Lab. Programmazione (CdL Informatica) & Informatica (CdL Matematica) a.a. 2022-23

Monica Nesi

Università degli Studi dell'Aquila

22 Novembre 2022

Introduzione alla Ricorsione

Finora abbiamo considerato metodi statici *iterativi*, ovvero metodi in cui vengono usati comandi iterativi o di ciclo per risolvere i problemi dati.

Molti problemi possono essere risolti utilizzando un approccio diverso, basato sulla *ricorsione*.

Un concetto si dice *ricorsivo* quando nella sua definizione si utilizza il concetto stesso, ovvero il concetto è definito in termini di se stesso o richiama se stesso.

Un esempio classico di definizione ricorsiva è quello della funzione *fattoriale*.

Definizione ricorsiva del fattoriale

In alcune lezioni precedenti abbiamo richiamato la definizione del fattoriale di un numero naturale:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1) \times \ldots \times 2 \times 1 \quad \text{per } n \geq 1 \end{aligned}$$

ed abbiamo scritto un metodo iterativo (basato su tale definizione) per calcolare il fattoriale in Java.

La definizione del fattoriale però può essere data anche come segue:

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \quad \text{per } n \geq 1 \end{aligned}$$

dove, nella seconda clausola, il concetto di fattoriale (che si sta definendo) compare anche *a destra della definizione*.

Questo significa che la definizione è data in modo *ricorsivo*.

Definizioni ricorsive

Nelle definizioni ricorsive occorre avere:

- *almeno un caso base*, ovvero una parte della definizione in cui non vi è ricorsione, ed

- *uno o più casi (o clausole) ricorsivi*, ovvero parti della definizione in cui il concetto che si sta definendo ricorre nella definizione.

Le definizioni ricorsive sono a volte caratterizzate da una minore efficienza rispetto alle soluzioni iterative, ma in generale sono più leggibili, più semplici e più facili da verificare corrette (o meno).

Esistono vari tipi di ricorsione.

La ricorsione nel fattoriale è detta *lineare*, in quanto nella clausola ricorsiva si ha un'*unica chiamata* alla funzione definita ricorsivamente.

Numeri di Fibonacci

I numeri di Fibonacci sono un altro classico esempio di definizione ricorsiva.

Definizione dei numeri di Fibonacci:

$$\begin{aligned} fib(0) &= 1 \\ fib(1) &= 1 \\ fib(n) &= fib(n-1) + fib(n-2) \quad \text{per } n \geq 2 \end{aligned}$$

Abbiamo:
- due casi base (per gli argomenti 0 ed 1);
- nella clausola ricorsiva la funzione è chiamata due volte su argomenti diversi. Si parla di *ricorsione branching (o binaria)*.

Terminazione della ricorsione

In una definizione ricorsiva occorre garantire che in un *numero finito di applicazioni* del concetto ricorsivo si giunga ad uno dei casi base e quindi la computazione termini (*ricorsione benfondata*).

Ciò significa garantire che ad *ogni chiamata ricorsiva* di una funzione, tale funzione venga applicata su uno o più argomenti tali da assicurare la terminazione della computazione (i.e., *riduzione dello spazio di ricerca*).

Nel caso della definizione ricorsiva del fattoriale, a partire da un qualsiasi argomento iniziale n, la computazione termina in un tempo finito, in quanto ad ogni chiamata ricorsiva del fattoriale l'argomento decresce di un'unità e si avvicina al caso base.

Lo stesso ragionamento vale per la definizione ricorsiva dei numeri di Fibonacci.

Tipi ricorsivi o induttivi

Sia la funzione del fattoriale che la serie dei numeri di Fibonacci sono definiti sui numeri naturali, che possono essere definiti ricorsivamente (o *induttivamente*) come segue:

$$Nat \ ::= \ 0 \mid succ\, Nat$$

Tale definizione asserisce che un numero naturale in Nat è la costante 0 oppure il numero dato dall'operatore $succ$ applicato ad un elemento di tipo Nat (da cui segue la ricorsione).

In base a tale definizione, un qualsiasi numero naturale k è rappresentato tramite $succ(succ(\ldots(0)\ldots))$, ovvero l'applicazione k volte di $succ$ a partire dalla costante 0.

Il *Principio di Induzione Naturale* si basa su una definizione induttiva dei numeri naturali.

Tipi ricorsivi o induttivi (cont.)

Le definizioni ricorsive di vari concetti sono immediate quando il tipo degli elementi, su cui un concetto viene definito, può essere definito in modo ricorsivo o induttivo.

Nell'Informatica esistono molti tipi di dati (e.g., espressioni e comandi in un linguaggio di programmazione) e strutture dati (e.g., alberi, liste, pile, code) definiti in modo ricorsivo.

Su tali tipi e strutture è immediato e naturale definire funzioni, metodi, etc., in modo ricorsivo.

Non solo, ma anche fare ragionamenti e dimostrazioni di proprietà tramite *principi di induzione strutturale*.

In Java è semplice definire metodi ricorsivi che implementano funzioni definite ricorsivamente.

