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 4-6-2019 – a.a. 2018-19 – Tempo: *4 ore* – somma punti: 33

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**.

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.

Analisi algoritmo

Si considerino i metodi Java di seguito illustrati.

```
static void incr(byte[] a, int n, int p) {
    byte b = 0;
    for(int j = 0; j < p; j++)
        b = (byte)(b+a[n-p+j]);
    a[n] = b;
}
```

```
static byte[] raddoppia(byte[] a) {
    byte[] b = new byte[2*a.length];
    for(int i = 0; i < a.length; i++) b[i] = a[i];
    for(int i = a.length; i < b.length; i++)
        incr(b, i, a.length);
    return b;
}
```

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 `raddoppia(byte[] a)` in funzione di z, la dimensione dell'input. **Non usare simboli che non sono definiti** (ad esempio: n riferita alla dimension dell'input non è definita). [4/30]

(b) Spiegare brevemente qual è la relazione tra le componenti di `b` e quelle di `a`. [2/30]

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

In questo problema si fa riferimento a *grafi semplici*. I grafi sono rappresentati attraverso *liste di adiacenza*. A ciascun nodo u è dunque associata una lista collegata contenente degree(u) elementi, ciascuno dei quali è il riferimento a uno dei vicini di u. I nodi sono rappresentati dalle classi/strutture `GraphNode`/`graph_node`, mentre il grafo è rappresentato dalle classi/strutture `Graph`/`graph`.

La gestione delle liste di nodi deve essere effettuata mediante il 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 del grafo.

Sono inoltre già disponibili le primitive di manipolazione grafo: creazione di grafo vuoto, get lista nodi, get lista nodi vicini di nodo dato, inserimento di nuovo nodo, inserimento nuovo arco (*da implementare, vedi sotto*), get label/valore (stringa) di un dato nodo, cancellazione arco, cancellazione nodo (e relativi archi incidenti), cancellazione grafo, stampa grafo, lettura grafo da file (rappresentato come lista di archi non diretti). 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.

2.1 Implementare la funzione/metodo `addEdge` della classe `Graph` (modulo `graph` in C) che aggiunge un arco. La funzione/metodo riceve in ingresso i riferimenti ai nodi estremi dell'arco. Si noti che il grafo rappresentato deve essere semplice e privo di cappi. Ciò significa che i) non bisogna aggiungere archi già esistenti e ii) non bisogna aggiungere cappi. [3/30]

2.2 Implementare il metodo `connectedComponents` (funzione `connected_components` in C) della classe `GraphServices` (modulo `graph_services` in C), che stampa a schermo le componenti connesse del grafo. Più precisamente, per ogni componente connessa, il metodo/funzione deve stampare la lista dei vertici che ne fanno parte (i loro valori). Si esegua il programma `driver` (`Driver` in Java) per un esempio.

Suggerimento: se implementata in modo opportuno, la soluzione di questo punto potrebbe essere utile per risolvere il punto successivo. [4/30]

2.3 Implementare il metodo/funzione `distances` della classe `GraphServices` (modulo `graph_services` in C) che, dato un nodo sorgente, stampa per ogni altro nodo appartenente alla componente connessa della sorgente, il numero minimo di archi che lo separano da essa. Si esegua il programma `driver` (`Driver` in Java) per un esempio. [3/30]

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.

Algoritmi

(a) Si consideri un grafo $G = (V, E)$ orientato *completo* (per ogni coppia di vertici $u, v \in V$, esistono entrambi gli archi *diretti* (u, v) e (v, u)) di n vertici. Com'è fatto l'albero di visita di una DFS applicata a G? Si descriva la sua struttura. [5/30]

(b) Dobbiamo realizzare una mappa (rispetto a chiavi *intere*). Una volta popolata con coppie (chiave, valore), la mappa dovrà gestire le seguenti operazioni (tra parentesi, la frequenza prevista per ciascuna operazione): i) `search(k)` (40% di tutte le operazioni); ii) `range_query(k1, k2)` (60% di tutte le operazioni). Sia n il numero di coppie presenti dopo il popolamento della struttura dati. Assumendo che, tipicamente, $k_2 - k_1 = O(\log n)$ nelle operazioni di range query, quale struttura dati usereste tra una tabella hash e un AVL? *Motivare la risposta.* [4/30]

(c) Mostrare come è possibile modificare un albero AVL in modo da implementare una coda di priorità con le stesse prestazioni (asintotiche) di un heap rispetto alle consuete operazioni di ricerca/estrazione del minimo, inserimento/cancellazione di una chiave. Non si richiede la descrizione degli algoritmi che implementano le operazioni menzionate sopra, ma soltanto di descrivere *chiaramente* come queste ultime (e la struttura dati sottostante) vanno modificate per conseguire l'obiettivo. [4/30]

BST: max-gap

Dato un BST t contenente *chiavi intere*, progettare un algoritmo (pseudo-codice) che determini e restituisca la massima differenza fra due chiavi adiacenti (rispetto all'ordinamento delle chiavi stesse). L'algoritmo *deve* avere un costo lineare rispetto al numero di chiavi presenti nel BST. Inoltre, l'algoritmo *deve* essere *in-place*, ossia non deve far uso di strutture dati di appoggio, a parte eventualmente una quantità di memoria ausiliaria pari

a $O(1)$ (tuttavia, saranno particolarmente apprezzate soluzioni che non fanno alcun uso di memoria ausiliaria). Quindi ad esempio, copiare tutte le chiavi in un array ausiliario per risolvere il problema non va bene.
Suggerimento: Per iniziare, si pensi a un semplice algoritmo lineare (che, come scritto sopra, non costituisce una soluzione accettabile) nel caso in cui sia possibile usare un array come struttura dati appoggio. [4/30]