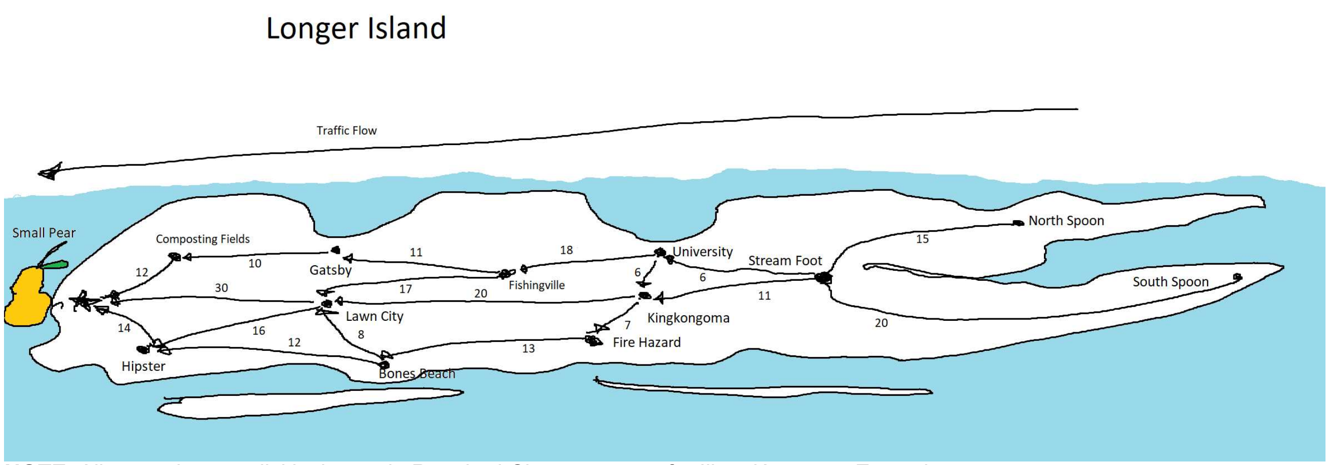
HOMEWORK 7 - due Tuesday, May 3rd no later than 7:00pm

REMINDERS:

- **Be sure your code follows the coding style for**
- **Make sure you read the warnings about academic dishonesty.** ***Remember, all work you submit for homework assignments MUST be entirely your own work. Also, group efforts are not allowed.***
- **Login to your grading account and click "Submit Assignment" to upload and submit your assignment.**
- **You are allowed to use any built-in Java API Data Structure classes to implement this assignment except where noted.**
- **Do not submit the Jar file along with your Java files. Otherwise, 10 points will be deducted.**
- **You may use Scanner, InputStreamReader, or any other class that you wish for keyboard input.**

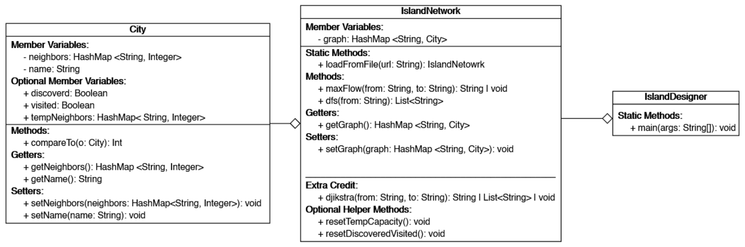
This is my last 214 hw, so at this point I don't mind being honest and saying that Long Island traffic is a pretty terrible thing. Therefore, as your local omniscient narrator, I have decided to start over again, and I have created the Longer Island. I'd like to see how terrible traffic on my island will be, so I have hired you to write a simulation to make sure my island works. Because I am not a benevolent narrator, I'm just interested in people getting to work (they can figure out their own way home). Generally on Longer Island, people head toward the local city, the Small Pear in order to go to work. All the major highways are one way (toward the Small Pear in the morning, and away in the evening), conveniently forming a DAG (Directed Acyclic Graph). I'm interested in answering two questions mainly: where can a person get from a given city (a question that can be answered with a Depth First Search from a given city) and also what is the maximum amount of flows between any two cities (I will give you an algorithm for this as well).
I would like you to make a program which can calculate these for me. Since I'm not sure I'm a terribly good island designer, I'd like to be able to load other designs at a later date, and also load islands other people on the internet have designed. Therefore, you will use a library to load files from the internet (it also works with files from your local machine), so I can try other island configurations.
Your input will be a list of cities, and a list of roads from various cities, with their carrying capacities in terms of car per minute.



NOTE: All exceptions explicitly thrown in Required Classes except for IllegalArgumentException are custom exceptions that need to be made by you.

