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

Appello (straordinario) del 4-4-2018 – a.a. 2017-18
Tempo a disposizione: *4 ore* – somma punti: 36

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`;
- 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`;
- altri file `probl1.<matricola>.txt`, `probl3.<matricola>.txt`, `probl4.<matricola>.txt`, ecc. contenenti, rispettivamente, gli svolgimenti degli altri problemi; tali file vanno posti nella cartella `ESAME` e debbono avere estensione `.txt`.

È possibile consegnare tabelle, calcoli e disegni su materiale cartaceo integrativo, che verrà esaminato solo a condizione che risulti ben leggibile e sia referenziato all'interno dei file `probl*.txt` di cui sopra.

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 process(String s) {
  process("", s);
}
static void process(String prefix, String s) {
  int n = s.length();
  if (n == 0) System.out.println(prefix);
  else for (int i = 0; i < n; i++)
    process(prefix + s.charAt(i), s.substring(0, i) + s.substring(i+1, n));
}
```

Sviluppare, *argomentando adeguatamente* (il 50% del punteggio dell'esercizio sarà sulle argomentazioni addotte), quanto segue:[1]

(a) Determinare il costo temporale asintotico di caso peggiore dell'algoritmo descritto da `process(String)` in funzione della dimensione dell'input, assumendo costante il costo del metodo `substring`. [4/30]

(b) Come varia il costo del caso (a) se assumiamo che il costo di `substring(x, y)` sia proporzionale a `y - x`? [2/30]

[1] `public String substring(int beginIndex, int endIndex)`. Returns a new string that is a substring of this string. The substring begins at the specified `beginIndex` and extends to the character at index `endIndex` - 1. Thus the length of the substring is `endIndex - beginIndex`.

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

Con riferimento agli alberi binari di ricerca (BST), impiegati per realizzare una mappa ordinata con chiavi `int` *non negative*, risolvere al computer quanto segue, in Java o in C. Si impieghi la rappresentazione basata su classe/struttura `BST` e classe/struttura `BinNode`. Per le specifiche esatte di rappresentazione si vedano i file forniti a supporto.

- Realizzare una funzione/metodo `BST_insert` per inserire in un BST una nuova chiave; la funzione/metodo deve restituire un riferimento/puntatore al nodo appena inserito; se la chiave è già presente, non fare alcun inserimento e restituire un riferimento/puntatore al nodo che la contiene. [4/30]
- Realizzare una funzione/metodo `isBST` che, dato un albero, determini se questo è un BST. In particolare, l'output (di tipo `int`) deve essere così organizzato:

output	significato
-1 :	funzione non implementata
0 :	non è un BST
$+1$:	è un BST

[3/30]

- Realizzare una funzione/metodo `isBalanced` che, dato un albero, determini se questo è bilanciato. In particolare, l'output (di tipo `int`) deve essere così organizzato:

output	significato
-1 :	funzione non implementata
0 :	non è bilanciato
$+1$:	è bilanciato

[3/30]

È necessario implementare i metodi in `bst.c` o `Tree.java` identificati dal commento `/*DA IMPLEMENTARE*/`. In tali file è permesso sviluppare nuovi metodi se si ritiene necessario. *Non è assolutamente consentito modificare metodi e strutture già implementati.* È invece possibile modificare il file `driver` per poter effettuare ulteriori test.

Problema 3 Insiemi

Si descrivano opportuni algoritmi e opportune strutture dati concrete per realizzare efficientemente e con side effect le seguenti operazioni su insiemi, indicandone i costi. Si assuma che ogni insieme sia sottoinsieme di un insieme universo U di grande dimensione. Usare pseudo-codice.

(a) Operazioni insiemistiche classiche:

- makeset(e): dato $e \in U$, costruire e restituire $\{e\}$ (dopo l'operazione e è disponibile per eventuali ulteriori makeset).
- union(A, B): dati A e B, determinare e restituire $A \cup B$ (A e B non sono più disponibili dopo l'operazione).
- intersection(A, B): dati A e B, determinare e restituire $A \cap B$ (A e B non sono più disponibili dopo l'operazione).

[4/30]

(b) Operazioni fra insiemi *disgiunti*, la cui unione dà U (assumere che gli insiemi siano costruiti attraverso una opportuna sequenza di operazioni makeset):

- makeset(e): dato $e \in U$, costruire e restituire $\{e\}$ (dopo l'operazione e non è più disponibile per un ulteriore makeset).
- find(e): dato $e \in U$ determinare e restituire l'insieme che lo contiene (o un riferimento ad esso).
- union(A, B): dati A e B, determinare e restituire $A \cup B$ (A e B non sono più disponibili dopo l'operazione).

[4/30]

Problema 4 Rete stradale

La rete stradale di una grande regione è modellata attraverso un *grafo stradale*, ove i nodi rappresentano incroci fra due o più strade e gli archi rappresentano tratti stradali privi di incroci (che sono presenti solo alle loro estremità). Si assume per semplicità che per ogni coppia di incroci esista al più un tratto stradale che li collega e che non ci siano strade che, partendo da un incrocio, conducano all'incrocio stesso. Possono esistere strade a senso unico. Ciascun arco è pesato con la lunghezza del tratto stradale che rappresenta (reale positivo).

Il governo regionale intende dotarsi di uno strumento che gli consenta di individuare, a partire da un incrocio dato s, il tratto stradale più critico, definito come quel tratto che, in caso di blocco, incrementa maggiormente la distanza media fra s e gli altri incroci. Qualora questo tratto critico non sia unico occorre individuarli tutti.

Ciò premesso, si chiede di sviluppare quanto segue.

(a) Descrivere un algoritmo (pseudo-codice) che, preso in input un grafo stradale $G = (V, E)$ e un vertice $s \in V$, determini il valore medio della distanza fra s e gli altri vertici. In formule, indicando con $d(x, y)$ la distanza[2] fra x ed y, l'algoritmo deve determinare la quantità $\overline{d(s)}$ definita come:

$$\overline{d(s)} = \frac{\sum_{t \in V \setminus \{s\}} d(s,t)}{|V| - 1}$$

Valutare il costo dell'algoritmo. [6/30]
N.B. Lo pseudo-codice privo di indentazione riceve penalizzazione del 20%.

(b) Descrivere un algoritmo (pseudo-codice) che, preso in input un grafo stradale $G = (V, E)$ e un vertice $s \in V$, determini e restituisca l'arco $e \in E$ tale che, in caso di blocco della circolazione sul tratto stradale corrispondente ad e, il valore $\overline{d(s)}$ incrementi maggiormente. Nel caso e non sia unico, l'algoritmo deve determinare e restituire l'elenco di tutti tali archi. Valutare il costo dell'algoritmo. [6/30]
N.B. Lo pseudo-codice privo di indentazione riceve penalizzazione del 20%.

[2] È opportuno rammentare che per *distanza* si intende la lunghezza del percorso più breve.