Problema 1 Analisi algoritmo

Si considerino i metodi Java di seguito illustrati.

```
// c.length > k >= 0
static double horner(double c[], int k, double x) {
    double r = c[k];
    if(k < c.length - 1)
        return x*horner(c, k+1, x) + r;
    return r;
}

static double horner(double c[], double x) {
    return horner(c, 0, x);
}
```

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

a)
L' algoritmo horner(double[], double) consiste in una chiamata all'alg

horner(double[], int, double), ed il suo costo temporale asintotico

é dato dal costo di quest'ultimo.horner(double[], int, double) ha lo stesso input del primo alg, più un intero k

compreso fra 0 e la dimensione dell'array in input. L' alg consta di una

assegnazione, di un confronto, e di una chiamata ricorsiva, che ne è

l'istruzione dominante. La chiamata verrà eseguita al più n= c.lenght() volte,

nel caso in cui il valore iniziale di k sia 0 esattamente c.lenght() volte

(cioè la dimensione dell'array in input), pertanto il costo asintotico

temporale di horner(double[], double) sarà Theta(n), vale a dire che il costo

cresce linearmente con la dimensione dell'input.

(b) Definire il significato di *ricorsione di coda* e spiegare se `horner(double[], int, double)` presenta una ricorsione di coda oppure no. [2/30]

Per ricorsione di coda si intende quel tipo di ricorsione lineare ove la chiamata ricorsiva costituisce l'ultima istruzione eseguita da un algoritmo.

In questi casi è spesso possibile eliminare la ricorsione con un semplice ciclo.

horner(double[], int, double) presente una ricorsione di coda in quanto la chiamata ricorsiva costituisce l'ultima istruzione eseguita, di fatto quasi l'unica, eccetto naturalmente per il caso base.

3 ALGORITMI

(a) Per il problema $\mathcal{A}$ sono noti un algoritmo di costo $\Theta(n^2)$, un secondo di costo $\Theta(n^2 \log n)$ e un lower bound $\Omega(n \log n)$. Poiché molti ricercatori stanno ancora studiando il problema è lecito attendersi la scoperta di nuovi e più efficienti algoritmi. Alla luce delle informazioni riportate, qual è il limite inferiore all'upper bound di un nuovo formidabile algoritmo? Discutere dettagliatamente. [4/30]

a)

Per quanto è dato sapere il miglior upper bound ottenibile risulta essere omega(nlog(n)) se non fosse così infatti verrebbe meno quanto detto in ipotesi, ovvero che il problema è Ω(n log n).

(b) Pare che nel 2050 saranno disponibili nuovi processori e nuove architetture di calcolo in cui i confronti fra valori avranno costo nullo, mentre le assegnazioni di valori atomici continueranno ad avere costo costante. Quale algoritmo di ordinamento diverrebbe in quel caso preferibile? Discutere in dettaglio.[1] [5/30]

b)

Dato che i confronti tra valori avranno costo nullo conviene utilizzare un algoritmo di sorting semplice e robusto come selection sort, infatti questo effettua molti confronti ma sposta ogni valore della sequenza sola volta (N-1 scambi).

l'algoritmo di selection sort si basa su una ricerca del valore massimo all'interno della sequenza da ordinare, partendo dal primo elemento si cerca nella restante sequenza il massimo e lo si scambia, si procede così fino a terminare la sequenza che risulterà ordinata.

(c) Ad Alice viene posto il seguente problema: due grafi orientati rappresentati tramite liste di adiacenza vengono forniti in input a un algoritmo: come può questo stabilire se i due grafi sono lo stesso grafo? Alice è conscia del fatto che per lo stesso grafo esistono molte rappresentazioni per lista di adiacenza che differiscono per l'ordine con cui gli adiacenti di un nodo compaiono in ogni lista; ciascun nodo ha un campo informativo `key`, intero positivo, che univocamente identifica un nodo del grafo. Puoi aiutarla? [4/30]

c)

Partendo dal presupposto che due grafi uguali contengono nodi uguali, ogni nodo avrà la stessa chiave key.

Una possibile soluzione consiste nel:

1) Per ogni nodo trovare il corrispettivo dell'altro grafo (tramite chiave key).
2) eseguire una BFS* semplificata per ognuno di quei nodi in cui, ad ogni livello si controlla la cardinalità delle code di priorità e gli elementi che essa contiene, se questi differiscono il grafo non è lo stesso.

(Essenzialmente nella coda di priorità che verra creata per ognuno dei due nodi ad ogni livello avrò i nodi connessi a quello esaminato, se le due code differiscono il grafo non sarà uguale in quanto diretto.)

Ovviamente si può ammortizzare il costo medio con controlli che riguardano la cardinalità di nodi e archi all'interno del grafo (se un grafo ha n vertici e n archi anche l'altro li dovrà avere).

*

BFS:

è una visita in ampiezza del grafo, in cui si costruiscono sequenze che contengono i nodi adiacenti appartenenti al nodo esaminato.