Ricorsione in Java: fattoriale

Scrivere un metodo ricorsivo che calcola il fattoriale di un numero naturale `n`. Alla definizione già vista aggiungiamo il requisito seguente: se al metodo viene passato un numero intero negativo, il metodo restituisce `-1`.

```
public static int fattR (int n) {
  if (n < 0) return -1;
  if (n == 0) return 1;      //caso base
  return n*fattR(n-1);       //caso ricorsivo
}
```

Notare come il metodo `fattR` è semplicemente la definizione ricorsiva del fattoriale scritta nella sintassi di Java.

Ricorsione annidata

Il metodo `fattR` è un metodo con *ricorsione annidata (nested recursion)*, in quanto la chiamata ricorsiva al metodo `fattR` si trova dentro l'operazione di moltiplicazione.

Ad esempio, valutiamo l'espressione `fattR(5)` (ovvero, la chiamata del metodo `fattR` con parametro attuale 5):

```
fattR(5) =
5*fattR(5-1) =
5*(4*fattR(4-1)) =
5*(4*(3*fattR(3-1))) =
5*(4*(3*(2*fattR(2-1)))) =
5*(4*(3*(2*(1*fattR(1-1))))) =
5*(4*(3*(2*(1*1)))) =
5*(4*(3*(2*1))) =
5*(4*(3*2)) =
5*(4*6) =
5*24 = 120
```

Ricorsione in testa

I risultati intermedi della computazione possono essere calcolati durante lo svolgimento ricorsivo *al costo di* un metodo in più e di qualche parametro in più, secondo un approccio basato sul mettere la chiamata ricorsiva *in testa*.

Una soluzione con *ricorsione in testa* (in inglese, invece, si dice *tail-recursion*) per il fattoriale è data dai seguenti metodi:

```
public static int fattTR (int n) {
  if (n < 0) return -1;
  return fattTR(n,1);
}

public static int fattTR (int n, int r) {
  if (n == 0) return r;       //caso base
  return fattTR(n-1,n*r);     //caso ricorsivo
}
```

Ricorsione in testa: alcune osservazioni

- Abbiamo *due* metodi chiamati con lo stesso nome sfruttando l'*overloading* di Java;

- il primo metodo `fattTR` non è ricorsivo e ha un'intestazione uguale a quella del metodo `fattR` (tranne per il nome, che è necessario cambiare se tali metodi sono nella stessa classe);

- il secondo metodo `fattTR` è il metodo ricorsivo e ha il parametro formale `r` in più, oltre al parametro `n`;

- il primo metodo `fattTR` effettua eventuali controlli sui parametri in ingresso e poi invoca il metodo ricorsivo `fattTR` passandogli, oltre al parametro `n`, il valore `1` da legare al parametro `r`;

- il parametro `r` del metodo ricorsivo sfrutta la *modalità del passaggio dei parametri per valore* per calcolare i risultati intermedi della funzione fattoriale.
Tale parametro è inizializzato al valore `1` da restituire nel caso base.

Ricorsione in testa: esecuzione

Valutiamo ora l'espressione `fattTR(5)`:

```
fattTR(5) =
fattTR(5,1) =
fattTR(5-1,5*1) =
fattTR(4-1,4*5) =
fattTR(3-1,3*20) =
fattTR(2-1,2*60) =
fattTR(1-1,1*120) =
120
```

I numeri di Fibonacci

Scrivere un metodo ricorsivo che calcola l'`n`-esimo numero di Fibonacci. Alla definizione già vista aggiungiamo il requisito seguente: se al metodo viene passato un numero intero negativo, il metodo restituisce `-1`.

```
public static int fibR (int n) {
  if (n < 0) return -1;
  if (n <= 1) return 1;          //casi base
  return fibR(n-1)+fibR(n-2);    //caso ricorsivo
}
```

Anche il metodo `fibR` è semplicemente la definizione ricorsiva dei numeri di Fibonacci scritta nella sintassi di Java.

Segue quindi che è molto leggibile, ma molto inefficiente nella computazione a causa della ricorsione branching annidata (molti numeri vengono calcolati più volte).

Metodo ricorsivo per i numeri di Fibonacci: esecuzione

Valutiamo l'espressione `fibR(4)`:

```
fibR(4) =
fibR(4-1) + fibR(4-2) =
(fibR(3-1) + fibR(3-2)) + (fibR(2-1) + fibR(2-2)) =
((fibR(2-1) + fibR(2-2)) + 1) + (1+1) =
((1+1)+1)+2 =
(2+1)+2 =
3+2 =
5
```

Ricorsione con array

Finora abbiamo trattato molti problemi che lavorano su array.

Ora l'obiettivo è fare esperienza di programmazione in Java utilizzando la ricorsione.

Quindi vorremmo scrivere delle soluzioni ricorsive per i problemi visti, in alternativa alle soluzioni iterative già date.

Però l'array *non* è una struttura dati definita ricorsivamente.

Quindi come lavorarci sopra in modo ricorsivo?

Tipicamente gli array vengono esaminati scorrendoli con indici che indicano le posizioni degli elementi.

Possiamo applicare un *ragionamento ricorsivo* incrementando o diminuendo opportunamente tali indici per ridurre lo spazio di ricerca in un array ad ogni chiamata ricorsiva, seguendo un *approccio basato sulla ricorsione in testa*.

