Informatica Generale
Homework 1/2020: The best of '10s Homeworks.

Ivano Salvo – Sapienza Università di Roma – 7/5/2020

L'obiettivo di questo Homework 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[1].

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.).

Esercizio 1 (2018) (Torre di Hanoi). Nella dispensa **D3** sezione **3** *"Iterare è umano, ricorrere è divino"* è descritta una soluzione ricorsiva al rompicapo della Torre di Hanoi. Il programma genera una *sequenza di mosse*, dove una mossa è una coppia di interi (da, a) che indicano, rispettivamente, il piolo di partenza e piolo di arrivo: potendo solo togliere il disco in cima a un piolo mettendolo in cima ad un altro piolo, una mossa è univocamente identificata dai numeri che rappresentano i due pioli. Non c'è il problema di verificare la legalità di una mossa (piolo *da* non vuoto, piolo *da* diverso dal piolo a, disco più piccolo sopra a disco più grande) perché quella soluzione è *corretta per costruzione*.

Tuttavia, se vogliamo programmare un gioco che interagisce con un umano che può provare a fare delle mosse a suo piacimento (eventualmente scorrette

[1] 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 ☺).

che il programma deve riconoscere come illegali) dobbiamo 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 dischi; 2. numero dei pioli (esistono variazioni del gioco con 4 o più pioli); 3. un array di `rodState`, tipo che descrive lo stato di ciascun piolo. A sua volta, `rodState` contiene: (a) il numero massimo di dischi, *high*; (b) il numero di dischi effettivamente presenti sul piolo, *top*; (c) 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 $nDisks$ (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, $nDisks$ corrisponde sempre a $high$, numero massimo di dischi ospitabile su ciascun piolo (ma anche qui esistono variazioni del problema). 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, listOfMoves 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, listOfMoves l);` esegue una sequenza di mosse (memorizzata come una lista di coppie in cui nodi contengono i due interi *from* e *to*) *arrestandosi*, eventualmente, alla prima mossa illegale. La funzione ritorna *in ogni caso* il numero di mosse legali eseguite.

Esercizio 2 (2017)(ULAM RELOADED) Riconsideriamo la successione di Ulam definita nell'Homework **1**, Es. **2**. Stavolta dovete scrivere una funzione C di prototipo:

```
int nextU(listDCFirstLast U){
     /* PREC: U contiene ordinatamente
        i primi k>=2 elementi della successione u */
```

che, sotto la precondizione che `U` contenga *ordinatamente* i primi $k \geq 2$ elementi della successione u, calcola il $k+1$-esimo, lo restituisce come risultato e lo aggiunge in coda a `U`. Il tipo `listDCFirstLast` è il tipo *lista doppiamente concatenata* in cui c'è un descrittore della struttura dati contenente un puntatore al primo e all'ultimo elemento della lista. Inoltre, ogni nodo contiene un pointer al successivo e un pointer al precedente. Queste liste sono definite nel file `listDC.h` (che **dovete** includere) e alcune utilità sono nel file `listDC.c` che potete usare (ma **non dovete** consegnare eventuali funzioni che usate presenti in questo file). Per esemplificare ulteriormente cosa devono fare le funzioni richieste, confido che risulti eloquente la Figura 1.

Risultato in memoria ***dopo*** l'esecuzione di `u=initializeU()`(a) e ***dopo*** l'esecuzione di 1, e 4 chiamate di `nextU(u)` (× per NULL)

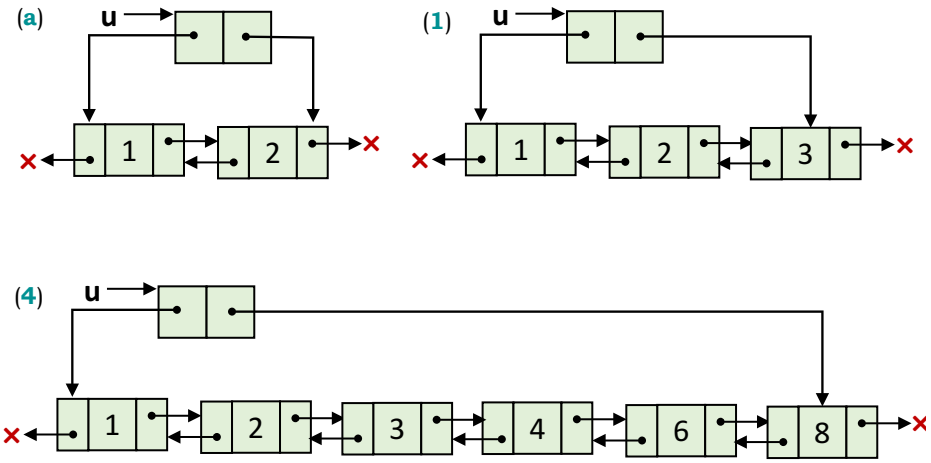

Figura 1: Esempio di esecuzione della funzione `nextU` (Esercizio **2**).

Esercizio 3 (2019)(Crivello di Eulero) Tutti dovreste conoscere l'algoritmo del crivello di Eratostene per generare tutti i numeri primi fino a un certo n. In questo algoritmo, prima vengono scritti tutti i numeri fino a n e poi vengono cancellati i multipli di 2, poi quelli di 3, e in generale quelli del prossimo numero non ancora cancellato p (che è necessariamente un primo).

