Sviluppare Programmi Corretti

Ivano Salvo
Università di Roma "La Sapienza"
email: salvo@di.uniroma1.it

Anno Accademico 2005-06

Presentazione

La presente dispensa si propone di presentare il materiale di un ciclo di esercitazioni di 10 lezioni ed è stata pensata sulla base dell'esperienza di un analogo corso tenuto nell'A.A. 2001-02. L'argomento principale è lo sviluppo di programmi iterativi e ricorsivi corretti, utilizzando *asserzioni logiche*.

Alcuni argomenti trattati, non fanno parte strettamente del corso di esercitazioni, ma appartengono comunque ai temi propri del corso di Programmazione, e sono stati inseriti in quanto propedeutici alle esercitazioni.

Vengono presentati un buon numero di esercizi, il cui svolgimento corretto dovrebbe assicurare un comodo passaggio dell'esame. Esercizi particolarmente difficili vengono segnalati con il simbolo ★, mentre gli esercizi contrassegnati dal simbolo ♣ sono in effetti "complementi" alla teoria presentata e quindi sono utili soprattutto ad una assimilazione "attiva" dei concetti. Il simbolo ★ può anche comparire a segnalare una sezione che richiede un certo sforzo e che, tipicamente, non è stata presentata nelle lezioni in aula.

Si tratta di una bozza e quindi è probabile la presenza di imperfezioni o errori. I lettori sono invitati a segnalare le imperfezioni e gli errori rilevati al docente, in modo da migliorare la presente dispensa.

Parte Prima: Ricorsione, Iterazione, Asserzioni Logiche

1 Ricorsione ed Iterazione

In questa sezione verranno riviste nozioni relative ai meccanismi base di controllo: iterazione e ricorsione, che dovrebbero essere già noti dal precedente corso di Programmazione 1. L'obiettivo è quello di confrontare questi due meccanismi di controllo e introdurre delle metodologie per valutare la correttezza dei programmi. Accompagneremo le considerazioni e l'introduzione di nuovi concetti con due semplici esempi.

1.1 La funzione fattoriale

La funzione fattoriale, indicata in matematica con il simbolo !, 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 di un intero $n+1$ 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.

Scrivere una funzione `C` iterativa che calcola il fattoriale è molto facile e segue uno schema di programmazione ben noto, simile a quello necessario per scrivere un programma che calcola una sommatoria. 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!).

```
int fattIt(int n)
{ int f=1; /* definizione della variabile accumulatore */
  int i=0; /* variabile contatore */

  while (i!=n)
  { i++;
    f *= i;
   }
  return f;
}
```

La versione ricorsiva, viceversa, è sostanzialmente una traduzione in `C` della definizione induttiva del fattoriale. Vedremo numerosi altri esempi, in cui un programma `C` sarà analogo ad una definizione induttiva.

```
int fattRec(int n)
{ if (n==0) return 1; /* caso base */
      else return n*fattRec(n-1); /* passo induttivo */
}
```

Le equazioni ricorsive che definiscono il fattoriale sono semplicemente state tradotte in linguaggio `C`: i diversi casi della definizione vengono discriminati da un costrutto di tipo `if ... else...` e abbiamo sostituito la notazione postfissa del simbolo di fattoriale ! con le chiamate ricorsive alla funzione `fattRec`.

Da un punto di vista concreto le due funzioni eseguono esattamente le stesse operazioni (n prodotti). Per motivi che saranno visti meglio più avanti 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).

Nel caso specifico non è particolarmente difficile capire cosa calcola la funzione `fattIt`, ma vedremo più avanti esempi in cui la versione iterativa di una corrispondente funzione ricorsiva non è affatto semplice da trovare (ad esempio il problema della Torre di Hanoi, considerato in una successiva sezione). Questo perchè 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 fibonacci

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

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

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

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 `C`, come nel caso del fattoriale:

```
int fibRec(int n)
{ if (n<=1) return n; /* caso base */
      else return fibRec(n-1) + fibRec(n-1); /* passo induttivo */
}
```

[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.

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 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.

```
int fibIt(int n)
{ inf fib1=0; /* variabile per il penultimo numero calcolato */
  int fib2=1; /* variabile per l'ultimo numero calcolato */
  int finbew; /* variabile per memorizzare il nuovo numero calcolato */
  int i = 2;

  if (n<2) return n;
  while (i<=n)
  { fibnew = fib1 + fib2;
    fib1 = fib2;
    fib2 = fibnew;
    i++;
   }
   return fibnew;
 }
```

A differenza del caso del fattoriale, 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. 1.2 viene esemplificato l'albero delle chiamate ricorsive generato per effetto di una chiamata a `fibRec(4)`.

E' facile dimostrare (per induzione!) che il calcolo di $fib(1)$ e $fib(0)$ verranno invocati rispettivamente $fib(n)$ e $fib(n-1)$ volte (vedi figura 1.2), 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.

2 Introduzione all'uso di asserzioni logiche

A questo punto risulta interessante chiedersi: è possibile dimostrare che le funzioni scritte calcolano esattamente quanto desiderato? Cosa succede quando vengono forniti input incoerenti (ad esempio viene chiesto di calcolare il fattoriale di un numero negativo)?

Vedremo che sarà opportuno completare la specifica di una procedura con dei commenti che esprimono cosa viene calcolato da una procedura (*postcondizioni*), sotto opportune assunzioni sui dati in ingresso (*precondizioni*): questo automaticamente, vedremo, fornisce una tecnica di prova per dimostrare la correttezza delle funzioni ricorsive. Infine vedremo una tecnica di prova per i programmi iterativi (*invarianti*).

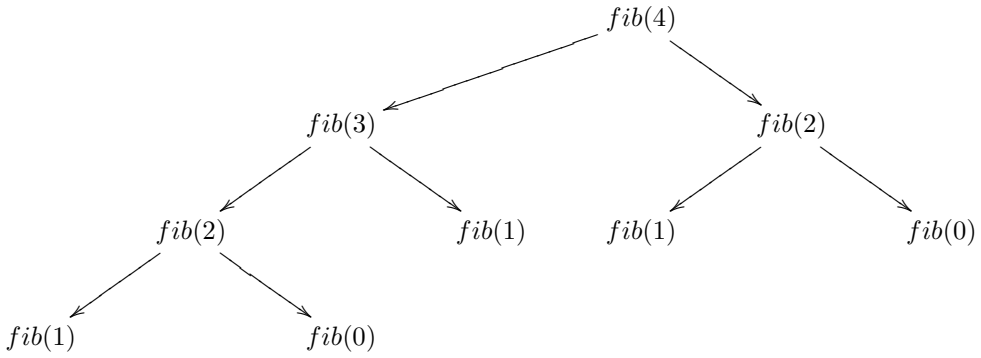

Figure 1: Attivazione delle chiamate ricorsive della funzione `fibRec(4)`

2.1 Precondizioni

Ritorniamo ai codici della funzione `fattIt` e `fattRec`. Cosa accade qualora il parametro di ingresso sia negativo? In entrambi i casi, le funzioni non terminano, in quanto in `fattIt` il parametro `i` viene inizializzato a 0 e continua a crescere non raggiungendo mai il valore negativo `n`, mentre `fattRec` continua a chiamare se stessa su valori più piccoli senza incontrare mai la condizione di chiusura della chiamate ricorsive, causando probabilmente un errore di sistema (*stack overflow*).

