Cognome e Nome: Numero di Matricola:

Docente:

Spazio riservato alla correzione

1	2	3	4	5	6	totale
/15	/15	/15	/15	/12	/18	/90

Non usare altri fogli, usare solo lo spazio sottostante. Fogli differenti da questo non saranno presi in considerazione per la correzione.

1. Sia f un flusso in una rete di flusso G con sorgente s e destinazione t. Dimostrare che G_f non contiene augmenting path allora $|f| = c(S,T)$ per un taglio (S,T).

2. Si supponga che un grafo G abbia la seguente lista delle adiacenze

   1 – (2,3,4)
   2 – (1,3.4)
   3 – (1,2,4)
   4 – (1,2,3,6)
   5 – (6,7,8)
   6 – (4,5,7)
   7 – (5,6,8)
   8 – (5,7)

   a. Disegnare il grafo
   b. Fornire l'albero BFS ottenuto applicando la visita BSF a partire dal vertice 1
   c. Fornire la foresta ottenuta applicando la visita DFS a partire dal vertice 1 (per ogni vertice indicare il valore del campo d ed il campo f)

3. Aggiungere alla classe **LinkedTree** il metodo **Position find(Object e)** che restituisce in output la posizione di un nodo dell'albero il cui elemento è uguale ad **e**. Se nessun nodo contiene **e**, allora il metodo deve lanciare l'eccezione **ElementNotFound.** Per l'implementazione non è possibile utilizzare i metodi **elements()** e **positions().** Quale è la complessità del metodo proposto? Giustificare la risposta.

4. Scrivere la funzione Java

   List merge(Comparator c, List L1,List L2)

   che riceve come argomenti due liste ordinate e restituisce una lista ordinata contenente gli elementi di entrambe le liste. I confronti tra gli elementi devono essere effettuati utilizzando il comparatore passato come argomento. Quale è la complessità di tempo della funzione proposta? Giustificare la risposta.

5. Scrivere una funzione Java **PQ-Sort** che ricevuti in input una sequenza ed un comparatore restituisce in output una sequenza ordinata. La funzione **PQ-Sort**, per l'ordinamento, deve far uso di una coda a priorità. Quale è la complessità di tempo della funzione? Giustificare la risposta.

6. Uno **SmartStack** è un contenitore che memorizza oggetti di tipo **Item** definiti come segue:

```
public class Item {
        private int numItem;
        private String itemName;

        public Item(String str)  {itemName=str; numItem=1;}
        public upNumber()      { numItem++; }
        public downNumber()  { numItem--; }
        public getNumber()     { return numItem; }
        public getItem()         { return itemName; }
}
```

Implementare l'interfaccia

```
public interface SmartStack {
        public int size();
        public void smartPush (String element);
        public String smartPop() throws SmartStackEmptyException;
}
```

tenendo presente il seguente funzionamento dei metodi

smartPush (String element) – se lo **SmartStack** contiene un oggetto **Item** il cui **itemName** è uguale alla stringa **element**, allora per quello oggetto viene incrementato di uno il valore della variabile **numItem**. Altrimenti, viene inserito al top dello **SmartStack** un nuovo oggetto **Item** contenente la stringa **element** con **numItem** settato ad 1.

String smartPop() – restituisce la stringa **itemName** contenuta nell'oggetto **Item** al top dello **SmartStack**. Il valore **numItem** corrispondente viene decrementato di uno. Se tale valore diventa nullo, allora rimuove l'oggetto **Item** dal top dello **SmartStack**.

size() – restituisce il numero degli elementi presenti nello **SmartStack** contando ogni elemento **numItem** volte.