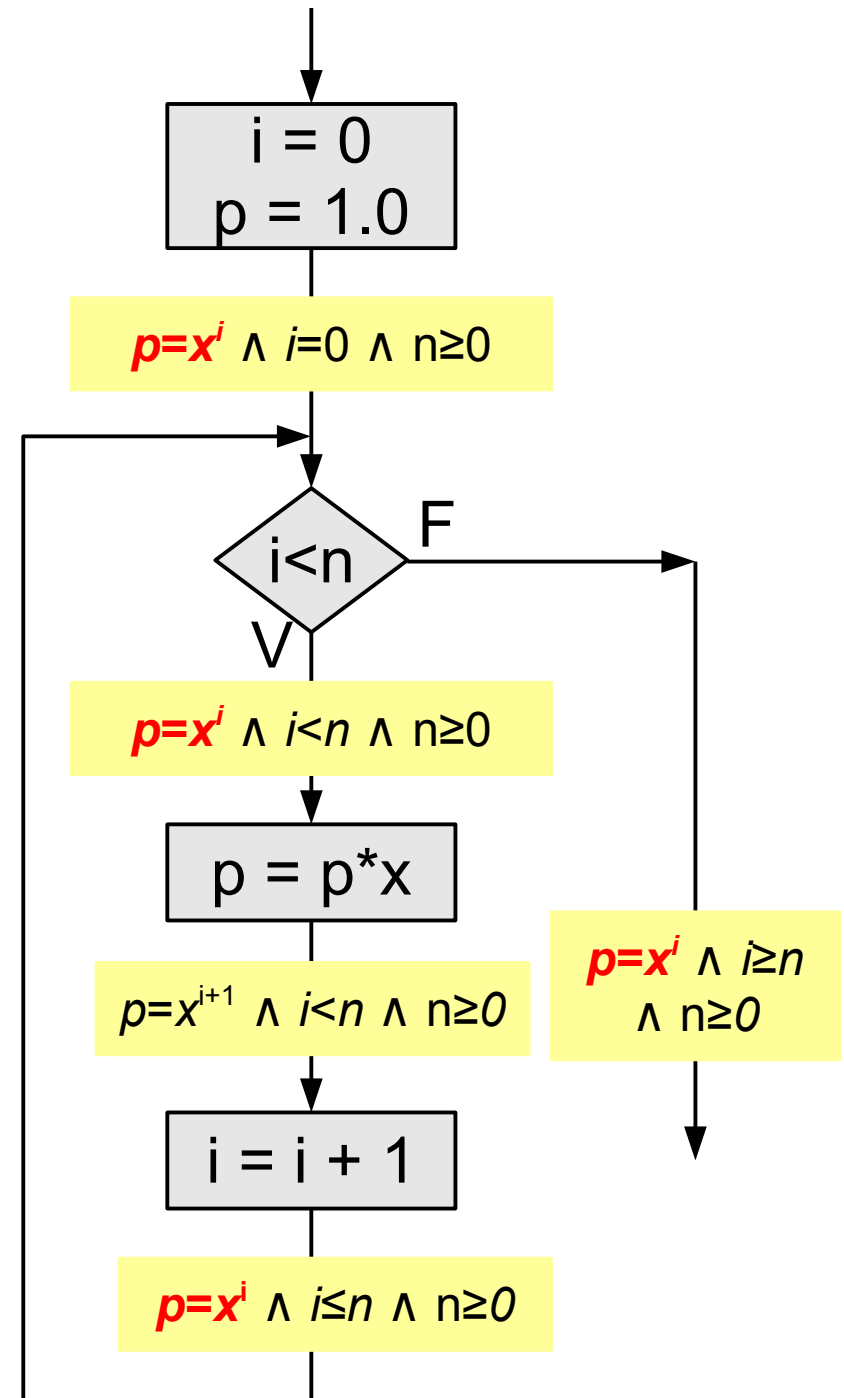
Es.: elevamento a potenza

- Dato x e un intero $n \geq 0$, calcolare x^n

```
/* preconditione: n≥0 */  
int n = ...;  
int i;  
double x = ... , p = 1.0;  
for (i=0; i < n; i++) {  
    p = p * x;  
}
```

Invariante: $(p = x^i)$

Questa invariante è **troppo debole** perché non permette di dimostrare $p = x^n$

