Informatica Generale
Homework 3/2018
Per Fare un Albero...

Ivano Salvo – Sapienza Università di Roma – 1/6/2018

Esercizio 1 (Albero delle chiamate ricorsive). Considerate la seguente funzione ricorsiva che calcola i coefficienti binomiali.

```
int cbin(int n, int k){
   if (n==k || n==0) return 1;
   return cbin(n-1,k-1)+cbin(n-1,k);
   }
```

Scrivere una funzione `C` di prototipo:

```
cBinTree cBinInvocations(int n, int k)
  /* PREC: n>=0, 0<=k<=n */
```

che genera l'albero delle chiamate ricorsive della funzione `int cbin(int n, int k)`, memorizzando in ogni nodo i valori dei parametri n e k e il valore calcolata

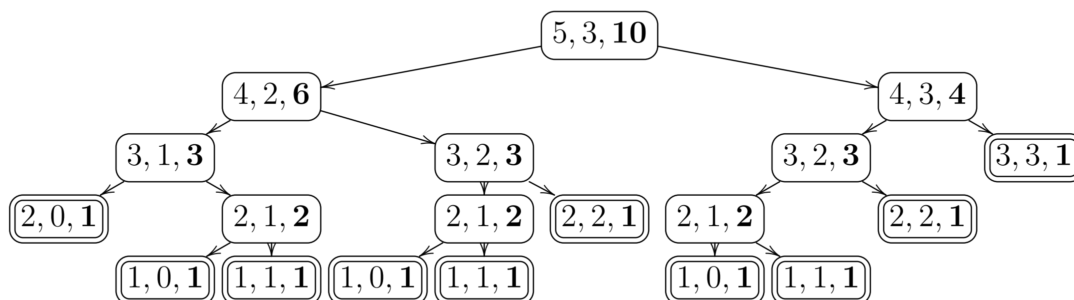

Figura 1: Albero delle chiamate ricorsive di `cbin(5,3)` (in grassetto i risultati ritornati), che deve essere generato dalla funzione `cBinInvocation(5,3)`.

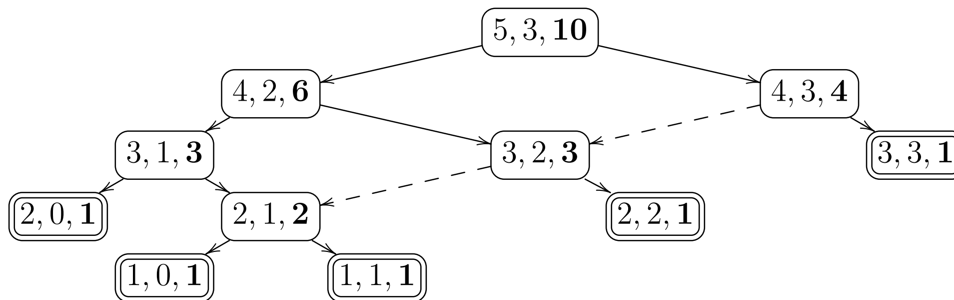

Figura 2: “Albero” (grafo aciclico) che deve essere generato dalla funzione `cBinIvnocationSharing(5,3)`, evitando replicazione di sottoalberi.

dalla in tale chiamata. Ogni nodo di un `cbinTree` contiene tre valori (vedi file `cbinTree.h`).

Esempio Se fosse invocata con paramentri $n = 5$ e $k = 3$, la funzione `cBinInvocations` dovrebbe produrre l'albero in Fig. 1, dove il valore ritornato è scritto in grassetto. Notate, in vista del prossimo esercizio, che ci sono numerosi sotto-alberi ripetuti nell'albero delle chiamate. Osservate anche che, ogni cammino rappresenta le possibili attivazioni di `cbin` che possono essere presenti nello stesso momento sullo stack di sistema.

Esercizio 2 (Grafo Aciclico delle chiamate ricorsive). Come nell'esercizio precedente, ma questa volta allocando un unico nodo per eventuali chiamate ripetute (con gli stessi valori per i parametri n e k), evitando quindi la replicazione di sotto-alberi. Osservate che in questo caso ci sono cammini diversi che finiscono nello stesso nodo (osservate che non occorre cambiare la definizione del tipo `cBinTree`, né funzioni di stampa o di visita).

Dovete scrivere una funzione C di prototipo :

```
cBinTree cBinInvocationSharing(int n, int k)
  /* PREC: n>=0, 0<=k<=n */
```

Esempio: In Fig. 2, l'“albero” risultante sullo stesso esempio dell'Esercizio **1**. Gli archi tratteggiati sono quelli che “riportano” su nodi già precedentemente allocati.

Esercizio 3 (Albero di un gioco). In questo esercizio dovrete generare *tutte* le posizioni del Tris (o Filetto, o TicTacToe, o oxo – quest'ultimo il nome a cui ho ispirato i nomi dei tipi).