Si tratta di un comportamento anomalo ed indesiderato, ma prevedibile visto che il fattoriale è definito solo su interi positivi. Le due funzioni funzionano correttamente solo quando è verificata la asserzione logica $n \geq 0$. Un'asserzione che specifica le proprietà che si devono verificare al momento dell'attivazione di una procedura o funzione affinché questa dia i risultati desiderati si chiama *precondizione* oppure *asserzione iniziale*. Scriveremo le precondizioni come commenti dentro il codice sorgente dei programmi, come segue:

```
int fattRec(int n)
/* PREC: n >= 0
 */
{ if (n==0) return 1;              /* caso base */
       else return n*fattRec(n-1); /* passo induttivo */
}
```

In questo modo il programmatore che definisce la funzione `fattRec` specifica oltre che al suo codice, anche il suo corretto utilizzo[2]: in tal modo un programmatore utente della definizione della funzione `fattRec` è informato che dovrà evitare chiamate scorrette che non rispettano la precondizione. Non solo, ma anche il programmatore 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

[2]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

stessa proprietà: infatti se $n = 0$ non attivo alcuna chiamata ricorsiva, mentre $n > 0$ implica $n - 1 \geq 0$.

2.2 Postcondizioni

L'altra clausola contrattuale a cui deve adempiere chi definisce una funzione, è la specifica di quanto la funzione calcola, ovviamente sotto le ipotesi descritte nella precondizione. Questa asserzione logica viene detta *postcondizione* oppure *asserzione finale*. Scriveremo anche queste dentro il testo del programma sotto forma di commento:

```
int fattRec(int n)
/* PREC: n >= 0
 * POST: ritorna n!
 */
{ if (n==0) return 1;              /* caso base */
       else return n*fattRec(n-1); /* passo induttivo */
}
```

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

Nel nostro caso dobbiamo dimostrare che, ricevendo in input un intero positivo, la funzione `fattRec` ne restituisce il fattoriale. Cioè che sotto le ipotesi della precondizione, la funzione garantisce la postocondizione, assumendo che le chiamate ricorsive restituiscano valori corretti (bisognerà comunque verificare che le chiamate ricorsive rispettano la precondizione). 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 $n!$.

2.3 Invarianti

Vediamo infine come sia possibile stabilire la correttezza di un programma iterativo. Un ciclo produce la ripetizione di un blocco di comandi, fino al verificarsi di una condizione: di conseguenza si tratta di ripetere la stessa computazione (al variare eventualmente del valore di alcune variabili): quindi è naturale pensare che durante l'esecuzione di un ciclo ci siano delle proprietà che rimangono invarianti (e che anch'esse verosimilmente dipendono dal valore delle variabili che vengono modificate durante il ciclo).

Un *invariante* è una asserzione logica che esprime una proprietà sempre verificata durante l'esecuzione di un ciclo. Vediamo come si può valutare la correttezza di un ciclo, riprendendo le versioni iterative delle funzioni che calcolano rispettivamente il fattoriale e l'ennesimo numero della successione di fibonacci. Scriveremo anche gli invarianti, come le altre annotazioni logiche (precondizioni e postcondizioni) come commenti all'interno dei programmi. Ecco di nuovo la funzione fattoriale opportunamente annotata:

```
int fattIt(int n)
/* PREC: n >= 0
 * POST: ritorna n!
 */
{ int f=1; /* definizione della variabile accumulatore */
  int i=0; /* variabile contatore */

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

Informalmente è facile osservare che prima dell'esecuzione del primo ciclo, la variabile `f` contiene il valore 1 ed `i` il valore 0. Quindi, la proprietà invariante è verificata all'ingresso del ciclo. Ad ogni iterazione, se all'inizio del ciclo la variable `f` contiene il valore `i`!, si incrementerà di 1 la variabile `i` ed `f` viene riaggiornata opportunamente.

Nella prossima sezione, vedremo brevemente un *sistema formale* per studiare la correttezza dei programmi, espresso in termini di regole logiche. Concludiamo questa sottosezione, vedendo il programma iterativo che calcola l'ennesimo numero di fibonacci, annotato con gli opportuni invarianti.

```
int fibIt(int n)
{ inf fib1=0;    /* variabile per il penultimo numero calcolato */
  int fib2=1;    /* variabile per l'ultimo numero calcolato */
  int finbew =1; /* variabile per memorizzare il nuovo numero calcolato */
  int i = 2;

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

2.4 Terminazione

Una proprietà auspicabile dei programmi è la *terminazione*[3]. Un programma termina se dopo un numero finito di passi la sua computazione termina. L'esempio più banale di

[3]in realtà esistono alcune applicazioni in cui è utile considerare processi che non terminano, ad esempio nel campo dei sistemi operativi o dei controlli dei sistemi. Pensate banalmente ad un qualsiasi programma che

programma non terminante è il seguente:

```
int beato ()
{ while (1) {}
   }
```

che è il programma che non fa nulla per l'eternità. Osservate che in generale un programma può fare un numero *illimitato* di passi, quindi non prevedibile a priori[4]. Vedremo ora, una semplice tecnica per dimostrare che un programma termina, applicabile sia nel caso dei cicli che nel caso delle funzioni ricorsive. Occupiamoci del primo caso, lasciando il secondo caso alle riflessioni del lettore.

Per dimostrare che un ciclo termina, è sufficiente trovare una funzione, detta *funzione di terminazione*, dipendente dallo stato della computazione (dipenderà tipicamente dal valore delle variabili modificate nel corpo del ciclo e/o coinvolte nella guardia) sempre positiva e sempre decrescente durante il ciclo. Più formalmente:

Definizione 2.1 [Funzione di Terminazione] Sia φ una proprietà invariante per il ciclo `while (b) C`. t è detta funzione di terminazione se gode delle seguenti proprietà (indichiamo con b l'asserzione logica associata alla guardia del ciclo):

1. $\varphi \wedge b \Rightarrow t \geq 0$
2. t decresce ad ogni iterazione;

Teorema 2.2 *Supponiamo esista una funzione di terminazione t per il ciclo* `while (b) C`. *Allora il ciclo termina dopo un numero finito di iterazioni.*

Dim: Supponiamo esista la funzione t con le proprietà della definizione 2.1 e che il ciclo non termini. Allora, indicando con S_i $(i \in N)$ lo stato all'i-esima iterazione, si avrebbe una successione infinita strettamente decrescente di numeri naturali $t(S_0) > t(S_1) > \ldots t(S_n) > t(S_{n+1}) > \ldots > 0$. Il che è assurdo, perché, per le proprietà dei numeri naturali, nei numeri naturali non esistono sequenze infinite strettamente decrescenti. ✓

Lasciamo al lettore lo studio della terminazione per funzioni definite ricorsivamente: in tal caso, la funzione di terminazione dipende dai parametri, e in particolare dalle variabili che occorrono nelle condizioni sotto le quali vengono attivate nuove chiamate ricorsive

2.5 Triple di Hoare

Vedremo brevemente come sia possibile *formalizzare* i ragionamenti visti sopra con un sistema di prova che permette di effettuare dimostrazioni formali. La formalizzazione rigorosa dei ragionamenti sulle asserzioni logiche avviene attraverso regole di deduzione che specificano come i programmi possano soddisfare una certa asserzione logica. Ciò permette di

gestisce l'interazione col calcolatore. Anche in questi casi, tuttavia ci sono nozioni simili alla terminazione: ad esempio è auspicabile che un programma, anche non terminante, comunque dopo un numero finito di passi torni in uno stato in cui interagisce con i dispositivi di input/output.