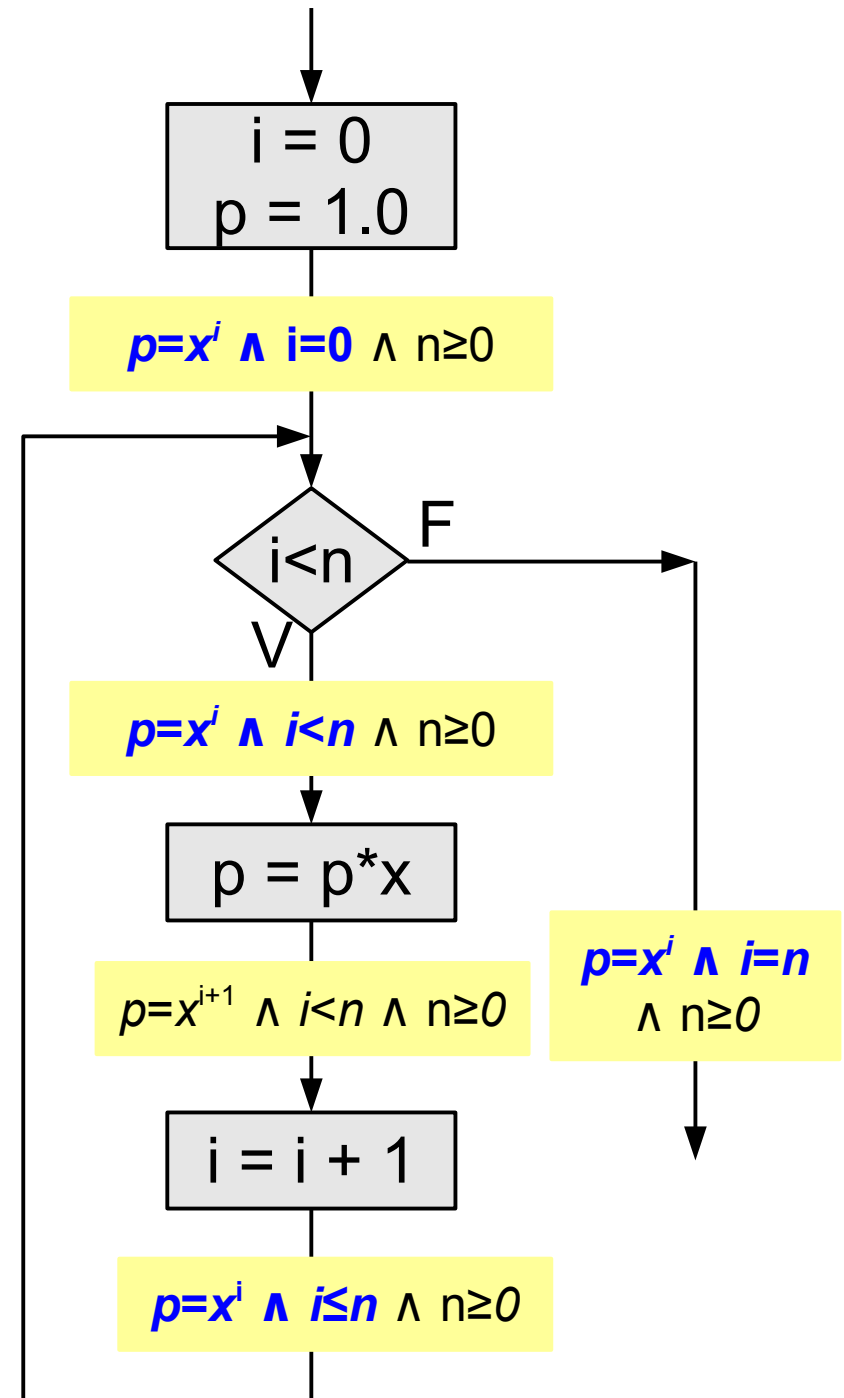


Es.: elevamento a potenza

- Dato x e un intero $n \geq 0$, calcolare x^n

```
/* preconditione: n≥0 */  
int n = ...;  
int i;  
double x = ... , p = 1.0;  
for (i=0; i < n; i++) {  
    p = p * x;  
}
```