Anche cominciando un ciclo di cancellazione da p^2 e proseguendo a passo p (cancellando quindi $p^2 + ip$ fino a che $p^2 + ip \leq n$), viene fatto del lavoro inutile. Infatti, ad esempio, cancellando i multipli di 3, vengono cancellati il 12, il 18, il 24 etc. che sono anche multipli di 2. Lo stesso accade con i multipli di 5, in quanto vengono cancellati il 30, il 40, il 45 etc. Si può dimostrare che la complessità di questo algoritmo è $\mathcal{O}(n \ln \ln n)$ per generare i primi fino ad n. Evitando di ri-cancellare multipli già cancellati, possiamo ottenere un algoritmo, il *crivello di Eulero* appunto, di complessità $\mathcal{O}(n)$ in quanto $\ln \ln n$ è il numero medio di divisori primi di un numero composto (e quindi anche il numero medio di volte che un numero composto viene ricancellato durante l'esecuzione del Crivello di Eratostene).

Tuttavia non è ovvio "saltare" in modo efficiente i numeri già cancellati per trarre vantaggio da quest'idea. La soluzione che vi propongo di implementare consiste nell'usare un vettore di coppie di naturali, *succ* e *prec*, come una *lista doppiamente concatenata* in cui nella posizione i, se i non è stato cancellato, *succ* è il numero di posizioni che occorre saltare per andare al prossimo numero non cancellato, mentre *prec* è il numero di posizioni che occorre saltare (all'indietro) per andare al precedente numero non cancellato.

Definiamo un tipo `Pair` che è una coppia di interi `succ` e `prec` e definiamo un vettore di `Pair` (vedi file `eulero.h`). Questo vettore va inizializzato con tutti 1 (che significa appunto che tutti i numeri sono ancora potenziali primi). Quindi lo stato del vettore, inizialmente è il seguente (dove al solito # significa 'non rilevante'):

pos	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
succ	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
prec	#	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Dopo aver cancellato i multipli di due, il vettore avrà i seguenti valori:

pos	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
succ	1	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#
prec	#	1	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#	2	#

Ora, partendo da 3 posso facilmente saltare sui numeri non cancellati. Moltiplicando questi per 3 ottengo quelli da cancellare in questa iterazione, e cioè 9, 15, 21, ottenendo la seguente situazione:

pos	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
succ	1	2	#	2	#	4	#	#	#	2	#	4	#	#	#	2	#	4	#	#	#	2	#
prec	#	1	#	2	#	2	#	#	#	4	#	2	#	#	#	4	#	2	#	#	#	4	#

Nell'esempio, a questo punto ho finito, perché il prossimo numero non cancellato è il 5 e $5^2 > 24$. Partendo da 2 e scorrendo il vettore usando i puntatori *succ* posso stampare tutti i numeri non cancellati che sono a questo punto necessariamente primi (vedi funzione `printPrimes` nel main fornito).

Voi dovete scrivere una funzione:

```
Pair* eulerSieve(int n);
   /* PREC: n>2 */
```

che restituisce un vettore di coppie da cui sia possibile ricostruire tutti i numeri primi da 2 a n.

Osservazioni: I puntatori *prev* servono essenzialmente per effettuare in modo efficienti le operazioni di cancellazione. Le cancellazioni sono 'problematiche' perché sono operazioni distruttive sulla struttura dati e potrebbero, se fatte troppo presto o con poca cura, rendere inconsistente lo stato del vettore.

Sperimentazioni: Verificare che questo programma risulta effettivamente più efficiente del crivello di Eratostene. Ovviamente, il guadagno asintotico ($\ln \ln n$) è modesto e fa operazioni più complicate. Occorrerà provarlo per un qualche n sufficientemente grande (ordine di migliaia o milioni...).

Note Pratiche e Concettuali

Potete usare le funzioni contenute nei file `listOfMoves.c`, `hanoi.c` e `listDC.c`, che forniscono alcune operazioni base come inserimento, creazione e stampa di liste, ma **non dovete consegnarle**. Per compilare il tutto, al solito, dovreste usare i comandi (supponendo le funzioni richieste più eventuali funzioni ausiliarie nei files `LilyEvans.1.c`, `LilyEvans.2.c` e `LilyEvans.3.c`):

```
gcc -std=c99 listOfMoves.c hanoi.c main-2020-2-1.c LilyEvans.1.c
gcc -std=c99 listDC.c main-2020-2-2.c LilyEvans.2.c
gcc -std=c99 main-2020-2-3.c LilyEvans.3.c
```

Dovete includere le librerie `listDC.h`, `listOfMoves.h`, `hanoi.h`, `eulero.h` (oltre alle librerie standard `stdlib.h`, e `stdio.h` ovunque necessario. Questo non causa problemi. Se andate a vedere le definizioni di `list.h` vedrete che le definizioni stanno dentro un costrutto:

```
#ifndef LIST_H
#define LIST_H
typedef
...
#endif
```

questo permette di compilare separatamente i files, ed evitare eventuali doppie definizioni. Tutte le librerie standard sono ovviamente fatte in questo modo.

Osservate infine che le "nostre" librerie vanno incluse con i comandi:

```
#include "listOfMoves.h"
#include "hanoi.h"
```

mentre le librerie di sistema con le parentesi angolate. Questo informa il compilatore di cercare nella directory corrente invece che nelle directory in cui sono archiviate le librerie standard.