Sviluppare Programmi Corretti II: Programmi Ricorsivi

Metodologie di Programmazione
Ivano Salvo
Sapienza Università di Roma
email: salvo@di.uniroma1.it

Anno Accademico 2013-14

Presentazione

In questa seconda dispensa analizzeremo lo sviluppo di programmi ricorsivi corretti, utilizzando ancora una volta *asserzioni logiche*.

Verranno inoltre analizzati vantaggi e svantaggi della ricorsione rispetto all'iterazione.

1 Ricorsione ed Iterazione

In questa sezione verrà introdotto un nuovo e potente meccanismo di controllo: la *ricorsione*. Una funzione è *ricorsiva* quando al suo interno contiene una chiamata a sè stessa. Da un punto di vista matematico, la cosa non sorprende: molte definizioni matematiche sono inerentemente ricorrenti (o *induttive*) e definiscono un concetto in termini dello stesso concetto (in una versione più semplice) e di alcuni *casi base*. In qualche corso, qualche professore avrebbe potuto lasciarsi sfuggire che:

... un numero naturale è zero, o il successore di un numero naturale...

Da un punto di vista informatico, non dovrebbe stupire che sulla pila di sistema possano convivere più *record di attivazione* della stessa funzione, ciascuna con il proprio stato locale di esecuzione. Nel seguito confronteremo la ricorsione con l'iterazione e introdurremo delle metodologie per valutare la correttezza dei programmi ricorsivi (essenzialmente basate sull'*induzione*).

1.1 La funzione fattoriale

Così come il programma `HelloWorld` è il prototipo di tutti i programmi Java, la funzione fattoriale è il prototipo di tutte le funzioni ricorsive. Il fattoriale, che in matematica è indicato con il simbolo postfisso !, può essere informalmente definita come segue: il fattoriale di un intero n è il prodotto di tutti i numeri da 1 ad n, cioè $1 \times 2 \times \ldots \times (n-1) \times n$. Esiste anche una più elegante definizione *induttiva*,

ossia una definizione che si limita a definire il valore della funzione fattoriale su 0 e il valore del fattoriale su un naturale successore $n+1$ in funzione del fattoriale di n:

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) \times n! \end{aligned}$$

Una proprietà chiave dei numeri naturali è quella che definizioni di questo tipo effettivamente individuano in modo univoco una funzione. L'importante è che a destra dell'uguale la funzione che si sta definendo per induzione sia applicata a numeri più piccoli e che siano considerati dei *casi base* (in questo caso lo 0).

Scrivere una funzione TinyJava iterativa che calcola il fattoriale è molto facile e segue uno schema di programmazione ben noto, assimilabile a quello già visto per il calcolo del prodotto o dell'esponenziale ed è mostrata in Fig. 1. Si eseguono i prodotti accumulando i risultati parziali in una variabile. La variabile viene inizializzata ad 1, l'elemento neutro del prodotto. La funzione tratta correttamente il caso base (non viene eseguito nessun prodotto e la variabile accumulatore rimane col suo valore iniziale, 1, cioè 0!).

```
public static int fattIt(int n){
/* REQUIRE: n>=0.
 * EFFECTS: returns n!.
 */
  int f=1;
  int i=0;

  while (i!=n){
      /* INV: f=i!, TERM: n-i */
       i++;
       f = mult(f,i);
     }
     return f;
}
```

Figura 1: Funzione **iterativa** per calcolare il fattoriale

La versione ricorsiva (Fig. 2), viceversa, è sostanzialmente una traduzione in TinyJava della definizione induttiva del fattoriale. Le equazioni ricorsive che definiscono il fattoriale vengono semplicemente tradotte in TinyJava discriminando i diversi casi della definizione con un costrutto condizionale `if` e sostituiendo la notazione postfissa del simbolo di fattoriale ! con le chiamate ricorsive alla funzione che stiamo definendo `fattRec`.

Non si tratta di una coincidenza fortunata. Facciamo altri esempi meno popolari. Possono essere definite per induzione tutte le funzioni sui naturali. Ad esempio la somma per induzione sul secondo argomento:

$$\begin{aligned} m+0 &= m \\ m+(n+1) &= (m+n)+1 \end{aligned}$$

```
public static int fattRec(int n){
/* REQUIRE: n>=0.
 * EFFECTS: returns n!.
 */
 if (n==0) return 1;                        /* caso base */
      else return mult(n,fattRec(n-1));     /* passo induttivo */
  }
```

Figura 2: Funzione **ricorsiva** per calcolare il fattoriale

A questa definizione corrisponde immediatamente un corrispondente programma ricorsivo, presentato in Fig. 3.

```
public static int sommaRec(int m, int n){
/* REQUIRE: n>=0.
 * EFFECTS: returns n!.
 */
 if (n==0) return m;
      else return sommaRec(m,pred(n))+1;
  }
```

Figura 3: Funzione ricorsiva per calcolare la somma di due naturali

L'eleganza della ricorsione si dovrebbe già intuire da questi esempi: `fattRec` e `sommaRec` ad esempio non hanno bisogno di variabili accumulatore o contatore a differenza delle loro corrispettive funzioni iterative. Avevamo già dato definizioni induttive per descrivere la moltiplicazione egiziana e il massimo comun divisore con l'algoritmo di Euclide. Vediamo in Fig. 4 e 5 le corrispondenti funzioni ricorsive. Confrontatele con le relative funzioni iterative già viste.