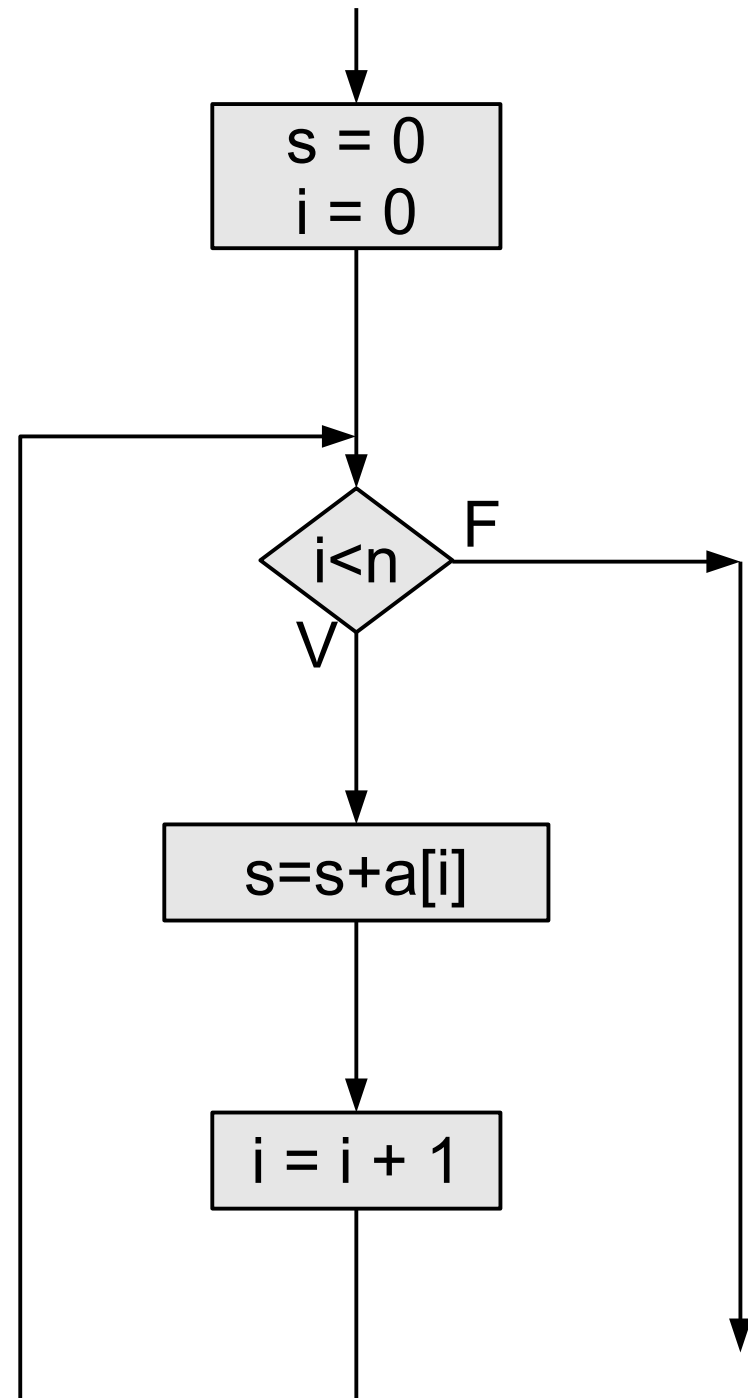
Invariante: $(p = x^i \wedge i \leq n)$



Es.: somma di un array

- Calcolare la somma degli elementi di un array
 - L'array vuoto ha somma 0

```
/* precondition: n ≥ 0 */  
int i, s=0;  
for (i=0; i<n; i++) {  
    s = s + a[i];  
}
```

