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 straordinario del 16-4-2019 – 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.

Problema 1 Analisi algoritmo

Si considerino i metodi Java di seguito illustrati.

```
static void p(int i, int[] a) {
    int v = a[i];
    while(++i < a.length)
        if(v > a[i]) a[i-1]=a[i];
        else break;
    a[i-1] = v;
}
```

```
static int[] s(int[] a, int i) {
    if(i == a.length - 1) return a;
    p(i, s(a, i+1)); return a;
}

static int[] s(int[] a) {
    if(a.length > 0) return s(a, 0);
    else return a;
}
```

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 `s(int[])` in funzione di z, dimensione dell'input. **Non usare simboli che non sono definiti** (ad esempio: n non è definito). [4/30]

(b) Il metodo `s(int[])` descrive un celebre algoritmo. Quale? (Spiegare perché). [2/30]

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

In questo problema si fa riferimento a *grafi semplici*. 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 e una label (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) 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/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 label (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.

2.1 Realizzare una funzione/metodo `conn_comp` che, dato un grafo semplice $G = (V, E)$, calcola e restituisce il numero di componenti connesse di G. [3/30]

2.2 Realizzare una funzione/metodo `contract` che, dato un grafo semplice G, modifica il grafo *contraendone* tutti i vertici di grado 2. *Contrarre* significa eliminare il vertice e connettere direttamente i due vertici che erano ad esso collegato (e che non erano fra loro collegati). *Esempio:* v è un vertice di grado 2, ed è connesso a u e w dagli archi $e_1 = (u, v)$ ed $e_2 = (w, v)$. La *contrazione* è possibile se non esiste un arco (u, w). In tal caso, si cancellano v, e_1 ed e_2, e si inserisce un nuovo arco (u, w). [4/30]

2.3 Realizzare una funzione/metodo `count_k3` che, dato un grafo semplice G, contratto come descritto nel quesito precedente, calcola e restituisce il numero di sottografi completi di 3 nodi contenuti in G. [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.

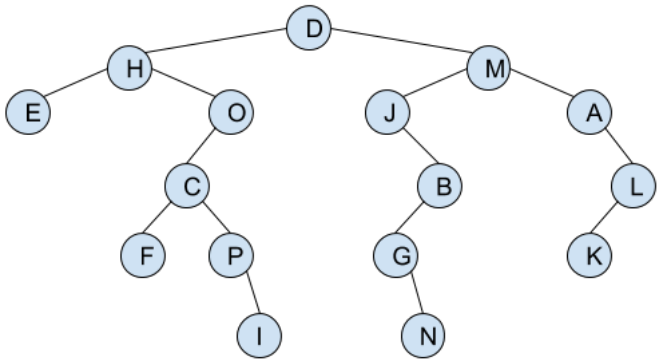
Per il superamento della prova al calcolatore è necessario conseguire almeno 5 punti. Si prega di non fare usi di package nel codice Java.

Problema 3 Algoritmi

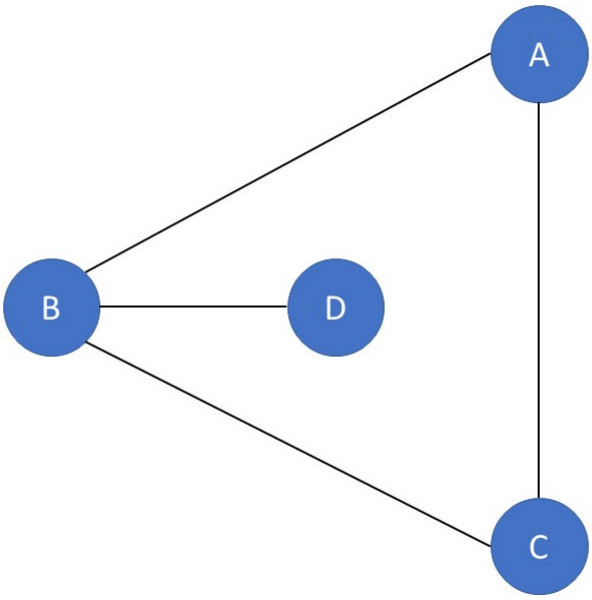
(a) Illustrare un algoritmo (codice o pseudo-codice) che, dato un array di interi, ne permuta i valori in modo da trasformarlo nella rappresentazione di un max-heap. [5/30]

