008AA – ALGORITMICA E LABORATORIO

Appello del 27 giugno 2011

SOLUZIONI

Esercizio 1.
Si scriva una funzione `LunStr()` che, dato in input un array di stringhe `InStr[]`, restituisca un array contenente la lunghezza di ogni stringa in `InStr[]`. Non si usi alcuna funzione di libreria per il calcolo della lunghezza di una stringa.

Soluzione.

```
int * lunStr(char **a, int size){
     int * b = malloc(size*sizeof(int));
     int i = 0;
     int j;
     int cont = 0;

     for (j = 0; j < size; j++) {
          while (*(a[j]+i) != '\0') {
              i++;
              cont++;
          }
          b[j] = cont;
          cont = 0;
          i = 0;
     }
     return b;
}
```

Esercizio 2.
Progettare un algoritmo che, dato un array di interi in ingresso, verifichi *efficientemente* che tale array soddisfi la proprietà di *heap*.

1. Dare un programma in pseudocodice per l'algoritmo proposto.
2. Discutere la complessità dell'algoritmo proposto.

Soluzione.
Sfruttiamo il fatto che il nodo di chiave $a[i]$ nello heap è figlio del nodo di chiave $a[i/2]$. Dunque, per verificare la proprietà di heap occorre controllare se $a[i] \leq a[i/2]$, per ogni $i = 1, 2, ..., n-1$.

```
VerificaHeap(a, n) {
   i = 1;
   isHeap = TRUE;
   while (i < n && isHeap) {
      if (a[i] <= a[i/2]) i++;
      else isHeap = FALSE;
   }
   return isHeap;
}
```

L'algoritmo esegue una scansione dell'array, dunque la sua complessità è $O(n)$.

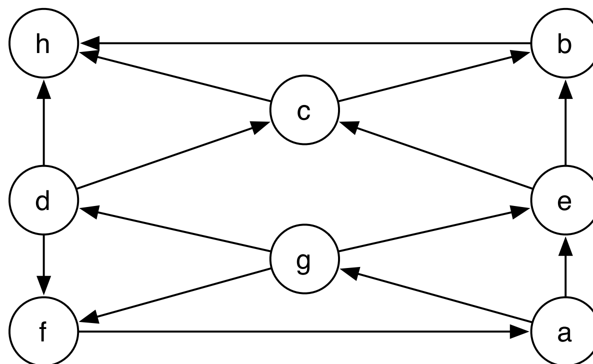
Esercizio 3.
Un nodo v in un albero binario si dice **0-bilanciato** se le altezze dei sottoalberi radicati nei suoi due figli sono uguali. Dato un albero binario, progettare un algoritmo *efficiente* che determini il numero di nodi 0-bilanciati e analizzarne la complessità.

Soluzione.
L'algoritmo restituisce la coppia `<numero di nodi 0-bilanciati, altezza>`.

```
ContaNodiBilanciati(u)
   if (u == null) return <0,-1>;
   if (u.sx == null && u.dx == null) return <1,0>;
   <nodiSx,hSx> = ContaNodiBilanciati(u.sx);
   <nodiDx,hDx> = ContaNodiBilanciati(u.dx);
   nodi = nodiSx + nodiDx;
   if (hSx == hDx) nodi++;
   h = 1 + max{hSx, hDx};
   return <nodi, h>
```

L'algoritmo esegue una visita dell'albero, la sua complessità in tempo è $T(n) = O(n)$.

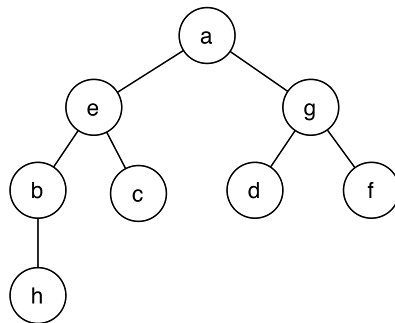
Esercizio 4.
È dato il seguente grafo orientato, rappresentato con liste di adiacenza ordinate alfabeticamente:



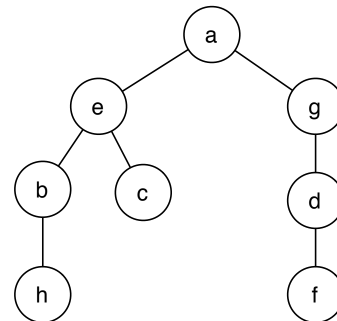
1. Indicare l'ordine di visita BFS e DFS dei vertici del grafo, partendo dal vertice a.

2. Disegnare gli alberi BFS e DFS ottenuti con le visite.
3. Indicare la classificazione degli archi indotta dalla visita DFS.

Soluzione.

1. **visita BFS**: a, e, g, b, c, d, f, h
   visita DFS: a, e, b, h, c, g, d, f
2. 

Albero BFS



Albero DFS

3. **Archi dell'albero**: (a, e), (a, g), (e, b), (e, c), (b, h), (g, d), (d, f)
   Archi all'indietro: (f, a)
   Archi in avanti: (g, f)
   Archi trasversali a sinistra: (c, b), (c, h), (d, c), (d, h), (g, e)