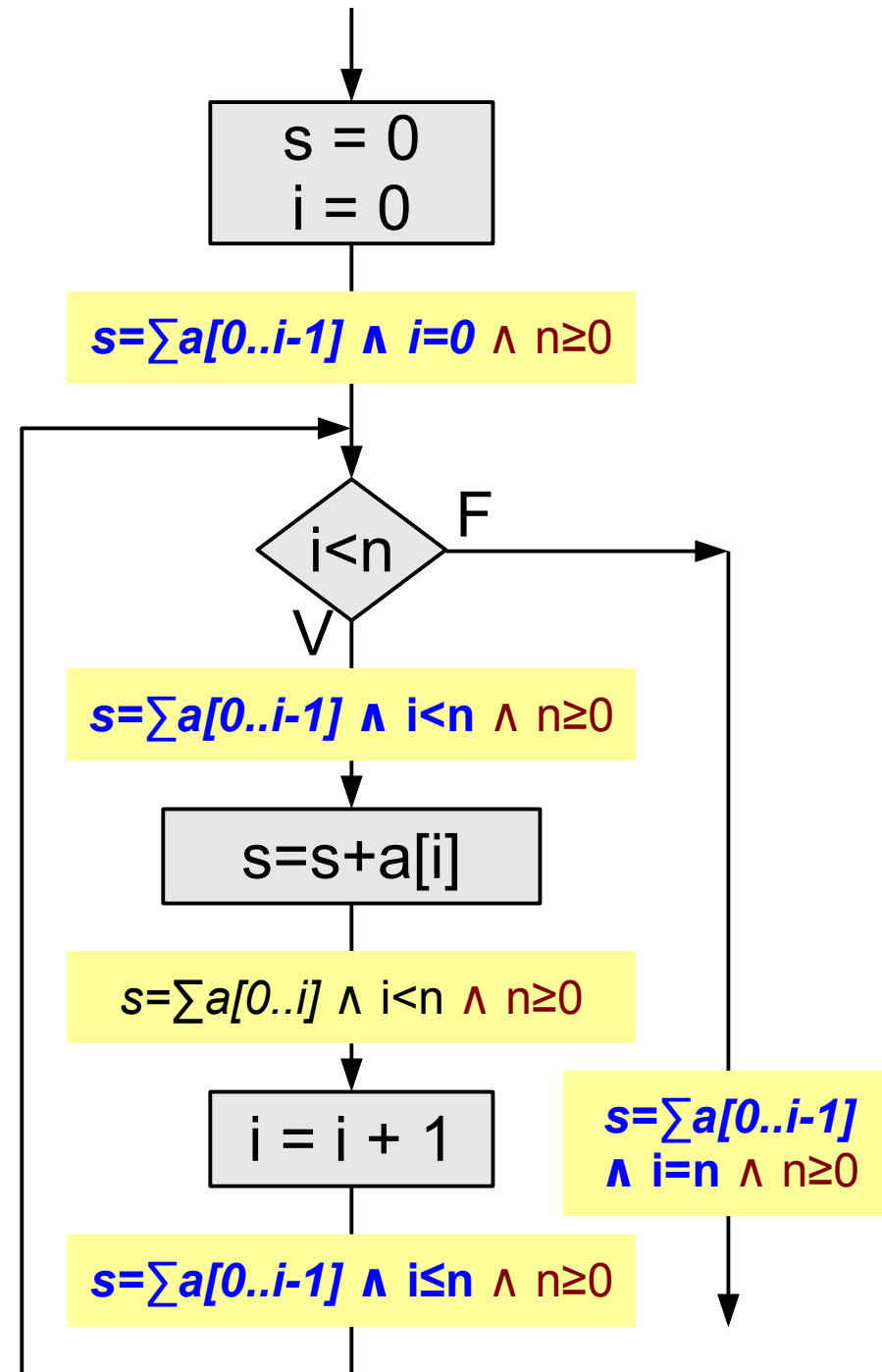


Es.: somma di un array

- Calcolare la somma degli elementi di un array
 - L'array vuoto ha somma 0

```
/* preconditione: n ≥ 0 */  
int i, s=0;  
for (i=0; i<n; i++) {  
    s = s + a[i];  
}
```

Invariante: $(s = \sum a[0..i-1] \wedge i \leq n)$



Osservazioni conclusive

- L'uso di asserzioni e invarianti è uno strumento potente per dimostrare la correttezza dei programmi
- Ha però degli svantaggi:
 - Non può essere applicato sempre (certe proprietà sono indimostrabili)
 - Anche su programmi semplici, le dimostrazioni possono diventare molto laboriose
- Come alternativa si ricorre al testing, che a sua volta ha delle limitazioni:
 - Tranne in casi banali, è impossibile definire un insieme di test esaustivi
 - Quindi **non dimostra l'assenza di errori, ma solo la loro presenza**