Algoritmi e Strutture di Dati – A.A. 2017-2018
Esempio di prova scritta del 19 gennaio 2018 – D.M. 270-9CFU
Libri e appunti chiusi
Tempo = 2:00h

☐ Note (es: correzione veloce, eventuali indisponibilità, ecc.) ……
……
……

Cognome: _____ Nome: _____Matricola:_____

DOMANDA SULLA COMPLESSITA' ASINTOTICA

Discuti la complessità computazionale nel caso peggiore (in termini di O-grande, Omega e Theta) della seguente procedura in funzione del numero n di elementi dell'albero. Assumi che AGGIUNGI-IN-TESTA faccia un numero di operazioni costante, mentre AGGIUNGI-IN-CODA faccia un numero di operazioni proporzionali alla lunghezza della lista corrente.

```
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,0)
return L

FUNZ-RIC(v,L,depth)
 if(v==NULL) return
 if(depth < 10)
     AGGIUNGI-IN-CODA(L,v.info)
 else
     AGGIUNGI-IN-TESTA(L,v.info)

 FUNZ-RIC(v.left,L,depth+1)
 FUNZ-RIC(v.right,L,depth+1)
```

ALGORITMO IN LINGUAGGIO C

Scrivi in linguaggio C il codice della funzione

```
grafo_array* componente_connessa(grafo_oggetti* g, nodo* n)
```

che accetti in input un puntatore ad grafo non orientato `g` rappresentato tramite oggetti ed il riferimento ad un nodo `n` e produca in output il riferimento ad un grafo non orientato rappresentato tramite un array di liste di adiacenza che contiene esclusivamente i nodi e gli archi della componente connessa che contiene il nodo `n`. Si usino le seguenti strutture:

```
typedef struct nodo_struct {
        elem_nodi* pos;  // posizione nodo
        elem_archi* archi;
        int color;
} nodo;

typedef struct arco_struct {
        elem_archi* pos;  // posizione arco
        nodo* from;
        nodo* to;
        elem_archi* frompos;  // posizione arco
        elem_archi* topos;      // posizione arco
} 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;
        arcoo* info;
} elem_archi;  // elemento di una lista di archi

typedef struct {
        int numero_nodi;
        int numero_archi;
        elem_archi* archi;
        elem_nodi* nodi;
} grafo_oggetti;
```

```
typedef struct elem_lista_int {
        struct elem_lista_int* prev;
        struct elem_lista_int* next;
        int info;
} elem_int;   // elemento di una lista di interi

typedef struct {
        int numero_nodi;
        elem_int** A;  // liste di adiacenza
} grafo_array;
```

Si dispone di una libreria che implementa una tabella hash `htable` di dimensione dinamica in cui le chiavi sono di tipo `nodo*` e i valori sono di tipo `int`, con le seguenti funzioni (dichiarate nel file `htable.h`):

```
htable* crea_tabella()
void aggiungi_elemento(htable* t, nodo* key, int i)
void rimuovi_elemento(htable* t, nodo* key)
int contiene_chiave(htable* t, nodo* key)
int trova_valore(htable* t, nodo* key)
void distruggi_tabella(htable * t)
```

È possibile utilizzare qualsiasi libreria nota e implementare qualsiasi funzione di supporto a quelli richiesti.

SOLUZIONI

DOMANDA SULLA COMPLESSITA' ASINTOTICA

Risposta sufficiente nel compito d'esame:

- La funzione FUNZIONE(T) non fa che richiamare FUNZ-RIC(v,L,depth) e dunque ha la stessa complessità asintotica nel caso peggiore.
- La funzione FUNZ-RIC(v,L,depth), assumendo che la dimensione dell'input sia sufficientemente grande, esegue una visita dell'albero e un numero lineare di inserimenti in testa (il numero di inserimenti in coda si può supporre costante). La sua complessità è dunque $\Theta(n)$

Risposta dettagliata per comprendere il fenomeno:

