Ingegneria degli Algoritmi
21 febbraio 2020

Esercizio 1
Trovare i limiti superiore e inferiori più stretti possibili per la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 3T(\lfloor n/3 \rfloor) + 4T(\lfloor n/4 \rfloor) + 12T(\lfloor n/6 \rfloor) + n^2 & n > 6 \\ 1 & n \leq 6 \end{cases}$$

Soluzione
Poiché l'equazione di ricorrenza ha una componente non ricorsiva, è immediato affermare che sia $\Omega(n^2)$.
Per completare lo studio, procediamo per tentativi. Appurato che sia $\Omega(n^2)$, è lecito chiederci se sia $\Theta(n^2)$. Per dimostrarlo, rimane da provare che sia anche $O(n^2)$.

- Caso base: $T(n) = 1 \leq cn^2$, per tutti i valori di n compresi tra 1 e 6, ovvero:

  $$c \geq 1, c \geq \frac{1}{4}, c \geq \frac{1}{9}, c \geq \frac{1}{16}, c \geq \frac{1}{25}, c \geq \frac{1}{36}$$

  Tutte queste disequazioni sono soddisfatte da $c \geq 1$.
- Ipotesi induttiva: $T(n) \leq ck^2$, per $k < n$.
- Passo induttivo:

  $$\begin{aligned} T(n) &= 3T(\lfloor n/3 \rfloor) + 4T(\lfloor n/4 \rfloor) + 12T(\lfloor n/6 \rfloor) + n^2 \\ &\leq 3c\lfloor n/3 \rfloor^2 + 4c\lfloor n/4 \rfloor^2 + 12\lfloor n/6 \rfloor^2 + n^2 \\ &\leq 3cn^2/9 + 4cn^2/16 + 12cn^2/36 + n^2 \\ &\leq 11/12cn^2 + n^2 \leq cn^2 \end{aligned}$$

  L'ultima disequazione è soddisfatta da $c \geq 12$.

Abbiamo quindi dimostrato che $T(n) = \Theta(n^2)$ per $n_0 = 1$ e $c \geq 12$.

Esercizio 2
Sia dato un albero binario di ricerca `A` non bilanciato. Scrivere l'algoritmo della funzione `bilancia()` (in pseudocodice o in python) che accetti in input tale albero `A` e restituisca un nuovo albero binario di ricerca `B` che risulti bilanciato. Fornire un istanza di input e mostrare i passi che vengono compiuti da tale algoritmo sull'istanza di esempio. Discutere informalmente la complessità computazionale dell'algoritmo proposto.

Soluzione
Un approccio "semplice" per risolvere questo esercizio consiste nell'eseguire una visita in ordine simmetrico dell'albero binario in input ed inserire i nodi, uno ad uno, all'interno di un albero auto-bilanciato (come un RB-tree o un AVL). Questa soluzione ha una complessità pari ad $O(n\ logn)$, dovuta alla scansione di tutti i nodi e dal costo necessario all'inserimento (e ribilanciamento) dell'albero di destinazione.

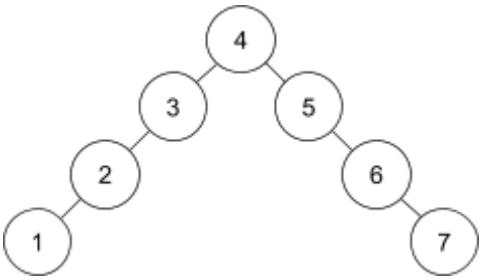
Una soluzione più elegante può essere implementata utilizzando un vettore di appoggio, costruito effettuando comunque una visita in ordine simmetrico. Si noti che, poiché l'albero in input seppur sbilanciato è comunque un albero binario di ricerca, la visita in ordine simmetrico produce un vettore in cui gli elementi sono ordinati. Questa visita ha costo $O(n)$.

