Iterare è Umano, Ricorrere è Divino

Ivano Salvo – Sapienza Università di Roma

Anno Accademico 2016-17

La presente dispensa si propone di introdurre il lettore alla programmazione ricorsiva in C. Analizzeremo pregi e difetti delle soluzioni ricorsive a problemi via via più complessi e sarà discusso il legame tra versioni iterative e ricorsive di programmi che risolvono lo stesso problema.

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 sé 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, ricordando cosa accade in seguito a una chiamata di funzione, 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. Anzi, la pila di sistema è stata ideata proprio per eseguire correttamente programmi ricorsivi. Linguaggi che non ammettono la ricorsione, come ad esempio il primitivo FORTRAN, possono allocare *staticamente* (cioè all'atto del caricamento in memoria del programma) i record di attivazione delle funzioni, senza bisogno di allocarli e deallocarli *dinamicamente* (cioè durante l'esecuzione del programma).

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 C, 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 definito 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, cioè un naturale nella forma $n+1$, 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 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, 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;

    for (int i=1; i!=n; i++)  f *= i;
        /* INV: f=i!, TERM: n-i */
    return f;
 }
```

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

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

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:

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

A questa definizione corrisponde immediatamente un programma ricorsivo:

```
int sommaRec(int m, int n){
    if (n==0) return m;
         else return sommaRec(m,pred(n))+1;
}
```

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 corrispondenti funzioni ricorsive. Avevamo già dato definizioni induttive per descrivere il massimo comun divisore con l'algoritmo di Euclide e la moltiplicazione egiziana. Vediamo in Fig. 1 e 2 le corrispondenti funzioni ricorsive. Confrontatele con le relative funzioni iterative già viste.

Approfittiamo di questi esempi per impratichirci con il C. 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. Nella funzione `mulRec` facciamo conoscenza con gli operatori aritmetici (che ci eravamo programmati a partire dal +1) `%` che calcola il resto della divisione intera, e `/` che calcola il quoziente della divisione intera (quando applicato a due operandi interi). Osservate che nella condizione `if (n%2)` speculiamo sul fatto che se `n` è dispari, `n%2` valuta a 1 che verrà interpretato come `true`. Più in generale, ogni comando del tipo `if (E!=0) C` potrà essere scritto semplicemente `if (E) C`, così come ogni comando del tipo `if (E==0) C` potrà essere scritto semplicemente `if (!E) C`.

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, 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

```
int mcdRec(int m, int n){
    if (m<n) return mcdRec(n-m,m);
    if (n<m) return mcdRec(m-n,n);
    return n;
    }
```

Figura 1: MCD di Euclide Ricorsivo

```
int mulEgyRec(int m, int n){
    if (!n) return 0;
    if (n%2) return m+mulRec(m,n-1);
    return mulRec(m+m,n/2));
    }
```

Figura 2: Moltiplicazione Egiziana

(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 *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 (vedi esercizio 1.3(6) qui sotto).

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:

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

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.

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){
 /*  PREC: n>=0 */
     if (n<=1) return n;
     return fibRec(n-1) + fibRec(n-2);
 }
```

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

[1] La successione prende il nome dal matematico italiano Leonardo Pisano detto Fibonacci (Pisa ∼1170–∼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, a volte sorprendenti, applicazioni in matematica.

```
int fibIt(int n){
    int fib_2=0; /* penultimo numero */
    int fib_1=1; /* ultimo numero */
    int fib=1;   /* nuovo numero  */

    if (n<2) return n;
    for (int i=2; i<=n; i++){
    /* INV: fib = fib(i) & fib_1 = fib(i-1) &
     * & fib_2 = fib(i-2) */
        fib = fib_1 + fib_2;
        fib_2 = fib_1;
        fib_1 = fib;
        }
    return fib;
    }
```

Figura 3: Funzione Fibonacci Iterativa

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 (vedi Fig. 3).