[4]In generale infatti, non è possibile scrivere un programma che preso in input un altro programma, dica se questo termina o meno.

dare dimostrazioni formali, e di conseguenza verificabili *meccanicamente*. L'idea è quella di provare enunciati del tipo:

$$\psi \quad \texttt{C} \quad \varphi$$

che hanno il seguente significato intuitivo: se ψ è una asserzione logica soddisfatta dallo stato (ψ dipende in genere da valori assunti dalle variabili usate dal programma), allora l'asserzione φ sarà soddisfatta nello stato ottenuto dopo l'esecuzione del comando `C`, nel caso in cui `C` *termini*. Le regole di deduzione sono date per induzione sulla *struttura sintattica* dei comandi.

Osserviamo che useremo la seguente convenzione: se `x` è una variabile, allora nell'asserzione logica x avrà il valore contenuto nella variabile `x`. Inoltre indicheremo con $\varphi[x]$ una formula logica che dipende da x e con $\varphi[E/x]$ la formula logica ottenuta rimpiazzando le eventuali occorrenze di x con l'espressione E.

$$\frac{\varphi\ \{\texttt{C}\}\ \psi \quad \psi\ \{\texttt{D}\}\ \phi}{\varphi\ \{\texttt{C;D}\}\ \phi}\ (\textsc{Sequenza}) \qquad \frac{\varphi \wedge b\ \{\texttt{C}\}\ \varphi \qquad \varphi \wedge \neg b \Rightarrow \psi}{\varphi\ \{\texttt{while(b) C}\}\ \psi}\ (\textsc{While})$$

$$\frac{\varphi \wedge b\ \{\texttt{C}\}\ \psi \quad \varphi \wedge \neg b\ \{\texttt{D}\}\ \psi}{\varphi\ \{\texttt{if (b) C else D}\}\ \psi}\ (\textsc{If}) \qquad \frac{}{\varphi[E/x]\ \{\texttt{x = E}\}\ \varphi[x]}\ (\textsc{Assegnazione})$$

Figure 2: Triple di Hoare (Regole fondamentali)

2.6 Esempio: Fattoriale

Occupiamoci ora di dimostrare formalmente che la funzione `fattIt` scritta prima è corretta. Ispirati dalla regola (While), dobbiamo far vedere che:

1. l'invariante $f = i!$ è verificato all'ingresso del ciclo;
2. l'invariante è mantenuto dai comandi nel corpo del ciclo;
3. l'invariante e la negazione della guardia implicano la postcondizione;
4. il ciclo termina.

All'ingresso del ciclo, le istruzioni `f=1; i=0` producono uno stato in cui l'invariante è soddisfatto, in quanto per definizione di fattoriale $1 = 0!$. Per la regola assegnazione, ho:

$$\begin{array}{ccc} f = i! & \{\texttt{i++}\} & f = (i-1)! \\ f = (i-1)! & \{\texttt{f=f*i}\} & f = i! \end{array}$$

Sono nelle condizioni di poter applicare la regola (Sequenza), che mi permette di concludere:

$$f = i! \quad \{\texttt{i++; f=f*i}\} \quad f = i!$$

La negazione della guardia del ciclo while, `(i!=n)`, è appunto $i = n$, che consente di sostituire n ad i nella proprietà invariante ed ottenere, quanto desiderato, cioè la postcondizione $f = n!$.

L'ultima cosa da verificare, è la terminazione del ciclo: consideriamo la funzione $n - i$: è positiva per $0 \leq i \leq n$. Inoltre è sempre strettamente decrescente, perché la variabile `i` viene incrementata nel corpo del ciclo. Per dimostrare la terminazione, l'invariante $f = i!$ non è sufficiente, ed è quindi necessario far vedere che anche l'asserzione $0 \leq i \leq n$ è invariante per il ciclo in questione. Osserviamo che la precondizione $n > 0$ e l'inizializzazione di i a 0, garantiscono che l'asserzione $0 \leq i \leq n$ sia soddisfatta all'ingresso del ciclo, mentre l'applicazione della regola per l'assegnazione permette di far vedere che si tratta di una proprietà invariante, cioè che l'enunciato:

$$0 \leq i \leq n \wedge i \neq n \quad \{\texttt{i++; f=f*i}\} \quad 0 \leq i \leq n$$

è derivabile. Infatti, $i \leq n \wedge i \neq n$ implica $i < n$ e quindi $i + 1 \leq n$. Applicando la regola per l'assegnazione, abbiamo $0 \leq i + 1 \leq n$ $\{\texttt{i++}\}$ $0 \leq i \leq n$. Non è necessario ragionare sull'altro comando del corpo del ciclo, in quanto l'assegnazione di `f` non influenza la preservazione di questa proprietà invariante.

Esercizi

1. Dimostrare per induzione che la chiamata di funzione `fibRec(n)` restituisce l'n-esimo numero di fibonacci.

2. Dare un argomento informale del fatto che una chiamata alla funzione `fibRec(n)` genera $fib(n)$ chiamate del tipo `fibRec(1)`.

3. Dimostrare per induzione che una chiamata alla funzione `fibRec(n)` genera $fib(n)$ chiamate del tipo `fibRec(1)` e $fib(n-1)$ chiamate del tipo `fibRec(0)`.

4. Scrivere una funzione *ricorsiva* che calcoli la funzione di fibonacci con un numero lineare di operazioni. [**Sugg**: usare dei parametri ausiliari nelle chiamate ricorsive per simulare lo stato analogamente alla versione iterativa]

5. ♣Si generalizzi la soluzione dell'esercizio precedente descrivendo un metodo generale per trasformare una funzione iterativa in ricorsiva.

6. (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 `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.

7. Scrivere una funzione iterativa ed una ricorsiva che calcolino il Massimo Comun Divisore, sfruttando l'algoritmo di Euclide. Corredare le funzioni di opportuni invarianti,

funzione di terminazione, precondizioni e postcondizioni. Provarne la correttezza. Ricordiamo che l'algoritmo di Euclide si fonda sulle seguenti equaglianze ($n, m > 0$):

$$\begin{array}{lcll} \mathsf{mcd}(n,n) & = & n & \\ \mathsf{mcd}(m,n) & = & \mathsf{mcd}(m-n,n) & \text{se } m > n \\ \mathsf{mcd}(m,n) & = & \mathsf{mcd}(n-m,m) & \text{se } n > m \end{array}$$

8. Scrivere una funzione iterativa e ricorsiva che calcolino l'esponenziale, corredando le funzioni di opportune asserzioni.

9. Scrivere una funzione iterativa ed una ricorsiva che calcolino l'esponenziale sfruttando le seguenti uguaglianze:

$$\begin{array}{rcl} m^{2n} & = & (m \times m)^n \\ m^{2n+1} & = & m \times m^{2n} \\ m^0 & = & 1 \end{array}$$

   Valutare il numero di prodotti e somme richieste e confrontare con l'algoritmo fornito all'esercizio precedente.

