Invarianti, principio di induzione e correttezza dei programmi

Consideriamo il principio di induzione per i numeri non negativi, detti anche **numeri naturali**.

Sia P una proprietà (espressa da una frase o una formula che contiene la variabile n che varia sui numeri naturali). Supponiamo che:

- (**Base dell'induzione**) P(k) sia vera quando n = k, e che
- (**Passo induttivo**) assumendo che P sia vera quando n prende un valore arbitrario i ≥ k (**ipotesi induttiva**), si riesca a dimostrare che P è vera anche quando n prende il valore i + 1.

Il **principio di induzione** afferma che con queste premesse P è vera per tutti i valori di n ≥ k. Una giustificazione (intuitiva) del principio di induzione si può avere dall'osservazione che per ogni numero naturale n ≥ k: n = k, altrimenti n > k ed n ha la forma k + 1 + 1 + ... + 1 (dove la somma + 1 viene eseguita un numero finito di volte). Spesso si abbrevia l'operazione x + 1 come s(x) (il successore di x). In generale allora si vede che un numero naturale è 0 oppure il successore di qualche numero naturale.

Esempio

Alcuni esempi di proprietà per le quali ci interesserà dimostrare per induzione che sono vere per tutti i numeri naturali.

(1) Consideriamo un frammento di codice della forma:

```
while (b)
        S;
```

La proprietà P è vera dopo n iterazioni del ciclo while: proprietà di questo tipo sono utilizzate per stabilire che P è una proprietà invariante del ciclo in questione.

(2) Se E(n) è un'espressione aritmetica che contiene la variabile n, l'equazione

$$f(n) = E(n)$$

stabilisce che la funzione f per l'argomento n ha lo stesso valore dell'espressione E(n). Se immaginiamo che la funzione f sia definita ricorsivamente, si può dimostrare per induzione che f(n) = E(n) per ogni valore naturale di n, stabilendo così la correttezza della definizione ricorsiva della funzione il cui valore per n è dato da E(n).

Lo sviluppo di un semplice programma e la dimostrazione della sua correttezza

Consideriamo il problema di calcolare il quoziente q ed il resto r della divisione di due numeri interi $X \geq 0$ e $D > 0$. L'algoritmo usuale consiste nel sottrarre ripetutamente D a X, aumentando ogni volta di 1 il valore di q che inizialmente ha valore 0.

Schematicamente, l'algoritmo è il seguente:

1. fino a quando $X \geq D$ esegui le seguenti azioni: sottrai D a X; aumenta q di 1
2. quando $X < D$, poni $r = X$

Un programma Java che realizza questo algoritmo è il seguente:

```
class divisione {
public static void main (String[] args) {
        int X, D, q, r;
        X = 14;
        D = 3;
        q = 0;
        r = X;
        while (r >= D) {
                r = r - D;
                q = q + 1;
                }
        System.out.println ("Il quoziente è: " + q);
        System.out.println ("Il resto è: " + r);
        }
}
```

Come si può dimostrare che il programma precedente è corretto? Prima di tutto, serve una specifica precisa del problema da risolvere:

La **condizione di ingresso** del programma, cioè la proprietà che i dati in ingresso X e D devono soddisfare, è che

$X \geq 0$ e $D > 0$

(la seconda proprietà serve ad evitare casi di divisione per 0).

La **condizione di uscita** del programma, cioè la proprietà che i dati in uscita q ed r devono soddisfare, è che

$X = q * D + r$, con $r < D$.

Questa proprietà dice proprio che q ed r sono, rispettivamente, il quoziente ed il resto della divisione intera di X per D.