Il lettore faccia anche attenzione all'invariante: l'asserzione logica $fib = fib(i)$ sarebbe ovviamente sufficiente a dimostrare la correttezza della funzione, ossia che $fib = fib(n)$ 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 fib_1 e fib_2 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. 4 viene esemplificato l'albero delle chiamate ricorsive generato per effetto di una chiamata a `fibRec(4)`.

È facile dimostrare (per induzione!) che il calcolo di $fib(1)$ e $fib(0)$ verranno invocati rispettivamente $fib(n)$ e $fib(n-1)$ volte (vedi Fig. 4), 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.

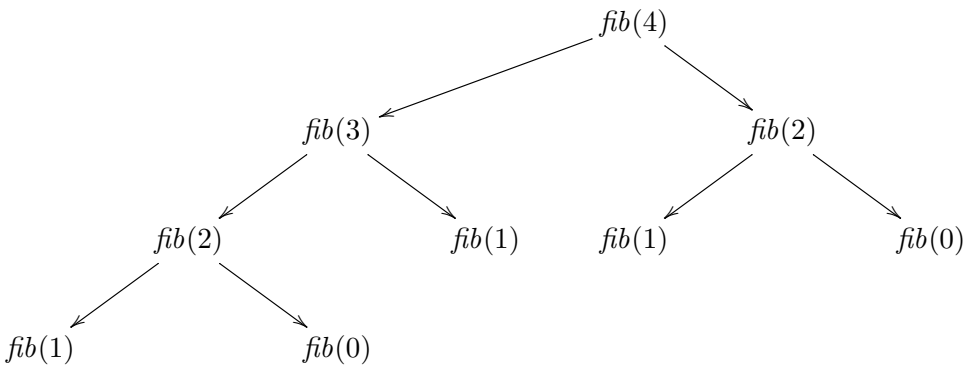

Figura 4: Attivazione delle chiamate ricorsive della funzione `fibRec(4)`

1.3 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 `C` che non contiene cicli, che non termina.

5. 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.

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 $fib(n)$ 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 `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 Funzioni Ricorsive con Asserzioni Logiche

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, le funzioni `fattIt` e `fattRec` non terminano nel caso in cui il parametro di ingresso sia negativo. Chi avesse fatto l'esperienza di sperimentare il programma predecessore su input minori o uguali zero, avrà scoperto che sorprendentemente, dopo una leggera attesa, tale programma termina fornendo un risultato "corretto" (le virgolette sono d'obbligo, perchè di fatto sui naturali il predecessore su 0 – e a maggior ragione su input negativi – non è definito). Chiamando `fattRec` con parametri negativi, il programma termina in modo anomalo con un errore (anche qui inaspettato) di *segmentation fault*: il motivo è che `fattRec` continua a chiamare sé 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 prevedibile 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 $2^{31}-1$. L'eseguibile prodotto da `gcc` non si scompone più di tanto e con-

[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, ad esempio usando `assert`, della libreria `assert.h`.

tinua a dare valori privi di senso senza informare l'utente di cosa stia effettivamente succedendo[3]. Non si fanno grandi progressi neanche con altri tipi interi.

A parte questioni numeriche[4] 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*.

Definendo le precondizioni, il programmatore che definisce la funzione `fattRec` oltre ad implementare il codice, specifica anche il suo corretto utilizzo[5]: 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 $n - 1 \geq 0$.

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.

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:

- i casi base siano trattati correttamente;
- le eventuali chiamate ricorsive rispettino le precondizioni;
- la funzione garantisca la postcondizione assumendo che le chiamate ricorsive garantiscano la postcondizione;

[3] 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. 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] magari prima di fine corso programmeremo gli interi di lunghezza illimitata.

[5] 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.

- 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 $n!$. Dobbiamo infine dimostrare la terminazione, ma assumendo $n \geq 0$ il valore del parametro `n` decresce a ogni chiamata.

2.3 Fibonacci Ricorsivo Efficiente

Spero che nessuno di voi, vedendo l'esempio di Fibonacci, abbia affrettatamente concluso 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 (int i=2; i<=n; i++){
    fib = fib_1 + fib_2;
    fib_2 = fib_1;
    fib_1 = 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 `fib_1`, `fib_2` 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 fatte durante un'attivazione *non influenzano* il valore delle variabili in una successiva attivazione, o al rientro da quella attivazione.

