Hadoop

1. Describe your map/reduce algorithm for solving the three's company problem.

a. Describe the operation of the mapper and reducer. How does this combination solve the three's company problem?

When the mapper sees an entry, it stores the first token as the Vertex and the other L tokens as the Friends. For the key it outputs the $\binom{L-1}{2} = (L-1)(L-2)/2$ entries which are {Vertex, X, Y} in sorted order, where X and Y are all the combinations of friends. The value is always 1.

The reducer counts the number of entries of each key. Whenever there are 3 entries of the same key, it prints the permutations of that key.

For example, suppose the input is:

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

The result of the mapper will be:

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

All entries with the same key are guaranteed to be sent to the same reducer. Then the output will be as desired:

```
<1,2,3>
<2,1,3>
<3,1,2>
```

An informal proof of correctness:
Assume that the data is in the specified format (i.e. each vertex has a unique row, all friendships are symmetric, and no friends are repeated in a row). If ABC is a triangle, the key is printed by the mapper exactly once for each of the rows A, B, and C (and for no other rows). If ABC is not a triangle, it will be printed at most once (note: no key is ever printed twice; the logical possibilities for any key are 0, 1, or 3. This is because if A has friends B&C, and B has friends A&C, then by symmetry it must be the case that C has friends A&B).

b. What is the potential parallelism? How many mappers does your implementation allow for? Reducers?

Suppose that there are N users, each with L friends. The potential parallelism is to have N mappers, since each mapper processes at least one line. Depending on the input, there may be as many as $O(N*L^2)$ reducers.

c. What types are used as the input/output to the mapper? Motivate the transformation.

The (key, value) input to the mapper is (LongWritable, Text), which is TextInputFormat. I used this for ease because it is the default InputFormat.

The mapper's output is (Text, IntWritable). The Text was the potential triangle (i.e. A B C), and the IntWritable was always 1 (representing the count for that potential triangle).

2. Design On combiners

a. Why did you leave the combiner class undefined in Step 4?

Although a combiner class may have lowered the network communication, it is not worth implementing because it cannot reduce communication by more than a factor of 3. For large data with relatively sparse connectivity, it reduces communication by an arbitrarily small factor, and yet it costs time and resources to run the combiner.

b. Generalize the concept: What sort of computations cannot be conducted in the combiner?

The combiner can only be used for functions which can be iterated.
Formally speaking, let A, B, ..., Z be sets. The combiner can be used only when the reducer has the property:

$$r(A \cup B \cup \ldots \cup Z) = r(r(A) \cup r(B) \cup \ldots \cup r(Z))$$

(Note: If I wanted to implement a combiner, I would need to change my code from "count += 1" to "count += value" so that it would have the necessary property)

3. Analyze the parallel and serial complexity of the problem and your M/R implementation (in Big-O notation). You should assume that there are n friends list each of length l, i.e. n users that each have l friends.

a. What is the fundamental serial complexity of the problem? Think of the best serial implementation.

My best serial time complexity: $O(n*L^2 \log n)$

If any row can be found in O(1) time, the time is $O(n*L^2 \log L)$. This is a correct assumption when the entries are integers {1,...,n}. However, in the general case, it is $O(n*L^2 \log n)$, which is achieved as follows:

```
Sort the rows by their first entry
For each row X
     Sort the elements (keeping the first element in place)
For each row X
     For each friend Y in List(X)
          For each friend Z in List(Y)
               If X is in List(Z)
                    Print <X, Y, Z>
```

The rows are sorted in O(n*L log n) time. After that, sorting each row is O(n*L log L). The first loop iterates n times, and the second iterates O(L) times. List(Y) is found in O(log n) time, and the third loop iterates O(L) times. List(Z) is found in O(log n) time, and the conditional statement is evaluated in O(log L) time. The print is O(1).

This results in an overall run-time of:
$O(n*L \log n) + O(n*L \log L) + n * O(L) * (O(\log n) + O(L)*(O(\log n) + O(\log L)))$
$= O(n*L^2 \log n)$, using the fact that $L \leq n$

In retrospect, creating the above serial algorithm was quite unnecessary. I could have simply:
Run my mapper: $O(n*L^2)$
Sort the output: $O(n*L^2 \log(n*L^2))$
Run my reducer: $O(n*L^2)$
This achieves the same run-time of $O(n*L^2 \log n)$, again using the fact that $L \leq n$

b. How much work in total (over all mappers and reducers) does the Map/Reduce algorithm perform?

$O(n*L^2)$
This is because the mappers compute a total of no more than $n*L^2$ triplets, and then the reducers process their data in linear time.

c. How much work is performed by each mapper? By each reducer?

Let M be the number of mappers. MapperWork = $O(n*L^2/M)$. This is because each mapper has n/M rows.
Let R be the number of reducers. ReducerWork = $O(n*L^2/R)$. This is because each reducer has on average $nL^2/2R$ entries (there can be much fewer entries, depending on the data). Note that this is dependent on Hadoop's shuffling to minimize skew; Hadoop must guarantee that for some constant C, no reducer is given more than $CnL^2/2R$ entries.

d. Based on your answers to the above, describe the tradeoff between complexity and parallelism (qualitatively, you have already quantified it in the previous steps).

At first glance, the algorithm appears to be embarrassingly parallelizable. However, the implementation details of Hadoop may prevent totally linear speed-up (i.e. perhaps network latency during the shuffling step will add a serial component).