Algoritmi e Strutture Dati - 31/01/2018

Esercizio 1

Procedendo per tentativi, proviamo con $\Theta(n \log n)$. È facile vedere che la ricorrenza è $\Omega(n \log n)$, per via della sua componente non ricorsiva. Proviamo quindi a dimostrare che $T(n) = O(n \log n)$.

- Caso base: $T(1) = 1 \leq c \cdot 1 \log 1 = 0$, è falso. Dobbiamo quindi calcolare altri case base:

$$\begin{aligned}
T(2) &= T(1) + T(0) + 2\log 2 = 1 + 1 + 2\log 2 = 4 \leq c \cdot 2\log 2 \Rightarrow c \geq 2 = c_2 \\
T(3) &= T(1) + T(1) + 3\log 3 = 1 + 1 + 3\log 3 \leq c \cdot 3\log 3 \Rightarrow c \geq 1 + \frac{2}{3\log 3} = c_3 \\
T(4) &= T(2) + T(1) + 4\log 4 = 4 + 1 + 4\log 4 \leq c \cdot 4\log 4 \Rightarrow 1 + \frac{5}{4\log 4} = c_4 \\
T(5) &= T(2) + T(1) + 5\log 5 = 4 + 1 + 5\log 5 \leq c \cdot 5\log 5 \Rightarrow c \geq 1 + \frac{5}{5\log 5} = c_5
\end{aligned}$$

  Tutti i casi precedenti fanno riferimento a $T(1)$, che non è stato possibile dimostrare. Abbiamo quindi dovuto dimostrarli uno ad uno. Poiché $T(6)$ è pari a $T(3) + T(2) + 6\log 6$, è espresso tramite casi base già dimostrati ($T(2)$ e $T(3)$) e possiamo utilizzare l'induzione per andare avanti. È facile vedere (senza calcolatrice!) che c_2, c_3, c_4, c_5 sono inferiori o uguali a 2, quindi $c \geq 2$ rispetta tutte le disequazioni sopra.

- Ipotesi induttiva: $T(k) \leq ck \log k$, per $2 \leq k < n$

- Passo induttivo:

$$\begin{aligned}
T(n) &= T(\lfloor n/2 \rfloor) + T(\lfloor n/3 \rfloor) + n\log n \\
&\leq c\lfloor n/2 \rfloor \log\lfloor n/2 \rfloor + c\lfloor n/3 \rfloor \log\lfloor n/3 \rfloor + n\log n \\
&\leq c\, n/2 \log\, n/2 + c\, n/3 \log\, n/3 + n\log n \\
&\leq c\, n/2 \log n + c\, n/3 \log n + n\log n \\
&\leq\, 5/6 cn\log n + n\log n \leq cn\log n
\end{aligned}$$

  L'ultima disequazione è rispettata per $c \geq 6$.

Abbiamo quindi dimostrato che $T(n) = O(n \log n)$, per $c \geq 6$ e $m = 2$.

Esercizio 2

È possibile visitare l'intero albero, con una visita in profondità, e restituire il numero di elementi che sono compresi fra i due limiti:

```
int occurrences(TREE T, int min, int max)
  if T = nil then
    return 0
  int nl = occurrences(T.left, min, max)
  int nr = occurrences(T.right, min, max)
  return nl + nr + iif(min ≤ T.key ≤ max, 1, 0)
```

Questo algoritmo ha costo $\Theta(n)$, qualunque siano i valori cercati. Non è molto efficiente quando l'intervallo è molto ristretto.
Per rendere più efficiente questo algoritmo, è sufficiente notare che:

- Se stiamo visitando un nodo in cui la chiave è minore a *min*, il nodo che stiamo considerando non va contato, ed è possibile che ci siano valori compresi fra *min* e *max* nel sottoalbero destro, ma sicuramente non nel sottoalbero sinistro. Quindi applicheremo ricorsivamente la funzione al sottoalbero destro.

- Altrimenti, se stiamo visitando un nodo in cui la chiave è minore a *max*, il nodo che stiamo considerando non va contato, ed è possibile che ci siano valori compresi fra *min* e *max* nel sottoalbero sinistro, ma sicuramente non nel sottoalbero destro. Quindi applicheremo ricorsivamente la funzione al sottoalbero sinistro.