Per inciso, ne approfittiamo per impratichirci con JAVA. Nella funzione `mcdRec` osserviamo che non mettiamo il ramo `else`: infatti è inutile, in quanto se la condizione valuta a `true`, viene eseguita un'istruzione `return` che sospende l'esecuzione della funzione e quindi comunque le istruzioni successive all'`if` non sarebbero eseguite.

Da un punto di vista concreto le versioni iterative e quelle ricorsive delle varie funzioni eseguono esattamente le stesse operazioni. Siccome l'allocazione dell'activation record di una procedura è un'operazione più costosa del salto indietro a rivalutare la guardia di un ciclo, la versione iterativa sarà leggermente più efficiente, mentre la versione ricorsiva è più vicina alla definizione matematica e mostra meno dettagli implementativi (variabili contatori e accumulatori). In particolare,

```
public static int mcdRec(int m, int n){
    if (minore(m,n)) return mcdRec(n-m,m);
    if (minore(n,m)) return mcdRec(m-n,n);
    return n;
    }
```

Figura 4: Algoritmo di Euclide Ricorsivo

```
public static int mulRec(int m, int n){
    if (n==0) return 0;
    if (divide(2,n)) return somma(m,mulRec(m,n-1));
    return mulRec(somma(m,m),quoziente(n,2));
    }
```

Figura 5: Moltiplicazione Egiziana ricorsiva

il trattamento delle variabili accumulatore nelle funzioni iterative `mcdEuclide` e `multiplyingLikeAnEgiptian` potrebbe non essere stato immediatamente evidente all'occhio del novizio.

Nel caso specifico non è particolarmente difficile capire cosa calcolino le varie versioni iterative, ma già nella prossima sezione vedremo esempi in cui la versione iterativa di una corrispondente funzione ricorsiva non è affatto semplice da trovare (la Sezione 3 è dedicata a problemi che non hanno soluzioni iterative semplici). Ciò che rende molto facile la traduzione delle funzioni ricorsive viste fin qui in iterative è sostanzialmente il fatto che queste funzioni eseguono al più *una sola chiamata ricorsiva* che chiude l'esecuzione della funzione. Funzioni ricorsive di questo tipo si chiamano *tail recursive* (cioè ricorsive di coda), e alcuni compilatori le traducono automaticamente in programmi iterativi, per i motivi di efficienza sopra visti. Esistono quindi facili regole per eliminare la ricorsione di coda.

La semplicità delle soluzioni ricorsive dipende dal fatto che i programmi ricorsivi mimano una naturale forma di ragionamento matematico, l'induzione, o se preferite un naturale modo di risolvere un problema: studiare i casi semplici (casi base) e ridurre la soluzione di istanze complicate a quella di istanze più semplici (passo induttivo).

A volte, tuttavia, la maggior semplicità del programma ricorsivo nasconde una complessità gestita implicitamente dal meccanismo computazionale che implementa la ricorsione, come vedremo nella prossima sezione.

1.2 La funzione di Fibonacci

La *successione di fibonacci*[1] viene induttivamente definita come segue:

$$\begin{aligned} \mathsf{fib}(0) &= 0 \\ \mathsf{fib}(1) &= 1 \\ \mathsf{fib}(n+2) &= \mathsf{fib}(n+1) + \mathsf{fib}(n) \end{aligned}$$

Essa definisce la successione di interi, $0, 1, 1, 2, 3, 5, 8, 13, 21, 23, 55, \ldots$. Di questa successione non è altrettanto evidente dare una definizione informale.

[1] La successione prende il nome dal matematico italiano Leonardo Pisano detto Fibonacci (Pisa $\sim$1170-$\sim$ 1250) che l'ha introdotta per studiare la riproduzione dei conigli, dando risposta alla seguente domanda: partendo da una coppia di conigli e supponendo che ogni coppia di conigli produca una nuova coppia ogni mese, a partire dal secondo mese di vita, quante coppie di conigli ci sono dopo n mesi? La successione di Fibonacci gode di numerose proprietà interessanti ed ha trovato successivamente moltre altre applicazioni in matematica.

Proponiamoci ora di scrivere due funzioni, una ricorsiva ed una iterativa, che preso in ingresso un intero n, calcolino l'n-esimo numero della successione di fibonacci. La funzione ricorsiva si può scrivere facilmente semplicemente traducendo le equazioni ricorsive in TinyJava, come nel caso del fattoriale (Fig. 6).

```
public static int fibRec(int n){
/* REQ: n>=0.
 * EFF: returns fib(n).
 */
 if (minUg(n,1)) return n;
 return fibRec(pred(n)) + fibRec(pred(pred(n)));
}
```

Figura 6: Funzione ricorsiva per il calcolo dei numeri di Fibonacci

La funzione iterativa è leggermente più complicata: tuttavia per scriverla è sufficiente osservare che per calcolare l'n-esimo numero di fibonacci è necessario conoscere i due precedenti numeri di fibonacci. Visto che conosciamo i primi due numeri della serie, possiamo pensare di calcolarli tutti a partire dal terzo (cioè $\mathsf{fib}(2)$ fino a quello desiderato. L'unica avvertenza è quella di mantenere sempre memorizzati gli ultimi due numeri calcolati per trovare il successivo numero di fibonacci (vedi Fig. 7).