La **correttezza** del programma (qualche volta si parla di questa condizione come di **correttezza parziale**) asserisce che:

> *per ogni dato in ingresso che soddisfa la condizione di ingresso, se il programma termina, allora i dati in uscita soddisfano la condizione di uscita.*

Una condizione più esigente di correttezza è quella che si chiama **correttezza totale**:

> *per ogni dato in ingresso che soddisfa la condizione di ingresso, il programma termina e i dati in uscita soddisfano la condizione di uscita.*

Per stabilire che un programma soddisfa la specifica vi sono vari modi, ma la tecnica più conveniente consiste nel trovare quello che si chiama un invariante (di ciclo):

> ***invariante** (di un ciclo) è una proprietà che lega (tutte o alcune del)le variabili coinvolte nel ciclo, e che è vera dopo un numero arbitrario di iterazioni del ciclo. In particolare, è vera all'ingresso nel ciclo (cioè dopo 0 iterazioni).*

Ci sono molte proprietà invarianti del ciclo

```
  while (r >= D) {
        r = r - D;
        q = q + 1;
}
```

nel programma precedente, per esempio la proprietà $q \geq 0$, ma tra tutte le possibili proprietà ce ne sono alcune che sono più interessanti di altre. Consideriamo ora la proprietà:

(*) $X = q * D + r$

che è molto simile alla condizione di uscita del programma. Che si tratti veramente di un invariante è qualcosa che deve ancora essere dimostrato, ma per il momento assumiamo che lo sia.

Quando il ciclo termina (e prima o poi deve terminare, perché ad ogni iterazione a r viene sottratto il valore D che, per la condizione di ingresso, è un numero > 0, quindi prima o poi deve accadere che $r < D$) abbiamo che $X = q * D + r$ perché abbiamo assunto che questa proprietà sia invariante, ed inoltre si esce dal ciclo perché $r < D$. Ma allora è vera la proprietà

$X = q * D + r$, con $r < D$,

che è proprio la condizione di uscita del programma.

L'uso dell'invariante ci permette quindi di dimostrare che il programma è (parzialmente) corretto. In questo caso abbiamo già implicitamente dimostrato che il programma è anche totalmente corretto, perché abbiamo già visto che il ciclo deve terminare.

Resta da dimostrare che la proprietà (*) è proprio invariante. Questo si può fare **per induzione sul numero di iterazioni del ciclo**. Supponiamo che questo numero sia 0 (base dell'induzione) (ovviamente, la dimostrazione che (*) è invariante vale in generale, non solo per gli specifici valori di X e D che abbiamo scelto). Allora q = 0 (perché q non viene incrementato) e r = X. Allora X = q * D + r perché questo si riduce a dire che X = 0 * D + X, che è ovviamente vero. Supponiamo che il ciclo sia stato eseguito n volte, e che la proprietà (*) sia vera (**ipotesi induttiva**); vogliamo dimostrare ora che resta vera anche dopo la (n + 1)-esima iterazione. Durante questa iterazione vengono modificati i valori di q e di r, ottenendo valori

q' = q + 1
r' = r – D

dove q' ed r' sono i valori delle variabili q ed r dopo l'esecuzione delle istruzioni

r = r - D;
q = q + 1;

Allora calcoliamo:

q' * D + r' = (q + 1) * D + (r - D) = q * D + D + r - D = q * D + r = X

dove l'ultimo passaggio sfrutta l'ipotesi induttiva.

Per induzione si conclude allora che la proprietà (*) è vera per qualsiasi numero di iterazioni del ciclo, quindi (*) è invariante.

Esempio (Quadrato di un numero naturale) Vediamo un altro esempio della tecnica appena usata per dimostrare la correttezza del programma per la divisione intera, utilizzandola questa volta per sintetizzare un programma per calcolare il quadrato di un numero naturale N. La condizione di ingresso sarà dunque:

N ≥ 0

mentre la condizione di uscita sarà:

Y = X * X e X = N

dove Y è il dato in uscita ed X una variabile ausiliaria utilizzata come contatore. L'invariante appropriato in questo caso è la formula

(*) Y = X * X.

