Algoritmi e Strutture Dati – Soluzione 17/01/2022

Esercizio A1 – Complessità – Punti ≥ 8

È facile vedere che $T(n)$ è $\Omega(n^2)$, per via della parte non ricorsiva. Verifichiamo se $T(n)$ è anche $O(n^2)$.

- **Caso base**: per $n = 1, \ldots, 7$, si deve dimostrare che $T(n) = 1 \leq cn^2$; questa disequazione è vera per qualunque $c \geq 1$, $c \geq 1/4$, $c \geq 1/9$, $c \geq 1/16$, $c \geq 1/25$, $c \geq 1/36$, $c \geq 1/49$ rispettivamente; è quindi vera per ogni $c \geq 1$.
- **Ipotesi induttiva**: ipotizziamo che per ogni valore k, $1 \leq k < n$: $T(k) \leq k^2$;
- **Passo induttivo**: consideriamo i casi con $n \geq 8$, per i quali $\lfloor n/2 \rfloor$ e $\lfloor n/8 \rfloor$ sono maggiori o uguali a 1 (il primo caso base); applichiamo la sostituzione e otteniamo:

$$
\begin{aligned}
T(n) &= 2T(\lfloor n/2 \rfloor) + 31T(\lfloor n/8 \rfloor) + n^2 - n \\
&\leq 2c\lfloor n/2 \rfloor^2 + 31c\lfloor n/8 \rfloor^2 + n^2 - n && \text{Sostituzione} \\
&\leq 2cn^2/4 + 31cn^2/64 + n^2 - n && \text{Eliminazione } \lfloor\ \rfloor \\
&\leq 2cn^2/4 + 31cn^2/64 + n^2 && \text{Eliminazione } -n \text{ in quanto negativo} \\
&= \frac{63}{64}cn^2 + n^2 \stackrel{?}{\leq} cn^2 && \text{Passaggio algebrico}
\end{aligned}
$$

L'ultima disequazione è vera per $c \geq 64$.

Abbiamo quindi dimostrato che $T(n) = O(n^2)$, per $m = 1$ e $c = 64$. Poiché $T(n)$ è anche $\Omega(n^2)$, abbiamo che $T(n^2) = \Theta(n^2)$.

Esercizio A2 – Alberi made up – Punti ≥ 10

È possibile risolvere il problema con una semplice visita in profondità, con costo computazionale $O(n)$. Se l'albero è vuoto, allora rispetta le regole richieste. Altrimenti, si calcolano il numero di figli.

- Se *level* mod $2 = 0$ (il livello *level* è pari), il nodo deve avere 0 o 2 figli (ovvero, *children* mod $2 = 0$);
- se *level* mod $2 = 1$ (il livello *level* è dispari), il nodo deve avere 1 figlio (ovvero, *children* mod $2 = 1$).

Sfruttiamo il fatto che la parità di *level* e di *children* deve coincidere e controlliamo la corrispondenza alle regole del nodo con l'espressione *children* mod 2 == *level* mod 2. Se la regola viene rispettata localmente, si verifica che venga rispettata anche nei sottoalberi.

```
boolean isMakeup(Tree T)
    return makeupRec(T, 0)
```

```
boolean makeupRec(Tree T, int level)
    if T == nil then
        return true
    else
        int children = iif(T.left == nil, 0, 1) + iif(T.right == nil, 0, 1)
        if children mod 2 == level mod 2 then
            return makeupRec(T.left, level + 1) and makeupRec(T.right, level + 1)
        else
            return false
```

