008AA – ALGORITMICA E LABORATORIO
Primo appello, 21 Gennaio 2014

SOLUZIONI

Esercizio 1. *(8 punti)*
È dato un array a **ordinato** di n interi il cui valore può essere solo 0 o 1. Si consideri il problema di contare il numero di occorrenze del numero 1 in a.

1. Descrivere un algoritmo che richiede tempo $O(n)$.
2. Dimostrare che un qualunque algoritmo che risolve il problema suddetto richiede tempo $\Omega(\log n)$ al caso pessimo.
3. Descrivere un algoritmo di tipo divide et impera che richiede tempo $\Theta(\log n)$ nel caso pessimo, indicando e risolvendo la corrispondente relazione di ricorrenza.

SOLUZIONE

1.

   Conta1(a)

   ```
   i = 0;
   while(a[i] == 0) i++;
   return n - i;
   ```

2. Misuriamo il costo in tempo valutando il numero di confronti effettuati da un generico algoritmo che risolve il problema. Il numero di soluzioni possibili è $s(n) = n + 1$, infatti il numero di occorrenze di 1 pu essere compreso tra 0 e n (estremi inclusi). Un algoritmo generico A che risolve il problema deve quindi poter distinguere tra almeno $n + 1$ casi possibili. Ogni confronto à luogo a 2 possibili risposte (uguale o diverso da 0). Dopo t confronti, A può distinguere tra 2^t casi. Dunque: $2^t \geq n + 1$, da cui $t = \Omega(logn)$.

3.

   Conta1(a, sx, dx)

   ```
   if (sx > dx) return 0;
   if (sx == dx) return a[sx];
   cx = (sx + dx)/2;
   if (a[cx] == 0) return Conta1(a, cx+1, dx);
   else return (dx - cx + 1) + Conta1(a, sx, cx -1);
   ```

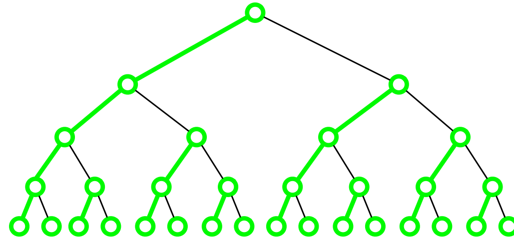
   Complessità: $T(n) = T(n/2) + O(1) = O(\log n)$.

Esercizio 2. *(14 punti)*
Dato un albero binario T, una catena sinistra di T è una sequenza di r nodi ($r \geq 1$) legati uno all'altro dal puntatore sinistro. Una catena massimale sinistra è una catena che non è contenuta in nessun'altra catena sinistra. Detto L_T il numero di catene massimali sinistre

1. Indicare le catene massimali su un albero binario completamente bilanciato di altezza 4.
2. Dimostrare che se T è completamente bilanciato e ha $n = 2^k - 1$ nodi, $L_T = 2^{k-1}$.
3. Si definisca un algoritmo efficiente che calcoli il numero di catene massimali sinistre L_T per un qualsiasi albero binario T.

SOLUZIONE

1.



2. Per induzione.

   Caso base. Per $k = 1$, l'albero contiene un solo nodo e una catena massimale: $n = 2^1 - 1 = 1$, $L_T = 1 = 2^{1-1}$.

   Passo. Sia T un albero completamente bilanciato con $n = 2^k - 1$ nodi. T è composto dal nodo radice, e da due sottoalberi T_{sx} e T_{dx} completamenti bilanciati di $2^{k-1} - 1$ nodi ciascuno. Applicando l'ipotesi induttiva sui sottoalberi, e tenendo presente che il numero di catene massimali di T è dato dalla somma delle catene massimali dei suoi sottoalberi (infatti la radice di T estende una catena massimale di T_{sx} e non dà origine a nuove catene massimali) si ottiene $L_T = L_{T_{sx}} + L_{T_{dx}} = 2^{(k-1)-1} + 2^{(k-1)-1} = 2^{k-1}$.

3. **Catene(u)**

```
if (u == NULL) return 0;
if (u.sx == NULL && u.dx == NULL) return 1;
if (u.sx == NULL) return 1 + Catene(u.dx);
else return Catene(u.sx) + Catene(u.dx);
```

   Complessità: $T(n) = O(n)$

Esercizio 3. *(8 punti)*
Dato un grafo orientato, progettare un algoritmo efficiente per determinare se il grafo è un DAG, ovvero un grafo orientato aciclico. Analizzare la complessità dell'algoritmo proposto.

SOLUZIONE
Un grafo orientato è ciclico se e solo se contiene almeno un arco all'indietro. Si ricordi che, nel corso di una visita del grafo, un arco (u, v) è detto *arco all'indietro* se v è già stato scoperto, ma la scansione della sua lista di adiacenza non è ancora terminata. Per verificare se un grafo orientato è un DAG possiamo modificare un algoritmo di visita del grafo (ad esempio la BFS) in modo che si arresti non appena si incontra un arco all'indietro. Per classificare gli archi coloriamo i vertici nel seguente modo. All'inizio i vertici sono tutti *bianchi*, diventano *grigi* quando sono scoperti, e *neri* quando termina la scansione della loro lista di adiacenza. Dunque un arco (u, v) è un arco all'indietro se v è di colore grigio.

DAG(G)

```
for (s = 0; s < n; s++) colore[s] = bianco;
for (s = 0; s < n; s++)
   if (colore[s] == bianco)
      Q = nuovaCoda();
      colore[s] = grigio;
      Q.Enqueue(s);
      while(!  Q.isEmpty())
          u = Q.dequeue();
          for (x = Adj[u].inizio; x ≠ NULL; x = x.next)
              v = x.dato;
              if (colore[v] == grigio) return FALSE;
              if (colore[v] == bianco) {colore[v] = grigio; Q.enqueue(v);}
          colore[u] = nero;
return TRUE;
```