Dato un vettore ordinato, sfruttando la proprietà degli alberi binari di ricerca, è possibile costruire un albero di ricerca bilanciato sfruttando la tecnica del *divide et impera*. Infatti, scegliendo ricorsivamente come radice di un sottoalbero l'elemento "al centro" del (sotto)vettore, si garantisce che i sottoalberi destro e sinistro siano composti dallo stesso numero di elementi. Anche questa costruzione, che visita gli elementi del vettore tutti una e una sola volta, ha costo $O(n)$. La complessità computazionale dell'algoritmo finale, quindi, avrà costo $O(n)$.

Una possibile implementazione in python è la seguente. Lo pseudocodice ricalca questa implementazione in maniera molto simile..

```
1. class Node:
2.     def __init__(self,data):
3.         self.data=data
4.         self.left=None
5.         self.right=None
6.
7. def BST2Array(root, arr):
8.     if not root:
9.         return
10.    BST2Array(root.left, arr)
11.    arr.append(root)
12.    BST2Array(root.right, arr)
13.
14.   def Array2BST(arr, start, end):
15.       if start > end:
16.           return None
17.       mid = (start + end) // 2
18.       node = arr[mid]
19.       node.left = Array2BST(arr, start, mid - 1)
20.       node.right = Array2BST(arr, mid + 1, end)
21.       return node
22.
23.   def bilancia(A):
24.       arr = []
25.       BST2Array(A, arr)
26.       return Array2BST(arr, 0, len(arr) - 1)
```

Un input di esempio, ed i passi di esecuzione dell'algoritmo, sono i seguenti.



Vettore di appoggio dopo la visita in ordine simmetrico:

1	2	3	4	5	6	7

La costruzione dell'albero bilanciato avviene prendendo come radice l'elemento centrale, assegnando la porzione destra e sinistra del vettore ai sottoalberi destro e sinistro rispettivamente, e ripetendo ricorsivamente la costruzione, come segue.

(a)

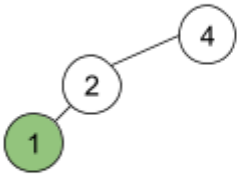
1	2	3	**4**	5	6	7

(b)

1	**2**	3	4	5	6	7

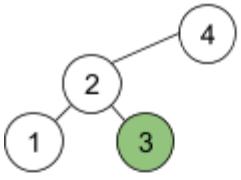
(c)

1	2	3	4	5	6	7



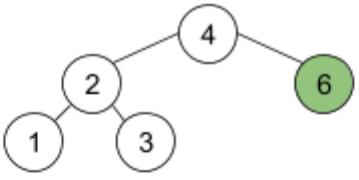
(d)

1	2	**3**	4	5	6	7



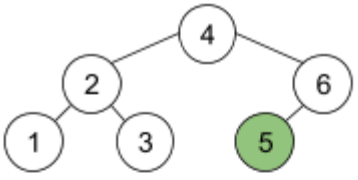
(e)

1	2	3	4	5	**6**	7



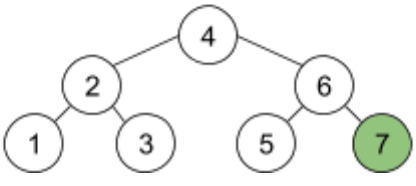
(f)

1	2	3	4	**5**	6	7



(g)

1	2	3	4	5	6	**7**



Esercizio 3

Nel gioco di Minecraft, una carta geografica altimetrica è rappresentata da una matrice $n \cdot n$ di celle quadrate, ognuna delle quali riporta un'altitudine intera misurata in blocchi. Un valore minore o uguale a zero indica un mare, mentre un valore positivo indica un'isola.

Scrivere un algoritmo che prenda in input una matrice A di interi e la sua dimensione positiva n, e restituisca l'altezza media dell'isola più elevata.

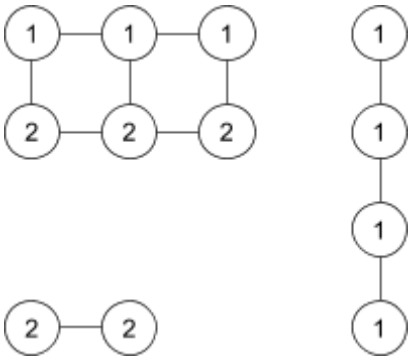
Discutere informalmente la correttezza della soluzione proposta e calcolare la complessità computazionale.

