Algoritmi e Strutture Dati - Seconda provetta
31/05/12

Esercizio 1

Soluzione calcolata con il codice `Flow.java` contenuto nella cartella dei compiti. Dateci un'occhiata, è interessante per la sua compattezza – a parte l'inizializzazione delle matrici e i commenti, sono solo 21 righe di codice.

Esercizio 2

E' possibile notare che quando si incontra una sequenza crescente, è possibile scegliere uno e uno solo dei suoi elementi; simili considerazioni valgono per una sequenza decrescente. Inoltre, è possibile notare che il primo elemento può essere sempre considerato come appartenente alla sequenza. A questo punto, si utilizza una variabile dir per memorizzare la "direzione" in cui si sta andando nella sequenza zig-zag. Tutte le volte che si incontra un elemento che soddisfa la sequenza zig-zag, si incrementa un contatore e si cambia direzione.

```
zigzag(integer[] Y, integer n)
  integer dir ← 1
  integer tot ← 1
  integer last ← 1
  integer i ← 1
  while i ≤ n do
    if (Y[i] − Y[last]) · dir > 0 then
      last ← i
      dir ← −dir
      tot ← tot + 1
    i ← i + 1
  return tot
```

Esercizio 3

E' possibile risolvere il problema tramite programmazione dinamica utilizzando la seguente formulazione ricorsiva. Sia $C[i,t]$ il numero di modi distinti per dare il resto t con i primi i tagli di banconote; $C[i,t]$ può essere espresso ricorsivamente come segue:

$$C[i,t] = \begin{cases} 1 & t = 0 \\ 0 & t < 0 \vee i = 0 \\ C(i-1,t) + C(i,t-B[i]) & \text{altrimenti} \end{cases}$$

Come vedete è molto semplice: esiste un solo modo per restituire un resto $t = 0$, indipendentemente dalle banconote rimaste. Se non ho più banconote oppure ho un resto negativo, non c'è modo di dare il resto richiesto quindi restituisco 0. Altrimenti, posso fare due scelte: smetto di usare la banconota i-esima oppure la uso ancora una volta, e sommo il numero di modi che ottengo nei due casi. Notate che quando considero una banconota i, posso selezionarla quante volte voglio, ma non posso più selezionare banconote le banconote in $B[i+1 \ldots n]$; questo evita di generare permutazioni.
L'algoritmo può essere scritto tramite memoization:

```
resto(integer[] B, integer i, integer t, integer[][] C)
  if t = 0 then
    return 1
  if t < 0 or i = 0 then
    return 0
  if C[i,t] = ⊥ then
    C[i,t] = resto(B, i − 1, t, C) + resto(B, i, t − B[i], C)
  return C[i,t]
```

Il costo computazionale è $O(nT)$; l'esecuzione inizia con $\mathsf{resto}(B, n, T)$.

Consideriamo il caso di esempio: $B[\,] = \{1, 2, 5\}, n = 3, T = 4$

$$\begin{aligned}
C[3,4] &= C[2,4] + C[3,-1] = 3 + 0 = 3\\
C[2,4] &= C[1,4] + C[2,2] = 1 + 2 = 3\\
C[2,2] &= C[1,2] + C[2,0] = 1 + 1 = 2\\
C[1,4] &= C[0,4] + C[1,3] = 0 + 1 = 1\\
C[1,3] &= C[0,3] + C[1,2] = 0 + 1 = 1\\
C[1,2] &= C[0,2] + C[1,1] = 0 + 1 = 1\\
C[1,1] &= C[0,1] + C[1,0] = 0 + 1 = 1
\end{aligned}$$

Esercizio 4

Si noti innanzitutto che avendo $2n$ valori, il mediano non è un singolo valore, ma una coppia. Restituiremo quindi una coppia di valori, non un valore singolo.
Se n è dispari, vi è un solo mediano per entrambi i vettori e si può considerare il mediano in posizione m_x di X e il mediano m_y di Y; se n è pari, vi sono due mediani in entrambi i vettori, e consideriamo il mediano "sinistro" in posizione m_x per X e il mediano "destro" in posizione m_y per Y. Supponendo di considerare il vettore X dall'indice b_x (begin) all'indice e_x (end) e il vettore Y dall'indice b_y all'indice e_y, possiamo ottenere le seguenti formule:

$$\begin{aligned}
m_x &= \lfloor (b_x + e_x)/2 \rfloor\\
m_y &= \lceil (b_y + e_y)/2 \rceil
\end{aligned}$$

A questo punto possono darsi tre casi:

- Se $X[m_x] < Y[m_y]$, tutti i valori a "sinistra" di m_x sono più piccoli di $X[m_x]$ e tutti i valori a "destra" di m_y sono più grandi di $Y[m_y]$; ovvero $X[i] < X[m_x] < Y[m_y] < Y[j]$, per $i < m_x$ e $j > m_y$. Inoltre per costruzione i valori a destra e a sinistra sono in numero uguale, quindi possiamo ridurci al sottoproblema che si ottiene scartando i valori a "sinistra" di m_x e a "destra" di m_y.

- Se $Y[m_y] < X[m_x]$, tutti i valori a "destra" di m_x sono più grandi di $X[m_x]$ e tutti i valori a "sinistra" di m_y sono più piccoli di $Y[m_y]$; ovvero $Y[i] < Y[m_y] < X[m_x] < X[j]$, per $i < m_y$ e $j > m_y$. Inoltre per costruzione i valori a destra e a sinistra sono in numero uguale, quindi possiamo ridurci al sottoproblema che si ottiene scartando i valori a "destra" di m_x e a "sinistra" di m_y.

- Se $X[m_x] = Y[m_y]$, allora tutti i valori a "sinistra" sia di m_x che di m_y sono minori di $X[m_x] = Y[m_y]$, e tutti i valori a "destra" sia di m_x che di m_y sono maggiori di $X[m_x] = Y[m_y]$, e per costruzione il numero di valori a destra è uguale al numero di valori a sinistra. Quindi $X[m_x] = Y[m_y]$ sono i due valori mediani.

Il caso base si ha quando rimangono quattro valori (due sul lato X e due sul lato Y); a questo punto i valori mediani possono trovarsi ovunque, entrambi in X, entrambi in Y oppure divisi fra i due vettori. E' sufficiente trovare i mediani fra i quattro valori rimasti, operazione che richiede tempo $O(1)$ ed è identificata da mediana4 nel codice sottostante.
La descrizione è più lunga del codice:

```
mediana(integer[] X, integer[] Y, integer b_x, e_x, b_y, e_y)
  if e_x − b_x = 1 then return mediana4(X, Y, b_x, e_x, b_y, e_y)
  integer m_x = ⌊(b_x + e_x)/2⌋
  integer m_y = ⌈(b_y + e_y)/2⌉
  if X[m_x] < Y[m_y] then return mediana(X, Y, m_x, e_x, b_y, m_y)
  if Y[m_y] > X[m_x] then return mediana(X, Y, b_x, m_x, m_y, e_y)
  return (X[m_x], Y[m_y])
```

Il costo computazionale è $O(\log n)$.