- Altrimenti, la chiave memorizzata nel nodo è compresa nell'intervallo (quindi bisogna sommare 1) e possono essere presenti nodi compresi nell'intervallo sia nel sottoalbero sinistro che in quello destro.

```
int occurrences(TREE T, int min, int max)
  if T = nil then
  |  return 0
  else if T.key < min then
  |  return occurrences(T.right, min, max)
  else if T.key > max then
  |  return occurrences(T.left, min, max)
  else
  |_ return 1 + occurrences(T.right, min, max) + occurrences(T.left, min, max)
```

La complessità di questo algoritmo dipende da n, dalla dimensione dell'intervallo e dalla struttura dell'albero. Se l'intervallo comprende tutti i nodi, il costo dell'algoritmo è comunque $O(n)$. Anche quando l'intervallo è piccolo (al limite contenente un solo elemento), è possibile progettare un albero che viene interamente visitato dall'algoritmo, con costo $\Theta(n)$. Ideare tale albero è lasciato per esercizio. Tuttavia, per alberi completi e con intervalli che racchiudono un numero di nodi $k \ll n$, il costo è $O(\log n + k)$.

Esercizio 3

Se si considerano le celle come nodi di un grafo non orientato e le coppie di celle adiacenti come archi, il problema da risolvere corrisponde ad identificare la componente connessa più grande.
È possibile e accettabile costruire un grafo a partire dalla matrice, applicando poi l'algoritmo per le componenti connesse. Tuttavia, è più semplice adattare la struttura dalla funzione cc(), ovvero facendo partire una DFS da ogni cella contenente alberi che non sia già stato visitato in precedenza.
Per ridurre la quantità di codice, andremo a modificare direttamente la matrice. Se ciò non fosse accettabile, è necessario aggiungere una funzione wrapper che copia la matrice in una matrice di appoggio.
Adotteremo queste convenzioni: una cella con valore 1 è un albero che deve ancora essere visitato; una cella con valore 0 o era un prato in origine oppure è già stato visitato.
La visita viene fatta in maniera ricorsiva, in profondità, dalla funzione dfs(), che ritorna il numero di nodi raggiunti dalla visita. Se le coordinate del nodo sono corrette e il nodo non è ancora stato visitato, viene marcato come visitato (direttamente sulla matrice) e viene contato come un nodo; vengono poi visitati ricorsivamente i quattro nodi adiacenti.
La complessità è $\Theta(n^2)$, in quanto l'algoritmo corrisponde ad una visita in profondità di un grafo con n^2 nodi e $4n^2 - 4n$ archi.

```
int searchForest(boolean[][] A, int n)
  maxsofar = 0
  for i = 1 to n do
  |  for j = 1 to n do
  |  |  if A[i, j] = 1 then
  |  |  |_ maxsofar = max(maxsofar, dfs(A, i, j, n))
  return maxsofar
```

```
int dfs(boolean[][] A, int i, int j, int n)
  if 1 ≤ i ≤ n and 1 ≤ j ≤ n and A[i, j] = 1 then
  |  A[i, j] = 0
  |  return 1 + dfs(A, i − 1, j, n) + dfs(A, i, j − 1, n) + dfs(A, i + 1, j, n) + dfs(A, i, j + 1, n)
  else
  |_ return 0
```

Esercizio 4

Il problema viene risolto con una ricerca binaria modificata. Si noti che l'algoritmo seguente utilizza indici dei vettori compresi fra 1 e n; per indici compresi fra 0 e $n - 1$ bisogna scambiare pari con dispari nella descrizione seguente, e aggiustare i valori di conseguenza.
Si parte con l'osservazione che le coppie consecutive $A[i], A[i+1]$ che si trovano *prima* del valore cercato sono tali per cui i è dispari, $i + 1$ è pari; le coppie che si trovano *dopo* il valore cercato sono tali per cui i è pari, $i + 1$ è dispari.

Quindi, supponendo di esaminare ricorsivamente il sottovettore $A[i \ldots j]$ e di avere almeno tre elementi contenuti in esso, si calcola l'elemento mediano $m = (i+j)/2$. Ci sono quattro casi: a seconda se m sia pari o dispari,e a seconda che sia uguale all'elemento successivo o precedente. In tutti i casi, si esamina un sottovettore grande al più la metà.
Se invece il sottovettore $A[i \ldots j]$ ha dimensione 1, allora contiene l'elemento cercato e si può restituire tale valore.

```
int single(ITEM[] A, int n)
    return singleRec(A, 1, n)
```

```
int singleRec(ITEM[] A, int i, int j)
    if i = j then
        return A[i]
    int m = ⌊(i + j)/2⌋
    if m mod 2 = 1 then
        if A[m] = A[m + 1] then
            return singleRec(A, m + 2, j)
        else
            return singleRec(A, i, m)
    else
        if A[m − 1] = A[m] then
            return singleRec(A, m + 1, j)
        else
            return singleRec(A, i, m − 1)
```

La complessità è pari a $O(\log n)$, essendo una ricerca dicotomica. binaria.