Esercizio A3 – Serie aritmetica bucata – Punti ≥ 12

Ovviamente, il problema si risolve tramite ricerca dicotomica.
Innanzitutto, nella funzione wrapper viene calcolata la ragione della serie aritmetica, ovvero il valore d che viene sommato man mano ai termini. Poichè $A[1] = x$ e $A[n] = x+nd$, possiamo ottenere d calcolando $(A[n]-A[1])/n = (x+nd-x)/n = nd/n = d$. Si noti che non è possibile utilizzare $A[2] - A[1]$, in quanto il termine mancante potrebbe essere incluso fra $A[1]$ e $A[2]$. Allo stesso modo, non è corretto prendere il minimo fra $A[2] - A[1]$ e $[n] - A[n-1]$, perché il vettore potrebbe avere due elementi. Nel progettare la funzione ricorsiva che esplora il sottovettore $A[i \ldots j]$, abbiamo assunto che l'elemento mancante sia sempre compreso fra i due estremi $A[i]$ e $A[j]$; il caso base è quindi costituito da un vettore di due elementi, nel qual caso il valore da restituire è $A[i] + d$.

Nel caso siano presenti più di due termini, viene calcolato l'elemento mediano m e si verifica se $A[m] = A[i] + d \cdot (m - i)$, ovvero se tutti gli elementi delle serie fra i ed m sono presenti. Se così è, si chiama riscorsivamente la funzione sul sottovettore $A[m \dots j]$; altrimenti si richiama la funzione sul sottovettore $A[i \dots m]$.
Il costo dell'algoritmo è $O(\log n)$ in quanto basato su una ricerca dicotomica.

```
missing(int[] A, int n)
  int d = (A[n] − A[1])/n
  return missingRec(A, 1, n, d)
```

```
missingRec(int[] A, int i, int j, int d)
  if j == i + 1 then
    return A[i] + d
  else
    int m = ⌊(i + j)/2⌋
    if A[m] == A[i] + d · (m − i) then
      return missingRec(A, m, j, d)
    else
      return missingRec(A, i, m, d)
```

Alcune note:

- Notate che questo algoritmo funziona correttamente anche se il valore d è negativo. Nella versione del compito data in aula, non veniva specificato. A chi l'ha chiesto, abbiamo detto che potevano assumere che d fosse positivo. Nella versione attuale del compito, ho indicato che d deve essere positivo.
- La soluzione proposta da molti studenti utilizzava una divisione $A[i \dots m]$ oppure $A[m + 1 \dots i]$, che ha come caso base $i = j$, ovvero un singolo elemento immediatamente successivo all'elemento mancante; in questo caso si restituisce $A[i] - d$.

Esercizio B1 – Ottali – Punti ≥ 8

Il problema può essere risolto utilizzando backtrack, in modo simile a tanti esercizi precedenti; in particolare, è molto simile all'Esercizio B2 (PrintBits) del 26/7/21. È possibile fare riferimento a quell'esercizio e adattarlo di conseguenza.
In questa versione, utilizziamo il parametro prev per ricordarci qual era il valore precedente e semplificare la condizione che evita il ripetersi dello stesso valore. All'inizio, il primo valore precedente è -1 che ci assicura che la cifra ottale meno significativa può essere scelta fra 0 e 7.

```
octals(int n)
  int[] S = new int[1 . . . n]
  octalsRec(S, n, −1)
```

```
octalsRec(int[] S, int i, int prev)
  if i == 0 then
    print S
  else
    for d = 0 to 7 do
      if d ≠ prev then
        S[i] = d
        octalsRec(S, i − 1, d)
```

Il numero di stringhe stampate è pari a $8 \cdot 7^{n-1} = {}^{8}/_{7} \cdot 7^n$, ognuna lunga n; la complessità è quindi $\Theta(n \cdot 7^n)$.

Esercizio B2 – Longest repeating sequence – Punti ≥ 10

Il problema può essere risolto con programmazione dinamica, notando la somiglianza con LCS. Idealmente, si tratta di utilizzare la funzione lcs(T, T) applicata due volte alla sequenza T, con il vincolo addizionale che se due caratteri sono uguali, devono avere indice diverso.