UML

The UML Diagram for all the classes specified below is as follows:


City
Member Variables:
- neighbors: HashMap <String, Integer>
- name: String
Optional Member Variables:
+ discoverd: Boolean
+ visited: Boolean
+ tempNeighbors: HashMap< String, Integer>
Methods:
+ compareTo(o: City): Int
Getters:
+ getNeighbors(): HashMap <String, Integer>
+ getName(): String
Setters:
+ setNeighbors(neighbors: HashMap<String, Integer>): void
+ setName(name: String): void
IslandNetwork
Member Variables:
- graph: HashMap <String, City>
Static Methods:
+ loadFromFile(url: String): IslandNetowrk
Methods:
+ maxFlow(from: String, to: String): String I void
+ dfs(from: String): List<String>
Getters:
+ getGraph(): HashMap <String, City>
Setters:
+ setGraph(graph: HashMap <String, City>): void
Extra Credit:
+ djikstra(from: String, to: String): String I List<String> I void
Optional Helper Methods:
+ resetTempCapacity(): void
+ resetDiscoveredVisited(): void
IslandDesigner
Static Methods:
+ main(args: String[]): void

Required Classes

City
Write a fully-documented class named City that will represent each vertex/node of the graph. Each node needs to know what roads lead out of it, and what the capacity of the roads is. The city should implement comparable, and the compare method should look at the name of the city only, so we can print a list of cities alphabetically.

- HashMap<String,Integer> neighbors:the key is the name of the city, the Integer is the cost of the road
- String name: the name of the city
- public int compareTo(City o) - for the comparable

Optional:

- HashMap<String,Integer> tempNeighbors: - this may be useful for figuring out maximum network flow (you can keep track of how much flow you have left on a given edge)
- Boolean discovered
- Boolean visited
- Any other fields/variables you want

IslandNetwork
Write a fully documented class called IslandNetwork that holds the graph. For extra credit, implement Djikstra's algorithm for shortest path, using the edge capacities as weights. You may add additional fields and methods as desired, however, you must be able to load a graph from a file, and then calculate the DFS from any node in the graph, as well as maximal network flow from any node to any other node (printing "no flow" when applicable) for full credit.

- public static IslandNetwork loadFromFile(String url) - loads the file from the given URL location
    - Please note: Big Data also is capable of loading files stored on the local machine. In order to do so please specify file location in the same way you normally do for loading files in a normal directory).
- private HashMap<String, City> graph - this stores the cities in the graph
- public String (or void with prints or whatever other return type is convenient) maxFlow(String from, String to)
- public List<String> dfs(String from)
- EXTRA CREDIT: public String (or List<String> or void with prints or whatever other return type you want) djikstra(String from, String to)
- Methods to add cities to the graph

Optional helper methods:

- resetTempCapacity() - resets the temporary capacities of a given node
- resetDiscoveredVisited() - resets discovered and visited for dfs

IslandDesigner
Write a fully documented class called island designer that allows the user to run Depth First Searches from any City on the Island to any other City on the Island, as well as allowing the user to find the maximum network flow from any City on the Island to any other City on the Island. For extra credit, also implement a shortest path algorithm that shows the shortest path from any city on the Island to any other City on the Island.

- Public static void main(String[] args)

Useful Algorithms (Pseudocode)

Depth First Search:

```
DFS at a node(visiting a node):
    node.visited=true
      for neighbor in node.neighbors():
          if(!neighbor.Discovered)
                neighbor.setDiscovered(true)
                print(neighbor)
                visit(neighbor)
```

Flow at a node:

1. Find a path from source to destination

1.1 There exists a path from the source to the destination if there exists a path from any of the neighbors to the destination and the
available capacity from the source to the neighbor from which there exists a path is >=0 and also the available capacity from the
node to the neighbor is >=0. The total available capacity is equal to the minimum available capacity along any given edge, and all
the available capacities along the path should be decremented by this amount.
2. The capacity of a path from a node to its destination is the minimum of the capacity of the edge from the node to its neighbor and the capacity of the path from the neighbor to the destination
3. Decrement the temp capacity of each edge on the path by this value
4. Print the path with the capacity of the path
5. Repeat (find new path+flow along path) from source to Dest until the available capacity along all paths is 0 (or there are no paths available)

To find a path:
There exists a path from a node to the destination:
if there exists a path from one of its neighbors to the destination (capacity is (min of capacity of the edge and capacity of the
path)).
base case: if the node is a neighbor to the desired destination
then, the path capacity is the available capacity from the node to the destination