Ad esempio, nella matrice seguente ci sono tre isole, una $2 \cdot 3$ in alto a sinistra con altezza media 1.5, una $4 \cdot 1$ a destra con altezza media 1, una $1 \cdot 2$ in basso a sinistra con altezza media 2. L'isola con altezza media più alta è quella in basso a sinistra, e quindi l'algoritmo deve restituire 2.

1	1	1	0	1
2	2	2	-1	1
0	-1	0	-2	1
0	0	-1	-2	1
2	2	-1	0	0

Soluzione

Il problema può essere risolto facilmente interpretando la matrice come se fosse un grafo. Le celle con valori minori o uguali a zero non ci interessano e possono essere "scartate" nella trasformazione da matrice a grafo. Ogni cella della matrice è quindi un nodo, connesso alle sole celle adiacenti che hanno un valore maggiore di zero. La matrice di esempio può quindi essere trasformata nel seguente grafo.



A questo punto, è sufficiente effettuare una visita in ampiezza di ciascuna componente connessa del grafo, calcolando la somma delle chiavi dei nodi e contando il numero di nodi visitati. La scelta dell'altezza media massima diviene a questo punto banale.

Nell'implementazione dell'algoritmo, occorre necessariamente sfruttare una struttura dati d'appoggio (ad esempio, matrice/vettore di booleani, coda, insieme, ...), per garantire che una singola componente connessa sia visitata una ed una sola volta dall'algoritmo. Si noti che, nello pseudocodice proposto più avanti, si utilizza una funzione di appoggio che restituisce due valori di ritorno. Tale funzionalità è direttamente disponibile in python, pertanto si fornisce unicamente lo pseudocodice come soluzione di riferimento.

```
isolaPiuAlta(M, n):
      maxH ← 0
      visited ← [0] * (n*n)
      for r ← 1 to n:
            for c ← 1 to n:
                  if M[r][c] > 0 and not visited[r*c] then:
                        height, count ← isolaDFS(M, n, r, c, visited)
                        maxH = max(maxH, height / count)
      return maxH

isolaDFS(M, n, r, c, visited)
      if 1≤r<n and 1≤c<n and M[r][c] > 0 and not visited[r*c] then:
            visited[r*c] ← 1
            h1, c1 ← isolaDFS(M, n, r - 1, c)
            h2, c2 ← isolaDFS(M, n, r + 1, c)
            h3, c3 ← isolaDFS(M, n, r, c - 1)
            h4, c4 ← isolaDFS(M, n, r, c + 1)
            height ← h1 + h2 + h3 + h4 + M[r][c]
            count ← c1 + c2 + c3 + c4 + 1
            return height, count
      return 0, 0
```

L'algoritmo di `isolaPiuAlta()` itera su tutta la matrice, pertanto è garantito che se un'isola esiste, questa verrà individuata. L'utilizzo di un vettore `visited` condiviso tra le due funzioni garantisce che ogni cella di un'isola sia visitata una ed una sola volta.
L'algoritmo di `isolaDFS()` esplora in tutte le direzioni, ricorsivamente, a partire da una cella di un'isola. È pertanto garantito che tutta l'isola verrà esplorata. Il numero di invocazioni ricorsive potrebbe essere ridotto—vengono effettuate molte chiamate inutili—tuttavia la soluzione presentata è corretta.
Poiché ogni cella può essere visitata una ed una sola volta, la complessità è data dall'istruzione dominante di `isolaPiuAlta()`, ovvero il controllo nell'if. Pertanto, la complessità di questo algoritmo è $O(n^2)$.

Sarebbe analogo trasformare esplicitamente la matrice in un grafo, inserendo all'interno di ciascun nodo un flag per indicare se tale nodo è già stato visitato o meno. A quel punto, si sarebbe potuto utilizzare un algoritmo di DFSpiù "tradizionale". La complessità computazionale asintotica non cambierebbe, ma chiaramente avremmo costanti nascoste più elevate, poiché pagheremmo comunque il costo di trasformazione da matrice a grafo.