Correttezza (e terminazione)

March 3, 2021

Obiettivi: introdurre l'analisi qualitativa degli algoritmi, basata su tecniche di verifica come induzione e invarianti
Argomenti: i bug, correttezza (totale e parziale), verifica, pre- e postcondizioni, induzione ("semplice" e completa), verifica di algoritmi ricorsivi con induzione, invarianti di ciclo, verifica di algoritmi con invarianti di ciclo, terminazione.

1 Bug, correttezza, verifica, pre- e postcondizioni

I bug, termine inventato da Edison, sono inevitabili: "It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise — this thing gives out and [it is] then that "Bugs" — as such little faults and difficulties are called — show themselves and months of intense watching, study and labor are requisite before commercial success or failure is certainly reached." (T. Edison, da Wikipedia)

Le loro conseguenze possono essere devastanti:
1962: la sonda Mariner 1 si schianta su Venere per un errore nel software di controllo del volo
1981: una TV nel Quebec decreta la vittoria alle elezioni di un partito sconosciuto; usavano un software difettoso per la rilevazione dei risultati
1983: un sistema computerizzato sovietico rileva un attacco nucleare con cinque missili balistici inesistenti
1985: alcuni pazienti vengono irradiati con dosi eccessive di raggi X dal sistema Therac-25 a controllo software
1993: il processore Pentium, che incorpora il coprocessore aritmetico, sbaglia le divisioni in virgola mobile
1996: l'Ariane 5 esce dalla sua rotta e viene distrutto in volo a causa di un errore di conversione dei dati da 16 a 32 bit
1999: usando un nuovo sistema, le poste inglesi recapitano mezzo milione di nuovi passaporti ad indirizzi sbagliati o inesistenti
1999: Y2K è il celeberrimo millenium-bug
(continua...)

Bisogna **verificare la correttezza degli algoritmi (e i programmi** che li implementano).

Un algoritmo è **corretto** se per ogni input **fornisce l'output corretto** (correttezza totale).

Un algoritmo è **parzialmente corretto** se per ogni input **se termina fornisce l'output corretto**.

Un "verificatore ideale", dato un problema P a un algoritmo A, dovrebbe essere in grado di rispondere: "**SI**, A è corretto per P" oppure "**NO**, A non è corretto per P e gli errori sono...".

Tale verificatore non esiste ma ci sono tanti progetti di ricerca che puntano a sviluppare strumenti per aiutare la verifica di algoritmi e programmi.

Verificare un algoritmo vuole dire controllare se esso soddisfa le specifiche del problema computazionale che deve risolvere. Le specifiche sono descritte tramite le **precondizioni** e le **postcondizioni**.

Per esempio, per il problema della divisione intera:
Precondizioni: $a \geq 0, b > 0$ numeri interi;
Postcondizioni: il risultato è un intero q tale che $a = b \cdot q + r$ con $0 \leq r < b$.

2 Induzione "semplice" e completa

Schema dell'**induzione "semplice"**:
Sia $P(n)$ una enunciato che dipende da una variabile intera $n \in \mathbb{N}$.
Se $P(n)$ vale per $n = 0$ e per ogni $n \in \mathbb{N}$ vale $P(n) \implies P(n+1)$ allora $P(n)$ vale per ogni $n \in \mathbb{N}$.

Cioè

$$(P(0) \wedge (\forall n \in \mathbb{N}.P(n) \implies P(n+1))) \implies \forall n \in \mathbb{N}.P(n)$$

dove $P(0)$ è il **caso base** e $(\forall n \in \mathbb{N}.P(n) \implies P(n+1))$ è il **passo induttivo**. Inoltre, $P(n)$ viene chiamata l'**ipotesi induttiva**.

In parole: una proprietà $P(n)$ che vale per $n = 0$ (caso base) e se vale per n allora vale per $n + 1$ (passo induttivo) vale per ogni $n \geq 0$.

Il caso base può essere anche $P(k)$:

$$(P(k) \wedge (\forall n \geq k.P(n) \implies P(n+1))) \implies \forall n \geq k.P(n)$$

quindi la proprietà vale da k in poi.

Il passo induttivo può essere anche nella forma $P(n-1) \implies P(n)$.

