| Esame di | |
|---|---|
| | **Algoritmi e strutture dati (parte di Fondamenti di informatica II 12 CFU)** |
| | **Algoritmi e strutture dati (V.O., 5 CFU)** |
| | **Algoritmi e strutture dati (Nettuno, 6 CFU)** |

**Appello del 5-9-2018, prova del 20-9-2018 – a.a. 2017-18**
**Tempo a disposizione: *4 ore* – somma punti: 35**

**Istruzioni**

Lanciare la macchina virtuale Oracle VirtualBox e lavorare all'interno della cartella `ESAME`, avendo cura di creare all'interno della cartella stessa:

- un file `studente.txt` contenente, una stringa per riga, cognome, nome, matricola, email; in tutto quattro righe, memorizzando il file nella cartella `esame`; è possibile aggiungere una quinta riga contenente eventuali informazioni addizionali che intendi porre all'attenzione del docente;
- una cartella `java.<matricola>`, o `c.<matricola>`, ove al posto di `<matricola>` occorrerà scrivere il proprio numero di matricola, **contenente i file prodotti per risolvere il Problema 2** (in tale cartella si copi il contenuto dell'archivio `c-aux.zip` o `java-aux.zip`); tale cartella va posizionata nella cartella `esame`;
- tre altri file `probl1.<matricola>.txt`, `probl3.<matricola>.txt` e `probl4.<matricola>.txt`, contenenti, rispettivamente, gli svolgimenti dei problemi 1, 3 e 4; i tre file vanno posti nella cartella `esame`.

Attenzione: i simboli < e > usati nei nomi dei file fanno parte del linguaggio dei metadati e **non debbono essere inclusi nei nomi reali**. *È possibile consegnare **materiale cartaceo integrativo** solo contenente **equazioni e disegni**, che verrà esaminato solo a condizione che risulti ben leggibile e a completamento di quanto digitalmente redatto.*

Per l'esercizio di programmazione (Problema 2) è possibile usare qualsiasi ambiente di sviluppo disponibile sulla macchina virtuale. Si raccomanda però di controllare che i file vengano salvati nella cartella `java.<matricola>`, o `c.<matricola>`. Si consiglia inoltre per chi sviluppa in C di compilare da shell eseguendo il comando `make` e poi eseguire `driver` per verificare la correttezza dell'implementazione. Analogamente si raccomanda per chi sviluppa in Java di compilare da shell eseguendo il comando `javac *.java` e poi eseguire `java Driver` per verificare la correttezza dell'algoritmo.

**N.B. Le implementazioni debbono essere compilabili. In caso contrario, l'esame non è superato.**

## Problema 1 Analisi algoritmo

Si considerino i metodi Java di seguito illustrati.

```
// c.length > k >= 0
static double horner(double c[], int k, double x) {
    double r = c[k];
    if(k < c.length - 1)
        return x*horner(c, k+1, x) + r;
    return r;
}

static double horner(double c[], double x) {
    return horner(c, 0, x);
}
```

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

(a) Determinare il costo temporale asintotico dell'algoritmo descritto da `horner(double[], double)` in funzione della dimensione dell'input. [4/30]

(b) Definire il significato di *ricorsione di coda* e spiegare se `horner(double[], int, double)` presenta una ricorsione di coda oppure no. [2/30]

**Problema 2 Progetto algoritmi C/Java [soglia minima: 5/30]**

In questo problema si fa riferimento a *grafi* con archi pesati. I grafi sono rappresentati attraverso *liste di incidenza*, ovvero liste di adiacenza in cui appaiono, per ogni nodo, gli archi incidenti invece dei nodi adiacenti. A ciascun nodo $u$ è dunque associata una lista collegata contenente degree($u$) elementi, ciascuno dei quali descrive un arco (incidente $u$); inoltre ad ogni nodo sono associati un numero progressivo (a iniziare da 0) automaticamente assegnato dalla primitiva di inserimento nodo ed un nome (stringa) fornita come input alla stessa primitiva. Similmente, a ciascun arco è automaticamente assegnato un numero progressivo (a iniziare da 0). Nodi ed archi sono rappresentati dalle classi/strutture `GraphNode`/`graph_node` e `GraphEdge`/`graph_edge`; il grafo è rappresentato dalla classe/struttura `Graph`/`graph`.

La gestione delle liste (di archi e/o di nodi) fa uso del tipo `linked_list` (C) o la classe `java.util.LinkedList<E>` (Java); per tali tipi sono già disponibili, nel codice sorgente fornito, o nella classe `java.util.LinkedList<E>`, le primitive di manipolazione. Si noti che gli elementi delle liste sono dei contenitori che includono puntatori ai nodi/archi del grafo.

(A fine testo è disponibile un disegno che mostra la rappresentazione di un semplice grafo di esempio).

