Correzione Primo Esonero, Esercizio 2

Ivano Salvo – Sapienza Università di Roma

Anno Accademico 2015-16

Esercizio 2: *Considerare le combinazioni di n oggetti presi k a k. Supponiamo di rappresentare una di tali combinazioni con un vettore di k interi compresi tra* 1 *ed n memorizzati nel vettore in ordine* crescente*.*

Punto 1 *Scrivere una funzione C* `int nextCbn(int c[], int n, int k)` *che presa in input una combinazione c di k interi tra* 1 *ed n, modifica c in modo che rappresenti la prossima combinazione nell'ordine lessicografico e torna 1. Se la combinazione c in ingresso è l'ultima,* `nextCbn` *torna 0.*

Ovviamente, occorre prima capire il problema. Scriviamo, ad esempio, le 10 combinazioni di numeri da 1 a 5, presi 3 a 3, nell'ordine lessicografico: [1,2,3], [1,2,4], [1,2,5], [1,3,4], [1,3,5], [1,4,5], [2,3,4], [2,3,5], [2,4,5], [3,4,5]. Osservate che, finchè possibile, è sufficientemente incrementare l'ultimo elemento, fino alla combinazione $[1, 2, 5]$. A quel punto, occorre incrementare il 2, ma la prossima combinazione semplicemente contiene, da quel punto in poi, numeri consecutivi in ordine crescente, ed è infatti $[1, 3, 4]$. Facciamo un ultimo esempio. Arrivati a $[1, 4, 5]$ non è possibile nè incrementare il 5, nè il 4, perchè il 5 è già presente nella combinazione. A questo punto è possibile incrementare solo l'1, continuando, riempendo ancora a destra con numeri consecutivi, ottenendo $[2, 3, 4]$. Ovviamente, giunti a $3, 4, 5$, non c'è nessun numero che si possa incrementare, e quindi non ci sono combinazioni successive e la nostra funzione tornerà 0.

L'algoritmo, che ovviamente dipende dal fatto che rappresentiamo le combinazioni con i numeri in ordine crescente, consiste nel trovare, *partendo da destra*, il primo indice in cui è possibile incrementare. Si incrementa quel numero di 1 e poi si posizionano i suoi successivi, uno alla volta alla sua destra. Per fare questo lavoro, è conveniente definire una funzione `void firstCmb(int* c, int l, int q)` che carica nel vettore c, l elementi consecutivi a partire dal valore q come in Fig. 1.

```
void firstCmb(int* v, int l, int q) {
     int i;
     for(i=0; i<l; i++) v[i]=q++;
}
```

Figura 1: Funzione che sistema la parte destra della combinazione

Come riconoscere il punto dove incrementare? Chiaramente per l'ultimo elemento in posizione $k-1$, deve essere $c[k-1] < n$. Per un elemento interno in posizione $i < k-1$, deve essere $c[i] < c[i+1]-1$. Quindi non appena si è trovato il punto giusto i, si sistema la coda della combinazione, invocando `firstCmb(&c[i],k-i,c[i]+1)`, che incrementa $c[i]$ di 1 e riempie la coda con numeri consecutivi nelle $k-i$ posizioni rimanenti (senza tale funzione è comunque sufficiente un facile ciclo `for`).

Alternativamente, è sufficiente osservare che nella zona "non incrementabile", $c[i] = n-k+i+1$ (idea non mia, ma emersa in un compito e che rispetto alla mia soluzione, consente di non trattare a parte l'ultimo elemento).

Se non si trova nessun punto in cui sia possibile incrementare, si torna 0 senza fare nulla. Il risultato in Fig. 2.

```
int nextCmb(int c[], int n, int k){
  /* PREC: c vettore di k elementi,
   * contiene numeri distinti tra 1 ed n,
   * c ordinato crescente
   */

   for (int j=k-1; j>=0; j--)
       if (c[j]<n-k+j+1){
          firstCmb(&c[j],k-j,c[j]+1);
          return 1;
       }
   return 0;
   }
```

Figura 2: Funzione `nextCmb`

Punto 2: *Usare la funzione* `nextCmb` *del punto precedente per scrivere una funzione iterativa C* `allCbn(int n, int k)` *che stampa in ordine crescente* tutte *le combinazioni di interi da* 1 *a n presi k a k [*Osservazione*: non occorre memorizzare le combinazioni!].*

Questo esercizio è un calcio di rigore al novantesimo! E fa portare a casa i 3 punti. Avendo la funzione `nextCmb`, è sufficiente caricare un vettore c la prima combinazione (osservare che qui posso riutilizzare `firstCmb`), e poi iterare `nextCmb` su c finchè tale funzione non torna 0. Ovviamente, ad ogni ciclo si stampa c. Il risultato in Fig. 3, dove viene usata una funzione `printCmb` per ottenere una stampa opportuna di una combinazione.

Punto 3: *Scrivere una funzione C ricorsiva* `allCbnRec(int n, int k)` *che usa lo schema ricorsivo del programma che calcola i coefficienti binomiali per stampare tutte le combinazioni di interi da* 1 *a n presi k a k.[omissis lungo suggerimento]*

Tutte le combinazioni di lunghezza k di n elementi, sono tutte le combinazioni che cominciano con 1, in cui devo sistemare gli $n-1$ elementi rimanenti a partire

```
void allCmb(int n, int k){
   int i;
   int v[k];

   firstCmb(v,k,1);
   printV(v,k);
   while (nextCmb(v,n,k)) printCmb(v,k);
}
```

Figura 3: Funzione `allCmb`

dal 2 **e** tutte le combinazioni che **non** contengono 1, in cui devo sistemare i numeri da 2 a n nelle k posizioni disponibili, e, ovviamente riapplicare ricorsivamente il ragionamento. Dovendole solo stampare sarà necessario avere tra i parametri della funzione un vettore che memorizzerà la combinazione in costruzione. Inoltre, quando k è 0 vuol dire che ho sistemato nella combinazione in costruzione tutti gli elementi. Quando $n = k$, allora significa che sono rimasti n elementi da sistemare in n posizioni e non possono che essere gli ultimi rimasti.

Riassumendo, dovrò scrivere una funzione ausiliaria `allCbnRecAux(int n, int k,int c[], int q, int j, int l)` in cui c è la combinazione in costruzione, q è il primo numero da sistemare nel sottoalbero ricorsivo corrente, j è il punto del vettore in cui sono arrivato, l è la lunghezza originaria delle combinazioni in costruzione. (in realtà j è sempre $l - k$ e potrebbe essere evitato, magari passando il pointer al primo punto "libero" del vettore c).

Il risultato in Fig. 4. Osservare che se $k = 0$, la funzione `firstCmb` non fa nulla (infatti ho 0 elementi da sistemare). Osservare anche che la seconda chiamata sovrascriverà la casella j di c, per cui non occorre preoccuparsi di annullare gli effetti delle assegnazioni (tipo `c[j]=q;`).

```
void allCmbAux(int n, int k, int c[], int q, int j, int l){

  if (k==0 || n==k){
     firstCmb(&c[j],l-j,q);
     printCmb(c,l);
     return;
   }
   c[j]=q;
   allCmbAux(n-1,k-1,c,q+1,j+1,l);
   allCmbAux(n-1,k,c,q+1,j,l);
}

void allCmbRec(int n, int k){
  int c[k];
  allCmbAux(n,k,c,1,0,k);
}
```

Figura 4: Funzione `allCmbRec`