Il lettore faccia anche attenzione all'invariante: l'asserzione logica $fib = \mathsf{fib}(i)$ sarebbe ovviamente sufficiente a dimostrare la correttezza della funzione, ossia che $fib = \mathsf{fib}(n)$ all'uscita del ciclo. Tuttavia, per dimostrare che questa asserzione è veramente un invariante per il ciclo in esame, è necessario avere delle assunzioni sui valori $fib1$ e $fib2$ da cui il prossimo valore di fib dipende. Anche questo è un fenomeno che si verifica molto spesso, guarda caso, nelle dimostrazioni per induzione, quando siamo obbligati a *rafforzare le ipotesi induttive*, cioè a dimostrare una proposizione più forte al fine di poter applicare il passo induttivo. In fondo, dietro alla metodologia degli invarianti, c'è un ragionamento induttivo sul numero di iterazioni del ciclo.

A differenza dei casi precedenti, le due funzioni `fibRec` e `fibIt`, si comportano in modo molto diverso: la funzione `fibIt` esegue il ciclo esattamente $n-2$ volte e ciascun ciclo costa una somma e 3 assegnazioni. La funzione ricorsiva, viceversa, per calcolare l'n-esimo numero di fibonacci, invoca il calcolo dell'$(n-1)$-esimo e dell'$(n-2)$-esimo. A sua volta il calcolo dell'$(n-1)$-esimo numero invocherà il calcolo dell' $(n-2)$-esimo e dell'$(n-3)$-esimo: già a questo punto è chiaro che parte del lavoro viene ripetuto inutilmente. In Fig. 8 viene esemplificato l'albero delle chiamate ricorsive generato per effetto di una chiamata a `fibRec(4)`.

È facile dimostrare (per induzione!) che i calcoli di $\mathsf{fib}(1)$ e $\mathsf{fib}(0)$ saranno eseguiti rispettivamente $\mathsf{fib}(n)$ e $\mathsf{fib}(n-1)$ volte (vedi Fig. 8), che è un numero che cresce esponenzialmente, quindi in questo caso la semplicità della funzione ricorsiva viene pagata a caro prezzo in termini di efficienza.

```
public static int fibIt(int n){
/* REQ: n>=0.
 * EFF: returns fib(n).
 */
 int fib2=0;
      /* variabile che memorizza il penultimo numero calcolato */
 int fib1=1;
     /* variabile che memorizza l'ultimo numero calcolato */
 int fib;
     /* variabile che memorizza il nuovo numero */
 int i=2;

 if (n<2) return n;
 while (i<n){
      i++;
      fib = fib1 + fib2;
      /* INV: fib = fib(i) & fib1 = fib(i-1) & fib2 = fib(i-2)*/
      fib2 = fib1;
      fib1 = fib;
      }
 return fib;
 }
```

Figura 7: Funzione iterativa per il calcolo dei numeri di Fibonacci

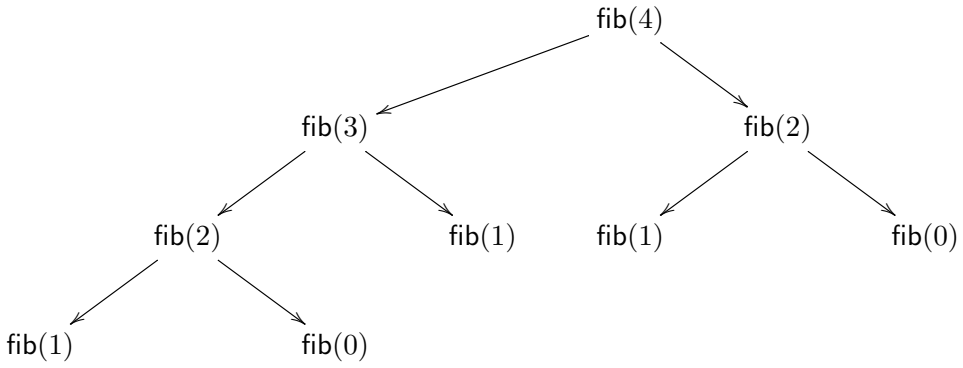

Figura 8: Attivazione delle chiamate ricorsive della funzione `fibRec(4)`

Esercizi e Spunti di Riflessione

1. Scegliere la vostra funzione ricorsiva preferita e cercate di scrivere le evoluzioni della pila di sistema durante l'esecuzione.

2. Riprendendo in considerazione l'esecutore che sa solo sommare 1 e testare l'uguaglianza con 0, scrivere funzioni *ricorsive* per il test di uguaglianza, il test di minore o uguale, la moltiplicazione, la divisione intera etc.

3. ★Forse leggermente più impegnativo, ancora una volta, sarà scrivere la funzione `predRec` che calcola ricorsivamente il predecessore. *Attenzione*: l'usuale schema ricorsivo visto in questa sezione non può essere seguito. Infatti in tutti gli esempi `f(n)` chiama `f(n-1)`, ma stavolta questo è vietato perchè ovviamente il nostro esecutore ipotetico non sa fare `n-1`.