Sono inoltre già disponibili le primitive di manipolazione grafo: creazione di grafo vuoto, get lista nodi, inserimento di nuovo nodo, get lista archi incidenti un dato nodo, inserimento nuovo arco, get chiave (progressivo) di un dato nodo/arco, get nome (stringa) di un dato nodo, get opposite di un nodo per un dato arco, cancellazione arco, cancellazione nodo (e relativi archi incidenti), cancellazione grafo, stampa grafo. Per dettagli sulle signature di tali primitive, così come per dettagli su quali porzioni di memoria allocata vengono liberate dalle varie primitive di cancellazione, si rimanda ai file sorgente distribuiti. Si noti che le primitive forniscono un insieme base per la manipolazione di grafi, ma i problemi proposti possono essere risolti usandone solo un sottoinsieme.

Tutto ciò premesso, risolvere al computer quanto segue, in Java o in C.

La ditta PowerPlants & Co., leader nel settore delle infrastrutture elettriche, ha rilevato un sensibile incremento nei costi di gestione dei suoi impianti di approvvigionamento delle maggiori città italiane. Per questo, decide di ricostruire da zero i collegamenti tra le principali centrali elettriche dislocate su tutta la nazione. Il progetto è ambizioso, la ditta vuole minimizzare la quantità di cavo necessario per mettere in collegamento tutte le centrali, pur mantenendo la connettività globale del sistema. Per risolvere questo problema, si richiede di:

2.1 Realizzare una funzione/metodo `getCompleteGraph` che, data una lista di $n > 3$ centrali elettriche, ritorna un puntatore al grafo completo avente un nodo per ogni centrale della lista, ed un arco per ogni coppia di nodi del grafo. Si noti che una centrale è rappresentata da un nome [stringa], ed una coppia di coordinate $x$ e $y$ [double]; inoltre, poiché la rappresentazione degli archi prevede un peso, è possibile inizializzare ad infinito il peso di ogni arco, utilizzando la costante `INF` appositamente definita. [3/30]

2.2 Realizzare una funzione/metodo `setWeightsInGraph` che, dato il grafo completo delle centrali elettriche, modifica i pesi degli archi del grafo secondo il criterio che segue. Per ciascun nodo $u$ individuare i tre nodi più vicini a $u$: $v$, $w$, $z$ e modificare i pesi degli archi $\{u, v\}$, $\{u, w\}$ e $\{u, z\}$ impostando le rispettive distanze euclidee (si noti che nel modulo/classe `GraphServices` viene fornita una funzione/metodo di supporto per calcolare la distanza euclidea di due centrali elettriche). [3.5/30]

2.3 Realizzare una funzione/metodo `getMinWire` che, dato un grafo pesato, ritorna una lista degli archi contenuti in un opportuno Spanning Tree del grafo, selezionato con l'obiettivo di minimizzare la quantità di cavo impiegato per connettere tutte le centrali elettriche. Si noti che per la risoluzione di questo punto, è possibile utilizzare i prodotti degli esercizi precedenti. [3.5/30]

È necessario implementare le funzioni in `graph_services.c` o i metodi in `GraphServices.java`, identificati dal commento `/*DA IMPLEMENTARE*/`. In tali file è permesso sviluppare nuovi metodi/funzioni ausiliari, se utile. *Non è assolutamente consentito modificare metodi e strutture già implementati.* È invece possibile modificare il file `driver.c`/`Driver.java` per poter effettuare ulteriori test.

Per il superamento della prova al calcolatore è necessario conseguire almeno 5 punti. Si prega di non fare usi di package nel codice Java.

**Problema 3 Algoritmi**

(a) Per il problema $\mathcal{A}$ sono noti un algoritmo di costo $\Theta(n^2)$, un secondo di costo $\Theta(n^2 \log n)$ e un lower bound $\Omega(n \log n)$. Poiché molti ricercatori stanno ancora studiando il problema è lecito attendersi la scoperta di nuovi e più efficienti algoritmi. Alla luce delle informazioni riportate, qual è il limite inferiore all'upper bound di un nuovo formidabile algoritmo? Discutere dettagliatamente. [4/30]

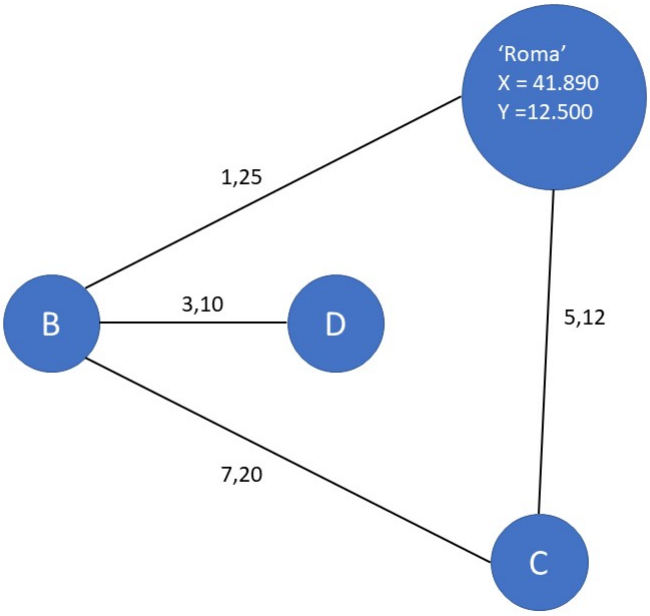
(b) Pare che nel 2050 saranno disponibili nuovi processori e nuove architetture di calcolo in cui i confronti fra valori avranno costo nullo, mentre le assegnazioni di valori atomici continueranno ad avere costo costante. Quale algoritmo di ordinamento diverrebbe in quel caso preferibile? Discutere in dettaglio.[1] [5/30]