Inizialmente avremo dunque X = 0 e Y = 0: l'invariante è ovviamente vera in questo caso, e questo stabilisce la base della dimostrazione induttiva che la proprietà (*) è invariante.

```
class quadrato {

        public static void main (String[] args) {
```

```
        int N, X, Y;

        N = … ; // inizializzazione
        X = 0;
        Y = 0;
        while (X < N) {
            Y = Y + 2 * X + 1;
            X = X + 1;
        }
        System.out.println ("Quadrato = " + Y);
    }
}
```

Per quanto riguarda il passo induttivo: l'ipotesi induttiva è che Y = X * X dopo l'n-esima iterazione; bisogna dimostrare che (*) resta vera dopo l'(n+1)-esima iterazione. Infatti, se Y' è il valore di Y dopo l'esecuzione dell'istruzione Y = Y + 2 * X + 1, mentre X' è il valore di X dopo l'esecuzione dell'istruzione X = X + 1, possiamo calcolare

Y' = Y + 2 * X + 1
= (X * X) + 2 * X + 1 (per ipotesi induttiva)
= (X + 1) * (X + 1)
= X' * X'

da cui si conclude che (*) è invariante. Poiché il ciclo termina (infatti il valore di N – X decresce strettamente ad ogni iterazione), all'uscita dal ciclo avremo X = N (perché la condizione del while è falsa e sempre X ≤ N), quindi per l'invariante Y = N * N, che mostra che la condizione di uscita è soddisfatta dal dato in uscita Y, perciò il programma è corretto.

Definizioni ricorsive di funzioni

Immaginiamo di volere definire una funzione f : N $\rightarrow$ A, dove N è l'insieme dei numeri naturali ed A un insieme qualsiasi. Si può allora utilizzare il seguente **schema di ricorsione**:

f(0) = a
f(n+1) = E(f(n))

dove a è un elemento di A e con la notazione E(f(n)) si indica che l'espressione E( ) può utilizzare il valore f(n). Una giustificazione intuitiva di questo schema si può ottenere considerando la struttura dei numeri naturali: la funzione f è definita per 0 perché la prima clausola dello schema ne fornisce il valore a; supponiamo invece che k sia un numero positivo, e che quindi k = n + 1 per qualche numero naturale n. Si può immaginare di avere già calcolato il valore di f(n) (la funzione f viene calcolata "dal basso", partendo dall'argomento 0), e si può quindi calcolare E(f(n)) che dà il valore di f(n+1). (Una giustificazione rigorosa di questo metodo di definizione di funzioni è al di là della portata di queste lezioni.)

Vediamo solo un esempio di applicazione di questo schema, riprendendo un esempio già visto trattato mediante un programma iterativo dimostrato corretto mediante invarianti:

Esempio (La funzione quadrato) Si può definire ricorsivamente il quadrato di un numero naturale mediante le clausole:

$q(0) = 0$
$q(n+1) = q(n) + 2*n +1$

Vediamo che le clausole precedenti definiscono effettivamente la funzione desiderata, dimostrando per induzione che la proprietà $q(n) = n*n$ è vera per ogni valore di n.

(Base dell'induzione)
$q(0) = 0$ (per definizione)
$= 0*0$

(Passo indutttivo)
$q(n+1) = q(n) + 2*n + 1$ (per definizione)
$= n*n + 2*n + 1$ (per ipotesi induttiva)
$= (n+1)*(n+1)$ (per proprietà algebriche)

Osservazione Le clausole della definizione ricorsiva della funzione q(n) consentono anche di calcolare il valore di questa funzione per un valore arbitrario dell'argomento. Per esempio:

