ASD (FI2 12 CFU - tutti), ASD (5 CFU)

Appello del 16-6-2016 – a.a. 2015-16 – Tempo a disposizione: *120 minuti* – somma punti in gioco: 33

Problema 1

`[Punti: (a) 2/30; (b) 2/30]; (c) 2/30]`

Si considerino i metodi Java di seguito illustrati.

```
// assumere n > 0
static int[] m1(int n) {
  int[] p = new int[n];
  for(int i=0; i < n; i++) p[i]=i+1;
  return p;
}

// assumere a.length > 0
static double m2(int[] a) {
    double t = 0;
    double[] temp = new double[a.length-1];
    for(int i = 0; i < temp.length; i++)
        temp[i] = (a[i] + a[i+1])/2.0;
    for(int i = 1; i < temp.length/2; i++)
        t += Math.sqrt(temp[temp.length-i]-temp[i-1]); // assumere Theta(1)
    return t;
}

// assumere n > 0
static double m(int n) {
    return m2(m1(n));
}
```

Sviluppare, *argomentando adeguatamente* (il 50% del punteggio dell'esercizio sarà sulle argomentazioni addotte), quanto segue:

(a) Determinare il costo asintotico dell'algoritmo descritto da `m1(int)` in funzione di $z = |\mathtt{n}|$, dimensione dell'input.

(b) Determinare il costo asintotico dell'algoritmo descritto da `m2(int[])` in funzione di $w = |\mathtt{a}|$, dimensione dell'input.

(c) Determinare il costo asintotico dell'algoritmo descritto da `m(int)` in funzione di $z = |\mathtt{n}|$, dimensione dell'input. In particolare, evidenziare le componenti di costo dovute a `m1` e quelle dovute a `m2`.

Problema 2

`[Punti: 9/30]`

Si considerino alberi di arità k (ogni nodo ha massimo k figli), rappresentati gestendo esplicitamente per ciascun nodo un puntatore al primo figlio e un puntatore al prossimo fratello.

Scrivere un metodo Java, oppure una funzione C, che, data la radice di un albero generico non vuoto, determini *in tempo lineare*, restituendola, la radice del più grande sottoalbero completo.[1] Nel caso di più sottoalberi completi massimali restituirne uno qualunque.

[1] Si rammenta che un albero *completo* è un albero in cui tutti i nodi interni hanno il massimo numero di figli e tutti i rami la stessa lunghezza.

Possibile bozza di soluzione

Si tratta di una bozza di soluzione in C, scritta di getto e non testata, per mostrare lo schema di ragionamento basato su visita post-order.

```
#include <stdio.h>
#include <stdlib.h>

/* tipo base per i nodi */
typedef struct Node {
    int key;
    struct Node *firstChild;
    struct Node *nextSibling;
} Node;

/*
    tipo creato per restituire la radice del sottoalbero più grande
    e la sua altezza
*/
typedef struct {
    Node *root;
    int height;
} CompleteTree;

/* costruttore per tipo CompleteTree */
CompleteTree *newCompleteTreeDescription(Node *root, int height) {
    CompleteTree *temp = malloc(sizeof(CompleteTree));
    temp->root = root;
    temp->height = height;
    return temp;
}

/* funzione principale */
CompleteTree *getMaxCompleteSubtree(Node *v, int k) {
    Node *temp;
    CompleteTree *buffer, *currMax;
    int cont, growable;

    /* passo base */
    if(v == NULL) return NULL;
    if(v->firstChild == NULL) return newCompleteTreeDescription(v, 0);

    /* passo ricorsivo (post-order) */

    temp = v->firstChild;
    cont = 1; /* conta i figli, questo è il primo */

    /* prima chiamata ricorsiva fuori dal ciclo per inizializzare currMax */
    currMax = getMaxCompleteSubtree(temp, k);
    if(currMax->root != temp)
        growable = 0; /* sottoalbero completo è più sotto */
    else
        growable = 1; /* per ora v potrebbe essere radice di sottoalbero completo */
    temp = temp->nextSibling;
    while(temp != NULL) {
        buffer = getMaxCompleteSubtree(temp, k);
        cont++;
        if(buffer->root != temp) growable = 0; /* sottoalbero completo è più sotto */
        if(buffer->height > currMax->height) {
            growable = 0;
            currMax = buffer;
        } else if(buffer->height < currMax->height) growable = 0;
```

```
        temp = temp->nextSibling;
    }
    if(growable && cont == k) { /* trovato sottoalbero completo più grande */
        int h = currMax->height+1;
        free(currMax);
        return newCompleteTreeDescription(v, h);
    } else return currMax;
}
```

Problema 3

Miscellanea argomenti.

1. Descrivere un algoritmo efficiente per determinare il k-esimo elemento di un insieme[2] di n elementi. (2 punti)

2. Svolgere la parte appropriata:

   - Solo a.a. 2015-16:
     descrivere (tramite pseudo-codice) la struttura dati *union-find* per la gestione efficiente di una collezione di insiemi disgiunti (operazioni: MAKESET, FIND, UNION). (3 punti)
   - Solo aa.aa. precedenti al 2015-16:
     descrivere (tramite pseudo-codice) l'algoritmo di Kruskal, determinandone il costo.[3] (3 punti)

3. Si faccia riferimento alla gestione di un grafo semplice e *sparso*.[4] Confrontare l'efficienza temporale delle seguenti operazioni, usando, rispettivamente, liste di adiacenza e matrice di adiacenza: test di adiacenza di due vertici assegnati, determinazione di tutti i vertici adiacenti a uno dato, determinazione dei vertici di grado massimo e minimo. (4 punti)

4. Fornire un esempio di grafo semplice, connesso e pesato che ammette uno spanning tree (minimum spanning tree per gli studenti degli aa.aa. precedenti al 2015-16) diverso dall'albero dei cammini minimi. (2 punti)

Problema 4

`[Punti: (a) 5/30; (b) 2/30]`

Si consideri il grafo del World Wide Web, i cui vertici sono pagine web e i cui archi rappresentano dei collegamenti (link) da una pagina a un'altra pagina. Si assuma per semplicità che tale grafo abbia dimensioni compatibili con la quantità di RAM disponibile in un sistema di elaborazione.[5]

Ciò premesso, data una pagina web p e un positivo $k > 0$, si vuole sapere se esiste una sequenza di navigazione che a partire da p porti di nuovo a p in non più di k click. Più formalmente, se esiste un ciclo semplice $(p_0, p_1, p_2, \ldots, p_h)$, $1 \leq h \leq k$, con $p = p_0 = p_h$ e $p_i \neq p$ per $i \in \{1, \ldots, h-1\}$.

Si richiede di:

(a) Presentare un algoritmo (pseudo-codice) che dato il grafo $G = (V, E)$, p e k, restituisca un ciclo semplice del tipo descritto, se esiste, ovvero restituisca NULL.

(b) Scegliere una rappresentazione per il grafo in modo da aumentare l'efficienza dell'algoritmo di cui al passo precedente.

---

[2] Sull'insieme è definita una relazione d'ordine totale.

[3] Il costo va giustificato.

[4] Un grafo $G = (V, E)$ è detto *sparso* se $|E| \in O(|V|)$.

[5] Tale ipotesi non è realistica.