Fundação CECIERJ - Vice Presidência de Educação Superior a Distância

Curso de Tecnologia em Sistemas de Computação
Disciplina: Programação III
AP3 2° semestre de 2014.

Nome –

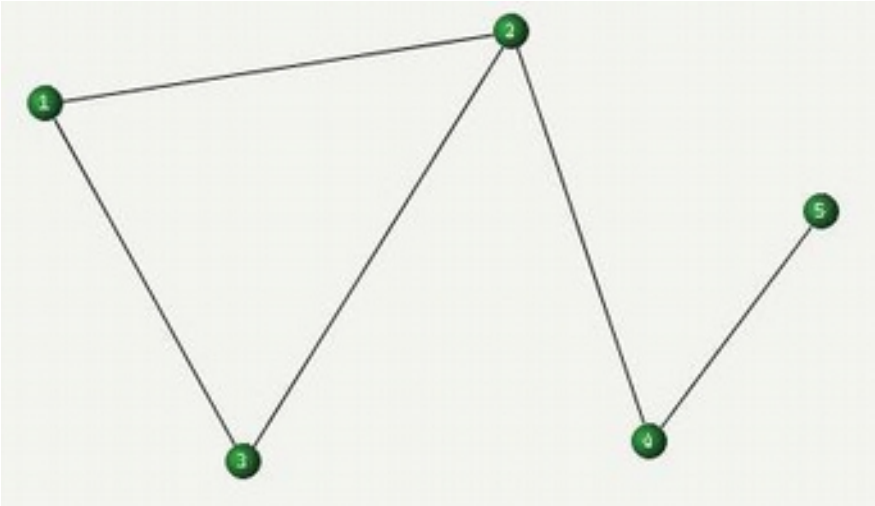
Assinatura –

Observações:

1. Prova sem consulta e sem uso de máquina de calcular.
2. Use caneta para preencher o seu nome e assinar nas folhas de questões e nas folhas de respostas.
3. Você pode usar lápis para responder as questões.
4. Ao final da prova devolva as folhas de questões e as de respostas.
5. Todas as respostas devem ser transcritas nas folhas de respostas. As respostas nas folhas de questões não serão corrigidas.

Questão 1) (5.0 pontos)

Escreva um programa que leia **SOMENTE UMA VEZ** de um arquivo texto, cujo o nome é informado como parâmetro de entrada, um grafo e informe, na tela, se o grafo é conexo ou não. Um grafo *G* é conexo se há pelo menos uma aresta ligando cada par de vértices deste grafo *G*. A figura abaixo mostra um exemplo deste tipo de grafo, onde **{1, 2, 3, 4, 5}** é o conjunto de vértices:



O modelo do arquivo de entrada para o exemplo supracitado é o seguinte:

1 2
1 3
2 3
2 4
4 5

Para o exemplo acima, a impressão informará que o grafo é conexo.

LEMBRE-SE: SEU PROGRAMA DEVE EXECUTAR COM QUAISQUER GRAFOS INFORMADOS COMO PARÂMETRO DE ENTRADA. SE O SEU PROGRAMA RESOLVER SOMENTE O PROBLEMA DO GRAFO SUPRACITADO, SUA QUESTÃO SERÁ SEVERAMENTE DESCONTADA.