$q(5) = q(4+1)$
$= q(4) + 2*4 + 1$
$= q(3+1) + 2*4 +1$
$= q(3) + 2*3 + 1 + 2*4 + 1$
$= q(2 +1) + 2*3 + 1 + 2*4 + 1$
$= q(2) + 2*2 +1 + 2*3 + 1 + 2*4 + 1$
$= q(1+1) + 2*2 +1 + 2*3 + 1 + 2*4 + 1$
$= q(1) + 2*1 + 1 + 2*2 +1 + 2*3 + 1 + 2*4 + 1$
$= q(0+1) + 2*1 + 1 + 2*2 +1 + 2*3 + 1 + 2*4 + 1$
$= q(0) + 2*0 + 1+ 2*1 + 1 + 2*2 +1 + 2*3 + 1 + 2*4 + 1$
$= 0 + 2*0 + 1+ 2*1 + 1 + 2*2 +1 + 2*3 + 1 + 2*4 + 1$
$= 1 + 2 + 1 + 4 + 1+ 6 + 1 + 8 + 1$
$= 25$

Esempi

(1) Dato un insieme A con una funzione $f : A \to A$, si definisce la funzione $f^{(n)} : A \to A$ per ricorsione, mediante le clausole:

$f^{(0)}(a) = a$
$f^{(n+1)}(a) = f(f^{(n)}(a))$

in generale si vede che $f^{(n)}(a) = f(f(...f(a)...))$.
(2) Le clausole ricorsive

$s(0) = 1$

s(n+1) = 2/s(n)

definiscono la sequenza 1, 2, 1, 2 ,1, 2, ...
(3) Il numero A(n) di modi in cui n persone possono essere assegnate a n poltrone può essere definito ricorsivamente mediante le clausole:

A(1) = 1
A(n+1) = A(n) * (n+1)

Si può dimostrare per induzione su n ≥ 1 che A(n) = n!

Come si progetta un metodo ricorsivo

Quanto segue è un tentativo di riassumere le lezioni introduttive sulla programmazione ricorsiva, ripercorrendo gli esempi fattti in aula nelle lezioni di Novembre. Per fare un breve sommario delle future attrazioni, studieremo

a) il fattoriale di un numero naturale
b) il quadrato di un numero naturale
c) Somma, prodotto ed elevazione a potenza di due numeri naturali
d) varianti della ricorsione con parametro per l'accumulazione del risultato
e) conversione della rappresentazione di un numero naturale da decimale a binario.

Prima però studiamo in generale in che cosa consiste la tecnica della ricorsione.

Consideriamo un metodo **ricorsivo** di tipo int, con un parametro di tipo int che si assume abbia sempre valori naturali, cioè ≥ 0, per implementare una funzione f : N $\rightarrow$ N. Il requisito che il metodo F deve implementare la funzione f si esprime mediante l'affermazione

per ogni n in N, F(n) = f(n)

La condizione di ingresso di F è appunto che il valore del parametro intero n sia ≥ 0. Quando si progetta il corpo del metodo per un generico parametro formale n che si assume essere un numero naturale, è utile indicare anche, come condizione di uscita del metodo, che deve valere F(n) = f(n):

```
static int F(int n) {
// condizione di ingresso: n ≥ 0; condizione di uscita: F(n) = f(n)

}
```

Operativamente, la tecnica della ricorsione richiede che vengano presi in considerazione due casi (il secondo caso ne comprende, in realtà, infiniti altri perché è uno schema):

Base della ricorsione: n = 0

Passo ricorsivo: n > 0, ed in questo caso la tecnica ci concede di fare uso dell'invocazione F(n – 1) per calcolare il valore di F(n). Non solo: si può assumere che questa invocazione restituisca correttamente il valore f(n – 1).

Intenzionalmente ho utilizzato una terminologia che ricorda quella utilizzata nel descrivere la tecnica di dimostrazione per induzione (base dell'induzione e passo induttivo). Possiamo stabilire la seguente corrispondenza tra la costruzione del metodo ricorsivo F(int n) e la dimostrazione induttiva che, per ogni valore naturale di n, F(n) = f(n).

