Algoritmi e Strutture Dati – 03/07/2020

Esercizio A1

Limite asintotico inferiore Data la componente non ricorsiva, è facile vedere che $T(n) = \Omega(n^2)$.

Limite asintotico inferiore Supponiamo che $T(n) = O(n^2)$. Dobbiamo dimostrare che

$$\exists c > 0, \exists m > 0 : T(n) \leq cn^2, \forall n \geq m$$

- **Casi base**:

$$T(i) = 1 \leq c \cdot i^2 \Rightarrow c \geq 1/i^2$$

  per $1 \leq i \leq 5$; tutte queste disequazioni sono valide per $c \geq 1$.

- **Ipotesi induttiva**:

$$T(k) \leq ck^2, \forall k < n$$

- **Passo induttivo**:

$$\begin{aligned}
T(n) &= 3T(\lfloor n/3 \rfloor) + 4T(\lfloor n/4 \rfloor) + 5T(\lfloor n/5 \rfloor) + 6T(\lfloor n/6 \rfloor) + n^2 \\
&\leq 3c\lfloor n/3 \rfloor^2 + 4c\lfloor n/4 \rfloor^2 + 5c\lfloor n/5 \rfloor^2 + 6c\lfloor n/6 \rfloor^2 + n^2 && \text{Ipotesi induttiva} \\
&\leq cn^2/3 + cn^2/4 + cn^2/5 + cn^2/6 + n^2 && \text{Semplificazione limite inferiore} \\
&= \frac{20 + 15 + 12 + 10}{60} cn^2 + n^2 \\
&= 57/60cn^2 + n^2 = 19/20cn^2 + n^2 \leq cn^2
\end{aligned}$$

  L'ultima disequazione è vera per $c \geq 20$.

Abbiamo quindi dimostrato che $T(n)$ è $\Theta(n^2)$.

Esercizio A2

Il problema può essere risolto tramite tecnica divide-et-impera. Dato un nodo t, siano t_l e t_r i suoi sottoalberi sinistro e destro di dimensione s_l e s_r (potenzialmente zero, se il sottoalbero è costituito da un puntatore **nil**). Si chiamerà quindi ricorsivamente la procedura sul sottoalbero sinistro, con s_l valori; si piazzerà il valore $s_l + 1$-esimo nella radice; e quindi si richiamerà la procedura sul sottalbero destro, con s_r valori.
Poichè questo algoritmo è sostanzialmente una visita, la sua complessità sarà pari a $\Theta(n)$.

```
binaryInsert(TREE T, int[] A)
    biRec(T, A, 1)
```

```
biRec(TREE t, int[] A, int i)
    int leftSize = 0
    if T.left ≠ nil then
        leftSize = t.left.size
        biRec(t.left, A, i)
    t.value = A[i + leftSize]
    if T.right ≠ nil then
        biRec(t.right, A, i + leftSize + 1)
```

Lorenzo Dongili mi ha fatto notare che non era necessario conoscere la dimensione del sottoalbero. Sarebbe stato possibile piazzare i valori utilizzando una visita in profondità e utilizzando una variabile globale come indice del valore successivo (sarebbe anche possibile farlo senza variabile globale, ma verrebbe inutilmente complicato). La complessità di questa soluzione resterebbe ovviamente $\Theta(n)$.

Esercizio A3

Il primo passo è notare che se esiste un valore di maggioranza, deve essere il valore contenuto nella posizione mediana $\lfloor (n+1)/2 \rfloor$. Se n è dispari, questo è l'elemento centrale. Qualunque maggioranza deve includere questo elemento e altri $(n-1)/2$ elementi; infatti, a sinistra e destra di tale elemento ci stanno esattamente $(n-1)/2$ elementi. Se n è pari, sia $n/2$ che $n/2+1$ devono appartenere alla maggioranza, per lo stesso ragionamento di cui sopra.
Dato questo valore v, è sufficiente scrivere una procedura di ricerca dicotomica che assuma l'esistenza di tale valore nel vettore e restituisca l'indice più basso in cui tale valore si presenta. Il problema di trovare l'indice più basso è stato proposto nel compito del 7/2/2019. Qui riporto l'intero codice per completezza, ma sarebbe stato possibile semplicemente invocare la funzione, quindi l'esercizio era risolvibile in quattro righe di codice.
Conoscendo l'indice del primo elemento, è necessario vedere se l'elemento in posizione $A[\mathit{first} + \lfloor n/2 \rfloor]$ è uguale ad esso, nel qual caso esiste una maggioranza di elementi uguali, essendo il vettore ordinato.

```
int hasMajority(int[] A, int n)
    int candidate = A[⌊(n + 1)/2⌋])
    int first = searchFirst(A, 1, n, candidate)
    return A[first] == A[first + ⌊n/2⌋]
```

```
int searchFirst(int[] A, int i, int j, int v)
    if i == j then
        return i
    else
        int m = ⌊(i + j)/2⌋
        if v ≤ A[m] then
            return searchFirst(A, i, m, v)
        else
            return searchFirst(A, m + 1, j, v)
```

Il costo computazionale, dato dalla ricerca dicotomica, è $O(\log n)$.