Problema 4 Intersezione di due dizionari

Due dizionari ordinati basati su chiavi intere sono gestiti attraverso due BST. Si richiede di scrivere un algoritmo (codice o pseudo-codice) che, dati due BST del tipo descritto, determina l'intersezione dei due dizionari, restituendone le chiavi in una lista fornita in output, ove le chiavi sono ordinate e presenti senza ripetizioni. [6/30]

Algorithm intersetion

Input: due BST che rappresentano due dizionari

Output: una lista con le chiavi dei due dizionari ordinate e senza ripetizioni

```
l <- empty list
l1 <- empty list
l2 <- empty list
if t1.root == null and t2.root == null
    return l
else if t1.root != null and t2.root == null
    l <- inOrder(t1)
    return l
else if t1.root == null and t2.root != null
    l <- inOrder(t2)
    return l
else
    l1 <- inOrder(t1)
    l2 <- inOrder(t2)
    l <- merge(l1, l2)
```

Algorithm inOrder

Input: un albero BST e un lista l vuota

Output: lista contenente i nodi del BST visitati con visita in-ordine

```
node <- t.root
if node.hasLeft()
        inOrder(node.left(), l)
 l.add(node.getKey())
if node.hasRight()
        inOrder(node.right(), l)
return l
```

esercizio 2.1

```
/*
	 * Data una lista contenente un elenco di Centrali, ritorna un puntatore al grafo completo avente:
	 * - un nodo per ogni Centrale
	 * - un arco per ogni coppia di Centrali (x,y) distinte
	 */
	public static Graph getCompleteGraph(LinkedList<Centrale> l1) {
		Graph graph = new Graph();
		for(Centrale c1 : l1) {
			graph.addNode(c1);
		}
		for(GraphNode n1 : graph.getNodes()) {
			Centrale c1 = (Centrale)n1.object;
			for(GraphNode n2 : graph.getNodes()) {
				if(n1.equals(n2)) {
					continue;
				}
				Centrale c2 = (Centrale)n2.object;
				graph.addEdge(n1, n2, graph.INF);
			}
		}
		return graph;
```

```
    }
```

2.2

```
public static void setWeightsInGraph(Graph g) {
        if(g==null)
            return;
        double[] min=new double[3];
        double app;
        GraphNode[] mingn= new GraphNode[3];
        for(int i=0; i<3; i++)
        {
            min[i]=Graph.INF;
            mingn[i]=null;
        }
        Iterator<GraphNode> it1=g.getNodes().iterator();
        Iterator<GraphNode> it2;
        GraphNode gn1;
        GraphNode gn2;
        while(it1.hasNext())
        {
            gn1=it1.next();
            it2=g.getNodes().iterator();
            while(it2.hasNext())
            {
                gn2=it2.next();
                app=dist((Centrale)gn1.object, (Centrale)gn2.object);
                if(gn1.equals(gn2)!=true && app!=0)
                {
                    for(int i=0; i<3; i++)
                        if(app<min[i])
                            for(i=i; i<3; i++)
                            {
                                double app2;
                                GraphNode appgn;
                                app2=min[i];
                                min[i]=app;
                                app=app2;
                                appgn=mingn[i];
                                mingn[i]=gn2;
                                gn2=appgn;
                            }
                }
            }
            for(int i=0;i<3;i++)
            {
                Iterator<GraphEdge> ige=g.getIncidentEdges(gn1).iterator();
                //ge.setWeight(min[i]);
                GraphEdge ge;
                while(ige.hasNext())
```

```
                {
                    ge=ige.next();

if((Centrale)ge.getEdgeOpposite(gn1).object==(Centrale)mingn[i].object)
                        ge.setWeight(min[i]);
                }
                min[i]=Graph.INF;
                mingn[i]=null;
            }
        }
        }
```

2.3

```
/*
        * Dato un grafo, ritorna una lista degli archi che minimizzano la
quantita' di cavo per connettere tutte le Centrali di g.
        */
        public static LinkedList<GraphEdge> getMinWire(Graph g) {
                int[] dimSets = new int[g.getNodes().size()];
                MinHeap<GraphEdge> minHeap = new MinHeap<GraphEdge>();
                LinkedList<GraphNode> listOfNodes = new
LinkedList<GraphNode>();
                int i = 0;
                for(GraphNode node : g.getNodes()) {
                        dimSets[i] = 1;
                        node.key = i;
                        i++;
                        listOfNodes.add(node);
                        for(GraphEdge edge : node.incidentEdges) {
                                minHeap.insert(edge.weight, edge);
                        }
                }
                LinkedList<GraphEdge> edges = new LinkedList<GraphEdge>();
                while(!minHeap.isEmpty()) {
                        GraphEdge edge = minHeap.removeMin().getValue();
                        if(edge.n1.key != edge.n2.key) {
                                edges.add(edge);
                                int numOfSet1 = edge.n1.key;
                                int numOfSet2 = edge.n2.key;
                                if(dimSets[numOfSet1] <= dimSets[numOfSet2]) {
                                        for(GraphNode n : g.getNodes()) {
                                                if(n.key == numOfSet1) {
                                                        n.key = numOfSet2;
                                                }
                                        }
                                        dimSets[numOfSet2] += numOfSet1;
                                }else {
                                        for(GraphNode n : g.getNodes()) {
                                                if(n.key == numOfSet2) {
                                                        n.key = numOfSet1;
```

```
                                                            }
                                                    }
                                                    dimSets[numOfSet1] += numOfSet2;
                                            }
                                    }
                            }
                            return edges;
                    }
}

class Centrale {
        String name;
        double x;
        double y;

        public Centrale(String name, double x, double y) {
                super();
                this.name = name;
                this.x = x;
                this.y = y;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public double getX() {
                return x;
        }

        public void setX(double x) {
                this.x = x;
        }

        public double getY() {
                return y;
        }

        public void setY(double y) {
                this.y = y;
        }

        @Override
        public String toString() {
                return name;
        }
}
```