

## Question 2

Not yet answered

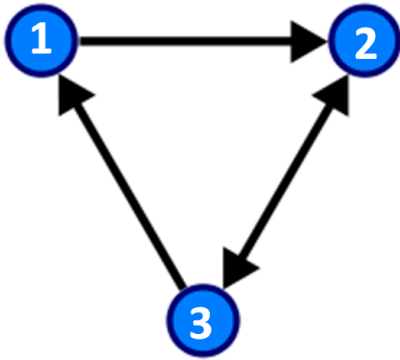
Marked out of 25.00

v1 (latest)

### Part 2: Java

This question relies on the mathematical concept of **graphs**. If you are not clear on the details, the following is a summary of the properties that will be needed.

#### Graphs



In graph theory, a **graph** is a collection of **nodes** and **edges**. The nodes can be thought of as objects, and every edge connects exactly two nodes. If there is an edge between two nodes, they can be thought of as related – for example, graphs are used to represent concepts such as transit connections or social media friendships.

We will consider only **directed graphs**, which are a specific type of graphs where each edge has a direction. That is, for every edge in a directed graph, one of the nodes is the starting point, and the other is the end point. For example, the image to the left represents a directed graph consisting of three nodes (labelled 1, 2, and 3) and four edges: (1,2), (2,3), (3,2), and (3,1) (the double-headed arrow represents an edge in both directions between nodes 2 and 3).

A directed graph can be represented as an **adjacency list**: this representation stores, for each node, the list of nodes that it is connected to. For example, the above directed graph can be represented by the following adjacency list:

- 1 -> (2)
- 2 -> (3)
- 3 -> (1, 2)

**Important note:** all classes created in this part should be put in the *graphs* package

#### Java Task 2a: Node class (6 marks)

The first task is to define a class that can be used to store a node in a directed graph represented as an adjacency list. A Node should have a label (an integer), as well as a list of its neighbours. The internal details of this class are up to you, but the following is the required behaviour.

This class should have a single constructor with the following signature:

- **public Node (int label)**

It should also have the following public methods:

- **public void addNeighbour (Node node)**
  - o Adds the given node to this node's adjacency list
- **public List<Node> getNeighbours()**
  - o Returns all of the nodes in the current adjacency list
- **public int getLabel()**
  - o Returns this node's label (as set in the constructor)

Your **Node** class should also override the following methods:

- **toString():** should produce a string consisting only of the node label
- **equals()** and **hashCode():** should determine equality based on the node label

#### Java Task 2b: GraphParser class (8 marks)

This is a class that has one purpose: to read the specification of a graph from a string, and to return an array of **Node** objects corresponding to that graph.

The string format is as follows:

- First line: the number of nodes in the graph
- Second line: the number of edges in the graph

- Each remaining line specifies a single edge in the graph, as a space-separated pair of numbers

You can assume that the nodes in the graph are numbered from 1 to  $n$ , where  $n$  is the number of nodes.

For example, the input for the sample graph above would be the following string (including newlines):

```
3
4
1 2
2 3
3 2
3 1
```

Your **GraphParser** class should have one **static** method, with the following signature:

- **public static Node[] parseGraph (String spec)**

This method should process the indicated string and should create and return an array of **Node** objects corresponding to the specified graph. The returned array should contain the nodes in order – that is, **nodes[0]** should be the node with ID 1, **nodes[1]** should contain the node with ID 2, and so on.

You can assume that the string contents will be correct – you do not need to do any error checking within your **parseGraph** method.

You are free to add any **private** helper methods that you want (although this is not required), but the above should be the only **public** method in the **GraphParser** class.

Hint 1: the method **String.split()** will likely be very useful here. If **str** is a String, then **str.split(*delim*)** splits the string on ***delim*** and returns a **String[]** corresponding to the split elements. In particular:

- **str.split("\n")** splits a string into its individual lines
- **str.split(" ")** splits a string into the components separated by spaces

Hint 2: if **str** is a String containing an integer (e.g., "3"), the method **Integer.valueOf(str)** returns the corresponding **int** (e.g., 3)

## Java task 2c: Edge class (5 marks)

Next, you should define a class to represent a **directed edge** in a graph. An edge consists of a pair of nodes, one representing the starting point and one representing the end point. You should define the class, including appropriate data types and access modifiers. You should also include a constructor that initialises all fields, appropriate **equals()** and **hashCode()** methods, as well as a **toString()** method that produces a human-readable version of the edge in the format **(1, 2)**.

## Java task 2d: GraphExplorer class (5 marks)

This class must provide a single **static** method, with the following signature:

- **public static Set<Edge> listEdges (Node[] nodes)**





This method should compute and return the complete set of edges in the given graph, using the **Edge** class you defined in Task 2c.


For the sample graph, this would return a set containing four edges: (1,2), (2,3), (3,2), and (3,1).

## Java submission

Be sure that your classes are all in the **graphs** package and that the files are named appropriately: **Node.java**, **GraphParser.java**, **Edge.java**, **GraphExplorer.java**. Submit the final version of each file on Moodle.

Maximum size for new files: 100 MB

 [Files](#)   



You can drag and drop files here to add them.

Accepted file types

Text file .java