4. Scrivere una funzione TinyJava che non contiene cicli, che non termina.

5. Scrivere una funzione TinyJava che non contiene cicli, che ricevendo come paramentro un intero n ritorna 1 se n è pari, non termina se n è dispari.

6. ♣Proporre un metodo generale per trasformare qualsiasi funzione che esegue un'unica chiamata ricorsiva in iterativa.

7. ♣Dare una prova informale del fatto che che una chiamata a `fibRec(n)` causa $\mathsf{fib}(n)$ chiamate del tipo `fibRec(1)`. ♣Dare una prova formale per induzione.

8. (Coefficienti Binomiali) Ricordiamo la definizione di coefficiente binomiale ($n \geq 0, 0 \leq k \leq n$):
$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$
Scrivere una procedura ricorsiva `public static int coeffBin(int n, int k)` basata sulle seguenti relazioni:
$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \quad \text{e} \quad \binom{n}{0} = \binom{n}{n} = 1$$
Valutare il numero delle chiamate ricorsive ai casi base.

2 Asserzioni logiche applicate alle funzioni ricorsive

Come abbiamo già visto, è opportuno completare la specifica di una procedura con dei commenti che esprimono cosa viene calcolato da una funzione (*postcondizione*), sotto opportune assunzioni sui dati in ingresso (*precondizioni*): nel caso delle procedure ricorsive questo automaticamente fornisce una tecnica di prova per dimostrarne la correttezza.

2.1 Precondizioni e Postcondizioni

Come già visto nel caso del predecessore, e di altre funzioni, le funzioni `fattIt` e `fattRec` non terminano nel caso in cui il parametro di ingresso sia negativo. Chiamando `fattRec` con parametri negativi, il programma termina in modo anomalo con un errore (anche qui inaspettato) di *stack overflow*: il motivo è che `fattRec` continua a chiamare sè stessa allocando activation records fino a saturare la memoria destinata allo stack di attivazione delle procedure (sul mio calcolatore dopo 261696 chiamate ricorsive).

Si tratta di un comportamento anomalo ed indesiderato, ma tutto sommato legittimo visto che il fattoriale è definito solo su interi positivi. In un qualche senso, il programmatore della funzione `fattRec` ha il diritto di non preoccuparsi di cosa accada nei punti in cui la funzione ! non è specificata[2].

Cogliamo l'occasione di osservare che così scritte, le funzioni `fattRec` e `fattIt` danno risultati corretti solo per valori esigui di `n` anche quando interrogate con parametri positivi: ben presto, infatti, il valore di $n!$ supera il massimo intero rappresentabile (sul mio calcolatore $2^{31} - 1$). Linguaggi diversi si comportano in modi diversi in queste situazioni: tipicamente viene quantomeno segnalato un errore di *overflow* anche nei casi in cui il programma venga lasciato proseguire[3]. Non si fanno grandi progressi neanche con altri tipi interi (`long int` o `long long int`).

Quindi, a parte questioni numeriche, le due funzioni funzionano correttamente solo quando è verificata la precondizione $n \geq 0$.

Ricordiamo che specificando le precondizioni, il programmatore che definisce la funzione `fattRec` oltre ad implementare il codice, specifica anche il suo corretto utilizzo[4]: in tal modo un programmatore utente della definizione della funzione `fattRec` è informato che dovrà evitare chiamate scorrette che non rispettano la precondizione.

Nel caso di funzioni ricorsive, il programmatore stesso della funzione `fattRec` dovrà preoccuparsi che le chiamate ricorsive rispettino la precondizione: nel nostro caso, nell'ipotesi $n \geq 0$ la chiamata ricorsiva verrà effettuata con un parametro che soddisfa la stessa proprietà: infatti se $n = 0$ non attivo alcuna chiamata ricorsiva, mentre $n > 0$ implica $n - 1 \geq 0$.

[2] il modo più maturo di gestire questi casi è *sollevare un'eccezione* informando il chiamante che la funzione non è andata a buon fine. Le eccezioni verranno considerate nel seguito del corso.

[3] Purtroppo non siamo ancora abituati all'idea che il software possa uccidere, ma dovremo presto farlo: un banale errore di overflow, per esempio, sembra sia stato alla base al famoso incidente del razzo francese Ariane.

[4] In realtà sarebbe possibile evitare la non terminazione, usando la condizione `n<=0`: tuttavia in tal caso, benchè la procedura termini sempre, i risultati ottenuti sui numeri negativi (verrebbe restituito sempre 1 se il parametro di ingresso è negativo) sono comunque arbitrari, visto che la funzione fattoriale è definita solo sugli interi positivi

2.2 Dimostrazioni di Correttezza per Funzioni Ricorsive

Osserviamo ora come sia semplice valutare la correttezza di una funzione ricorsiva: basterà fare una semplice dimostrazione per induzione. Cioè occorrerà dimostrare che:

1. i casi base siano trattati correttamente;
2. le eventuali chiamate ricorsive rispettino le precondizioni;
3. la funzione garantisca la postcondizione assumendo che le chiamate ricorsive garantiscano la postcondizione;
4. che le chiamate ricorsive si applichino ad argomenti "più piccoli" rispetto un ad un ordinamento *ben fondato*, cioè senza catene decrescenti infinite.