10. Scrivere una funzione `C` che non contiene cicli, che non termina.

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

12. ♣Scrivere le regole di Hoare per i costrutti:

    (a) `do C while (b)`

    (b) `for (i=E, b, i++) C`

    (c)
    ```
    switch (c)
    { v1 : C1
      ...
      vn : CN
    }
    ```

    [**Sugg**: consultare un manuale del linguaggio `C`!]

13. ♣Scrivere le regole di Hoare per la chiamata di funzione (supporre che la funzione non modifichi lo stato globale!) [**Sugg**: usare precondizioni e postcondizioni].

3 Uso di invarianti in programmi su Array

Approfondiremo in questa sezione l'uso di asserzioni logiche nello studio della correttezza dei programmi, presentando alcuni semplici esempi di problemi sugli array; negli ultimi due esempi mostreremo l'uso degli invarianti non solo per dimostrare la correttezza del programma, ma come guida nello sviluppo del programma. Due esercizi svolti concluderanno questa prima parte.

3.1 Il problema del minimo

Il problema è molto semplice e consiste nel restituire l'elemento minimo di un vettore. La specifica della funzione sarà del tipo:

```
int minimo(int A[], int n)
/* PREC: n=lunghezza di A, n>0
 * POST: ritorna il valore min, tale che: per ogni 0<=i<n, min <= A[i]
 */
```

L'idea per risolvere questo problema è molto semplice ed è comune a molti programmi sui vettori: scandire il vettore da sinistra a destra memorizzando il minimo calcolato fino al punto in cui è arrivata la scansione del vettore. Più in dettaglio, si introduce una variabile `min` che viene inizializzata al valore di `A[0]` (che in effetti è anche il minimo del sottovettore contenente solo l'elemento `A[0]`). Si scandisce il vettore usando un indice `i`, avendo l'accortezza di riaggiornare la variabile `min` ogni qualvolta si trova un elemento `A[i]` che contiene un valore minore di quello già memorizzato in `min`. Forse è più semplice vedere il codice, corredato dell'opportuno invariante:

```
int minimo(int A[], int n)
/* PREC: n=lunghezza di A, n>0
 * POST: ritorna il valore min, tale che: per ogni 0<=i<n, min <= A[i]
 */
{ int min;
  int i = 0;

  min = A[0]; i=1;
  while (i < n)
  /* INV: forall j. 0<=j<i min<=A[i] */
  { if (A[i] < min) min=A[i];
    i++;
  }

  return min;
}
```

E' relativamente facile verificare che l'invariante è verificato all'entrata del ciclo. Infatti, esiste un solo j, tale che $0 \leq j < 1$, e la variabile `min` vale esattamente `A[0]`. Altrettanto evidente è che l'invariante e la negazione della guardia $(i = n)$ implicano l'asserzione finale. Esattamente come nel caso del fattoriale, $n - i$ è una funzione di terminazione, e sempre come nel caso del fattoriale si mostra che decresce ad ogni iterazione e che assume valori positivi se la guardia del ciclo è verificata $(i < n)$.

Infine è necessario verificare che l'invariante viene mantenuto dalla sequenza di comandi contenuta nel corpo del ciclo. Per applicare la regola (If) del sistema di deduzione delle triple di Hoare, occorre prima osservare che:

$$(\forall j.0 \leq j < i.min \leq A[j]) \wedge (A[i] < min) \quad \{\texttt{min = A[j]}\} \quad \forall j.0 \leq j < i+1.min \leq A[j] \quad (1)$$

e:

$$(\forall j.0 \leq j < i.min \leq A[j]) \wedge (min \leq A[j]) \quad \{\,\} \quad \forall j.0 \leq j < i+1.min \leq A[j] \quad (2)$$

(1) deriva dalla proprieta' transitiva del $\leq$. Infatti, $\forall j.0 \leq j < i.min \leq A[j] \wedge A[i] < min$ implica $\forall j.0 \leq j < i+1.A[i] \leq A[j]$. A questo punto è sufficiente sostituire min ad $A[j]$ come prescritto dalla regola (ASSEGNAZIONE) per ottenere la tripla (1).

Per (2), trattandosi del comando vuoto, è sufficiente osservare che l'asserzione sinistra implica la destra.

Usando (1) e (2) come premesse, dalla regola (IF) deduciamo che:

$$\forall j.0 \leq j < i.min \leq A[j] \quad \{\texttt{if (A[i] < min) min=A[i]}\} \quad \forall j.0 \leq j < i+1.min \leq A[j]$$

Applicando la regola dell'assegnazione, abbiamo anche:

$$\forall j.0 \leq j < i+1.min \leq A[j] \quad \{\texttt{i++}\} \quad \forall j.0 \leq j < i.min \leq A[j]$$

e l'asserzione destra è proprio la proprietà invariante. Ora, siamo nella condizione per applicare la regola (SEQUENZA), ottenendo esattamente quanto desiderato:

$$\forall j.0 \leq j < i.min \leq A[j] \quad \{\texttt{if (A[i] < min) min=A[i]; i++}\} \quad \forall j.0 \leq j < i.min \leq A[j]$$

3.2 Uguaglianza di due vettori

Vediamo ora un problema simile, cioè dire se due vettori contengono gli stessi elementi nelle stesse posizioni. I parametri di ingresso saranno i due vettori e la loro lunghezza. Cominciamo con lo scrivere la precondizioni e postcondizioni. La funzione ha senso solo se i due vettori hanno la stessa lunghezza positiva (eventualmente nulla. I vettori di lunghezza zero sono tutti uguali!) Supponiamo di ritornare il valore di una variabile booleana `eq`. Consideriamo le seguenti due asserzioni logiche:

$$\begin{aligned} \varphi &\equiv \forall i.0 \leq i < n.A[i] = B[i] \\ \psi &\equiv \exists j.0 \leq i < n.A[i] \neq B[i] \end{aligned}$$

Osserviamo che $\varphi = \neg\psi$, e ci aspettiamo che prima di restituire `eq`, valga la condizione:

$$(eq \wedge \varphi) \vee (\neg eq \wedge \psi)$$

Il fatto che i vettori A e B soddisfino φ, puo' essere stabilito solo verificando che *tutte* le uguaglianze prescritte dal quantificatore universale (su un dominio finito!) siano soddisfatte. Quindi, l'asserzione finale suggerisce che è necessario verificare (uno alla volta) che gli elementi dei due vettori sono a due a due uguali, continuando fino a quando φ è soddisfatta nella porzione di vettore analizzata. La verifica avrà termine quando ha luogo una delle due seguenti circostanze:

1. sono stati scanditi fino in fondo i vettori;
2. è stata trovata una coppia di elementi diversi, cioè $A[i] \neq B[i] \wedge i < n$.