RESPOSTA:

```
import java.io.*;
import java.util.*;

/*
A estrutura Vizinho é composta de:
  - no vizinho
  - referência para o próximo vizinho
(para fazer o encadeamento de vizinhos)
Esta classe é formada de construtor  e
método toString. O último é usado para verificar se a
estrutura está sendo criada de maneira correta.
*/
class Vizinho{
  int no_viz;
  Vizinho prox;

  Vizinho(int c){
    no_viz = c;
    prox = null;
  }

  public String toString(){
    return no_viz + " ";
  }
```

```
}

/*
A estrutura Lista é composta de:
  - no de origem
  - a lista de seus vizinhos
  - o proximo no do grafo
Esta estrutura é montada para ser usada no grafo
(estrutura principal criada para resolver o problema).
Esta classe tem o método construtor, um método para testar se o
vizinho já está na lista, um método para inserir vizinho na
primeira posição da lista de vizinhos, e o método toString.
*/
class Lista{
  int no;
  Vizinho prox_viz;
  Lista prox_no;

  Lista(int c){
    no = c;
    prox_viz = null;
    prox_no = null;
  }

  Vizinho pertence(int no){
    Vizinho resp = prox_viz;
    while((resp != null) && (no != resp.no_viz))
      resp = resp.prox;
    return resp;
  }

  void ins_Viz(int c){
    Vizinho v = pertence(c);
    if(v != null) return;
    v = new Vizinho(c);
    v.prox = prox_viz;
    prox_viz = v;
  }

  public String toString(){
    String resp = no + ": \n";
    Vizinho p = prox_viz;
    while(p != null){
      resp += p.toString();
      p = p.prox;
    }
    return resp + "\n";
  }
}
```

```
/*
A estrutura Grafo é desenvolvida para resolver o problema. Ela é
composta da referência para o primeiro nó da lista. Tem os
seguintes métodos:
  - construtor;
  - para verificar se um no existe na lista de nos do grafo;
  - para inserir nos; e
  - toString.
*/
class Grafo{
  Lista prim;

  Grafo(){  prim = null; }

  Lista pertence(int no){
    Lista resp = prim;
    while((resp != null) && (no != resp.no)) resp = resp.prox_no;
    return resp;
  }

  void insere(int no1, int no2){
    Lista p = pertence(no1);
    if(p == null){
      p = new Lista(no1);
      p.prox_no = prim;
      prim = p;
    }
    p.ins_Viz(no2);
    Lista q = pertence(no2);
    if(q == null){
      q = new Lista(no2);
      q.prox_no = prim;
      prim = q;
    }
    q.ins_Viz(no1);
  }

  public String toString(){
    String resp = "";
    Lista p = prim;
    while(p != null){
      resp += p.toString();
      p = p.prox_no;
    }
    return resp;
  }
}
```

```
public class Q1_AP3_2014_2{
  public static void main(String[] args) throws IOException{
    BufferedReader in;
    in = new BufferedReader(new FileReader(args[0]));
    try {
      Grafo g = new Grafo();
      String s, vs[];
      //leitura do arquivo de entrada uma única vez
      while((s = in.readLine()) != null){
       vs = s.split(" ");
       g.insere(Integer.parseInt(vs[0]),Integer.parseInt(vs[1]));
      }
      in.close();
      System.out.println(g);
      System.out.println(estaConectado(g));
    }
    catch (Exception e){
      System.out.println("Excecao\n");
    }
  }

  //método que conta a quantidade de nos do grafo
  static int conta (Grafo g){
    int resp = 0;
    Lista p = g.prim;
    while(p != null){
      resp++;
      p = p.prox_no;
    }
    return resp;
  }

  //método que verifica se, de um nó, é possível chegar
  //aos demais do grafo. É o método principal para resolver
  //o problema.
  static boolean estaConectado(Grafo g){
    //vet é o vetor que indica quais nós já foram alcançados.
    //Inicialmente, vet não possui um único nó.
    int i, n = conta(g), vet[] = new int[n];
    for(i = 0; i < n; i++) vet[i] = 0;

    //usada para enfileirar todos os nós que ainda precisam
    //ser testados.
    Deque<Lista> nos = new ArrayDeque<Lista>();

    //adiciono o primeiro nó do grafo.
    nos.addLast(g.prim);
```

```
    //enquanto a fila não está vazia
    while(nos.size() != 0){
      //removo o primeiro nó alcançável de g.
      //Inicialmente, ele é o apontado por prim.
      Lista aux = nos.remove();

      //indico que o nó foi alcançado.
      if(vet[aux.no - 1] == 0) vet[aux.no - 1] = 1;

      //busco este nó no grafo para incluir seus vizinhos
      //na fila, se eles não foram alcançados ainda.
      aux = g.pertence(aux.no);
      Vizinho viz = aux.prox_viz;
      while(viz != null){
        if(vet[viz.no_viz - 1] == 0)
          nos.addLast(new Lista(viz.no_viz));
        viz = viz.prox;
      }
    }

    //se a fila está vazia e existe uma posição do vetor que
    //não foi alcançada, isto indica que o grafo não
    //é conectado.
    for(i = 0; i < n; i++)
      if(vet[i] == 0) return false;
    return true;
  }
}
```