Prendendo ad esempio il fattoriale, dobbiamo dimostrare che, ricevendo in input un intero positivo, la funzione `fattRec` ne restituisce il fattoriale.

La base di induzione consiste nella banale verifica che per $n = 0$ viene restituito 1, come stabilito dalla definizione di fattoriale. Il passo induttivo consiste nel verificare che per $n > 0$ viene restituito $n!$. Ma ciò è banale perché per $n > 0$ viene attivata la chiamata ricorsiva a `fattRec(n-1)`: questa attivazione della procedura soddisfa le precondizioni in quanto $n > 0 \Rightarrow n - 1 \geq 0$; possiamo quindi usare l'ipotesi induttiva, e cioè che la chiamata restituisca $(n-1)!$. A questo punto, è sufficiente osservare che `fattRec` restituisce $n \times (n-1)!$, ma per definizione di fattoriale questo è esattamente $n!$.

Ad essere rigorosi dobbiamo infine dimostrare che n decresce ad ogni chiamata e che $n \geq 0$ sotto le ipotesi garantite dalla precondizione.

2.3 Fibonacci Ricorsivo Efficiente

Spero che, visto l'esempio di Fibonacci, nessuno di voi abbia concluso affrettatamente che i programmi ricorsivi siano irrimediabilmente meno efficienti. È infatti possibile, con funzioni ricorsive, mimare il comportamento di ogni funzione iterativa. Nel nostro caso dovremmo quindi usare la ricorsione per mimare il comportamento del seguente ciclo:

```
for (i=2; i<=n; i++){
    fib = fib1 + fib2;
    /* INV: fib = fib(i) & fib1 = fib(i-1) & fib2 = fib(i-2)*/
    fib2 = fib1;
    fib1 = fib;
    }
```

Per quanto riguarda il controllo, è facile rendersi conto che è sufficiente introdurre tra i parametri della funzione ricorsiva un parametro che gioca il ruolo dell'indice `i` e mantenere un parametro col valore di ingresso `n`, e: a) terminare le attivazioni

ricorsive quando il valore di `i` raggiunge il valore di `n`; b) incrementare ad ogni attivazione ricorsiva il valore del parametro `i`.

L'altro problema riguarda come mantenere lo stato della computazione, cioè come sia possibile mantenere i valori via via computati e memorizzati nelle variabili `fib1`, `fib2` e `fib`. Ricordiamo che le variabili dichiarate in una funzione sono *locali* e quindi per ciascuna variabile viene riallocata una nuova copia ad ogni attivazione ricorsiva della funzione, e di conseguenza eventuali assegnazioni a una variabile fatte durante un'attivazione *non influenzano* il valore di quella variabile in una successiva attivazione, o al rientro da quella attivazione.

Come è possibile comunicare i valori via via calcolati alle nuove attivazioni (dal chiamante al chiamato)? La risposta è semplice: usare i parametri. Dovremmo quindi arricchire l'interfaccia della funzione con dei parametri ausiliari che giocheranno un ruolo analogo alle variabili `fib1` e `fib2` nel programma iterativo. Il valore della variabile `fib`, viceversa, sarà comunicato attraverso il valore ritornato dalla funzione (dal chiamato al chiamante).

Siccome però vorremmo scrivere una funzione `fibRecEff` che accetta in input *un solo* parametro intero (e che quindi abbia la stessa interfaccia delle funzioni `fibRec` e `fibIt`), scriveremo una funzione ausiliaria `fibRecEffAux` con i parametri necessari a mimare le operazioni del programma iterativo. La funzione `fibRecEff` si limiterà a trattare i casi base e chiamare la funzione ausiliaria `fibRecEffAux` inizializzando opportunamente i parametri per la prima chiamata, esattamente come la funzione iterativa `fibIt` inizializza opportunamente le variabili `i, fib1` e `fib2` prima dell'ingresso nel ciclo. Ecco quindi il codice delle due funzioni:

```
private static int fibRecEffAux(int n, int i, int fib1, int fib2){
/* REQ: fib1=fib(i) & fib2=fib(i-1) & 2<=i<=n.
 * EFF: ritorna fib(n).
 */
 if (i==n) return fib1;
 return fibRecEffAux(n, i+1, fib1+fib2, fib1);
}

public static int fibRecEff(int n){
/* REQ: n>=0.
 * EFF: ritorna fib(n)
 */
 if (n<2) return n;
 return fibRecEffAux(n,2,1,1);
}
```

Dimostriamo ora per induzione che una chiamata alla funzione `fibRecEff(n)` effettivamente restituisce $\mathsf{fib}(n)$, sotto la precondizione $n \geq 0$. Per $n < 2$ è sufficiente verificare che $n = \mathsf{fib}(n)$ ed effettivamente la funzione restituisce n. Viceversa la

correttezza di `fibRecEff` dipende dalla correttezza di `fibRecEffAux`. Dimostriamo le seguenti cose:

