Algoritmi e Strutture di Dati – A.A. 2018-2019
Esame scritto del 04/02/19 – D.M. 270-9CFU
Libri e appunti chiusi – Tempo = 2:00h

9

☐ Note (vincoli, indisponibilità, preferenze, ecc.) ……
……
……

N.B.: gli esami orali si volgeranno dal 5 al 22 febbraio. Il docente non è disponibile da lunedì 24 in poi.

Cognome: _____ Nome: _____Matricola:_____

DOMANDA SULLA COMPLESSITA' ASINTOTICA (3 punti su 30)

Discuti la complessità computazionale nel caso peggiore (in termini di O-grande, Omega e Theta) delle seguenti procedure in funzione del numero n di elementi dell'albero. Assumi che

- SOTTOALBERO_CAMMINO verifichi che il sottoalbero radicato al nodo corrente sia un cammino e, nel caso peggiore, faccia un numero di operazioni proporzionale alla dimensione del sottoalbero radicato al nodo corrente
- AGGIUNGI_IN_TESTA faccia un numero di operazioni costante

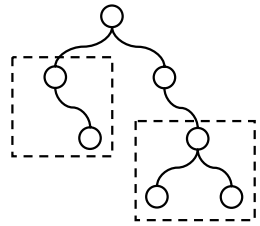
```
FUNZIONE(T)       /* T è un albero binario di interi */

L.head = NULL     /* L è una nuova lista (vuota) di interi */
FUNZ_RIC(T.root,L)
return L


FUNZ_RIC(v,L)

 if(v==NULL) return
 if(SOTTOALBERO_CAMMINO(v))
      AGGIUNGI_IN_TESTA(L,v.info)
 FUNZ_RIC(v.left,L)
 FUNZ_RIC(v.right,L)
```

ALGORITMO IN LINGUAGGIO C (27 punti su 30)

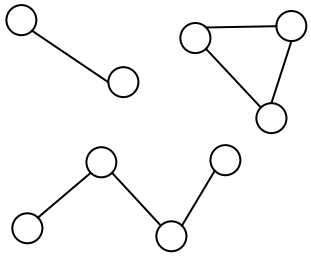
Scrivi in linguaggio C il codice della funzione

```
int verifica(int h, nodo_albero* a, grafo* g)
```

che accetti in input un intero **h**, un puntatore **a** alla radice di un albero binario di interi e un puntatore **g** ad grafo non orientato rappresentato tramite oggetti e riferimenti. La funzione restituisce 1 se il numero di nodi dell'albero **a** che sono radici di un sottoalbero di profondità **h** sono tanti quanti sono i nodi della componente più piccola del grafo **g**, altrimenti la funzione restituisce 0. Se grafo e albero sono entrambi vuoti (cioè uguali a NULL) la funzione ritorna 1. Se uno è vuoto e uno no, allora ritorna 0. Se non ci sono sottoalberi di profondità **h** in **a** (cioè se il loro numero è 0) la funzione ritorna 0.



Albero a



Grafo g

Per esempio l'albero **a** in figura ha due sottoalberi di profondità h=1 (rinchiusi nei quadrati) e la componente connessa più piccola del grafo **g** ha 2 nodi, dunque `verifica(1,a,g)` ritorna 1 (true) mentre `verifica(2,a,g)` ritorna 0 (false) perché **a** contiene un solo sottoalbero di profondità 2 ma **g** non ha una componente connessa con un solo nodo.

Usa le seguenti strutture (che si suppone siano contenute nel file "`strutture.h`"):

```
typedef struct nodo_struct {
        elem_nodi* pos;  /* posizione nodo nella
                            lista del grafo */
        elem_archi* archi;   // lista archi incidenti
        int color;
} nodo;

typedef struct arco_struct {
        elem_archi* pos;  // pos. arco lista grafo
        nodo* from;
        nodo* to;
        elem_archi* frompos;  // pos. arco nodo from
        elem_archi* topos;      // pos. arco nodo to
} arco;

typedef struct elem_lista_nodi {
        struct elem_lista_nodi* prev;
        struct elem_lista_nodi* next;
        nodo* info;
} elem_nodi;   // elemento di una lista di nodi
```

```
typedef struct elem_lista_archi {
        struct elem_lista_archi* prev;
        struct elem_lista_archi* next;
        arco* info;
} elem_archi;  // elemento di una lista di archi

typedef struct {
        int numero_nodi;
        int numero_archi;
        elem_archi* archi;  // lista degli archi
        elem_nodi* nodi;    // lista dei nodi
} grafo;

/* struttura per l'albero binario */

typedef struct nodo_albero_struct {
        struct nodo_albero_struct* left;
        struct nodo_albero_struct* right;
        int info;
} nodo_albero;
```

È possibile utilizzare qualsiasi libreria nota e implementare qualsiasi funzione di supporto a quella richiesta.

