Algoritmi e Strutture Dati - Prova d'esame
07/09/11

Esercizio 1

Poniamo $n = 2^k$. Sostituendo nella ricorrenza otteniamo:

$$\begin{aligned} T(2^k) &= 4T(\sqrt{2^k}) + \log^2 2^k \\ &= 4T(2^{k/2}) + k^2 \end{aligned}$$

Sostituiamo quindi la variable $T(2^k)$ con $S(k)$ e otteniamo (tramite Master Theorem)

$$S(k) = 4S(k/2) + k^2 = \Theta(k^2 \log k)$$

Ri-esprimendo la funzione nei termini di $T(n)$ e $k = \log n$, otteniamo

$$T(n) = \log^2 n \log\log n$$

Esercizio 2

L'idea è molto semplice: tramite una post-visita, otteniamo per ogni nodo v il numero di foglie contenute nel sottoalbero radicato in v e il massimo grado di sbilanciamento dei nodi contenuti nel sottoalbero radicato in v.
Per semplicità, assumiamo esiste una struttura dati RET con due campi: *leafs* contiene il numero di foglie e *max* contiene il massimo grado di sbilanciamento.
La complessità di una post-visita è ovviamente $O(n)$.

```
RET unbalance(TREE T)
  if T = nil then
    return newRET(0, 0)
  L ← unbalance(T.left)
  R ← unbalance(T.right)
  V ← new RET
  V.max ← max(L.max, R.max, |L.leafs − R.leafs|)
  V.leafs ← L.leafs + R.leafs
  return V
```

Esercizio 3

Una possibile soluzione è la seguente: per prima cosa, ordiniamo tutte le stringhe internamente, ovvero una per una. Ad esempio, montresor diventa emnoorrst. Tutte le stringhe che sono una anagramma dell'altra, una volta ordinate, coincidono.
Utilizziamo poi una tabella hash per identificare le stringhe che coincidono; si sarebbe potuto ordinare le stringhe, ma la complessità sarebbe superiore.
Il costo di questo algoritmo è $O(nk \log k + n)$.

```
anagrams(ITEM [ ][ ] S, integer n)
  HASH H ← Hash()
  for i ← 1 to n do
    sorted ← sort(S[i])
    SET S ← H.lookup(sorted)
    if S = nil then
      S ← Set()
    S.insert(S[i])
    H.insert(sorted, S)
  foreach x : H.lookup(x) ≠ nil do
    SET S ← H.lookup(x)
    if S.dim() > 1 then
      print S
```

Esercizio 4

E' possibile ottenera una soluzione modificando opportunamente l'algoritmo di Merge Sort. Durante l'operazione di merge, quando si seleziona il valore che si trova nella metà di destra, questo è invertito rispetto a tutti i valori che si trovano nella metà di sinistra e che non sono ancora stati inseriti nel vettore di appoggio. E' quindi sufficiente mantenere un contatore a cui verrà sommata la dimensione del vettore mancante. Questo valore, ritornato dall'operazione di merge, verrà progressivamente sommato dalla procedura MergeSort(). Costo dell'operazione, $O(n \log n)$.

```
CountInversion(integer A[ ], integer primo, integer ultimo)
  if primo < ultimo then
      integer mezzo ← ⌊(primo + ultimo)/2⌋
      return CountInversion(A, primo, mezzo) + CountInversion(A, mezzo + 1, ultimo) + Merge(A, primo, ultimo, mezzo)
```

```
Merge(integer A[ ], integer primo, integer ultimo, integer mezzo)
  integer i, j, k, h
  i ← primo; j ← mezzo + 1; k ← primo
  counter ← 0
  while i ≤ mezzo and j ≤ ultimo do
      if A[i] ≤ A[j] then
          B[k] ← A[i]
          i ← i + 1
      else
          counter ← counter + (mezzo − i + 1)
          B[k] ← A[j]
          j ← j + 1
      k ← k + 1
  j ← ultimo
  for h ← mezzo downto i do
      A[j] ← A[h]
      j ← j − 1
  for j ← primo to k − 1 do  A[j] ← B[j]
  return counter
```