1. La chiamata iniziale `fibRecEffAux(n,2,1,0)` rispetta le precondizioni. Infatti, essendo falsa la condizione dell'`if`, $n \geq 2$ e quindi $2 = i \leq n$. Inoltre $\mathsf{fib}(n) = 1 = fib2$ e $\mathsf{fib}(1) = 1 = fib1$;
2. se $i = n$ e vale la precondizione ($\mathsf{fib}(i) = fib1$), chiaramente la funzione restituisce $\mathsf{fib}(n)$;
3. altrimenti, se $i < n$, induttivamente, la correttezza della chiamata alla funzione `fibRecAuxEff(n,i,fib1,fib2)` dipende solo dalla correttezza della chiamata `fibRecAuxEff(n,i+1,fib1+fib2, fib1)`. L'unica cosa da far vedere è che la nuova chiamata rispetta le precondizioni, ma:
   - $i \leq n \wedge i \neq n$ implica $i + 1 \leq n$;
   - per definizione della funzione di fibonacci, se $fib1 = \mathsf{fib}(i) \wedge fib2 = \mathsf{fib}(i-1)$, allora $fib1 + fib2 = \mathsf{fib}(i+1)$;
   - analogamente, se $fib1 = \mathsf{fib}(i)$ allora avremo che incrementando i di 1 $fib1 = \mathsf{fib}(i-1)$ come richiesto.
4. infine, osserviamo che la sequenza di chiamate ricorsive termina, perché la funzione $n - i$ (positiva sotto le precondizioni) decresce ad ogni chiamata.

Ancora una volta ricordiamo che la funzione `fibRecEff`, pur facendo circa le stesse operazioni di `fibIt`, sarà comunque più inefficiente. Questo perché il passaggio di parametri e l'allocazione dei record di attivazione per una funzione hanno un costo significativo.

Un'ultima osservazione riguarda il fatto che abbiamo definito il metodo statico `fibRecEffAux` di tipo `private`. Anche se non siamo ancora entrati nei dettagli di cosa significhino i *modificatori di visibilità* `public` e `private`, osserviamo che l'utilizzatore di una libreria di funzioni sugli interi, *vuole* usare una funzione che calcola i numeri di fibonacci che prende in input un *unico parametro*. Inoltre, le precondizioni della funzione `fibRecEffAux` sono alquanto complesse e, tutto sommato, imperscrutabili per un utilizzatore esterno. Risulta pertanto *inutile* e anche *inopportuno* renderla disponibile a un potenziale utilizzatore di una libreria di funzioni sui naturali. `private` significa appunto che la funzione sarà visibile *solo all'interno* della classe che stiamo progettando e solo lì potrà essere invocata. È buona norma definire `private` tutti gli eventuali metodi ausiliari scritti in una classe.

Esercizi e Spunti di Riflessione

1. ★È possibile scrivere un programma ricorsivo che renda efficiente il calcolo dei coefficienti binomiali?
2. ★Provate a scrivere un programma iterativo che calcola i coefficienti binomiali.

3. ♣Sulla scorta dell'esempio di fibonacci ricorsivo efficiente, proporre un metodo generale per mimare il comportamento di un qualsiasi ciclo `while` o `for` con chiamate ricorsive.

4. ★♣Provate a immaginare come si potrebbe rendere efficiente il calcolo dei numeri di fibonacci con una funzione che mantiene la struttura ricorsiva di `fibRec`. Questo esercizio vi faccia riflettere sul fatto che è spesso possibile barattare memoria per efficienza.

5. Usare i metodi visti in questa sezione per dimostrare formalmente la correttezza di alcune funzioni ricorsive scritte negli esercizi di sezione 1.2.

6. ▶Calcolare il massimo numero intero per cui `fattRec` (o `fattIt`) dà risultati corretti sul vostro calcolatore. Provare con diversi tipi interi (`int`, `unsigned int`, `long int` e così via).

7. ▶Come nell'esercizio precedente, però definendo la variabile `f` in `fattIt` oppure il valore di ritorno di `fattRec` con un tipo che memorizza ""numeri reali"" (qui le virgolette non sono mai abbastanza!) come `float` o `double`.

3 Soluzioni inerentemente ricorsive

Concludiamo la nostra passeggiata nel mondo incantato della ricorsione con un esempio che mostra come la soluzione naturale di alcuni problemi sia *inerentemente* ricorsiva o induttiva. Non esistono cioè semplici od evidenti soluzioni iterative.

Dovreste aver già intuito a questo punto che il ciclo `while`[5] è sufficiente a calcolare qualunque funzione, e vedremo (studiando la struttura dati *pila*) come sia possibile simulare il comportamento di qualsiasi programma ricorsivo in modo sistematico simulando in modo esplicito il comportamento della pila di attivazione delle chiamate ricorsive. Tuttavia, questo tipo di soluzioni, "moralmente", sono ricorsive.

Per amore di equità va detto, che anche la funzione ricorsiva `fibRecEff` "moralmente" è iterativa, e soprattutto nei programmi con vettori, vedremo problemi la cui soluzione *naturale* è iterativa. Va infine detto che, come visto per `fibRecEff` (ed esercizi collegati), la simulazione di un'iterazione con la ricorsione è decisamente più semplice della simulazione sistematica della ricorsione attraverso l'iterazione.