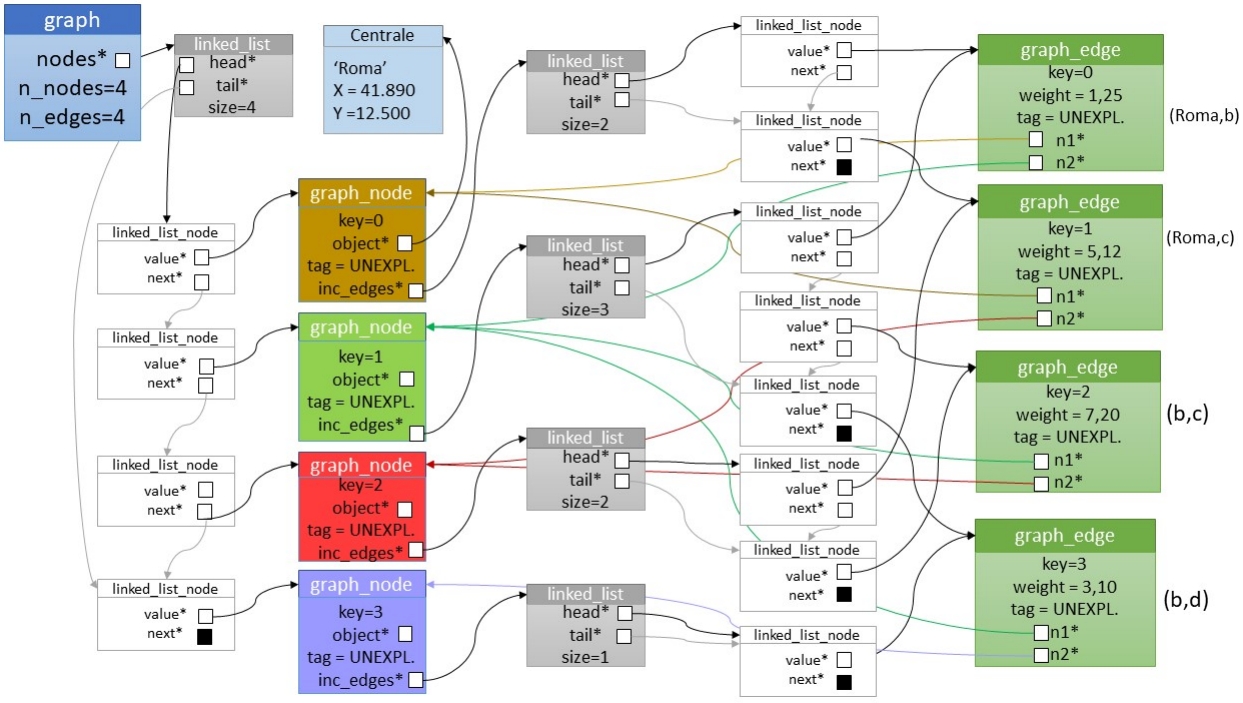
(c) Ad Alice viene posto il seguente problema: due grafi orientati rappresentati tramite liste di adiacenza vengono forniti in input a un algoritmo: come può questo stabilire se i due grafi sono lo stesso grafo? Alice è conscia del fatto che per lo stesso grafo esistono molte rappresentazioni per lista di adiacenza che differiscono per l'ordine con cui gli adiacenti di un nodo compaiono in ogni lista; ciascun nodo ha un campo informativo `key`, intero positivo, che univocamente identifica un nodo del grafo. Puoi aiutarla? [4/30]

**Problema 4 Intersezione di due dizionari**

Due dizionari ordinati basati su chiavi intere sono gestiti attraverso due BST. Si richiede di scrivere un algoritmo (codice o pseudo-codice) che, dati due BST del tipo descritto, determina l'intersezione dei due dizionari, restituendone le chiavi in una lista fornita in output, ove le chiavi sono ordinate e presenti senza ripetizioni. [6/30]

[1] L'eventualità ipotizzata ha valore puramente speculativo: alla luce delle nozioni oggi disponibili, non appare possibile.


'Roma'
X = 41.890
Y =12.500
1,25
3,10
5,12
7,20
B
D
C


graph
nodes*
n_nodes=4
n_edges=4
linked_list
head*
tail*
size=4
Centrale
'Roma'
X = 41.890
Y =12.500
linked_list_node
value*
next*
graph_node
key=0
object*
tag = UNEXPL.
inc_edges*
key=1
key=2
key=3
linked_list
head*
tail*
size=2
size=3
size=1
graph_edge
key=0
weight = 1,25
tag = UNEXPL.
n1*
n2*
(Roma,b)
key=1
weight = 5,12
(Roma,c)
key=2
weight = 7,20
(b,c)
key=3
weight = 3,10
(b,d)