Contare occorrenze in un array

Scrivere un metodo ricorsivo che, dati un array monodim. di interi `a` ed un intero `n`, restituisce il numero delle occorrenze di `n` in `a`.

Soluzione con ricorsione in testa

```
public static int occorrenzeRic(int[] a, int n) {
  return occorrenzeRic(a,n,0,0);
}


public static int occorrenzeRic(int[] a, int n,
int i, int c) {
  if (i == a.length)
    return c;
  if (a[i] == n)
    return occorrenzeRic(a,n,i+1,c+1);
  return occorrenzeRic(a,n,i+1,c);
}
```

Contare occorrenze in un array (cont.)

Il secondo metodo `occorrenzeRic` può essere scritto anche come segue:

```
public static int occorrenzeRic(int[] a, int n,
int i, int c) {
  if (i == a.length)
    return c;
  if (a[i] == n)
    c++;
  return occorrenzeRic(a,n,i+1,c);
}
```

N.B. Non scrivere `i++` e `c++` all'interno delle chiamate ricorsive, al posto di `i+1` e `c+1`! Non è corretto!

Sommare i numeri in un array

Scrivere un metodo ricorsivo che, dato un array monodim. di interi a, restituisce la somma degli elementi in a.

```
public static int sommaARic(int[] a) {
  return sommaARic(a,0,0);
}

public static int sommaARic(int[] a, int i,
int sum) {
  if (i == a.length)
    return sum;
  return sommaARic(a,i+1,sum+a[i]);
}
```

Verificare l'occorrenza di un elemento

Scrivere un metodo ricorsivo che, dati un array monodim. di interi a ed un intero `n`, restituisce `true` se `n` compare in a, `false` altrimenti.

```
public static boolean occorreRic(int[] a,
int n) {
  return occorreRic(a,n,0);
}

public static boolean occorreRic(int[] a,
int n, int i) {
  if (i == a.length)
    return false;
  if (a[i] == n)
    return true;
  return occorreRic(a,n,i+1);
}
```

Verificare l'occorrenza di un carattere in una stringa

Scrivere un metodo ricorsivo che, dati un carattere c ed una stringa s, restituisce `true` se c occorre in s, `false` altrimenti.

```
public static boolean occorreCarRic(char c,
String s) {
  return occorreCarRic(c,s,0);
}

public static boolean occorreCarRic(char c,
String s, int i) {
  if (i == s.length())
    return false;
  if (s.charAt(i) == c)
    return true;
  return occorreCarRic(c,s,i+1);
}
```

Contare occorrenze in un array bidimensionale

Scrivere un metodo ricorsivo che, dati un array bidim. di interi a ed un intero x, restituisce il numero delle occorrenze di x in a.

```
public static int occorrenzeBiRic(int[][] a,
int x) {
  return occorrenzeBiRic(a,x,0,0,0);
}


public static int occorrenzeBiRic(int[][] a,
int x, int i, int j, int c) {
  if (i == a.length)
    return c;
  if (j == a[i].length)
    return occorrenzeBiRic(a,x,i+1,0,c);
  if (a[i][j] == n)
    c++;
  return occorrenzeBiRic(a,x,i,j+1,c);
}
```

Sommare i numeri in un array bidimensionale

Scrivere un metodo ricorsivo che, dato un array bidimensionale di interi a, restituisce la somma degli elementi in a.

```
public static int sommaABiRic(int[][] a) {
  return sommaABiRic(a,0,0,0);
}

public static int sommaABiRic(int[][] a,
int i, int j, int sum) {
  if (i == a.length)
    return sum;
  if (j == a[i].length)
    return sommaABiRic(a,i+1,0,sum);
  return sommaABiRic(a,i,j+1,sum+a[i][j]);
}
```

Verificare l'occorrenza di un elemento in un array bidim.

Scrivere un metodo ricorsivo che, dati un array bidim. di interi a ed un intero x, restituisce `true` se x occorre in a, `false` altrimenti.

```
public static boolean occorreBiRic(int[][] a,
int x) {
  return occorreBiRic(a,x,0,0);
}

public static boolean occorreBiRic(int[][] a,
int x, int i, int j) {
  if (i == a.length)
    return false;
  if (j == a[i].length)
    return occorreBiRic(a,x,i+1,0);
  if (a[i][j] == n)
    return true;
  return occorreBiRic(a,x,i,j+1);
}
```

Confronto tra soluzione iterativa e soluzione ricorsiva

Confrontando le soluzioni iterative e ricorsive dei problemi considerati precedentemente, possiamo notare:

- i parametri in più passati al metodo ricorsivo corrispondono alle variabili locali del metodo iterativo;

- i valori a cui sono legati questi parametri nella prima chiamata del metodo ricorsivo sono i valori iniziali delle variabili locali nel metodo iterativo;

- i casi base della ricorsione corrispondono alle condizioni di uscita da un ciclo;

- le modifiche sui parametri attuali del metodo ricorsivo corrispondono alle modifiche sulle variabili locali del metodo iterativo.