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 dell'8-1-2019 – a.a. 2018-19 – Tempo a disposizione: *4 ore* – somma punti: 34

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** (esempio: **non** `c.<00000000>`, ma `c.00000000`). *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.

```
static int ternsearch(int a[], int v, int start, int end) { // array a ordinato in senso non decrescente
    if (end - start < 2) return base(a, v, start, end);
    int onethird = (end - start)/3;
    int i = start + onethird;
    if(a[i] > v) return ternsearch(a, v, start, i-1);
    else if (a[i] == v) return i;
    else {
        int j = i + onethird;
        if(a[j] > v) return ternsearch(a, v, i+1, j-1);
        else if (a[j] == v) return j;
        else return ternsearch(a, v, j+1, end);
    }
}

static int base(int a[], int v, int start, int end) { // array a ordinato in senso non decrescente
    while(start <= end) {
        if(a[start] == v) return start;
        start++;
    }
    return -1;
}

static int ternsearch(int a[], int v) { // array a ordinato in senso non decrescente
    return ternsearch(a, v, 0, a.length-1);
}
```

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 `ternsearch(int[], int)` in funzione della dimensione dell'input. [4/30]

(b) Confrontare i costi temporali e spaziali di `ternsearch(int[], int)` con quelli di un'ordinaria ricerca binaria su un array ordinato. [2/30]

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

In questo problema si fa riferimento a *grafi diretti* 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 sono associate due liste collegate: una per gli archi entranti ed una per gli archi uscenti; inoltre ad ogni nodo sono associati un numero progressivo (a iniziare da 0) automaticamente assegnato dalla primitiva di inserimento nodo ed 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 uscenti ed entranti rispetto ad 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 `mat2list` che, data la matrice di adiacenze di un grafo diretto G, restituisce la rappresentazione per lista di incidenze dello stesso grafo G. Si rammenta che la matrice di adiacenze è una matrice quadrata in cui ogni riga ed ogni colonna rappresentano un nodo del grafo. La cella $\langle i, j \rangle$ ha valore 1 se esiste un arco che connette i nodi i e j (in direzione da i verso j), 0 altrimenti. Si consiglia di utilizzare la funzione/metodo ausiliaria `char_gen(int seed)` per la generazione di caratteri pseudo-casuali, da assegnare come label ai nodi che verranno creati. Il `Driver` relativo a questo esercizio legge la matrice da standard input seguendo la seguente formattazione:

- La prima riga contiene la dimensione n della matrice (quadrata);
- Seguono n righe, ciascuna contenente n interi in $\{0, 1\}$.

[2.5/30]

2.2 Realizzare una funzione/metodo `is_strongly_connected` che, dato un grafo diretto semplice G, stabilisce se G è *fortemente connesso*. [3.5/30]

2.3 Realizzare una funzione/metodo `is_closed` che, dato un grafo diretto semplice G, stabilisce se G è uguale alla sua *chiusura transitiva*. [4/30]

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) Con riferimento alle tavole hash con *linear probing*, impiegate per implementare una mappa, descrivere accuratamente la gestione delle operazioni di inserimento e di eliminazione di chiavi. [4/30]

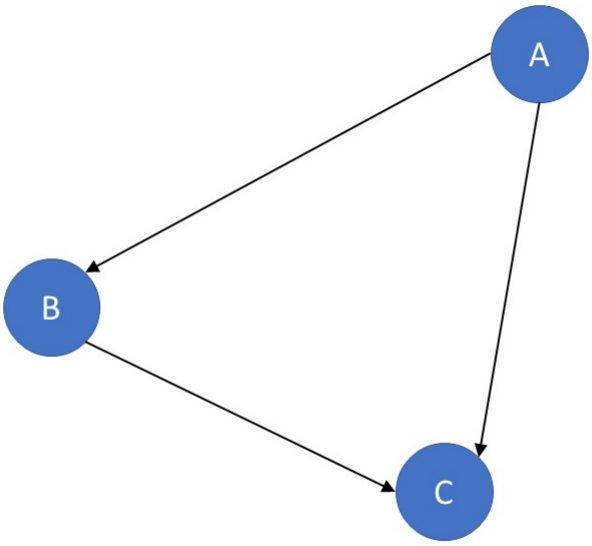
(b) Progettare un algoritmo (codice o pseudo-codice) che dati un insieme di n interi e un intero k ($1 \leq k \leq n$), determini i k interi più piccoli dell'insieme in tempo $o(n \log n)$, se $k \in o(n)$ (tempo asintoticamente migliore di $n \log n$ se k è asintoticamente più piccolo di n). [4/30]

