TPFI 2023/24
Hw 5: A taste of C

assegnato: 18 maggio 2024, consegna 5 giugno 2024

1. Type Safety

L'esistenza del tipo `void*` è il più evidente segno del fatto che in C sia un linguaggio non type-safe, nonostante la buona volontà dei compilatori moderni che hanno proibito diverse possibilità presenti nel C standard delle origini.

Scrivere un programma C che dà risultati diversi se compilato su un'architettura big-endian (il primo byte è il più significativo, tipico di Motorola, Sun, IBM...) oppure su una little-endian (il primo byte è il meno significativo, tipico di Intel e Digital...).

Per chi disponesse di un Mac con processori di famiglia M, potrebbe farmi sapere se sono little o big-endian. Io non lo so ☺.

2. Array Mutabili

Scrivere una funzione C di complessità $\Theta(n \log n)$ che elimina i duplicati da un vettore (analogo dell'esercizio **1** dell'Homework **1**), compattando gli elementi a sinistra, e restituendo come risultato il numero degli elementi rimasti.

Potete ovviamente usare strutture dati per mimare la vostra soluzione Haskell, ma quale potrebbe essere il modo migliore per rendere reversibile un ordinamento di un array in C?

Riflettere bene sul prototipo corretto della funzione, in accordo con il galateo del C e riflettere anche su quali potrebbero essere (e come definirle) eventuali strutture dati ausiliarie.

L'algoritmo elementare per risolvere questo problema è $\Theta(n^2)$, dove n è la lunghezza della lista.

3. Quello che in Haskell non si può fare I: Dati "non-funzionali"

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);
   }
```

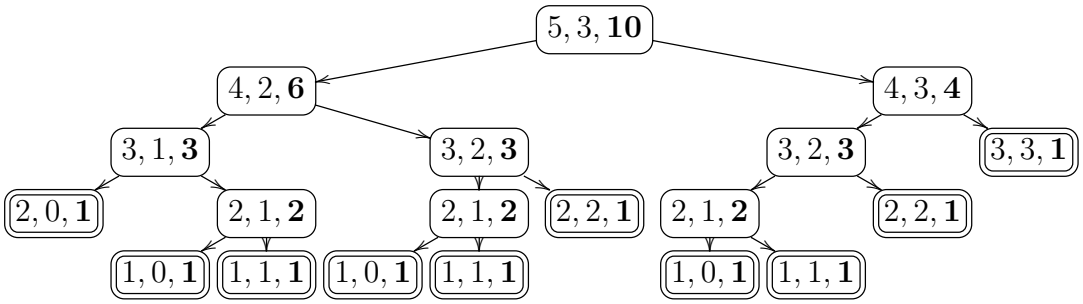

Figura 1: Albero delle chiamate ricorsive di `cbin(5,3)` (in grassetto i risultati ritornati), che deve essere generato dalla funzione `cBinInvocation(5,3)`.

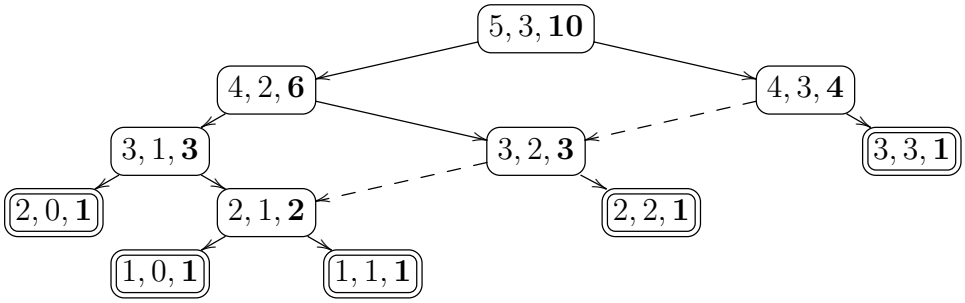

Figura 2: "Albero" (grafo aciclico) che deve essere generato dalla funzione `cBinIvnocationSharing(5,3)`, evitando replicazione di sottoalberi.

Dare prima la definizione di un tipo `cBinTree` adatto a memorizzare in un albero binario i valori dei parametri n e k e il valore calcolato dalla in tale chiamata (ogni nodo di un `cbinTree` contiene quindi tre valori.).

Scrivere poi una funzione `C` 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 calcolato dalla in tale chiamata (vedi Fig. 1).

Siccome ci sono numerosi sotto-alberi ripetuti nell'albero delle chiamate, è interessante costruire il grafo aciclico delle chiamate ricorsive, 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 (vedi Fig. 2). Osservate che in questo caso ci sono cammini diversi che finiscono nello stesso nodo e osservate anche che non occorre cambiare la definizione del tipo `cBinTree`, né funzioni di stampa o di visita.

Esempio: In Fig. 2, l'"albero" risultante sullo stesso esempio. Gli archi tratteggiati sono quelli che "riportano" su nodi già precedentemente allocati.

4. Quello che in Haskell non si può fare II: Crivello di Eulero

Scrivere una funzione C che implementi il *crivello di Eulero*. Non è ovvio "saltare" in modo efficiente i numeri già cancellati per trarne vantaggio nelle successive cancellazioni. 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.

Potete ad esempio definire un tipo `Pair` che è una coppia di interi `succ` e `prec` e definiamo un vettore di `Pair`. 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 # 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 (scrivere una funzione `printPrimes` allo scopo).

Voi dovete scrivere una funzione: `Pair* eulerSieve(int n);` 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 efficiente le operazioni di cancellazione. Le cancellazioni sono 'problematiche' perché sono operazioni distruttive sulla struttura dati e potrebbero, se fatte 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 è modesto ($\Theta(n)$ invece che $\Theta(n \ln \ln n)$) e fa operazioni più complicate. Occorrerà provarlo per un qualche n sufficientemente grande (ordine di migliaia o milioni...).