Informatica Generale
Homework 2/2018: Alcuni Strani Tipi

Ivano Salvo – Sapienza Università di Roma – 15/5/2018

L'obiettivo di questo homework (in particolare, l'esercizio **2**) è farvi vedere come la programmazione sia spesso guidata dalla progettazione delle strutture dati. Questa attività è così centrale nella programmazione al punto che i linguaggi di programmazione più usati oggi, i Linguaggi Orientati agli Oggetti (ad esempio SmallTalk (il primo), Java, oppure le 'estensioni' ad oggetti del C, C# ("C sharp") e C++, per esempio) sono motivati dalla necessità di affrontare in modo maturo proprio la progettazione delle strutture dati.

In C le strutture dati vengono progettate combinando arbitrariamente i costruttori di tipo `struct` (record), `[]` (array) e `*` (puntatori) e i mattoncini base, cioè i tipi predefiniti (`int`, `char`, `float` etc.).

Chi volesse saperne di più, al terzo anno c'è la possibilità di scegliere, tra le abilità Informatiche da 3 crediti il corso di Linguaggi di Programmazione, verbalizzabile anche come Altre conoscenze... (piccolo spot pubblicitario ☺).

Per voi, questa volta, sarà prevalentemente un compito "passivo", nel senso che io mi sono sobbarcato l'onere di decidere le strutture dati, e voi dovrete semplicemente scrivere funzioni che operano su di esse. Credo, tuttavia, si tratti comunque di un esercizio istruttivo.

L'esercizio **1** ripropone il problema dell'Homework **P**, esercizio **1** e introduce il tipo "lista di coppie".

L'esercizio **2** sarà probabilmente più difficile da capire che da risolvere, proponendo una struttura dati appena un po' complessa per memorizzare una posizione durante la soluzione del gioco della Torre di Hanoi (dispensa **D3**, sezione **3**).

L'esercizio **3** vi farà tornare nel fantastico mondo di Fibolandia, dove tutti sono matematici buontemponi: qui vedrete le "liste di liste": qui forse qualcuno vagheggerà delle liste *indipendenti* dai tipi dei valori che vengono memorizzati in esse. I curiosi possono cercare qualche informazione nella dispensa **D7**.

Esercizio 1 (Somma di Quadrati Reloaded). Considerate il problema di scrivere un numero come somma di due quadrati (Homework **P**, Esercizio **2**). Questa volta dovrete generate *tutte* (a meno di simmetrie) le coppie i cui quadrati, sommati, danno il numero in input.

Scrivere una funzione C di prototipo:

```
listOfPairs allPairsOfSquares(int n)
  /* PREC: n>=0 */
```

che alloca una lista di coppie (vedi files `listOfPairs.h` e `listOfPairs.c`) con *tutte* le coppie di interi (a, b) tali che $a \leq b$ e $a^2 + b^2 = n$. La lista dovrà essere ordinata in modo che le coppie siano ordinate lessicograficamente. Quindi, avendo due coppie (a_1, b_1) e (a_2, b_2) tali che $a_1^2 + b_1^2 = n = a_2^2 + b_2^2$ se $a_1 < a_2$ nella lista risultato dovrà comparire prima (a_1, b_1) e poi (a_2, b_2).

Esempi: Ricevuto in input un numero che non può essere scritto come somma di due quadrati, ad esempio, la funzione deve tornare lista vuota.

Ricevuto in input il numero 25, la lista calcolata (usando le notazioni della dispensa **D3**, e le usuali notazioni per le coppie) dovrà essere $\langle (0, 5), (3, 4) \rangle$.

Osservazione Pratica: Il file da consegnare dovrà contenere

```
#include "listOfPairs.h"
```

Se la vostra funzione fosse contenuta nel file `soluzione-2-1.c` e dovrà compilare ed eseguire con il comando

```
gcc -std=c99 listOfPairs.c soluzione-2-1.c main-2-1.c.
```

Esercizio 2 (Torre di Hanoi). Nella sezione **3** della dispensa **D3** *"Iterare è umano, ricorrere è divino"* è descritto il problema della Torre di Hanoi: viene descritta una elegante e semplice soluzione ricorsiva che risolve il rompicapo partendo da una prestabilita situazione iniziale. In quel caso, è considerata una soluzione semplicemente una *sequenza di mosse*, dove una mossa è una coppia, (da, a) che indicano, rispettivamente, il piolo di partenza e piolo di arrivo: siccome si può muovere solo il disco in cima alla pila questa identifica univocamente la mossa, mentre non c'è problema di verificarne la legalità (piolo *da* non vuoto, disco più piccolo sopra a disco più grande etc.) perché quella soluzione è *corretta per costruzione*.

