# Correzione Secondo Esonero, Esercizio 3

Ivano Salvo – Sapienza Università di Roma

Anno Accademico 2011-12

**Esercizio 3:** *Si consideri il problema di prendere in input una lista di interi e calcolare la lista in cui tutti gli elementi sono sostituiti con la somma dei successivi (elemento incluso). Ad esempio, presa in input la lista* $\langle 1, 7, 11\rangle$ *si vuole ottenere in output la lista* $\langle 19, 18, 11\rangle$.

**Punto 1:** *Specificare la funzione mediante equazioni ricorsive su sequenze;*

Sicuramente il modo più semplice era considerare una funzione ausiliaria sommaSeq che calcola la somma di tutta una sequenza e definire anche quest'ultima per ricorsione. Vediamo:

$$\begin{aligned}
\mathsf{sommaSucc}(\langle\,\rangle) &= \langle\,\rangle \\
\mathsf{sommaSucc}(a \cdot s) &= \mathsf{sommaSeq}(a \cdot s) \cdot \mathsf{sommaSucc}(s) \\
\mathsf{sommaSeq}(\langle\,\rangle) &= 0 \\
\mathsf{sommaSeq}(a \cdot s) &= a + \mathsf{sommaSeq(s)}
\end{aligned}$$

Questa specifica ricorsiva conduceva tuttavia a una immediata traduzione in un programma C di complessità quadratica. Un po' d'arguzia avrebbe potuto suggerire che data la lista di input $s = \langle a_1, a_2, \ldots, a_n\rangle$, nella lista risultato $s' = \langle a'_1, a'_2, \ldots, a'_n\rangle$ è soddisfatta la relazione $a'_i = a_i + a'_{i+1}$. Quindi il primo elemento della nuova lista si può semplicemente calcolare sommando la testa della lista di input con la testa della lista risultato calcolata sulla coda. Tuttavia, ciò necessita di una ulteriore accortezza: infatti, occorre assicurarsi che anche la coda a sua volta non sia vuota. Sono necessari, quindi, due casi base. Voilà:

$$\begin{aligned}
\mathsf{sommaSucc'}(\langle\,\rangle) &= \langle\,\rangle \\
\mathsf{sommaSucc'}(\langle\, a\,\rangle) &= \langle\, a\,\rangle \\
\mathsf{sommaSucc'}(a \cdot s) &= (a + \mathsf{head}(\mathsf{sommaSucc'}(s))) \cdot \mathsf{sommaSucc'}(s)
\end{aligned}$$

Discuteremo ulteriormente questa soluzione nel punto successivo.

**Punto 2:** *Scrivere una funzione* `sommaSuccessiviFun` *che crea una* nuova *lista;*

Come sempre, un programma che genera nuove liste si può ottenere con una facile traduzione delle equazioni ricorsive. Ovviamente, la traduzione di sommaSucc' (vedi Fig. 2) conduce a un programma più efficiente rispetto alla traduzione di sommaSucc (vedi Fig. 1), anche se non senza ombre.

```
lista sommaLista(lista L){
    if (isEmpty(L)) return 0;
    return head(L)+somma(tail(L));
}
lista sommaSuccFun(lista L){
    if (isEmpty(L)) return L;
    return addHead(sommaSuccFun(tail(L)), somma(L));
}
```

Figura 1: Versione funzionale ricorsiva quadratica

```
lista sommaSuccFun(lista L){
    if (isEmpty(L) || isEmpty(tail(L))) return L;
    return addHead(sommaSuccFun(tail(L)),
                     head(L)+head(sommaSuccFun(tail(L)))
                   );
}
```

Figura 2: Versione funzionale lineare (primo tentativo)

In particolare, la funzione in Fig. 2 per ogni elemento *attiva due chiamate* ricorsive di sè stessa. A parte l'inefficienza in tempo, va considerato il fatto che le invocazioni di `sommaSuccFun` *allocano memoria*, quindi viene generato un gran numero di liste ausiliarie (quante?) di cui si perdono i riferimenti (oltre ad essere inutili, quindi, non possono nemmeno essere deallocate). È sufficiente effettuare una sola chiamata ricorsiva e salvare il risultato in una variabile. Questo conduce al programma di complessità lineare in Fig. 3

Solo a titolo di cultura, nei linguaggi funzionali, dove sostanzialmente si programma come nelle nostre specifiche ricorsive e non c'è una vera e propria istruzione di assegnamento, per ovviare a questo problema viene usualmente considerata una espressione del tipo **let** $x = e_1$ **in** $e_2$ **end** che permette di assegnare il risultato di una espressione $e_1$ a una variabile $x$ che poi si può usare nell'espressione $e_2$. Vediamo quindi, come apparirebbe la nostra specifica ricorsiva:

$$\begin{aligned}
\mathsf{sommaSucc}''(\langle\,\rangle) &= \langle\,\rangle \\
\mathsf{sommaSucc}''(\langle\, a\,\rangle) &= \langle\, a\,\rangle \\
\mathsf{sommaSucc}''(a \cdot s) &= \textbf{let } t = \mathsf{sommaSucc}''(s) \textbf{ in } (a + \mathsf{head}(t)) \cdot t \textbf{ end}
\end{aligned}$$

Alcuni, con alterne fortune, hanno preferito cimentarsi con la scrittura iterativa della funzione `sommaSuccFun`. Anche se a lezione ho presentato le versioni `Fun` quasi

```
lista sommaSuccFun(lista L){
    lista M;
    if (isEmpty(L) || isEmpty(tail(L))) return L;
    M=sommaSuccFun(tail(L));
    return addHead(M, head(L)+head(M));
}
```