Più formalmente, sia $DP[i][j]$ la lunghezza delle sottosequenze ripetute massimali contenute nei primi i caratteri di T e nei primi j caratteri di T; tale valore può essere calcolato ricorsivamente come segue:

$$DP[i][j] = \begin{cases} 0 & i = 0 \textbf{ or } j = 0 \\ DP[i][j] + 1 & T[i] = T[j] \textbf{ and } i \neq j \\ \max(DP[i-1][j], DP[i][j-1]) & \text{altrimenti} \end{cases}$$

Il caso base è simile a quello della LCS: la lunghezza delle sottosequenze ripetute massimali quando uno dei due prefissi è vuoto è ovviamente 0. Se due caratteri in due posizioni distinte i e j sono uguali, si considera il sottoproblema dato da $DP[i-1][j-1]$ e si somma 1; altrimenti, si scarta uno o l'altro carattere (sottraendo uno all'indice) e si prende il massimo. Il codice è identico a quello della LCS, in cui è stata cambiata la condizione dell'**if**. La complessità è pari a $\Theta(n^2)$.

```
int lrs(Item[] T, int n)
    int[][] DP = new int[0...n][0...n]
    for i = 0 to n do
        DP[i][0] = 0
    for j = 0 to n do
        DP[0][j] = 0
    for i = 1 to n do
        for j = 1 to n do
            if T[i] == T[j] and i ≠ j then
                DP[i][j] = DP[i-1][j-1] + 1
            else
                DP[i][j] = max(DP[i-1][j], DP[i][j-1])
    return DP[n][n]
```

Esercizio B3 – Ben bilanciato – Punti ≥ 12

Sia $DP[i]$ il numero di alberi ben-bilanciati di dimensione i. Nei casi base, con $i = 0$ e $i = 1$, esiste un unico albero possibile (l'albero vuoto o l'albero composto da un solo nodo, rispettivamente). Se $i \geq 2$, possono darsi due casi:

- Se i è dispari, un nodo viene "consumato" dalla radice, restano $i-1$ nodi; $i-1$ è pari, quindi l'unico modo di rispettare il ben-bilanciamento è assicursi che entrambi i sottoalberi abbiano esattamente $(i-1)/2$ nodi. Il numero di alberi possibili è quindi pari a $DP[(i-1)/2]^2$ (per ognuno dei $DP[(i-1)/2]$ alberi a sinistra, ci sono $DP[(i-1)/2]$ alberi a destra, da cui il quadrato).
- Se i è pari, i restanti $i-1$ nodi sono in numero dispari; è quindi possibile che ci siano $\lfloor (i-1)/2 \rfloor$ nodi a destra e $\lceil (i-1)/2 \rceil$ nodi a sinistra, o viceversa. Il numero di alberi possibili è quindi pari a $2 \cdot DP[\lfloor (i-1)/2 \rfloor] \cdot DP[\lceil (i-1)/2 \rceil]$; il fattore 2 deriva dal fatto che il nodo in eccesso può essere a sinistra oppure a destra, e quindi contiamo i casi due volte.

La formula risultante è la seguente:

$$DP[i] = \begin{cases} 1 & i \leq 1 \\ DP[(i-1)/2]^2 & i \geq 2 \textbf{ and } i \text{ è dispari} \\ 2 \cdot DP[\lfloor (i-1)/2 \rfloor] \cdot DP[\lceil (i-1)/2 \rceil] & i \geq 2 \textbf{ and } i \text{ è pari} \end{cases}$$

Questa formula può essere trasformata in un algoritmo basato su memoization nel modo seguente:

```
countWellBalanced(int n)
    int[] DP = new int[1...n] = {-1}
    return wbRec(DP, i)
```

```
wbRec(int[] DP, int i)
if i ≤ 1 then
    return 1
else
    if DP[i] < 0 then
        if i mod 2 == 1 then
            DP[i] = (wbRec(DP, (i − 1)/2))^2
        else
            DP[i] = 2 · wbRec(DP, ⌊(i − 1)/2⌋) · wbRec(DP, ⌈(i − 1)/2⌉)
    return DP[i]
```

Poichè è necessario riempire tutte le n caselle del vettore DP, il costo è $\Theta(n)$.