Nel primo caso (riconoscibile dal fatto che l'indice `i` che scandisce il vettore ha raggiunto il valore n), significa che i due vettori sono uguali. Altrimenti, se sono uscito perché ho trovato due elementi diversi, l'indice su cui mi sono fermanto è strettamente più piccolo del valore di n (ricordate che in `C` un vettore di lunghezza n ha gli elementi identificati dalle posizioni $1, \ldots, n-1$ permette di dedurre la proprietà esistenziale ψ.

```
int uguali(int A[], int B[], int n)
/* PREC: n= lunghezza di A e B, n>=0
 * POST: ritorna 1 se 0<=i<n, A[i]=B[i], 0 se esiste i. A[i]!=B[i]
 */
{ int i=0;
  int eq;

  while (i<n && A[i]==B[i]) i++;
  /* INV: forall j. 0<=j<i A[j]==B[j] */

  eq = (i==n);
  return eq;
}
```

Per verificare che φ è un invariante per il ciclo, cioè: $\phi \wedge b$ {i++} φ (dove $b \equiv i < n \wedge A[i] = B[i]$ è la asserzione logica espressa dalla guardia) è sufficiente osservare che se due vettori sono uguali fino alla posizione $i-1$ e $A[i] = B[i]$, allora sono uguali fino alla posizione i, quindi $\varphi \wedge b$ implica $\forall j.0 \leq j < i+1.A[j] = B[j]$. A questo punto, la regola (ASSEGNAZIONE) permette di derivare la tripla:

$$\forall j.0 \leq j < i+1.A[j] = B[j] \quad \{\texttt{i++}\} \quad \forall j.0 \leq j < i.A[j] = B[j]$$

Ancora una volta, la terminazione è provata dalla funzione $n-i$ e dalla proprietà invariante $0 \leq i < n$.

Infine, l'asserzione finale è soddisfatta, osservando che, all'uscita del ciclo:

$$\begin{array}{rcll} eq & \equiv & i = n & \Rightarrow \quad \forall i.0 \leq i < n.A[i] = B[i] \\ \neg eq & \equiv & i < n & \Rightarrow \quad A[i] \neq B[i] \Rightarrow \exists i.0 \leq i < n.A[i] \neq B[i] \end{array}$$

3.3 Ricerca Binaria

Il ben noto problema proposto in questa sezione, consiste nel trovare un elemento all'interno di un *vettore ordinato* restituendo la sua posizione se viene trovato, oppure -1 in caso contrario. L'algoritmo di *ricerca binaria* sfrutta la seguente idea: controllo l'elemento mediano, se questo è l'elemento cercato, allora ho finito. Se è minore dell'elemento cercato, allora devo andare in cerca nella parte destra, altrimenti nella parte sinistra. In qualche modo, questo algoritmo è analogo al modo con cui cerchiamo una parola nel dizionario[5].

Allora mettiamoci in un'ipotesi leggermente semplificata e supponiamo di sapere che l'elemento cercato abbia un valore compreso tra il minimo ed il massimo del vettore; siccome il vettore è per ipotesi ordinato, avrò che è verificata la seguente relazione (A è il nome del vettore e *elem* l'elemento cercato):

$$A[0] \leq elem \leq A[n-1] \tag{3}$$

Ricordiamoci il nostro scopo: dimezzare l'intervallo di ricerca, assicurandoci che l'elemento cercato stia nell'intervallo scelto. Sarà quindi necessario introdurre due variabili, chiamate

[5] in raltà un essere umano fa delle ipotesi iniziali più forti su dove probabilmente la parola si trova, basate sull'esperienza di precedenti consultazioni e quindi si dirige sicuro verso la fine se ad esempio cerca una parola che comincia con la lettera "t"

inf e *sup* che delimitano l'intervallo di ricerca $[inf, sup]$. Vogliamo continuare la ricerca assicurandoci che valga l'asserzione:

$$A[inf] \leq elem \leq A[sup] \tag{4}$$

che chiaramente è un'ottima candidata ad essere l'invariante del nostro ciclo.

L'ipotesi semplificativa (3) è facilmente assicurabile, anche se non prevista dalle precondizioni. Infatti se $elem < A[0]$ oppure $elem > A[n-1]$ possiamo concludere subito (visto che A è ordinato) che $\forall i.0 \leq i < n.A[i] \neq elem$ e ritornare il valore -1 senza ulteriori analisi. Avendo la proprietà (3), possiamo inizializzare una variabile `inf` a `0` e una `sup` a `n-1`, avendo (per sostituzione) soddisfatta (4).

Rimane il problema di come riaggiornare gli estremi dell'intervallo di ricerca; innanzittutto cerchiamo il punto medio. A patto che $inf \leq sup$, il comando: `m = (inf + sup) / 2` calcola il punto medio usando la divisione intera (se la variabile `m` è dichiarata intera). Osserviamo che la condizione $inf \leq sup$ esprime il fatto che il nostro intervallo di ricerca non sia vuoto. Si tratta quindi di una perfetta candidata al ruolo di guardia del ciclo. Se $A[m] = elem$, ho chiaramente finito la ricerca e restituisco il valore della variabile `m`. Altrimenti è sufficiente osservare che, sotto le ipotesi che il vettore sia ordinato:

$$\begin{array}{lcl} (A[m] < elem) \wedge (A[inf] \leq elem \leq A[sup]) & \Rightarrow & A[m] < elem \leq A[sup] \\ (A[m] > elem) \wedge (A[inf] \leq elem \leq A[sup]) & \Rightarrow & A[inf] \leq elem < A[m] \end{array}$$

E quindi, usando nel primo caso l'assegnazione `inf = m+1` e nel secondo `sup = m-1` mantengo la proprietà (4). Riassumendo formalmente in triple di Hoare, ho:

$$\begin{array}{lll} A[m] < elem \leq A[sup] & \{\texttt{inf = m+1}\} & A[inf] \leq elem \leq A[sup] \\ A[inf] \leq elem < A[m] & \{\texttt{sup = m-1}\} & A[inf] \leq elem \leq A[sup] \end{array}$$

a patto che $A[m] \neq elem$. Quindi applicando la regola (IF), ho:

$$A[m] \neq elem \quad \{\ \texttt{if (A[i]<elem) inf=m+1; else sup=m-1;}\} \quad A[inf] \leq elem \leq A[sup]$$

Nell'altro caso, $A[m] = elem$, è sufficiente uscire vittoriosi dal ciclo, senza preoccuparsi di mantenere l'invariante, quindi:

$$\begin{array}{l} (A[inf] \leq elem \leq A[sup]) \wedge (inf \leq sup) \\ \quad \{\ \texttt{m=(inf+sup) / 2;} \\ \quad\ \ \texttt{if (A[i]==elem) break;} \\ \quad\quad \texttt{else if (A[i]<elem) inf=m+1;} \\ \quad\quad \texttt{else sup=m-1}\} \\ \qquad\qquad\qquad\qquad\qquad\qquad A[inf] \leq elem \leq A[sup] \end{array}$$

Rimangono due problemi. La terminazione e come capire se siamo usciti dal ciclo perché abbiamo trovato l'elemento o perché abbiamo verificato che l'elemento non c'è. La terminazione si dimostra semplicemente prendendo la funzione $t = sup - inf$. E' ovvio che: $inf \leq sup \Rightarrow inf \leq \frac{inf+sup}{2} \leq sup$. E per sostituzione ho la validità della tripla di Hoare:

$$inf \leq sup \quad \{\texttt{m=(inf+sup) / 2}\} \quad inf \leq m \leq sup$$

che implica chiaramente le seguenti due relazioni:

$$sup - (m+1) < sup - inf \qquad (m-1) - inf < sup - inf$$

che implicano, che la funzione t sia strettamente decrescente sotto le condizioni sempre verificate all'interno del ciclo. Il lettore può allenarsi a derivare formalmente questa proprietà applicando le regole di Hoare (Assegnazione) e (If), usando le relazioni scritte sopra.