Figura 3: Versioni funzionale ottimizzata

```
void sommaSuccIterFun(lista L) {
    lista l1=L;
    lista new=NULL;
    lista l2;
    int s;

    while (L){
        s=0;
        l2=L;
        while (l2) {
            s+=l2->val;
            l2=l2->next;
        }
        new=addTail(new, s);
        L=L->next;
    }
    return new:
}
```

Figura 4: Funzione iterativa "ingenua" che genera una nuova lista

esclusivamente in versione ricorsiva, ciò non contravveniva alle specifiche dell'esercizio. La versione più naturale, riportata in Fig. 4, soffre di due fonti di inefficienza: il calcolo delle somme delle code delle liste ad ogni elemento (che ripete molti conti inutilmente) e l'aggiunta in coda (che in una lista semplice è necessariamente lineare nella lunghezza della lista).

Per risolvere la prima fonte di inefficienza, un'idea interessante poteva essere quella già vista al primo esonero nella soluzione al problema del baricentro, dove per calcolare in modo efficiente le somme dei suffissi si calcolava prima la somma totale del vettore e poi, scandendolo in avanti, si andavano a sottrarre via via gli elementi incontrati.

Per risolvere la seconda fonte di inefficienza, si poteva semplicemente creare la nuova lista aggiungendo in testa e alla fine rovesciarla, prima di restituire al chiamante il risultato. L'algoritmo finale in Fig. 5, che richiede 3 scansioni della lista, migliora significativamente la funzione quadratica in Fig. 4.

Un'ultima idea consisteva nel rovesciare subito la lista (allocando nuova memoria), risolvere un problema "duale" (che potremo chiamare somma dei precedenti), che ha una naturale soluzione iterativa, e infine ri-rovesciare (stavolta *senza* allocare nuova memoria) la lista ottenuta.

**Punto 3:** *Scrivere una funzione* `sommaSuccessiviRec` *ricorsiva che* modifica *la lista di ingresso;*

Con un piccolo sforzo le versioni funzionali che allocano una nuova lista, si possono trasformare nelle corrispondenti versioni ricorsive che *modificano la lista* (vedi Fig. 6). Questo passaggio, a volte, è delicato perché occorre stare molto attenti alle

```
lista sommaSuccFunIterEff(lista L) {
    lista aux=L;
    lista M=NULL;
    int s=0;

    while (aux){
        s += aux->val;
        aux=aux->next;
    }
    aux=L;
    while (aux){
        M=addHead(s,M);
        s -= aux->val;
        aux = aux->next;
    }
    return reversePlaceIter(M);
}
```

Figura 5: Funzione iterativa efficiente che genera una nuova lista

```
void sommaSuccRec(lista L){
    if (!L || !L->next) return;
    sommaSuccFun(L->next);
    L->val += L->next->val;
}
```

Figura 6: Funzione ottenuta per l'analogia dalla versione funzionale

eventuali conseguenze dei side-effects. In questo caso, tuttavia, tutto fila abbastanza liscio.

Osserviamo che le funzioni che risolvono questo punto non avevano alcuna necessità di restituire un valore come risultato, visto che la lista originaria *non viene mai* modificata nella sua struttura, ma solo nel contenuto dei nodi (che, ricordiamolo, è memoria passata implicitamente per *indirizzo*).

Possiamo quindi sfruttare il valore di ritorno per restituire al chiamante il valore della somma degli elementi della coda della lista, calcolo che può procedere in parallelo con la modifica della lista. Questa simpatica idea conduce al programma di Fig. 7, che, ovviamente, è il mio preferito.

**Punto 4:** *Scrivere una funzione* `sommaSuccessiviIter` *iterativa che modifica la lista di ingresso.*

```
int sommaSuccRec(lista L) {
    if (!L) return 0;
    L->val+=sommaSuccRec(L->next);
    return L->val;
}
```

Figura 7: Funzione ricorsiva che sfrutta il valore di ritorno

```
void sommaSuccIter(lista L) {
    lista aux=L;
    int s=0, h;

    while (aux){
        s += aux->val;
        aux=aux->next;
    }
    aux=L;
    while (aux){
        h=aux->val;
        aux->val = s;
        s-=h;
        aux = aux->next;
    }
}
```

Figura 8: Funzione Iterativa che modifica la lista di ingresso.

```
lista sommaSuccFunIter(lista L) {
    lista aux=copiaListaIter(L);
    sommaSuccIter(aux);
    return aux;
}

lista sommaSuccFun(lista L) {
    lista aux=copiaListaRec(L);
    sommaSuccRec(aux);
    return aux;
}
```

Figura 9: versioni funzionali che usano le versioni in place

Una funzione iterativa di complessità lineare che modifica è mostrata in Fig. 8: è più naturale della sorella che alloca una nuova lista, perchè modificando la lista in place, non necessita di fare inserzioni in testa e quindi non necessita del rovesciamento finale.

Anche qui, comunque, si poteva applicare l'idea del doppio rovesciamento (sempre in place stavolta) per poter fare le somme "in avanti".

Un'ultima osservazione: avendo programmato le versioni che modificano le liste in place, si possono ottenere facilmente le versioni che allocano nuova memoria (questo non è molto nello spirito della cosa, ma soddisfa alle richieste dell'esercizio!). A questo trucco è dedicata l'ultimo frammento di codice (Fig. 9). È sufficiente creare una copia della lista di ingresso (che dovrebbe essere di programmazione banale), e poi invocare l'algoritmo in place corrispondente.

Come dire, paghi uno, prendi due!