Esempio: dimostriamo con induzione che

$$\sum_{k=1}^{n}(2k-1) = n^2 \tag{1}$$

Caso base: la proprietà con $n = 1$ vale

$$\sum_{k=1}^{1}(2k-1) = 1 = 1^2$$

Passo induttivo: supposta che la proprietà in (1) è vera per n (questa è l'ipotesi induttiva), verifichiamo per $n + 1$

$$\sum_{k=1}^{n+1}(2k-1) = \left(\sum_{k=1}^{n}(2k-1)\right) + (2(n+1)-1) = n^2 + (2n+1) = (n+1)^2$$

dove abbiamo sostituito $(\sum_{k=1}^{n}(2k-1))$ con n^2 grazie all'ipotesi induttiva. Visto che la proprietà in (1) vale per $n = 1$ (caso base) e se vale per n allora vale per $n+1$ (passo induttivo), la proprietà vale per qualunque $n \geq 1$.

Schema dell'**induzione completa**:
Sia $P(n)$ una proprietà che dipende da una variabile intera $n \in \mathbb{N}$.
Se $P(n)$ vale per $n = 0, 1, ..., k$ e per ogni $n \geq k$ vale $(P(0) \wedge P(1) \wedge ... \wedge P(n)) \implies P(n+1)$ allora $P(n)$ vale per ogni $n \in \mathbb{N}$.

Cioè

$$(P(0) \wedge P(1) \wedge ... \wedge P(k) \ \wedge \ (\forall n \geq k.(P(0) \wedge P(1) \wedge ... \wedge P(n)) \implies P(n+1))) \implies \forall n \in \mathbb{N}.P(n)$$

dove $P(0) \wedge P(1) \wedge ... \wedge P(k)$ è il **caso base** e $(\forall n \geq k.(P(0) \wedge P(1) \wedge ... \wedge P(n)) \implies P(n+1))$ è il **passo induttivo**. Inoltre, $P(0) \wedge P(1) \wedge ... \wedge P(k)$ viene chiamata l'**ipotesi induttiva**.

In parole: una proprietà $P(n)$ che vale per $n = 0, 1, ..., k$ (caso base) e se vale per $0, 1, ..., n$ allora vale per $n + 1$ (passo induttivo) vale per ogni $n \geq 0$.

Dimostriamo il seguente teorema con induzione completa.

Teorema: Ogni numero naturale $n \geq 2$ è un numero primo oppure è esprimibile come prodotto di numeri primi.

Dimostrazione: Procediamo con induzione su n.
Caso base. L'asserto è vero per $n = 2$ perché 2 è un numero primo.
Passo induttivo. Dobbiamo dimostrare che se l'asserto è vero per ogni $n' \in \{2, 3, ..., n\}$ allora è vero per $n + 1$:

- se $n + 1$ è primo allora l'asserto è banalmente vero;
- se $n + 1$ non è primo allora si può esprimere $n + 1$ come il prodotto di due numeri più piccoli n_1, n_2, cioè $n + 1 = n_1 \cdot n_2$ con $1 < n_1 < n$ e $1 < n_2 < n$;
  per ipotesi induttiva, sia n_1 sia n_2 è un numero primo oppure è esprimibile come prodotto di numeri primi;
  segue che anche $n + 1 = n_1 \cdot n_2$ è esprimibile come prodotto di numeri primi.

Per il principio dell'induzione completa, l'asserto è vera per ogni $n \geq 2$. □

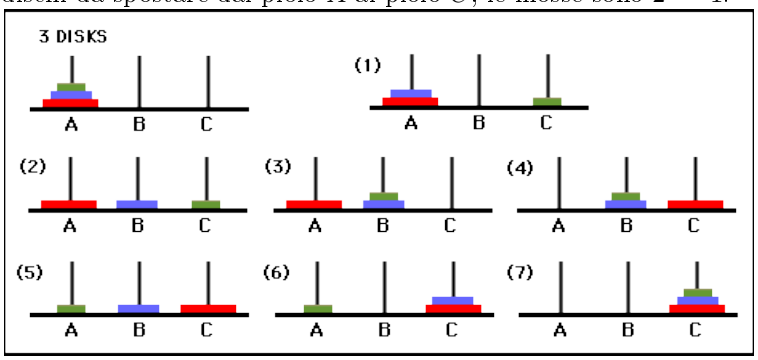
3 Dimostrazione di correttezza di algoritmi ricorsivi con induzione

Il problema delle torre di Hanoi. Dati tre pioli

- il piolo *sorgente* con sopra n dischi di diametro crescente,
- il piolo *d'appoggio* senza dischi,
- e il piolo *destinazione* senza dischi,

il compito è spostare la torre dal piolo sorgente al piolo destinazione, sfruttando il piolo d'appoggio, muovendo un disco alla volta, senza mai sovrapporre un disco più grande ad uno più piccolo.

Per esempio, con tre dischi da spostare dal piolo A al piolo C, le mosse sono $2^3 - 1$:



Sviluppiamo un algoritmo ricorsivo per risolvere il problema in presenza di n dischi.