Ricorsione	**Induzione**
Base della ricorsione: se n = 0, si restituisce il valore f(0)	Base dell'induzione: F(0) = f(0)
Passo ricorsivo: se n > 0, si restituisce un valore rappresentato da una espressione che può contenere (e normalmente la conterrà) la sottoespressione F(n – 1). L'espressione per il valore di F(n) è suggerita da una definizione ricorsiva per la funzione f.	Passo induttivo: assumendo (questa assunzione si chiama **ipotesi induttiva**) che F(n – 1) calcoli correttamente f(n – 1), si dimostra che F(n) calcola correttamente f(n). La dimostrazione del passo induttivo sfrutta la definizione ricorsiva della funzione f.

Vediamo ora come questa tecnica si applica nel caso di alcuni esempi, in ordine (leggermente) crescente di difficoltà.

a) **Il fattoriale di un numero naturale**
Il fattoriale n! di un numero naturale n è il prodotto 1 · … · n, con la convenzione che 0! = 1. Per dare una versione ricorsiva della definizione della funzione fattoriale (in quanto distinta dal metodo per calcolare questa funzione, che può essere ricorsivo o iterativo) bisogna riconoscere una struttura ricorsiva nel prodotto 1 · … · n. Questo è un esercizio creativo che può essere anche molto complicato. In questo caso però è semplice, basta raggruppare i fattori del prodotto in modo leggermente diverso (usando la proprietà associativa del prodotto)

$$(1 \cdot \ldots \cdot n - 1) \cdot n$$

per accorgersi che vale la formula

$$n! = n \cdot (n - 1)!,$$

ed ecco che abbiamo il passo ricorsivo della definizione della funzione fattoriale. Vediamo ora come progettare il metodo ricorsivo per calcolare questa funzione:

```
static int F(int n) {

}
```

Prima di tutto annotiamo il metodo con un commento che esprime la condizionedi ingresso e la condizione di uscita:

```
static int F(int n) {
// condizione di ingresso: n ≥ 0; condizione di uscita: F(n) = n!
```

```
}
```

e procediamo a scrivere le istruzioni per i due casi della definizione ricorsiva come alternative di una istruzione condizionale a due vie:

```
static int F(int n) {
// F(n) = n!
if (n == 0)
        …
else
        …
}
```

In realtà sappiamo già qualcosa che ci può essere utile: la tecnica della ricorsione ci permette, al passo ricorsivo, di usare F(n – 1) per calcolare (n – 1)!. Ma la definizione ricorsiva del fattoriale che abbiamo utilizzato prima ci dice che n! = n · (n – 1)!, quindi completiamo la definizione del metodo, prima con la base della ricorsione

```
static int F(int n) {
// F(n) = n!
if (n == 0)
        return 1;
else
        …
}
```

infatti 0! = 1; e infine con il passo ricorsivo

```
static int F(int n) {
// F(n) = n!
if (n == 0)
        return 1;
else
        return n * F(n – 1);
}
```

Esercizio: dimostrare la correttzza del metodo F per calcolare il fattoriale, cioè dimostrare che per ogni n ≥ 0, F(n) = n!. Ovviamente la dimostrazione è per induzione su n: **generalmente il parametro su cui avviene la ricorsione è quello su cui si procede induttivamente per dimostrare la correttezza del metodo**.

b) **Il quadrato di un numero naturale**
Anche in questo caso, come nel precedente, vogliamo trovare un metodo ricorsivo per calcolare la funzione che associa ad un numero n il suo quadrato n^2. Procediamo nello stesso modo di prima, osservando innanzitutto che $(n + 1)^2 = n^2 + 2 \cdot n + 1$, una formula già nota ai pitagorici che è il passo ricorsivo di una definizione ricorsiva, la cui base è $0^2 = 0$.
Il metodo ricorsivo è allora

```
static int Q(int n) {
// Q(n) = n^2
if (n == 0)
        return 0;
else
        return Q(n–1) + 2 * n – 1;
}
```

c) **Somma, prodotto ed elevazione a potenza di due numeri naturali**
In altri casi non è immediatamente chiaro quale parametro di un metodo scegliere come parametro di ricorsione. Consideriamo la somma di numeri naturali:

```
static int somma(int n, int m) {
// calcola la somma di m,n ≥ 0 }
```