Chiaramente la condizione $inf > sup$ implica che siamo usciti dal ciclo perchè non abbiamo trovato l'elemento ed abbiamo esaurito lo spazio di ricerca. Siamo finalmente pronti a scrivere il programma completo:

```
int ricBinIt(int A[], int n, int elem)
/* PREC: n= lunghezza di A, n>=0, ord(A)
 * POST: ritorna k se A[k]=elem, -1 altrimenti
 */
{ int m;
  int inf;
  int sup;

  if (elem<A[0] || elem>A[n-1]) return -1;

  while (inf <= sup)
  /* INV: A[inf]<=elem<=A[sup]
   * TERM: t=sup-inf */
  { m = (inf+sup) / 2;
    if (elem==A[m]) break;
       else if (elem<A[m]) inf=m+1;
       else sup=m-1;
   }
   if (inf>sup) return -1 else return m;
}
```

Il lettore è invitato a dimostrare che il comando:

```
if (elem<A[0] || elem>A[n-1]) return -1;
```

è di fatto inutile, e la funzione darebbe comunque risultati corretti. La dimostrazione, segue gli stessi passi, usando però il seguente invariante un po' più complicato:

$$\exists k.A[k] = elem \Rightarrow inf \leq k \leq sup$$

che esprime il fatto che se $elem$ c'è nel vettore, allora l'indice che occupa sta nell'intervallo che abbiamo selezionato.

3.4 Ricerca Binaria Ricorsiva

L'algoritmo di ricerca binaria si esprime in modo estremamente naturale in modo ricorsivo: supponiamo di sapere che l'elemento da cercare stia tra inf e sup. Trova il punto medio m. Se $A[m]$ e' l'elemento giusto, ritorna m altrimenti cerca nuovamente nell'intervallo $[m+1, sup]$ se $A[m] < elem$ oppure nell'intervallo $[inf, m-1]$ se $A[m] > elem$.

Vediamo il codice, come sempre annotato con le relative asserzioni logiche:

```
int ricBinRec(int A[], int inf, int sup, int elem)
/* PREC: ord(A), se esiste k. A[k]=elem allora inf<=k<=sup
 * POST: ritorna k se A[k]=elem, -1 altrimenti
 */
{ int m;

  if (inf>sup) return -1;
  m = (inf + sup) / 2;
  if (elem==A[m]) return m;
       else if (elem<A[m]) return ricBinRec(A, m+1, sup, elem);
                      else return ricBinRec(A, inf, m-1, elem);
}
```

Se provate a dimostrare la correttezza, osserverete che i ragionamenti non sono molti diversi da quelli fatti per il programma iterativo, solo che sarà leggermente più semplice.

3.5 ⋆Uguaglianza di due vettori a meno di shift

Consideriamo ora un problema che generalizza il problema dell'eguaglianza di due vettori[6]. Immaginiamo i due vettori come circolari (cioé l'elemento successivo a $A[n-1]$ è $A[1]$). Per semplificare le notazioni immaginiamo associata ad un array A una funzione *periodica* omonima A definita da:

$$A(i) = A[i \bmod n]$$

dove mod è il resto della divisione intera. Il nostro obiettivo sarà verificare se per due vettori è verificata la seguente proprietà:

$$eq \equiv \exists i.\forall k.0 \leq k < n.A(i+k) = B(k)$$

Osserviamo che questa proprietà può essere riscritta nella seguente:

$$eq \equiv \exists i, j.\forall k.0 \leq k < n.A(i+k) = B(j+k)$$

che ha il vantaggio di far giocare un ruolo simmetrico ai due vettori A e B. Indichiamo, sempre per comodità di notazione, con SA_i (rispettivamente SB_i) la sequenza ottenuta leggendo circolarmente il vettore A (risp. B) a partire dall'indice i, cioè la sequenza:

$$A[i]\ A[i+1] \ldots A[n-1]\ A[1] \ldots\ A[i-1]$$

In questo modo il nostro problema può essere riformulato nello stabilire la verità o meno dell'asserzione:

$$\exists i, j.SA_i = SB_j \tag{5}$$

Ragionando esattamente come nel caso di uguaglianza dei due vettori, tenendo presente solo la natura "circolare" del problema in questione, abbiamo che il seguente ciclo, se termina con $h = n$, stabilisce che i due vettori sono uguali, a meno di shift, e che i, j sono gli indici necessari a provare la proprietà esistenziale espressa da (5):

[6]La presente soluzione è dovuta al prof. Edger W. Dijkstra, recentemente scomparso, che con C. A. R. Hoare sviluppò la teoria delle asserzioni logiche. Numerosissimi sono i contributi del prof. Dijkstra alla teoria e alla pratica dei linguaggi di programmazione e degli algoritmi.

```
while (h<n && A[(i+h) % n] == B[(j+h) % n]) h++;
/* INV: forall k. 0<=k<h. A(i+k) = B(j+k) */
/* se h=n allora SA.i=SB.j
```

Tuttavia, se da questo ciclo si esce perché, per un qualche $h < n$, $A(i+h) \neq B(j+h)$, sorge la questione se sia il caso di cercare una nuova coppia i, j o se abbiamo finito, potendo concludere che i due vettori sono diversi. L'altra questione è legata alla domanda se sia possibile recuperare in parte il lavoro svolto, evitando di ricominciare daccapo con una nuova coppia i, j ed eventualmente riispezionare inutilmente sottovettori già analizzati. Ora è chiaro che fissando uno dei due indici, ad esempio i ad un valore arbitrario (ad esempio 0) e testando tutte le uguaglianze:

$$SA_0 = SB_j \quad (j = 0, \ldots n-1)$$

otteniamo un algoritmo che risolve questo problema. La complessità peggiore sarà nell'ordine di n^2 (provate ad esempio a vedere quanti confronti sono necessari nel caso in cui $SA_0 = \underbrace{000\ldots0}_{n-1}1$ e $SB_0 = \underbrace{000\ldots}_{n-2}11$). La domanda quindi è se, scoprendo che $SA_i \neq SB_j$, sia possibile recuperare parte del lavoro svolto e scegliere in modo più intelligente una nuova coppia di indici i, j. La cosa è possibile solo se stabiliamo una certo criterio con cui esplorare l'insieme delle coppie di sequenze SA_i e SB_j. L'osservazione chiave della soluzione che sarà proposta è che l'insieme delle sequenze può essere ordinato, ad esempio usando l'ordine *lessicografico*[7] (quello del dizionario per intenderci). Inoltre se esiste una coppia di indici i, j tale che $SA_i = SB_j$ anche gli insiemi $\{SA_i \mid i = 0, \ldots n-1\}$ e $\{SB_j \mid i = 0, \ldots n-1\}$ sono uguali ed inoltre hanno lo stesso elemento *massimo* nell'ordine lessicografico. Chiamiamo:

$$\begin{array}{rcl} AA & = & max_{0 \leq i \leq n-1}\ SA_i \\ BB & = & max_{0 \leq i \leq n-1}\ SB_i \end{array}$$