---

Big Data Sample Program

```
import big.data.DataSource;

public class BigDataCityRoadsExample {

    public static void main(String args[]){
        HashMap<String,Node> cities = new HashMap<String,Node>();

            DataSource ds = DataSource.connectXML("hw7.xml");
            ds.load();
            String cityNamesStr=ds.fetchString("cities");
            String[] cityNames=cityNamesStr.substring(1,cityNamesStr.length()-1).replace("\"","").split(",");
            String roadNamesStr=ds.fetchString("roads");
            String[] roadNames=roadNamesStr.substring(1,roadNamesStr.length()-1).split("\",\"");

        // Fill the HashMap here...
    }
}
```

Using a JAR in Eclipse:
Right click the project name in the ";Package Explorer"; tab (on the left, by default) - Select ";Build Path"; - Select ";Add External Archives..."; - Navigate to where you saved bigdata.jar and select it.
Using a JAR NetBeans:
Right click the project name in the ";Package Explorer"; tab (on the left, by default) - Click on ";Properties"; - Select ";Libraries"; on the left - Click on ";Add JAR/Folder"; on the right - Navigate to where you saved bigdata.jar and select it.
Using a JAR in IntelliJ: http://stackoverflow.com/questions/1051640/correct-way-to-add-external-jars-lib-jar-to-an-intellij-idea-project
Using a JAR on the command line: http://stackoverflow.com/questions/2096283/including-jars-in-classpath-on-commandline-javac-or-apt
Now you can ";import big.data.DataSource"; in your source code (or any other class from the big.data library that you need).

Note on Exceptions: all exceptions should be handled gracefully - they should be caught in the main, and the user should be notified by a nice printout. Your messages should clearly indicate what the problem is (bad index, full list, negative number, etc.). The program should continue to run normally after an exception is encountered. We will not be checking Input Mismatch cases.

Pretty Printing: Here is a tutorial on how to print tables neatly using printf in java. It is highly encouraged that you print the output neatly, as it makes grading much easier.
https://docs.oracle.com/javase/tutorial/java/data/numberformat.html

---

General Recommendations
You can feel free to add extra methods and variables if you need.

Note: please make sure that the menu is NOT case sensitive (so selecting A should be the same as selecting a).

Program Sample

```
Welcome to the Island Designer, because, when you're trying to stay above water, Seas get degrees!

please enter an url: https://www.cs.stonybrook.edu/~cse214/hw/hw7-images/hw7.xml

Map loaded.

Cities:     //Alphabetical Order
---------------------
Bones Beach
Composting Fields
Fire Hazard
Fishingville
Gatsby
Kingkongoma
Lawn City
North Spoon
Small Pear
South Spoon
Stream Foot
University

Road                                  Capacity
----------------------------------------------
Composting Fields to Small Pear           12
Lawn City to Small Pear                   30
Hipster to Small Pear                     14
Hipster to Small Pear                     16
Lawn City to Hipster                      16
Gatsby to Composting Fields               10
Fishingville to Lawn City                 17
Fishingville to Gatsby                    11
Bones Beach to Hipster                    12
Bones Beach to Lawn City                   8
Fire Hazard to Bones Beach                13
Kingkongoma to Fire Hazard                 7
Kingkongoma to Lawn City                  20
University to Kingkongoma                  6
University to Fishingville                18
Stream Foot to University                  6
Steam Foot to Kingkongoma                 11
North Spoon to Stream Foot                15
South Spoon to Stream Foot                20

Menu:
    D) Destinations reachable (Depth First Search)
    F) Maximum Flow
    S) Shortest Path (Extra Credit)
    Q) Quit
Please select an option: D
Please enter a starting city: University
DFS Starting From University:
Fishingville,Gatsby,Composting Fields, Small Pear,Lawn City, Hipster,Kingkongoma,Fire Hazard,BonesBeach
Please select an option: D
Please enter a starting node: South Spoon
DFS results (destinations reachable):
Stream Foot, Kingkongoma, Lawn City, Small Pear, Fire Hazard, Bones Beach, Hipster, Fishingville,
Gatsby, Composting Fields
Please select an option: F
Please enter a starting city: University
Please enter a destination: Hipster
Routing:
University->Fishingville->Lawn City->Hipster: 16
University->Kingkongoma->Fire Hazard->Bones Beach->Hipster: 6
Maximum Flow: 22
//Routing doesn't matter, so long as the right routes appear, and for each path, the maximum flow
equals the minimum of available flow rate among the edges within route.
Please select an option: F
Please enter a starting city: University
Please enter a destination: South Spoon
No route available!
Please select an option: S //Extra credit
Please enter a starting node: Kingkongoma
Please enter a destination node: Small Pear
Path: Kingkongoma->Fire Hazard->Bones Beach->Hipster->Small Pear
Cost: 46
Please select an option: Q
You can go your own way! Goodbye!
```

Extra Credit

You will get up to 5 points extra credit for implementing the Djikstras algorithm.