Una pessima soluzione sarebbe quella di dichiarare `fib_1`, `fib_2` e `fib` come variabili globali: come sappiamo, i side effects sono da evitare in quanto modificare

lo stato globale in modo non specificato dall'interfaccia delle funzioni, 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 `fib_1` e `fib_2` nel programma iterativo. Il valore della variabile `fib`, viceversa, sarà comunicato attraverso il valore ritornato dalla funzione.

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 a chiamare la funzione ausiliaria `fibRecEffAux` inizializzando opportunamente i parametri per la prima chiamata, esattamente come la funzione iterativa `fibIt` inizializza opportunamente le variabili `i, fib_1` e `fib_2` all'ingresso del ciclo. Ecco quindi il codice delle due funzioni:

```
int fibRecEffAux(int n, int i, int fib_1, int fib_2){
  /* PREC: fib_1=fib(i) & fib_2=fib(i-1) & 2<=i<=n
   * POST: ritorna fib(n)
   */
   if (i==n) return fib_1;
   return fibRecEffAux(n, i+1, fib_1+fib_2, fib1);
 }

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

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:

- 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 $fib(2) = 1 = fib_2$ e $fib(1) = 1 = fib_1$;
- se $i = n$ e vale la precondizione $fib(i) = fib_1$ e chiaramente la funzione restituisce $fib(n)$;
- 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 $fib_1 = fib(i) \wedge fib_2 = fib(i-1)$, allora $fib_1 + fib_2 = fib(i + 1)$;
  - analogamente, se $fib_1 = fib(i)$ allora avremo che incrementando i di 1 $fib_1 = fib(i - 1)$ come richiesto.

- 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.

2.4 Esercizi e Spunti di Riflessione

1. ★ È possibile scrivere un programma ricorsivo che renda efficiente il calcolo dei coefficienti binomiali?

2. ♣ 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.

3. ★ ♣ 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.

4. Usare i metodi visti in questa sezione per dimostrare formalmente la correttezza di alcune funzioni ricorsive scritte negli esercizi di sezione 1.3.

5. ▶ 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).

6. ▶ 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 due esempi che mostrano 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` sia sufficiente a calcolare qualunque funzione, ed è possibile simulare il comportamento di qualsiasi programma ricorsivo con un programma iterativo che simula 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 matematico):

> *"narra la leggenda che in un tempio Indù alcuni monaci siano impegnati a spostare su tre colonne di diamante 64 dischi d'oro di diversi diametri 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."*

In Fig. 5 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.

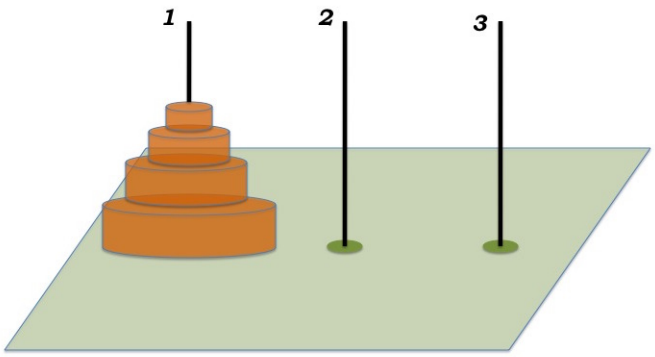

Figura 5: Il problema della Torre di Hanoi con 4 dischi

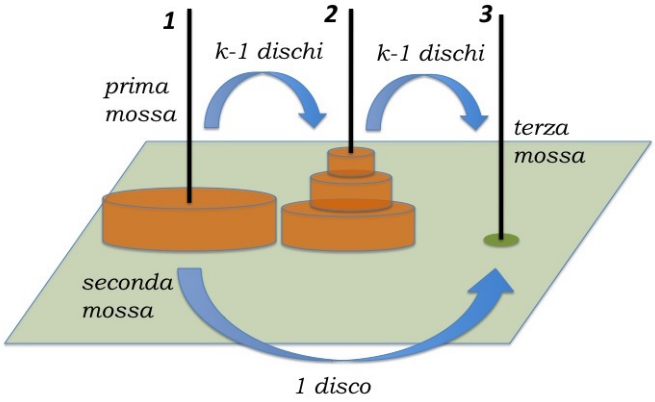

Figura 6: Soluzione del problema della Torre di Hanoi

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 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) 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. 6).

L'unica verifica da fare è che lo spostamento di $k-1$ dischi non vìola 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 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, in modo che sposti k dischi da un certo piolo sorgente a un piolo destinazione usando un terzo piolo ausiliario, indipendentemente da quali siano effettivamente i numeri che identificano il piolo sorgente, destinazione e appoggio. Il risultato è mostrato in Fig. 7.

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