Esercizio B1

Questo esercizio può essere risolto tramite tecnica greedy, con una dimostrazione semplice.
Innanzitutto, anche se non richiesto, notiamo che una sequenza di cifre può dare origine ad un numero palindromo se e solo se esiste al più una cifra il cui numero di occorrenze è dispari. Se esiste esattamente una cifra il cui numero di occorrenze è dispari, andrà al centro.
Dopo di che, la scelta greedy sarà quella di piazzare le cifre più alte all'esterno (quindi all'inizio), per ottenere un numero più grande.
Per prima cosa quindi realizziamo un conteggio del numero delle cifre, simile a quanto fatto con counting sort. Il costo di questa operazione é $O(n)$ (l'inizializzazione di un vettore contenente 10 elementi ha costo costante).
Dopo di che, piazziamo le cifre dall'esterno verso l'interno. Come richiesto, la versione presentata non effettua il controllo di palindromicità. È semplice aggiungere un ciclo su *count* che verifichi che il numero di elementi dispari sia al massimo uno.

Il costo della funzione é $O(n)$. Nel soluzioni proposte dagli studenti, ho visto versioni con costo $O(n \cdot n!)$ (generazione di tutte le permutazioni possibile e verifica di palindromicità), $O(n^3)$ (ricerca quadratica della coppia di valori massimi, scambio con i valori esterni, ripetizione nel settovettore dei valori interni), $O(n^2)$ (come sopra, ma con ricerca lineare della coppia di valori massimi), $O(n \log n)$ (più simile alla mia versione, ma con ordinamento al posto di un approccio più simile al counting sort).

```
boolean lpp(int[] A, int n)
    int[] count = new int[0...9] = { 0 }          % Initialized to zero
    for i = 1 to n do
        count[A[i]] = count[A[i]] + 1
    int start = 1
    int end = n
    int digit = 9
    while digit ≥ 0 and start ≤ end do
        while count[digit] > 1 do
            A[start++] = A[end-] = digit
            count[digit] = count[digit] − 2
        digit = digit − 1
    if start == end then
        for digit = 1 to 9 do
            if count[digit] == 1 then
                A[start] = digit
```

Esercizio B2

Ovviamente, il problema può essere risolto con una visita in profondità dell'albero, con l'unica accortezza di memorizzare il percorso svolto fino a quando non si raggiunge una foglia; a quel punto il percorso viene stampato interamente. Per memorizzare il percorso, utilizziamo uno stack; assumiamo che la stampa avvenga in ordine FIFO.

```
printAllPaths(Tree T)
    Stack path = new Stack()
    printRec(T, path)
```

```
int printRec(Tree T, Stack path)
    if T ≠ nil and T.value > 0 then
        path.push(T.value)
        if T.left == nil and T.right == nil then
            print path
        else
            printRec(T.left, path)
            printRec(T.right, path)
        path.pop()
```

È possibile considerare questo algoritmo come un algoritmo di backtracking, ma visto che alla fine non è altro che una visita dell'albero, ogni nodo verrà visitato una volta sola, con costo $O(n)$. Tuttavia, al momento di raggiungere una foglia, l'albero stamperà un percorso pari alla profondità della foglia.

Potremmo quindi dire che la complessità é $O(fh)$, dove f è il numero di foglie e h è la profondità massima. Poiché $f \leq n$, abbiamo che la complessità è $O(nh)$. Questo mi basterebbe per l'analisi di complessità.
Non è preciso invece affermare che essendo un algoritmo di backtracking, allora la complessità è $O(2^n)$ - ovviamente come limite superiore è corretto, ma la sua precisione lascia molto a desiderare. È sbagliato invece affermare che l'algoritmo ha costo lineare.

Analisi più approfondita – non richiesta Cerchiamo di analizzare meglio la complessità. Consideriamo il caso ottimo di un albero "lineare", dove tutti i nodi hanno esattamente un figlio tranne l'unica foglia. In questo caso, la complessità è $O(n)$. Il numero massimo di foglie si ottiene nel caso di un albero perfetto, con $n = 2^{h+1} - 1$ nodi e le foglie sono 2^h, con altezza h. In questo caso la complessità è $O(h2^h)$; espresso in termine di n, otteniamo $O(n \log n)$. Ma è l'albero che costa di più, in questo algoritmo?
È facile affermare che l'algoritmo ha costo $O(n^2)$, come limite superiore: ci sono al più $O(n)$ foglie e il percorso più lungo è $O(n)$. Ma esistono alberi con questa complessità? Si consideri un albero in cui $n/2$ nodi sono organizzati in una catena lineare di profondità $n/2$. Dall'ultimo nodo di questa catena parte un albero perfetto, di dimensione $n/2$, con $\approx n/4$ foglie. Da ognuna di queste foglie si dovrà stampare una catena lunga $O(n)$, e quindi il costo sarà $O(n^2)$.

Esercizio B3

Il problema è simile al problema della più lunga sottosequenza crescente. Utilizziamo la programmazione dinamica e realizziamo una tabella DP tale per cui $DP[i]$ contiene il valore massimale della sottosequenza k-limitata il cui ultimo valore è in posizione i-esima.
Il valore DP può essere calcolato tramite la seguente formula ricorsiva:

$$DP[i] = \begin{cases} A[1] & i = 1 \\ A[i] & i > 1 \textbf{ and } \forall j, 1 \leq j < i : |A[i] - A[j]| > k \\ \max\{DP[j] : 1 \leq j < i \wedge |A[i] - A[j]| \leq k\} + A[i] & \text{altrimenti} \end{cases}$$

Al termine del calcolo della tabella DP, il valore da restituire è il massimo contenuto nella tabella.
Il costo computazionale è $O(n^2)$.

```
int ksequence(int[] A, int n, int k)
    int[] DP = new int[1...n]
    for i = 1 to n do
        DP[i] = A[i]                    % Se non troviamo valori k-limitati, si memorizza A[i]
        for j = 1 to i − 1 do
            if |A[j] − A[i]| ≤ k and DP[j] + A[i] > DP[i] then
                DP[i] = DP[j] + A[i]
    return max(DP)
```