Caso base. Quando il numero di dischi da spostare è 1, il problema è triviale.

Ricorsione. Supponendo che siamo in grado di risolvere il problema correttamente con $n - 1$ dischi, si può risolvere il problema con n dischi in tre passaggi:
1. sposta $n - 1$ dischi dal piolo sorgente al piolo d'appoggio;
2. sposta un disco dal piolo sorgente al piolo destinazione;
3. sposta $n - 1$ dischi dal piolo d'appoggio al piolo destinazione.

In pseudo-codice:

```
1: MoveTower(n, A, B, C)      ▷ n dischi da spostare dal piolo A al piolo C utilizzando
                                il piolo B come appoggio;
                                Precondizione: A non è vuota e la base di A ha
                                un diametro più piccolo del disco più in alto sia di
                                B che di C;
                                Postcondizione: n dischi sono spostati dal piolo A
                                al piolo C;
2: if n = 1 then
3:     move 1 disk from A to C
```

4: **else**
5: MoveTower$(n-1, A, C, B)$
6: move 1 disk from A to C
7: MoveTower$(n-1, B, A, C)$
8: **end if**

Simulazione dell'algoritmo con 3 dischi da spostare dal piolo 1 al piolo 3 utilizzando il piolo 2 come appoggio. Dopo ogni spostamento viene indicato lo stato del sistema.

```
call_hanoi(n:3,source:1,spare:2,dest:3)
  call_hanoi(n:2,source:1,spare:3,dest:2)
    call_hanoi(n:1,source:1,spare:2,dest:3)
      move  1 -> 3:    2,0,1
    end_hanoi(n:1,source:1,spare:2,dest:3)
    move  1 -> 2:    1,1,1
    call_hanoi(n:1,source:3,spare:1,dest:2)
      move  3 -> 2:    1,2,0
    end_hanoi(n:1,source:3,spare:1,dest:2)
  end_hanoi(n:2,source:1,spare:3,dest:2)
  move  1 -> 3:    0,2,1
  call_hanoi(n:2,source:2,spare:1,dest:3)
    call_hanoi(n:1,source:2,spare:3,dest:1)
      move  2 -> 1:    1,1,1
    end_hanoi(n:1,source:2,spare:3,dest:1)
    move  2 -> 3:    1,0,2
    call_hanoi(n:1,source:1,spare:2,dest:3)
      move  1 -> 3:    0,0,3
    end_hanoi(n:1,source:1,spare:2,dest:3)
  end_hanoi(n:2,source:2,spare:1,dest:3)
end_hanoi(n:3,source:1,spare:2,dest:3)
```

Teorema: L'algoritmo precedente è corretto.
Dimostrazione: Procediamo con induzione su n.
Caso base. Con $n = 1$ è triviale che l'algoritmo sia corretto.
Passo induttivo. Dobbiamo dimostrare che se l'algoritmo è corretto con $n-1$ dischi (ipotesi induttiva) allora è corretto con n dischi. Con n dischi:

- all'inizio i numeri di dischi sui pioli sono: $n, 0, 0$
- MoveTower$(n-1, A, C, B)$ sposta, grazie all'ipotesi induttiva, correttamente $n-1$ dischi dal piolo A al piolo B; dopodiché abbiamo $1, n-1, 0$ dischi sui pioli
- dopo "move 1 disk from A to C" la situazione è $0, n-1, 1$
- MoveTower$(n-1, B, A, C)$ sposta, grazie all'ipotesi induttiva, correttamente $n-1$ dischi dal piolo B al piolo C; dopodiché abbiamo $0, 0, n$ dischi sui pioli.

Quindi se l'algoritmo è corretto con $n-1$ dischi allora è corretto con n dischi.
Per il principio di induzione, l'algoritmo è corretto per ogni $n \geq 1$. □

Il problema della divisione intera.
Precondizioni: $a \geq 0, b > 0$ numeri interi;
Postcondizioni: i risultati sono due interi q, r tali che $a = b \cdot q + r$ con $0 \leq r < b$ (dove q è il quoziente e r è il resto).

Secondo le postcondizioni dobbiamo avere $a = b \cdot q + r$ con $0 \leq r < b$, cioè

$$a - q \cdot b = a - \underbrace{b - b - ... - b}_{q} = r < b$$

Se $a < b$ allora la soluzione è $q = 0$ con $r = a$. Se $a \geq b$ allora

$$(a - b) - \underbrace{b - b - ... - b}_{q'} = r < b$$

e quindi q' e r sono quoziente e resto di $a - b$ diviso b e $q = q' + 1$.