```
void hanoi(int sorg, int aux, int dest, int n){
   /* sposta n dischi da sorg a 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 7: Programma che risolve il problema della Torre di Hanoi

torri. Per esempio la situazione in Fig. 6 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:

```
void muovi(int s, int d){ printf("%d --> %d\n",s,d);}
```

3.2 ⋆ Generazione degli anagrammi di una parola

Purtroppo, siccome gli anagrammi sono *sequenze* (cioè dipendono dall'ordine) non ci sono sufficienti gli insiemi per formalizzare il problema (infatti nella teoria degli insiemi abbiamo che ad esempio l'insieme $\{1, 3\}$ è uguale all'insieme $\{3, 1\}$).

Data la costante $\langle\,\rangle$ (*sequenza vuota*), un insieme di elementi A, e l'*operazione di concatenazione* $\cdot$, definiamo per induzione le sequenze di elementi di A, Seq$[A]$:

$$\langle\,\rangle \in \text{Seq}[A]$$
$$s \in \text{Seq}[A], a \in A \Rightarrow a \cdot s \in \text{Seq}[A]$$

Osservate che l'operazione di concatenazione è una funzione che permette di costruire nuove sequenze a partire da un elemento di A e altre sequenze più corte e quindi è una funzione $\cdot : A \times \text{Seq}[A] \mapsto \text{Seq}[A]$. Data la precedente definizione, una sequenza di elementi $a_1, a_2 \ldots, a_n$ ha la forma $a_1 \cdot a_2 \cdot \ldots \cdot a_n \cdot \langle\,\rangle$. Prendiamoci la libertà di indicare una tale sequenza con $\langle a_1, a_2, \ldots, a_n\rangle$, convenendo che $a \cdot \langle a_1, \ldots, a_n\rangle = \langle a, a_1, \ldots, a_n\rangle$.

A questo punto, l'idea per definire chi sono gli anagrammi di una sequenza (per induzione sulla struttura delle sequenze) dovrebbe cominciare ad apparrire all'orizzonte. Dovrebbe essere chiaro che l'insieme degli anagrammi di $\langle\,\rangle$ è ovviamente $\{\langle\,\rangle\}$. Gli anagrammi di una sequenza lunga n $\langle a_1, a_2, \ldots, a_n\rangle$ sarà l'insieme delle sequenze che cominciano per a_1 concatenato con tutti gli anagrammi di $\langle a_2, \ldots, a_n\rangle$, unito l'insieme delle sequenze che cominciano per a_2 concatenato tutti gli anagrammi di $\langle a_1, a_3 \ldots, a_n\rangle$, unito $\ldots$ e così via fino ad a_n.