In questo caso sarà chiaro che la scelta del parametro di ricorsione non altera la concezione del metodo. L'idea è in ogni caso la stessa: vedere come la somma di n ed m dipenda

- dalla somma di n ed m – 1, somma (n, m – 1), oppure
- dalla somma di n – 1 ed m, somma (n – 1,m)

In entrambi i casi, basta aggiungere 1 al risultato della chiamata ricorsiva. Immaginando di scegliere m come parametro su cui fare la ricorsione:

```
static int somma(int n, int m) {
// calcola la somma di m,n ≥ 0
if (m == 0)
        return n;
        // n + 0 = n
else
        return somma(n,m – 1) + 1;
        // n + m = n + (m – 1) + 1
}
```

Analogo ragionamento si applica al metodo ricorsivo per calcolare il prodotto di due numeri naturali: scegliamo uno dei parametri e facciamo la ricorsione su quello. Nel caso di funzioni non commutative come l'elevazione a potenza, la scelta del parametro di ricorsione può essere decisiva per l'esito dell'impresa. Allora bisogna ragionare così:

so come esprimere il valore di b^e in temini di $(b – 1)^e$ ed operazioni note? La risposta, dopo qualche riflessione, sarà presumibilmente "no", e allora si prova l'altra domanda:
so come esprimere il valore di b^e in termini di $b^{(e – 1)}$ ed operazioni note? Qui la risposta è "sì": infatti

$$b^e = b \cdot b^{(e-1)}$$

e possiamo usare questa idea per il metodo seguente:

```
static int exp(int b, int e) {
// calcola b elevato alla potenza e
if (e == 0)
        return 1;
else
        return exp(b,e – 1) * b;
}
```

d) **Varianti della ricorsione con parametro per l'accumulazione del risultato**
La somma di due numeri naturali si può calcolare, in modo iterativo, mediante l'algoritmo che toglie 1 da un addendo e lo aggiunge all'altro. C'è una versione ricorsiva di questo procedimento, che rapresenta un esempio semplice di una tecnica di ricorsione ampiamente diffusa, soprattutto in programmazione funzionale, in cui il risultato viene accumulato in uno dei parametri del metodo che calcola la funzione. Il risultato dell'accumulazione viene restituito quando si raggiunge il caso base della ricorsione.
Vediamo due esempi di questa tecnica:

```
static int somma(int n, int m){
        if (m == 0)
                return n;
        else
                return somma(n + 1, m – 1);
}

public static int fattoriale(int a,int n){
// calcola a * n!
        if (n == 0) return a;
        else return fattoriale(a*n,n-1);
}
```

Chiaramente, nell'ultimo caso si deve fare attenzione ai parametri attuali nella prima chiamata: se si vuole calcolare il fattoriale di n occorre invocare il metodo come fattoriale(1,n).

e) **Conversione della rappresentazione di un numero naturale da decimale a binario**
La conversione di un numero espresso in base 10 alla sua rappresentazione in base 2 segue un algoritmo che prescrive di continuare a dividere per 2 tenendo traccia dei resti delle divisioni. La sequenza dei resti letta al contrario è la rappresentazione in base 2 del numero dato. Per esempio, per il numero 6 abbiamo

```
6 : 2 = 3       resto 0
3 : 2 = 1       resto 1
1 : 2 = 0       resto 1
```

e la sequenza 110 è proprio la rappresentazione binaria di 6.
Come si può escogitare una versione ricorsiva di questo procedimento? Basta osservare che quello che resta da fare dopo avere diviso un numero n per 2 è esattamente il procedimento che fornisce

la rappresentazione binaria di n : 2. Rovesciare la sequenza dei resti si può fare sfruttando lo stack implicito nell'esecuzione della ricorsione:

```
static void conv (int n){   // c.i.: n >= 0, c.u.: n è convertito in base 2
        if (n / 2 != 0)
                conv(n / 2);
        System.out.print(n % 2);
        }
```