Questão 2) (5.0 pontos)

Suponha a classe Transporte abaixo, a qual é definida num sistema de uma transportadora e descreve, de forma geral, os veículos existentes numa transportadora.

```
class Transporte {
	int capacidade;
	double peso;
	List <Carga> carregamento;
	public Transporte(int capacidade, List<Carga> carregamento) {
		this.capacidade = capacidade;
		this.carregamento = carregamento;
	}
	public double cargaTotal() {
		int carga = 0;
		for (Carga c : this.carregamento)
			carga += c.volume();
		return carga;
	}
}
```

a) Defina a classe Carga, a qual representa os objetos que serão transportados. Para este sistema, basta saber as dimensões dos objetos, ou seja, largura, altura e profundidade. O volume é calculado pela multiplicação destas quantidades.
b) Os veículos existentes na transportadora são de 2 tipos: caminhões e picapes. Para os caminhões, o sistema necessita armazenar se eles são fechados (baú), se possuem cama (para viagens longas) e o números de eixos (para eventual cálculo de pedágio). Para as picapes, basta saber se ela é de cabine dupla, a qual permite o transporte de mais pessoas. Defina estes tipos.
c) Suponha que haja, se é que não há, uma regra no código de trânsito que limita a carga de um veículo de transporte relacionado ao seu número de eixos. Por exemplo, se um veículo tem peso total (peso + carga) de 2000 kg e 2 eixos, a razão peso/eixo é de 1000 (2000/2). Considerando que as picapes sempre possuem 2 eixos, obrigue que cada classe do tipo Transporte defina um método para o cálculo dessa razão.

Obs.: Nas respostas, considere conceitos de OO sempre que possível.

RESPOSTA:

```
package br.cederj.comp.ano2014;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Caminhao extends Transporte {
	boolean bau;
	boolean cama;
	int numEixos;

	public Caminhao(int capacidade, double peso, int numEixos, List<Carga> carregamento, boolean
bau, boolean cama) {
		super(capacidade, peso, carregamento);
		this.numEixos = numEixos;
		this.bau = bau;
		this.cama = cama;
	}

	public double razao() {
		return (this.cargaTotal() + peso) / this.numEixos;
	}
}

class Picape extends Transporte {
	boolean cabineDupla;

	public Picape(int capacidade, double peso, List<Carga> carregamento, boolean cabineDupla) {
		super(capacidade, peso, carregamento);
		this.cabineDupla = cabineDupla;
	}

	public double razao() {
		return (this.cargaTotal() + peso) / 2;
	}
}
```

```
class Carga {
	double altura, largura, profundidade;

	public Carga(double altura, double largura, double profundidade) {
		this.altura = altura;
		this.largura = largura;
		this.profundidade = profundidade;
	}

	public double volume () {
		return altura * largura * profundidade;
	}
}

abstract class Transporte {
	int capacidade;
	double peso;
	List <Carga> carregamento;

	public Transporte(int capacidade, double peso, List<Carga> carregamento) {
		this.capacidade = capacidade;
		this.peso = peso;
		this.carregamento = carregamento;
	}

	public double cargaTotal() {
		int carga = 0;
		for (Carga c : this.carregamento)
			carga += c.volume();
		return carga;
	}

	public abstract double razao();
}

// Apenas a título de ilustração (NÃO ERA REQUERIDO NA QUESTÃO)

public class AP3_2014_2_Q1 {
	public static void main(String[] args) {
		List <Transporte> frota = new ArrayList<Transporte>();
		List <Carga> cargasCaminhao = Arrays.asList(new Carga(10, 10, 10), new Carga(15,
15, 15));
		frota.add(new Caminhao (100, 300, 3, cargasCaminhao, true, false));
		List <Carga> cargasPicape = Arrays.asList(new Carga(5, 5, 5));
		frota.add(new Picape (100, 100, cargasPicape, true));
		double somaCargas = 0;
		for (Transporte t : frota)
			if (t.razao() > 500)
				somaCargas += t.cargaTotal();
		System.out.println("Volume total: " + somaCargas);
	}
}
```