3.1 La Torre di Hanoi

Forse vi siete già imbattuti nel problema della *Torre di Hanoi* (ci sono eleganti versioni in legno che potrebbero arredare con gusto la casa di un informatico):

[5] con assegnazione, sequenza e le espressioni +1 e test di uguaglianza con 0.

"narra la leggenda che in un tempio Indù alcuni monaci siano costantemente impegnati a spostare su tre colonne di diamante 64 dischi d'oro di diversi diamentri secondo le regole della Torre di Hanoi (a volte chiamata Torre di Brahma): ogni disco può essere spostato da una colonna all'altra senza mai che un disco di diametro maggiore sia posto sopra un disco di diametro inferiore. L'obiettivo è spostare i 64 dischi dalla prima alla terza colonna, facendoli passare eventualmente sulla seconda. I monaci spostano un disco ogni giorno e quando completeranno il lavoro, il mondo finirà."

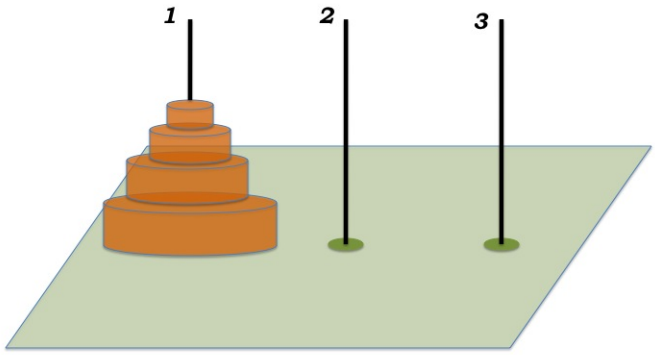

Figura 9: Il problema della Torre di Hanoi con 4 dischi

In Fig. 9 è raffigurato lo stato iniziale del problema. Il rompicapo può banalmente essere risolto con un solo disco. Due facili mosse lo risolvono nel caso di due dischi. Anche andando a tentativi, probabilmente riuscirete a risolverlo anche nel caso di 3 dischi. Tuttavia, al crescere della dimensione del problema (ossia del numero dei dischi), è necessaria una strategia ben precisa.

Confortati dalla soluzione dei casi facili, si può osservare che dovendo da spostare k dischi dal piolo 1 (o *sorgente*) al piolo 3 (o *destinazione*), usando il piolo 2 come *appoggio*, sarà sufficiente spostare $k-1$ dischi dal piolo sorgente al piolo ausiliario, spostare il disco di diametro maggiore rimasto finalmente libero di muoversi dal disco sorgente a quello destinazione, ed infine spostare i $k-1$ dischi dal piolo appoggio al piolo destinazione (sopra il disco di diametro maggiore) usando il piolo sorgente come appoggio. Abbiamo risolto il problema della Torre di Hanoi per un qualsiasi numero di dischi, semplicemente riconducendo una generica istanza del problema con k dischi alla soluzione di due istanze dello stesso problema con $k-1$ dischi e una istanza banale con 1 disco solo da muovere (vedi Fig. 10).

L'unica verifica da fare è che lo spostamento di $k-1$ dischi non viola le regole del gioco, ma ciò è vero, perché l'unico altro disco rimasto è quello di diametro maggiore e quindi può essere ignorato: ogni altro disco gli può essere appoggiato sopra.

L'ultima osservazione da fare, prima di scrivere un programma che risolve questo problema, è che i ruoli dei pioli 1, 2 e 3 cambiano durante la soluzione: ad esempio, il piolo 1 che è il piolo sorgente, viene usato come piolo di appoggio nella soluzione della seconda istanza con $k-1$ dischi. Occorre quindi parametrizzare la nostra soluzione, dimodochè sappia spostare in generale k dischi da un certo piolo sorgente a un piolo destinazione usando un terzo piolo come ausiliario, indipendentemente da quali siano effettivamente i numeri che identificano il piolo sorgente, destinazione e appoggio. Il programma è mostrato in Fig. 11.

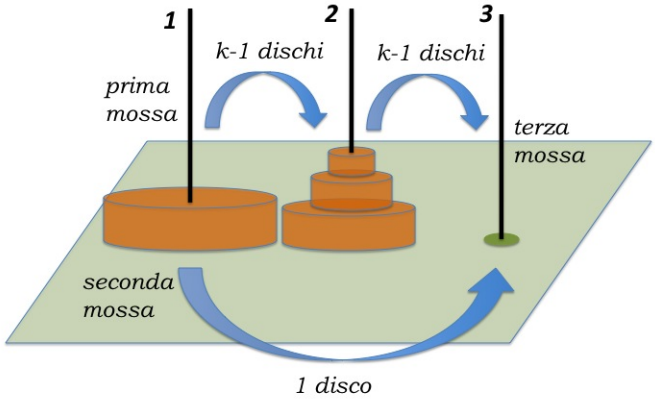

Figura 10: Soluzione del problema della Torre di Hanoi

```
public static void hanoi(int sorg, int aux, int dest, int n){
 /* REQ: n>0. {sorg, aux, dest}={1,2,3}.
  * EFF: sposta n dischi dal piolo sorg al piolo dest,
  *      usando aux come appoggio.
  */
  if (n==1) muovi(sorg, dest);
     else{
        hanoi(sorg, dest, aux, n-1);
        muovi(sorg, dest);
        hanoi(aux, sorg, dest, n-1);
        }
  }
```

