DS 4300: Large-Scale Storage and Retrieval
Learning Scala

Problem Description:

This assignment is designed to give you some experience in Scala programming as well as having you think about the implementation of various kinds of NoSQL databases.

Part A. Scala Warm-up

1. (10 pts) In class I showed how the following snippet of code can be used to read a comma-delimited text file, counting the *null* (empty) values in each column.

```
val nulls = Array[Int](0, 0, 0, 0)
for (line <- Source.fromFile(filename).getLines) {
  val toks = line.split(",", -1)
  for (i <- 0 until toks.length)
    if (toks(i) == "") nulls(i) = nulls(i) + 1
}
println(nulls.mkString(","))
```

   The following code would also work and employs a more functional style. Clearly explain what's going on here. Why does it achieve the same thing?

```
val nulls = Source.fromFile(filename).getLines
  .map(_.split(",", -1)).map(a => a.map(z => if (z == "") 1 else 0))
  .reduce((x, y) => (x zip y).map { case (u, v) => u + v })

println(nulls.mkString(","))
```

2. (10 pts) Write a recursive function to convert an unsigned integer to an n-bit binary string. What is the 32-bit binary for 1,234,567,890? The number of 1's in a binary number is known as the *weight.* Produce a plot of the weight of binary numbers from 0 to 1024. Here are your signatures:

```
toBinary(x: Int, bits: Int): String
weight(b: String): Int
```

3. (10 pts) When we have large-scale data in the form of billions of records (with ID = 1 to N) we might distribute those records to different storage nodes. Suppose we assign nodes using the following function:

   assigned_node = ID mod n (where n is the number of nodes in my network.)

   This is actually a terrible idea because if the number of nodes changes even slightly, a large fraction of records will usually have to be moved to a different node. Implement a function:

```
moved(records: Int, startN: Int, endN: Int): Double
```

that returns the fraction of records that would have to be re-assigned to a new node if records were re-partitioned from startN to endN nodes using the mod function. Compute moved(1000000, 100, 107).

Part B. Key-Value and Graph Stores in Scala

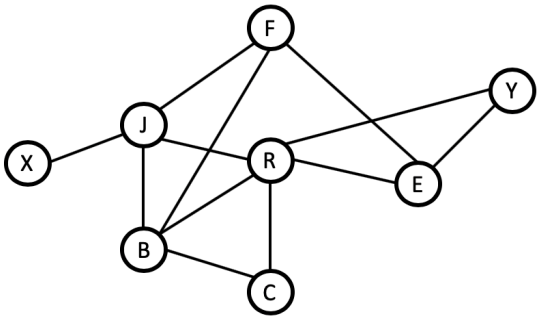
4. (35 pts) Redis is great, but what if we wanted our own very own embedded, in-memory redis-like key-value store? We could implement it in Scala of course. Implement a Scala-based key-value store as a single class. The class should have the following redis-like methods:

   get(key: String): String
   set(key: String, value: String)
   lpush(key: String, value: String)
   rpush(key: String, value: String)
   lpop(key: String): String
   rpop(key: String): String
   lrange(key: String, start: Int, stop: Int): List[String]
   llen(key: String)
   flushall()

5. (35 pts) There has been some experimentation on using Redis as backend storage for graph databases. Graph databases represent knowledge as graphs, and we will learn more about their capabilities later. Implement a rudimentary graph database built upon your in-memory key-value store. The graph database should be a single class with the following methods:

   addNode(v: String)
   addEdge(u: String, v: String)
   adjacent(v: String): List[String]
   shortestPath(u: String, v: String): List[String]

   Store the following graph in your graph database and test your database by finding the shortest path from x to y.



What to submit:
Submit your code and a file containing any program outputs to a git repository.
I recommend the following code files:

For Part A:

Binary.scala (object)
Partition.scala (object)

For Part B:
Redis.scala (class)
Graph.scala (class)
HW4.scala (object)