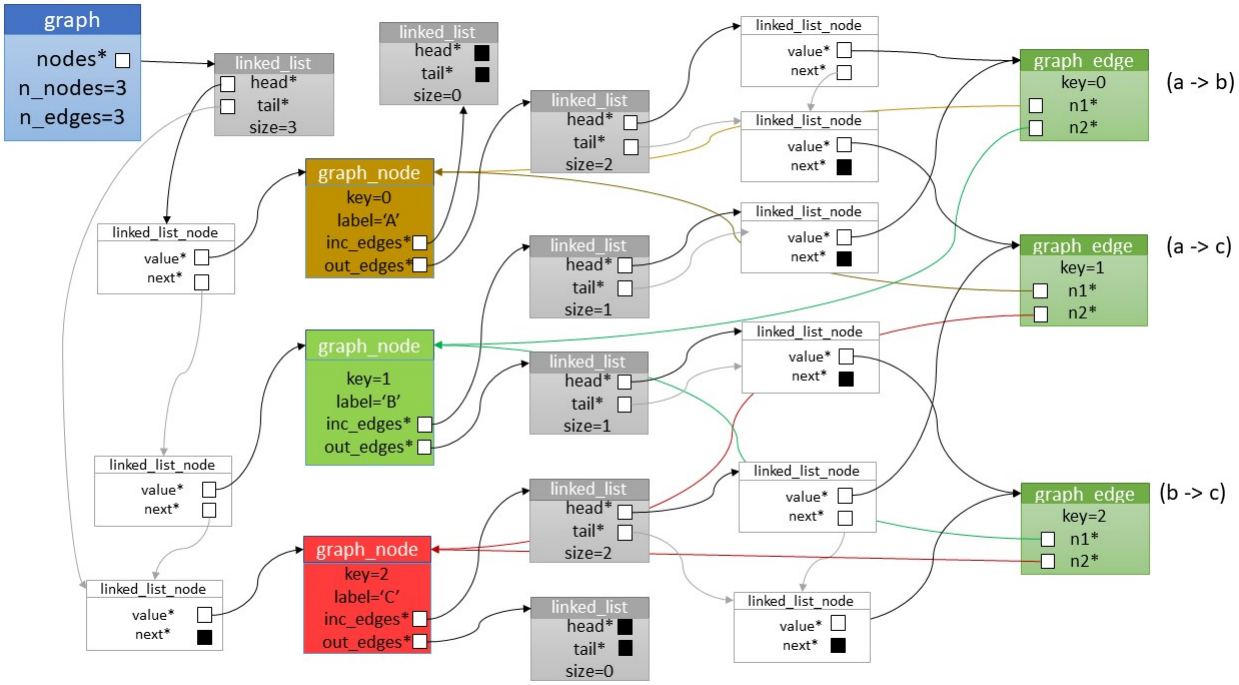
(c) Progettare un algoritmo (codice o pseudo-codice) che, dato un grafo semplice G, stabilisce se G è una *foresta*. [4/30]

Problema 4 Sottoalbero completo massimo

Dato un albero binario t, progettare un algoritmo (codice o pseudo-codice) che determini il numero di nodi del più grande sottoalbero completo presente in t.

Si richiamano per comodità le definizioni di albero binario completo e di sottoalbero. Un albero binario è detto *completo* se tutti i suoi nodi interni hanno arità (numero figli) due e tutti i rami hanno la stessa lunghezza. Dato un nodo v di un albero t (non necessariamente binario), il *sottoalbero di t radicato in v* è l'albero indotto da tutti i nodi di t che sono discendenti di v (incluso). [6/30]


A
B
C


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