Ma cosa accadrebbe se volessimo programmare un gioco che interagisce con un umano che può provare a fare delle mosse a suo piacimento (eventualmente scorrette, che il programma dovrebbe impedire)? O provare a risolvere

un problema più generale e cioè risolvere una qualsiasi istanza della Torre di Hanoi, con i dischi arbitrariamente posizionati sui pioli?[1] Occorre ovviamente memorizzare lo *stato* del gioco.

Nel file `"hanoi.h"` (da includere nella vostra funzione) trovate le definizioni del tipo `hanoiState` che contiene informazione su: 1. numero dei pioli (esistono variazioni del gioco con 4 o più pioli); 2. un array di `rodState`, tipo che descrive lo stato di ciascun piolo. A sua volta, `rodState` contiene: 1. il numero massimo di dischi, *high*; 2. il numero di dischi effettivamente presenti sul piolo, *top*; 3. l'array *disks* con le dimensioni di ciascun disco sul piolo (in posizione 0 c'è il disco alla base, in posizione 1 quello immediatamente sopra, e così via). I singoli dischi sono rappresentati da un numero intero tra 1 e $nDisk$ (numero dei dischi) che ne rappresenta la dimensione (in modo astratto. Se preferite rappresenta il numero d'ordine del disco, con 1 il più piccolo e $nDisk$ il più grande). Nei nostri esempi, $nDisk$ corrisponde sempre a $high$, numero massimo di dischi ospitabile su ciascun piolo.

Voi dovete implementare le seguenti tre funzioni:

```
int canMove(hanoiState *h, int from, int to);
void move(hanoiState *h, int from, int to);
int moveSeq(hanoiState *h, listOfPairs l);
```

dove:

- `int canMove(hanoiState *h, int from, int to);` verifica (*senza modificare* lo stato) se è possibile spostare un disco dal piolo *from* al piolo *to*;

- `void move(hanoiState *h, int from, int to);` sposta il disco in cima al piolo *from* e lo mette in cima al piolo *to* (ovviamente, questa dovrebbe essere chiamata solo *dopo* aver verificato con `canMove` se la mossa sia legale);

- `int moveSeq(hanoiState *h, listOfPairs l);` esegue una sequenza di mosse (memorizzata come una lista di coppie come all'esercizio **1**, interpretando *fst* come *from* e *snd* come *to*) *arrestandosi*, eventualmente, alla prima mosse illegale. La funzione ritorna il numero di mosse legali eseguite.

Stavolta la vostra soluzione (diciamo nel file `soluzione-2-2.c`) deve compilare con il comando:

```
gcc -std=c99 hanoi.c listOfPairs.c soluzione-2-2.c main-2-2.c
```

[1] l'analisi di questo problema potrebbe essere tema di tesi di laurea. A volte ci penso, ma di recente ho scoperto che qualcuno c'ha dedicato la vita e scritto libri!

Esercizio 3 (Ritorno a Fibolandia). Dopo l'esercizio **3** dell'Homework **P**, torniamo a Fibolandia. Questa volta occorre 'generare' tutte le sequenze ordinate di monete che permettono di dare un certo resto. Nei files `"listOfLists.h"` e `"listOfLists.c"` trovate definizioni di tipo e funzioni base delle "liste di liste", che nel campo informazione hanno un puntatore alla testa di una lista di interi, come quelle viste a lezione. Dovete quindi scrivere una funzione di prototipo:

```
listOfLists changeChoices(int n)
  /* PREC: 0<k<n */
```

che restituisce in output la lista di *tutti* i possibili resti. Le liste *devono* essere ordinate in modo crescente, e, a loro volta, nella lista di liste, le liste devono apparire nell'ordine lessicografico.

Dovete ricordarvi di mettere `#include "listOfLists.h"` nella vostra soluzione per poter usare le funzioni `cons` e `consLoL` etc.

Anche stavolta la vostra soluzione (diciamo nel file `soluzione-2-3.c`) deve compilare con il comando:

```
gcc -std=c99 listOfLists.c soluzione-2-3.c main-2-3.c
```

Esempio: Se l'input della funzione fosse 9, il risultato sarebbe la seguente lista di liste (esattamente nell'ordine come di seguito):

$$\langle\langle 1,1,1,1,1,1,1,1,1\rangle\langle 1,1,1,1,1,1,1,2\rangle, \langle 1,1,1,1,1,1,3\rangle, \langle 1,1,1,1,1,2,2\rangle,$$
$$\langle 1,1,1,1,2,3\rangle, \langle 1,1,1,1,5\rangle, \langle 1,1,1,2,2,2\rangle, \langle 1,1,1,3,3\rangle, \langle 1,1,2,2,3\rangle,$$
$$\langle 1,1,2,5\rangle, \langle 1,2,2,2,2\rangle, \langle 1,2,3,3\rangle, \langle 1,3,5\rangle, \langle 1,8\rangle, \langle 2,2,2,3\rangle, \langle 2,2,5\rangle, \langle 3,3,3\rangle\rangle$$