Quindi l'idea diventa quella di esplorare l'insieme di sequenze sempre cercando tra quelle *maggiori* nell'ordine lessicografico. Questo è possibile scomponendo la condizione di uscita dal ciclo $A(i+h) \neq B(j+h)$ nei suoi due sottocasi, e cioè $A(i+h) < B(j+h)$ e $A(i+h) > B(j+h)$. Nel primo caso avrò che $SA_i < SB_j$, mentre nel secondo che $SA_i > SB_j$. Concentriamoci sul primo caso (l'altro evidentemente è simmetrico). Ho che, per definizione di massimo, $SB_j \leq BB$. Ma avendo trovato h uguaglianze del tipo $A(i+k) = B(j+k)$, per $0 \leq k < h$, possiamo dire che tutte le sequenze $SA_{i+k} < SB_{j+k} \leq BB$. E a poco servirebbe quindi aggiornare i incrementandolo di 1. Possiamo direttamente occuparci di cominciare la nuova ricerca dall'elemento seguente a $i + h$, cioè $i + h + 1$. L'idea in buona sostanza è quella di cercare tra tutte le stringhe SA_i (e simmetricamente SB_j) finché sappiamo che $SA_k < BB$ $(0 \leq k < i)$ (e simmetricamente $SB_k < AA$ $(0 \leq k < j)$). Quindi gli invarianti della ricerca saranno:

$$\begin{array}{rcl} P[i,j,h] & \equiv & \forall k.0 \leq k < h.A(i+k) = B(j+k) \\ QA[i] & \equiv & \forall k.0 \leq k < i.SA_k < BB \\ QB[j] & \equiv & \forall k.0 \leq k < j.SB_k < AA \end{array}$$

Se ho verificato la proprietà $P[i, j, n]$, allora significa che ho trovato due indici i, j tale che $SA_i = SB_j$. Viceversa se sono arrivato a verificare la proprietà $QA[n]$ significa che tutte le sequenze SA_i sono strettamente minori di BB e quindi i due vettori sono diversi. Analoga conclusione posso trarre dalla verifica di $QB[n]$.

[7] l'ordine lessicografico tra due sequenze dipende dall'ordine del primo elemento su cui differiscono.

```
int equalShift(int A[], int B[], int n)
/* PREC: n>=0, n lunghezza di A e B
 * POST: ritorno 1 se esistono 0<=i,j<n tale che vale
 * P[i,j,n] = forall k, 0<=k<n, A(i+k)=B(j+k)
 */
{ int i = 0; int j = 0; int h = 0;
  while (i<n && j<n && h<n)
  /* INV: P[i,j,h] & QA[i] & QB[j]
   * TERM: 3n - (i+j+h)
   */
  { if (A[(i+h) % n] == B[(j+h) % n]) h++
           /* continuo a vedere se SA.i = SB.j */
       else if (A[(i+h) % n] < B[(j+h) % n]) {i=i+h+1; h=0}
           /* scopro che SA.i, SA.i+1, ..., SA.i+h sono minori di BB
            * (quindi vale QA[i+h])
            */
       else {j=j+h+1; h=0}
           /* scopro che SB.j, SB.j+1, ..., SB.j+h sono minori di AA
            * (quindi vale QB[i+h])
            */
  }
  eq = (n<h);
  return eq;
}
```

Siccome la guardia implica che tutte e tre le variabili `i, j, h` assumeranno valori minori di n durante il ciclo, osserviamo che la funzione $t = 3n - (i + j + h)$ è positiva durante l'esecuzione del ciclo. La terminazione si dimostra semplicemente osservando che la somma $i + j + h$ aumenta di uno ad ogni iterazione. In questo caso, la funzione di terminazione ci da anche informazione sulla complessità del programma, in quanto ci dice che il ciclo verrà eseguito al più $3n$ volte. Un eccellente progresso rispetto alla complessità n^2 del programma ingenuo!

Esercizi

1. Modificare la funzione `minimo` in modo che venga restituita la posizione dell'elemento minimo, piuttosto che il valore del minimo. Modificare opportunamente le asserzioni logiche.

2. Scrivere la funzione `minimo` senza scrivere cicli. Corredare la funzione di opportune asserzioni logiche e confrontarle con quelle scritte per il programma iterativo.

3. Scrivere la postcondizione che deve soddisfare una funzione che esegue l'ordinamento di un vettore.

4. Scrivere una procedura per l'ordinamento di un'array basata sull'idea di selezionare i minimi successivi e metterli all'inizio della porzione di vettore non ancora ordinato *Selection Sort*. Corredare la funzione di opportuni invarianti e dimostrarne la correttezza. [**Sugg**: usare la selezione del minimo come sottoprocedura ed usare le sue

precondizioni e postcondizioni per garantire la preservazione dell'invariante del ciclo piú esterno].

5. Si scriva una funzione `C` selectSort che ordina un vettore di interi senza scrivere cicli. Si corredi la funzione di opportune asserzioni logiche e se ne dimostri la correttezza. Confrontare le asserzioni con quelle scritte nella soluzione dell'esercizio precedente.

6. Scrivere la procedura per l'ordinamento *Insertion Sort* corredata di opportuni invarianti e dimostrarne la correttezza. [**Sugg**: si scriva prima una funzione `inserisci` che inserisce un elemento al posto giusto in un vettore ordinato e procedere come suggerito nell'esercizio precedente].

7. Scrivere una procedura `merge` che fonde due vettori ordinati restituendo come risultato un vettore ordinato. Corredare la funzione di opportuni invarianti e dimostrarne la correttezza.

8. Scrivere una funzione `C` che cerca (scandendo il vettore da sinistra a destra) un elemento in un vettore e restituisce la sua posizione:

```
int ricercaLineare(int A[], int n, int elem)
/* PREC: n = lunghezza di A, n>=0
 * POST: restituisce k se esiste k. A[k]=elem
 *       -1 altrimenti
 */
```

   Corredare di opportuni invarianti.

9. Scrivere una funzione `C` che verifica se due array sono uno la permutazione dell'altro, cioé contengono gli stessi elementi (non necesseriamente nello stesso ordine)

10. Ricordiamo al lettore il triangolo di Tartaglia[8]:

$$
\begin{array}{ccccccc}
1 & & & & & & \\
1 & 1 & & & & & \\
1 & 2 & 1 & & & & \\
1 & 3 & 3 & 1 & & & \\
1 & 4 & 6 & 4 & 1 & & \\
1 & 5 & 10 & 10 & 5 & 1 & \\
\ldots & & & & & &
\end{array}
$$

    L'elemento all'incrocio della n-esima riga e k-esima colonna (cominciando a contare da 0!) è proprio il coefficiente binomiale $\binom{n}{k}$. Detta T la matrice che rappresenta il triangolo di Tartaglia, la sua costruzione si può infatti fare usando la relazione $T[n,k] = T[n-1,k-1]+T[n-1,k]$, dopo aver scritto la prima riga e immaginando che l'unico 1 della prima riga sia circondato da 0. Chi avesse scritto la procedura ricorsiva per calcolare i coefficienti binomiali, si sarebbe dovuto accorgere (anche seguendo un

[8] chiamato di Pascal in Francia e di Newton in Inghilterra.

ragionamento informale) che le chiamate ai casi base, viene eseguita esattamente $\binom{n}{k}$ volte, un numero che cresce esponenzialmente.

Scrivere una procedura che calcola il coefficiente binomiale $\binom{n}{k}$ generando le prime n righe del triangolo di Tartaglia, restituendo il k-esimo elemento dell'n-esima riga. Usare solo un vettore (e non una matrice) di cui si occuperanno al più le prime n posizioni. Corredare di opportuni invarianti.