La funzione FUNZ-RIC(v,L,depth) esegue una visita ricorsiva dell'albero binario. Durante la visita viene aggiornata la lista L. Fino a profondità 9 gli indici dei nodi dell'albero vengono aggiunti in coda (con costo più elevato), dalla profondità 10 in poi vengono aggiunti in testa (con costo costante). Poiché stiamo studiando la complessità asintotica siamo autorizzati a ritenere la dimensione dell'albero sufficientemente grande da poter assumere che il numero dei nodi a profondità minore di 10 sia un numero costante, mentre il numero dei nodi a profondità maggiore di 10 sia un numero lineare. Ne deriva che la funzione ricorsiva esegue una visita dell'albero (che avrebbe una complessità $\Theta(n)$ se non ci fosse qualche altra operazione) più un inserimento in testa alla lista per ogni nodo (con una complessità totale aggiuntiva di $\Theta(n)$).

ALGORITMO IN LINGUAGGIO C

```
#include "htable.h"

/* qui ci vanno le strutture fornite nel testo */

/* visita DFS standard di un grafo rappresentato
   mediante oggetti e riferimenti */

void DFS(nodo* n) {

      n->color = 1; // coloro il nodo come visitato
      elem_archi ea = n->archi;
      while( ea != NULL) {
            nodo* altro_nodo = ea->info->from;
            if( altro_nodo == n) {
                  altro_nodo = ea->info->to;
            }
            if( altro_nodo->color == 0) {  // se non e' gia' visitato...
                  DFS(altro_nodo);          // ... lo visito
            }
            ea = ea->next
      }
}

/* aggiunge un arco al grafo rappresentato tramite un
   array di liste di adiacenza */

void aggiungi_arco(grafo_array g, int u, int v) {

      elem_int* ei = (elem_int*)malloc(sizeof(elem_int));
      ei->info = v;
      ei->next = g->A[u];
      ei->prev = NULL;   // lo aggiungero' in testa alla lista
      if( g->A[u] != NULL) { // c'era gia' un elemento in lista
            g->A[u]->prev = ei;
      }
      g->A[u] = ei;

    /* l'arco opposto viene aggiunto quando viene trovato
       a partire dal nodo v nel grafo_oggetti g */

}

/* funzione richiesta dal compito */

grafo_array* componente_connessa(grafo_oggetti* g, nodo* n) {

      elem_nodi* en = g->nodi;
      while (en != NULL) {
            en->info->color = 0; // marco tutti i nodi con 0
      }
      DFS(n);  // visito la componente di n e marco tutti i nodi con 1

    /* popolo una tabella hash che contiene le corrispondenze tra
       i nodi del grafo_oggetti e quelli del grafo_array.
       Cont e' un contatore dei nodi. Il suo valore viene via via
       usato come intero identificatore del nodo nel grafo_array. */

      htable* htab = crea_tabella();
      int cont = 0;
```

```
      en = g->nodi;
      while (en != NULL) {
            if (en->info->color == 1) {  // il nodo e' stato visitato
                  cont++; // conto questo nodo
                  aggiungi_elemento(htab,en->info,cont);
                        // usero' cont come identificatore di questo nodo
            }
      }

    /* creo il grafo di output */

      grafo_array* ga = (grafo_array*)malloc(sizeof(grafo_array));
      ga->numero_nodi = cont;
      ga->A = (elem_int**)calloc(ga->numero_nodi,sizeof(elem_int*));

    /* ora aggiungo gli archi al grafo di output */

      en = g->nodi;
      while (en != NULL) {
            if (en->info->color == 1) {  // en->info e' stato visitato

                  /* scorro la lista degli archi del nodo en->info
                     e aggiungo gli archi corrispondenti al
                     grafo di output */

                  elem_archi* ea = en->info->archi;
                  while ( ea != NULL ) {
                        nodo* altro_nodo = ea->info->from;
                        if( altro_nodo == en->info ) {
                              altro_nodo = ea->info->to;
                        }
                        aggiungi_arco(ga,trova_valore(htab,en->info),
                                         trova_valore(htab,altro_nodo));
                        ea = ea->next;
                  }
            }
            en = en->next;
      }
      distruggi_tabella(htab);
      return ga;
}
```