Visto che $a - b < a$, il precedente ragionamento indica un procedimento ricorsivo per risolvere il problema:

```
1: DIV-REC(a, b)        ▷ Pre: a ≥ 0, b > 0
                          Post: ritorna q, r tali che a = b q + r ∧ 0 ≤ r < b
2: if a < b then
3:     q, r ← 0, a
4: else
5:     q', r ← DIV-REC(a − b, b)
6:     q ← q' + 1
7: end if
8: return q, r
```

Teorema: Il precedente algoritmo è corretto.
Dimostrazione: Fissato $b > 0$, procediamo con induzione completa su a.
Casi base. Se $a < b$, cioè per ogni $a \in \{0, 1, ..., b - 1\}$ l'algoritmo restituisce $q = 0, r = a$ che è chiaramente corretto.
Passo induttivo. Dobbiamo dimostrare che, in caso di $a \geq b$, se l'algoritmo è corretto per ogni $a' < a$ allora l'algoritmo è corretto per a:

- se con $a' = a - b, b$ l'algoritmo restituisce q', r' allora con a, b restituisce $q = q' + 1, r = r'$
- secondo l'ipotesi induttiva la risposta con a', b è corretta e quindi

  $$a' = a - b = b \cdot q' + r' \wedge 0 \leq r' < b$$

  da dove segue che

  $$a = b \cdot q' + b + r' = b(q' + 1) + r' \wedge 0 \leq r' < b$$

  e visto che $q = q' + 1, r = r'$ abbiamo

  $$a = b \cdot q + r \wedge 0 \leq r < b$$

  e quindi il passo induttivo è dimostrato.

Per il principio di induzione completa, l'algoritmo è corretto per qualunque $a \geq 0, b > 0$. □

4 Dimostrazione di correttezza di algoritmi iterativi con invarianti

L'**invariante di ciclo** è una proposizione sul valore delle variabili "intorno" ad un ciclo con:

- *inizializzazione:* la proposizione vale immediatamente prima di entrare nel ciclo; e
- *mantenimento:* se la proposizione vale prima di eseguire il corpo del ciclo, allora vale anche dopo aver eseguito il corpo del ciclo.

L'invariante di ciclo è utile se, al termine del ciclo, aiuta a dimostrare la correttezza dell'algoritmo.

Esempio. Algoritmo iterativo per calcolare l'ennesimo numero della seria di Fibonacci ($F_1 = F_2 = 1, F_n = F_{n-2} + F_{n-1}, \forall n \geq 2$).

```
1: FIB-ITE(n)           ▷ Pre: n ≥ 1
                          Post: ritorna F_n
2: a, b, i ← 1, 1, 3
3: while i ≤ n do
4:     c ← a + b
5:     a ← b
```

6: $b \leftarrow c$
7: $i \leftarrow i + 1$
8: **end while**
9: **return** b

Teorema: l'invariante, che vale ogni volta prima di eseguire la riga 3 e ogni volta dopo aver eseguito la riga 7, è

$$b = F_{i-1} \wedge a = F_{i-2}$$

L'invariante garantisce la correttezza dell'algoritmo.
Dimostrazione: È immediato che l'invariante vale prima di entrare nel ciclo: $a = 1, b = 1, i = 3$ e quindi $b = F_{i-1} = F_2 \wedge a = F_{i-2} = F_1$.
Il ciclo mantiene l'invariante. Assumiamo che prima di eseguire il corpo del ciclo abbiamo $b = F_{i-1} \wedge a = F_{i-2}$. Alla variabile c viene assegnato $a + b = F_{i-2} + F_{i-1} = F_i$, alla variabile a viene assegnate $b = F_{i-1}$ e alla variabile b il valore $c = F_i$. Visto che i viene incrementato, dopo l'esecuzione del corpo del ciclo abbiamo di nuovo $b = F_{i-1} \wedge a = F_{i-2}$.
All'uscita del ciclo si ha $i = n + 1$, e quindi all'uscita $b = F_{i-1} \wedge a = F_{i-2}$ implica

$$b = F_{(n+1)-1} = F_n$$

e dunque l'algoritmo è corretto. □

Esempio. Algoritmo iterativo per calcolare il valore di un polinomio rappresentato dai suoi coefficienti $a_0, a_1, ..., a_n$ in un punto x (utilizzando la regola di Horner):

1: HORNER$(a_0, a_1, ..., a_n, x)$
2: $y \leftarrow 0$
3: $i \leftarrow n$
4: **while** $i \geq 0$ **do**
5: $y \leftarrow a_i + x \cdot y$
6: $i \leftarrow i - 1$
7: **end while**
8: **return** y