Ancora una volta si tratta di un albero, ma stavolta *non* binario: ad esempio, nel gioco completo, la radice contiene la posizione vuota, e quindi i ‘possibili’

futuri immediati sono 9, a seconda di dove il primo giocatore posizioni il proprio simbolo. I nodi al livello successivo avranno al più 8 futuri e così via: il numero di posizioni è quindi superiormente limitato da 9! (in realtà sono meno perchè a partire dal livello 5 alcune partite sono finite in quanto uno dei due giocatori ha raggiunto una posizione vincente). Volendo fare i raffinati, anche qui si potrebbe dare una rappresentazione compatta evitando replicazioni di sotto-alberi come nell'esercizio 2 (ma non richiesto), in quanto, diverse sequenze di mosse potrebbero portare alla stessa posizione.

Ogni nodo dell'albero conterrà (vedi file `OxOTree.h`):

1. L'informazione `turn` su chi (`o` oppure `x`) effettuerà la prossima mossa;
2. una matrice `pos` di caratteri 3×3, allocata dinamicamente, contenente i caratteri `o`, `x` oppure '.' (quest'ultimo per casella vuota);
3. il numero di caselle libere `free`;
4. un vettore di puntatori `next` ai sotto-alberi di dimensione `free`;

Qualora una griglia sia piena, il valore di `free` sarà 0 e il vettore ai sotto-alberi sarà il puntatore `NULL`. Analogamente, quando la partita è finita (perchè la griglia contiene un tris) anche se `free` non è 0, il vettore ai sotto-alberi sarà il puntatore `NULL`. Voi dovrete scrivere due funzioni `C` di prototipo:

```
OxOTree genera(char** p, int m, int n, char t);
char valuta(OxOTree g, char t);
```

dove `genera(p,3,3,t)` genera tutto l'albero del gioco a partire dalla posizione p (di dimensione 3×3), e t è un carattere che dice a chi tocca muovere. Se la posizione non fosse legale (ad esempio: 2 tris già presenti, deve muovere un giocatore che ha già un simbolo in più) `genera` deve restituire `NULL`.

Viceversa, `valuta(g,t)` valuta la situazione visitando l'albero del gioco g, dal punto di vista di del giocatore t, codificato con uno dei caratteri `o` e `x`, e restituisce un risultato in accordo con la seguente valutazione:

?: se la partita fosse l'albero vuoto (ad esempio prodotto da `genera` in quanto chiamata con una posizione illegale);

W: la partita è *vinta* dal giocatore t: c'è un tris vincente nella posizione iniziale di g, che evidentemente non può che essere una foglia;

L: la partita è *persa* dal giocatore t: c'è un tris vincente dell'avversario nella posizione iniziale di g, che evidentemente non può che essere una foglia;

D: la partita è *patta*: la griglia è piena e non ci sono vincitori (anche qui g non può che essere una foglia);

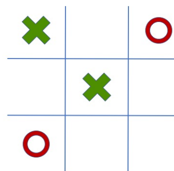

Figura 3: `o` vincente

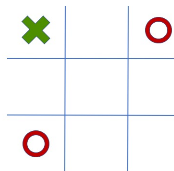

Figura 4: `x` perdente

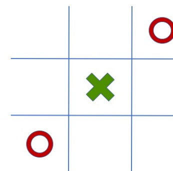

Figura 5: `x` pari

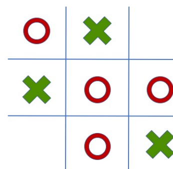

Figura 6: `x` pattante

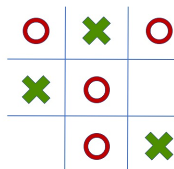

Figura 7: `x` sfavorevole

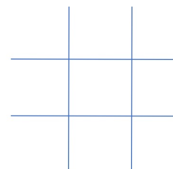

Figura 8: pari!

w: la partita può essere *forzatamente* vinta dal giocatore t, che ha una *strategia vincente*: a ogni livello, esiste una mossa di t che per ogni mossa dell'avversario porta a posizioni vinte (vedi Fig. 3);

l: la partita può essere *forzatamente* vinta dall'avversario, che ha una *strategia vincente*: a ogni livello, per ogni mossa di t esiste una mossa dell'avversario che conduce alla sconfitta di t (vedi Fig. 4);

d: *pattante* in quanto la partita finirà necessariamente patta: *tutte le foglie* finiscono in una posizione di tipo **D** (vedi Fig. 6);

f: la posizione è *favorevole*: non ci sono strategie vincenti, ma tutte le 'future' posizioni non patte sono vinte da t;

s: la posizione è *sfavorevole*: non ci sono strategie vincenti, ma tutte le 'future' posizioni non patte sono vinte dall'avversario (vedi Fig. 7);

p: la posizione è *pari*: non ci sono strategie vincenti, ma entrambi i giocatori possono ancora vincere (ovviamene a causa di un errore, vedi Fig. 5 e 8);

Trovate diverse funzioni già scritte per inizializzare, stampare, clonare, aggiungere/togliere un segno da una posizione in `posizione.h` e `posizione.c` e per inizializzare la radice dell'albero in `OxOTree.h` e `OxOTree.c`.