4 Esercizi Svolti

4.1 Fibonacci Ricorsivo Efficiente

L'idea di base è quella di mimare ricorsivamente il comportamento del programma iterativo scritto nelle precedenti sezioni. Dovremmo quindi usare la ricorsione per mimare il comportamento del ciclo:

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

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:

1. terminare le attivazioni ricorsive quando il valore di `i` raggiunge il valore di `n`;
2. 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` e `fib2` e `fibnew`. 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 fatte durante un'attivazione **non** influenzano il valore di quelle variabili in una successiva attivazione, o al rientro da quella attivazione. Una pessima soluzione sarebbe quella di dichiarare `fib1 e fib2` e `fibnew` come variabili globali: questo condurrebbe ad un fenomeno detto *side effect* assolutamente sconsigliato, in quanto una funzione che modifica lo stato globale in modo non specificato dall'interfaccia rende l'analisi dei programmi molto difficile. Ma allora come comunicare i valori via via calcolati alle nuove attivazioni? 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.

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` all'ingresso del ciclo.

Ecco quindi il codice delle due funzioni:

```
int fibRecEffAux(int n, int i, int fib1, int fib2)
/* PREC: fib1=fib(i) & fib2=fib(i-1) & 2<=i<=n
 * POST: ritorna fib(n)
 */
{ if (i==n) return fib1;
     else return fibRecEffAux(n,i+1,fib1+fib2, fib1);
}

int fibRecEff(int n)
/* PREC: n>=0
 * POST: ritorna fib(b)
 */
{ if (n<2) return n;
     else return fibRecEffAux(n,2,1,1);
}
```

Dimostriamo ora per induzione che una chiamata alla funzione `fibRecEff(n)` effettivamente restituisce $fib(n)$, sotto la precondizione $n \geq 0$. Per $n < 2$ è sufficiente verificare che $n = fib(n)$ 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 $fib(2) = 1$ e $fib(1) = 1$;

2. se $i = n$ e vale la precondizione ($fib(i) = fib1$), chiaramente la funzione restituisce $fib(n)$;

3. altrimenti, induttivamente la correttezza di `fibRecAuxEff(n,i,fib1,fib2)` dipende solo dalla correttezza di `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 $fib1 = fib(i) \wedge fib2 = fib(i-1)$, allora $fib1 + fib2 = fib(i+1)$.

4. infine, osserviamo che la sequenza di chiamate ricorsive termina, perché la funzione $n - i$ (positiva sotto le precondizioni) decresce ad ogni chiamata.

Concludiamo con un ulteriore spunto di riflessione. 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 (che sarà oggetto di studio più avanti) hanno un costo significativo.

4.2 Selection Sort

Vediamo in questa sezione, una ben nota procedura di ordinamento di vettori, detta *Selection Sort*, basata sul principio di trovare i minimi successivi della parte di vettore non ancora ordinato e piazzarli via via in coda al vettore già ordinato. Al generico passo i, la situazione è la seguente:

1. il vettore è ordinato fino alla posizione $i-1$. Formalmente:

$$\varphi[i] \equiv \forall j.0 \leq j < i.A[j] \leq A[j+1]$$

2. tutti gli elementi del vettore dalla posizione i in poi, sono maggiori degli elementi che stanno fino alla posizione $i-1$. Formalmente:

$$\psi[i] \equiv \forall k, j.(0 \leq j < i) \wedge (i+1 \leq k < n).A[j] \leq A[k]$$

A questo punto, conoscendo l'indice k_0, tale che $A[k_0] = min_{i \leq k < n} A[k]$, è chiaro che scambiando $A[k_0]$ con $A[i]$, avremo soddisfatte le proprietà $\varphi[i+1]$ e $\psi[i+1]$, perché $A[k_0]$ è maggiore di tutti gli elementi fino a $i-1$ (per $\psi[i]$) e per come è stato scelto, è minore di tutti gli elementi da $i+1$ in poi. Riscrivendo la procedura `minimo`, come segue, in modo che restituisca la posizione occupata dall'elemento minimo, invece che il suo valore (osservate le modifiche rispetto alla precedente funzione `minimo` e che dobbiamo introdurre un nuovo parametro perché ora ci serve calcolare il minimo non di tutto il vettore, ma a partire da una certa posizione[9]):

```
int minimo(int A[], int m, int n)
/* PREC: n=lunghezza di A, n>m>=0
 * POST: ritorna il valore k, tale che: per ogni m<=j<n, A[k] <= A[j]
 */
{ int min;
  int i = m;

  i=1; k = m;
  while (i < n)
  /* INV: forall j. m<=j<i A[k]<=A[j] */
  { if (A[i] < A[k]) k=i;
    i++;
  }

  return k;
}
```

possiamo scrivere la procedura di ordinamento `selectSort` come segue:

[9] tuttavia invarianti e dimostrazione di correttezza seguono esattamente i passi visti nella sezione precedente. Una interessante alternativa è quella di non introdurre il parametro ausiliare `m`, ma passare direttamente il puntatore al primo elemento da dove si vuol cominciare a cercare il minimo. La chiamata deve però impostare correttamente la lunghezza.

```
void selectSort (int A[], int n)
/* PREC: n=lunghezza di A, n>=0
 * POST: forall j. 0<=j<n-1 A[j] <= A[j+1]
 */
{ int i = 0;
  while (i< n-1)
  /* INV: forall j,k. 0<=j<i i<=k<n A[j] <= A[j+1] & A[j]<=A[k]*/
  { l = minimo(A, i, n);
    scambia(A[i],A[l]);
    i++;
  }
}
```

Lasciamo al lettore la verifica formale attraverso triple di Hoare che $\varphi \wedge \psi$ è una proprietà invariante, mentre è chiaro (per sostituzione) che $\varphi \wedge (i = n - 1)$ implica la postcondizione. Sempre il solito ragionamento (e la solita funzione di terminazione $n - i$) permettono di dedurre che il ciclo termina.

Il lettore attento avrà osservato che la procedura `selectSort`, pur avendo due cicli annidati (nel codice scritto questo è nascosto dalla chiamata alla procedura `minimo`), non è più difficile da analizzare rispetto altri programmi nelle sezioni precedenti. Questo perchè abbiamo convenientemente scomposto il problema in sottoproblemi e scritto ed analizzato la funzione, assumendo che un'altra funzione fosse corretta: non solo i ragionamenti formali sono facilitati dalla scomposizione di un problema complesso in sottoobbiettivi, ma anche quelli informali: in generale una buona tecnica di programmazione consiste nel suddividere un problema in sottoproblemi, risolverli separatamente e poi comporli in modo adeguato per risolvere il problema di partenza (metodologia *top-down*).

Vedremo nel seguito un atteggiamento diametralmente opposto, ma animato dallo stesso principio (noto come *divide et impera*): costruire nuovi tipi di dato e operazioni sempre più complesse per fornire ad altri programmatori delle azioni elementari via via più complesse e quindi adatte a risolvere problemi complessi in modo relativamente semplice (metodologia *bottom-up*). Sarà particolarmente importante che il comportamento degli strumenti software forniti ad altri programmatori sia specificato in modo preciso e non ambiguo, attraverso asserzioni logiche.