Per formalizzare ciò ci manca una notazione per la sequenza in cui tolgo l'elemento i-esimo. Data una sequenza $s = \langle a_1, a_2, \ldots, a_n\rangle$ conveniamo di denotare con s_i il suo elemento i-esimo a_i e con s_{-i} la sequenza $\langle a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n\rangle$ in cui ho rimosso a_i da s (anche questa operazione può essere definita ricorsivamente, vedi Esercizio 3.3(3) qui sotto). Indichiamo con $\sharp s$ la lunghezza di s. Siccome dobbiamo concatenare un elemento con tutte le sequenze di un insieme, definiamo l'operazione $a{*}S$, dove $a \in A$ e $S \subseteq \mathrm{Seq}[A]$, come $\{a{\cdot}s \mid s \in S\}$. Siamo finalmente pronti per dare una definizione induttiva rigorosa dell'insieme degli anagrammi di una sequenza:

$$A(\langle\,\rangle) = \{\langle\,\rangle\}$$
$$A(s) = \textstyle\bigcup_{i=1}^{\sharp s} s_i * A(s_{-i})$$

In alcuni linguaggi di programmazione (in particolare linguaggi funzionali come Haskell o ML), questo è già (modulo qualche piccolo aggiustamento sintattico) un programma che calcola gli anagrammi di una sequenza! Ma si tratta di linguaggi progettati appunto per scrivere programmi ricorsivi e che forniscono primitive molto semplici da usare per gestire tipi di dato come sequenze o insiemi.

Se avete capito il vero obiettivo della paziente costruzione dei programmi sui naturali, a partire dal +1 e test di uguaglianza, forse sarete persuasi che la seguente funzione potrebbe correttamente generare gli anagrammi di una sequenza, una volta che vi foste programmati le operazioni per gestire sequenze e insiemi di sequenze:

```
InsiemeSeq anagrammi(Sequenza s){
   InsiemeSeq A=insiemeVuotoSeq();
   if (seqVuota(s)) return creaInsieme(creaSeqVuota());
   for (int i=1; i<=lung(s); i++)
       A=unione(A, concat(iesimo(s,i), anagrammi(elim(s,i))));
   return A;
   }
```

In altre parole, l'algoritmo visto sopra è corretto, riferito ad un esecutore che sappia manipolare strutture dati come Insiemi e Sequenze. Certo, ci sarebbe un bel po' di lavoro, partendo dall'esecutore del linguaggio C, per costruire un esecutore in grado di eseguire questo algoritmo. Inoltre, se l'operazione `unione` è implementata correttamente, questo programma evita di inserire eventuali ripetizioni (che si hanno quando i caratteri di `s` non sono tutti distinti) in quanto $X \cup \{a\} = X$ se $a \in X$.

Ferme restando le equazioni ricorsive scritte sopra per gli anagrammi, che ci serviranno comunque da guida nella stesura del codice, esistono tuttavia modi più economici per scrivere un programma C che genera gli anagrammi di una sequenza di valori. Affronteremo questo problema nella sezione dedicata ai vettori.

3.3 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 delle equazioni ricorsive per definire l'operazione $\sharp s$.

3. Scrivere delle equazioni ricorsive per definire l'operazione s_{-i} per induzione su i. Assumere che $s_{-i} = s$ se $\sharp s < i$.

4. ★ Modificare la funzione `anagrammi`, in modo che funzioni correttamente nel caso in cui ci siano ripetizioni nella sequenza da anagrammare. Evitare quindi di generare sequenze già generate, non appena possibile.

5. 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 $\mathcal{P}(X)$.

6. 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 $\mathcal{P}_k(X)$.

7. ★ Definiamo insieme delle *partizioni ordinate* di un numero n l'insieme delle sequenze di numeri naturali positivi che danno come somma n. Formalmente, $\mathrm{Part}(n) = \{s \in \mathrm{Seq}[\mathbb{N}^+] \mid \sum_{i=1}^{\sharp s} s_i = n\}$. Ad esempio, $\mathrm{Part}(4) = \{\langle 4\rangle, \langle 2,2\rangle, \langle 1,3\rangle, \langle 3,1\rangle, \langle 1,1,2\rangle, \langle 1,2,1\rangle, \langle 2,1,1\rangle, \langle 1,1,1,1\rangle\}$. Scrivete equazioni che definiscano l'insieme delle partizioni ordinate per induzione su n.

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