Per individuare l'invariante del ciclo simuliamo l'algoritmo segnando il valore delle variabili subito dopo la riga 3 e subito dopo la riga 6. Si ottiene la seguente tabella:

i	y
n	0
$n-1$	a_n
$n-2$	$a_{n-1} + a_n x$
$n-3$	$a_{n-2} + a_{n-1}x + a_n x^2$
$n-4$	$a_{n-3} + a_{n-2}x + a_{n-1}x^2 + a_n x^3$
$\vdots$	$\vdots$

La relazione fra i e y in ogni riga della tabella precedente, cioè l'invariante del ciclo, è

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

Dimostriamo formalmente che l'equazione precedente è un invariante del ciclo dell'algoritmo e la usiamo per dimostrare la correttezza dell'algoritmo.

Teorema: l'invariante, che vale subito dopo la riga 3 e ogni volta dopo aver eseguito la riga 6, è

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k \tag{2}$$

L'invariante garantisce la correttezza dell'algoritmo.
Dimostrazione: Con $y = 0$ e $i = n$ l'invariante viene soddisfatto (la sommatoria è "vuota") e quindi si verifica prima di eseguire il ciclo. Per dimostrare che l'invariante viene mantenuto dal ciclo, bisogna dimostrare che *se l'invariante si verifica prima di eseguire il ciclo allora si verificherà anche dopo*, cioè si verificherà anche con i valori aggiornati delle variabili. Denotiamo con y' e i' i valori aggiornati. Per i e i' primo abbiamo

$$i' = i - 1, \quad i = i' + 1$$

Per quanto riguarda y' abbiamo

$$y' = a_i + x \cdot y$$

dove, avendo assunto che l'invariante vale per y e i, possiamo scrivere

$$y' = a_i + x \cdot \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

Portando x dentro la sommatoria si ottiene

$$y' = a_i + \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^{k+1}$$

e introducendo $k' = k + 1$ (e quindi $k = k' - 1$) abbiamo

$$y' = a_i + \sum_{k'=1}^{n-(i+1)+1} a_{k'-1+i+1} x^{k'} = a_i + \sum_{k'=1}^{n-i} a_{k'+i} x^{k'}$$

Ora è facile portare dentro la sommatoria anche a_i e otteniamo

$$y' = \sum_{k'=0}^{n-i} a_{k'+i} x^{k'}$$

Applicando $i = i' + 1$ abbiamo

$$y' = \sum_{k'=0}^{n-(i'+1)} a_{k'+i'+1} x^{k'}$$

L'ultima equazione dimostra che la relazione fra y' e i' è uguale a quella indicata da (2) e quindi l'invariante viene mantenuto dal ciclo.
All'uscita del ciclo si ha $i = -1$ e quindi

$$y = \sum_{k=0}^{n-(-1+1)} a_{k-1+1} x^k = \sum_{k=0}^{n} a_k x^k$$

cioè y contiene il valore del polinomio nel punto x e quindi l'algoritmo è corretto. □

5 Terminazione (accenni)

Il problema della terminazione in generale è difficile. Turing: Non esiste alcun algoritmo in grado di decidere se data una procedura ed un ingresso la procedura termina!

Invece è facile avere delle procedure che non terminano mai. Il comportamento può dipendere anche dall'input. Il seguente algoritmo termina con qualunque intero positivo ma non termina con numeri non interi.

```
1: SUM(n)
2: if n = 0 then
```

3: **return** 0
4: **else**
5: **return** n+SUM$(n-1)$
6: **end if**

(Cosa calcola l'algoritmo per $n \in \mathbb{N}$?)

Per costruire una funzione ricorsiva terminante conviene assicurarsi che qualcosa nel valore dei parametri e/o nella loro "dimensione" diminuisca.
Se la dimensione non può decrescere illimitatamente e tutti i casi minimali sono contemplati come "caso base", la funzione ricorsiva risulterà ovunque definita.

Per esempio:

1: STRANGESUM$(A[1..n])$
2: **if** $n = 1$ **then**
3: **return** $A[1]$
4: **end if**
5: **if** $n = 2$ **then**
6: **return** $A[1] + A[2]$
7: **end if**
8: **return** $A[n-1] + A[n]$+STRANGESUM$(A[1..n-2])$

La terminazione è garantita per qualunque $n \geq 1$ (cioè per qualunque vettore di lunghezza n): se n è pari si arriva al caso base della 5. riga; se n è dispari si arriva al caso base della 2. riga. (Cosa calcola l'algoritmo?)