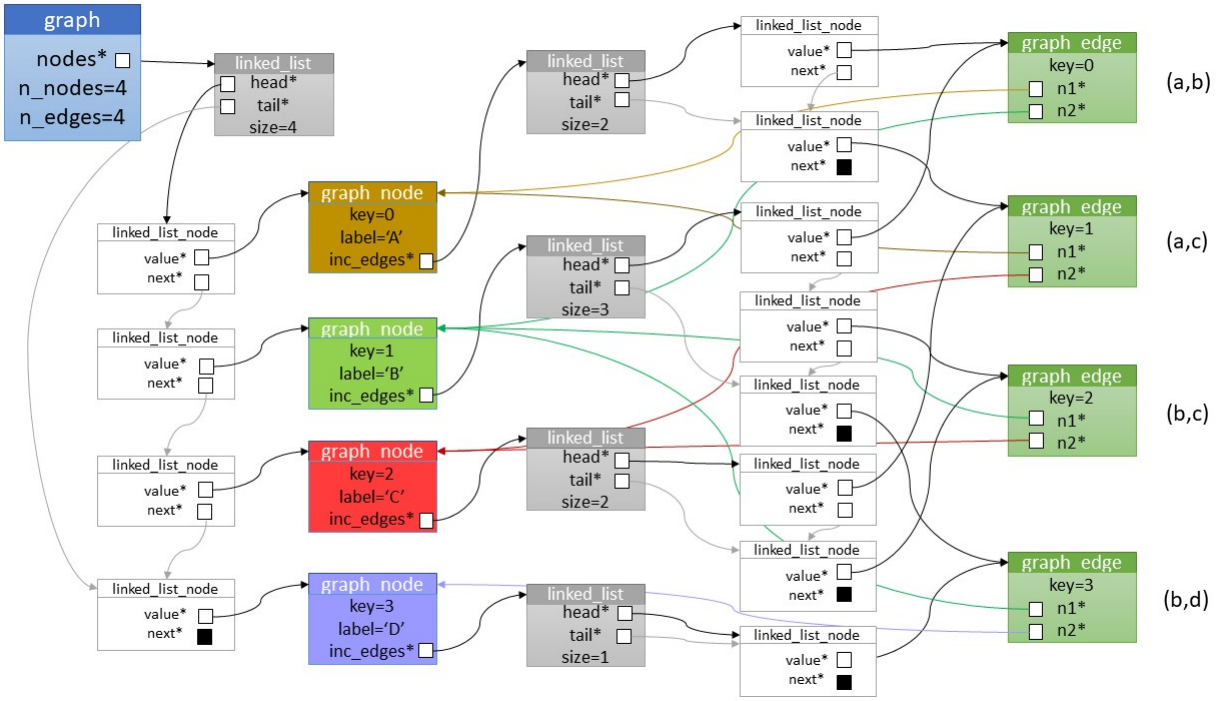
(b) Dato un albero binario t, progettare un algoritmo (codice o pseudo-codice) che determini se t è completo (tutti i suoi livelli hanno il massimo numero di nodi). [5/30]

(c) Con riferimento all'albero in figura, scrivere in quali sequenza vengono visitati i suoi vertici con: visita pre-order, visita in-order, visita post-order. [3/30]



(d) Definire gli alberi di Fibonacci (50%) e spiegare a cosa servono (50%). [4/30]


A
B
D
C


graph
nodes*
n_nodes=4
n_edges=4
linked_list
head*
tail*
size=4
linked_list_node
value*
next*
graph_node
key=0
label='A'
inc_edges*
key=1
label='B'
key=2
label='C'
key=3
label='D'
graph_edge
key=0
key=1
key=2
key=3
n1*
n2*
(a,b)
(a,c)
(b,c)
(b,d)
size=3
size=2
size=1