SOLUZIONE DOMANDA SULLA COMPLESSITA' ASINTOTICA

FUNZIONE(T) ha la stessa complessità asintotica di **FUNZ_RIC**(v,L) in quanto non fa altro che richiamare quest'ultima sulla radice dell'albero T (tutte le altre operazioni sono Theta(1)).

FUNZ_RIC(v,L) esegue **AGGIUNGI_IN_TESTA** condizionato alla proprietà verificata da **SOTTOALBERO_CAMMINO**(v). L'aggiunta in testa ha costo Theta(1) e dunque, anche se fosse eseguita per ogni nodo dell'albero, avrebbe costo totale Theta(n), che nulla aggiunge al costo Theta(n) già pagato dalla visita. La funzione **SOTTOALBERO_CAMMINO**(v), invece, ha un costo proporzionale alla dimensione del sottoalbero radicato sul nodo v. Se l'intero albero fosse effettivamente un cammino avremmo che il costo della funzione **SOTTOALBERO_CAMMINO** sarebbe n quando lanciata sulla radice, n-1 quando lanciata sul figlio della radice, e così via, portando un contributo complessivo di Theta(n^2). Dunque il costo di **FUNZ_RIC** (e di **FUNZIONE**) è Theta(n^2).

SOLUZIONE ALGORITMO IN LINGUAGGIO C

```
/* Funzione richiesta */

int verifica(int h, nodo_albero* a, grafo* g) {
     if (a == NULL && g == NULL) return 1;
     if (a == NULL || g == NULL) return 0;
     return numero_sottoalberi_prof(a, h) == dim_componente_minima(g);
}

/* Funzione ricorsiva che calcola il numero di sottoalberi dell'albero
   passato in input che hanno profondità esattamente h */

int numero_sottoalberi_prof(nodo_albero* a, int h) {
     if( a == NULL ) return 0;
     int risultato = numero_sottalberi_prof(a->left, h) +
                     numero_sottalberi_prof(a->right, h);
     if( prof_albero(a) == h)
          risultato++;
     return risultato;
}

/* Calcola la profondità di un albero. Un albero con la sola radice
   ha convenzionalmente profondità zero. */

int prof_albero(nodo_albero* a) {
     if (a == NULL) return -1;
     int l = prof_albero(a->left);
     int r = prof_albero(a->right);
     if( l > r ) return l+1;
     return r+1;
}

/* Calcola la dimensione della componente connessa più piccola del grafo
   passato in input */

int dim_componente_minima(grafo* g) {

     /* 1) coloro il grafo con l'etichetta 0 (non visitato) */

     elem_nodi* ln = g->nodi;
     while( ln != NULL ) {
          ln->info->color = 0;
          ln = ln->next;
```

```
      }

      /* 2) eseguo una serie di visite memorizzando la dimensione
            della componente più piccola nella variabile "piu_piccola" */

      /* questo primo lancio qui sotto è solo per inizializzare la
         variabile "piu_piccola" ad un valore che abbia senso (non
         posso inizializzarla a zero perché non sarebbe mai aggiornata) */

      int piu_piccola = DFS_size(g->nodi->info);

      ln = g->nodi;  // riutilizzo la variabile ln (non indispensabile)
      while( ln != NULL ) {
            if(ln->info->color == 0) {  // trovata componente da visitare
                  int size = DFS_size(ln->info);             // la visito
                  if( size < piu_piccola ) piu_piccola = size;
            }
            ln = ln->next;
      }
      return piu_piccola;
}

/* Esegue una visita in profondità del grafo riportando il numero dei
   nodi della componente connessa visitata */

int DFS_size(nodo* n){

      n->color = 1;
      int output = 1; // conto il nodo corrente
      elem_archi* le = n->archi;
      while( le != NULL ) {
            nodo* altro_nodo = nodo_opposto(le->info, n);
            if( altro_nodo->color == 0 )
                  output = output + DFS_size(altro_nodo);
            le = le->next;
      }
      return output;
}

/* Ritorna il nodo incidente sull'arco "e" e diverso dal nodo "n" */

nodo* nodo_opposto(arco* e, nodo* n) {
      if ( e->from == n ) return e->to;
      return e->from;
}
```