Figura 11: Funzione che risolve il problema della Torre di Hanoi

Abbiamo volutamente mantenuto una certa flessibilità su cosa significhi spostare un disco da un piolo `sorg` a un piolo `dest`, invocando una funzione che esegue l'operazione. Potrebbe trattarsi di una procedura che esegue un'animazione e fa vedere il disco sfilarsi dal piolo `sorg` e muoversi lentamente verso il piolo `dest`. Oppure potremmo pensare di ristampare a video ad ogni mossa lo stato delle tre torri. Per esempio la situazione in Fig. 10 potrebbe essere rappresentata con:

```
1 :   4
2 :   3  2  1
3 :   -
```

Al momento, tuttavia non abbiamo a disposizione abbastanza strutture dati per svolgere questo compito. Ci limiteremo a stampare a video le mosse nella forma `sorg-->dest`, quindi la funzione `muovi` come segue:

```
public static void muovi(int s, int d){
    System.out.println(s+"-->"+d);
    }
```

Esercizi e Spunti di Riflessione

1. Dimostrare per induzione che il numero di mosse necessario per risolvere il problema della Torre di Hanoi con n dischi è $2^n - 1$. Dedurre che, anche fosse vera, la leggenda non ha influenza sulle nostre vite! ▶Stimare il tempo che il vostro calcolatore impiegherebbe a risolvere il problema con 64 dischi.

2. Scrivere una funzione TinyJava che presi in input due interi positivi n e k ($n, k \geq 1$) restituisca in output il numero delle k-uple *ordinate* $\langle p_1, \ldots, p_k \rangle$ di interi positivi ($1 \leq p_1 \leq p_2 \leq \ldots \leq p_k$) il cui prodotto è n.

   Esempi: Per ogni k, se n è primo, il risultato deve sempre essere 1. Infatti, se n è primo, l'unica k-upla che dà come prodotto n è $\langle \underbrace{1, 1, 1, \ldots 1}_{k-1}, n \rangle$.

   Se n fosse 12 e k fosse 4, il programma dovrebbe rispondere 4. Infatti ho che 12 può essere ottenuto come prodotto delle seguenti k-uple: $\langle 1, 1, 1, 12 \rangle$, $\langle 1, 1, 2, 6 \rangle$, $\langle 1, 1, 3, 4 \rangle$, $\langle 1, 2, 2, 3 \rangle$.

3. ★Definiamo insieme delle *partizioni ordinate* di un numero n l'insieme delle sequenze di numeri naturali positivi che danno come somma n. Ad esempio, $\text{Part}(4) = \{\langle 4 \rangle, \langle 2, 2 \rangle, \langle 1, 3 \rangle, \langle 3, 1 \rangle, \langle 1, 1, 2 \rangle, \langle 1, 2, 1 \rangle, \langle 2, 1, 1 \rangle, \langle 1, 1, 1, 1 \rangle\}$. Scrivete equazioni che definiscano l'insieme delle partizioni ordinate per induzione su n.

4. ★Definiamo l'insieme $\text{Part}'(n)$ delle *partizioni* di un numero n come l'insieme ottenuto dalle partizioni ordinate eliminando tutte le sequenze che differiscono solo per l'ordine[6]. Ad esempio avremo che $\text{Part}'(4) = \{\{4\}, \{2,2\}, \{1,3\}, \{1,1,2\}, \{1,1,1,1\}\}$. Scrivete equazioni che definiscano l'insieme delle partizioni per induzione su n.

5. ★Sfruttare le equazioni scritte nell'esercizio precedente per scrivere una funzione `public static int partizioni(int n)` che conta il numero delle partizioni che si possono fare di un numero naturale n.

6. ★★Scrivere una funzione che stampa tutte le partizioni di un certo numero n.

7. Ricordiamo che dato un insieme X, l'*insieme delle parti* (o potenza) $\mathcal{P}(X)$ è:

$$\mathcal{P}(X) = \{Y \mid Y \subseteq X\}$$

   Dare una definizione induttiva dell'insieme delle parti che specifichi il significato di $\mathcal{P}(\varnothing)$ (attenzione! – qui c'è una piccola insidia!) e il significato di $\mathcal{P}(X \cup \{a\})$ in termini di $\mathcal{P}(X)$.

8. L'insieme $\mathcal{P}_k(X)$ dei sottoinsiemi di cardinalità fissata k, per $0 \leq k \leq |X|$, è definito come:

$$\mathcal{P}_k(X) = \{Y \mid Y \subseteq X \ \& \ |Y| = k\}$$

   Cominciare dalla soluzione dell'esercizio precedente, per dare una definizione induttiva dell'insieme $\mathcal{P}_k(X)$.

9. Provate a scrivere il codice Java che implementa le equazioni ricorsive scritte nei due precedenti esercizi. Quali "capacità elementari" dovreste supporre nel vostro esecutore per mantenere intatta la "semplicità" delle vostre equazioni ricorsive?

10. ★Riflettere su come sia possibile definire induttivamente l'insieme degli anagrammi di una parola.

[6] ciò equivarrebbe a considerare multiinsiemi piuttosto che sequenze.