Assignment 1: Algorithms in MapReduce

(Adapted from Bill Howe's programming assignment, Coursera course: Introduction to Data Science)

In this assignment, you will be designing and implementing MapReduce algorithms for a variety of common data processing tasks.

The goal of this assignment is to give you experience "thinking in MapReduce." We will be using small datasets that you can inspect directly to determine the correctness of your results and to internalize how MapReduce works.

Write your own code!

For this assignment to be an effective learning experience, you must write your own code! I emphasize this point because you will be able to find Python implementations of most or perhaps even all of the required functions on the web. Please do not look for or at any such code! **Do no share code with other students in the class!!**

Here's why:

- The most obvious reason is that it will be a huge temptation to cheat: if you include code written by anyone else in your solution to the assignment, you will be cheating. As mentioned in the syllabus, this is a very serious offense, and may lead to you failing the class.
- However, even if you do not directly include any code you look at in your solution, it surely will influence your coding. Put another way, it will short-circuit the process of you figuring out how to solve the problem, and will thus decrease how much you learn.

So, just don't look on the web for any code relevant to these problems. Don't do it.

Python MapReduce Framework

You will be provided with a python library called MapReduce.py that implements the MapReduce programming model. The framework faithfully implements the MapReduce programming model, but it executes entirely on a single machine -- it does not involve parallel computation.

Here is the word count example discussed in class implemented as a MapReduce program using the framework:

```
# Part 1
mr = MapReduce.MapReduce()

# Part 2
def mapper(record):
    # key: document identifier
    # value: document contents
    key = record[0]
    value = record[1]
    words = value.split()
    for w in words:
      mr.emit_intermediate(w, 1)

# Part 3
def reducer(key, list_of_values):
    # key: word
```

```
    # value: list of occurrence counts
    total = 0
    for v in list_of_values:
      total += v
    mr.emit((key, total))

# Part 4
inputdata = open(sys.argv[1])
mr.execute(inputdata, mapper, reducer)
```

In Part 1, we create a MapReduce object that is used to pass data between the map function and the reduce function; you won't need to use this object directly.

In Part 2, the mapper function tokenizes each document and emits a key-value pair. The key is a word formatted as a string and the value is the integer 1 to indicate an occurrence of word.

In Part 3, the reducer function sums up the list of occurrence counts and emits a count for word. Since the mapper function emits the integer 1 for each word, each element in the list_of_values is the integer 1.

The list of occurrence counts is summed and a (word, total) tuple is emitted where word is a string and total is an integer.

In Part 4, the code loads the json file and executes the MapReduce query which prints the result to stdout.

Submission Details

For each problem, you will turn in a python script, similar to wordcount.py, that solves the problem using the supplied MapReduce framework.

When testing, make sure MapReduce.py is in the same directory as the solution script.

To allow you to test your programs, sample data will be provided in the data folder, and the corresponding solutions will be provided for each problem in the solutions folder.

Your python submission scripts are required to have a mapper function that accepts at least 1 argument and a reducer function that accepts at least 2 arguments. Your submission is also required to have a global variable named mr which points to a MapReduce object.

If you solve the problems by simply replacing the mapper and reducer functions in wordcount.py, then this condition will be satisfied automatically.

What you will turn in these five files:

- Program that implements a MapReduce algorithm to count words of different sizes in documents: `size_count.py`
- Program that implements a MapReduce algorithm that lists all the friends for each person called: `list_friends.py`
- Program that implements a MapReduce algorithm to identify pairs of friends (symmetric friendship) called: `pair_of_friends.py`
- Program that implements a MapReduce algorithm to identify mutual friendships for a pair of friends called: `mutual_friends.py`
- Program that implements a MapReduce algorithm for relational join: `relational_join.py`

Problem 1

Create a python program called `size_count.py` that implements a mapReduce algorithm to count the words of each size (large, medium, small, tiny) in a document. Given a set of documents, you will parse each document to identify individual words. You will determine the length of each word and add to the count of words of each size: large (10 or more letters), medium (5 to 9 letters), small (2 to 4 letters), and tiny (1 letter).

Use Combiner logic in the Map task to summarize the counts for the given chunk in the Map task.

Mapper Input

The input is a 2 element list: `[document_id, text]`, where `document_id` is a string representing a document identifier and `text` is a string representing the text of the document. The document text may have words in upper or lower case and may contain punctuation. You should treat each token as if it was a valid word; that is, you can just use value.split() to tokenize the string. (Note: this may treat a punctuation mark like ":" or "." as if it is a tiny word—that is OK.)

Reducer Output

The output should be a (document_id, size_list) tuple where document_id is a string and size_list is a list of tuples that contain (size, count) values, such as: [('large,' 17), ('medium', 34), ('small', 20), ('tiny', 5)]

You can test your solution to this problem using the data file books.json:

```
python size_count.py books.json
```

You can verify your solution against size_count.json.

Problem 2

Consider a simple social network dataset consisting of a set of key-value pairs `(person, friend)` representing a friend relationship between two people. Create a python program called `list_friends.py` that implements a MapReduce algorithm to produce a complete list of friends for each person.

Map Input

Each input record is a 2 element list `[personA, personB]` where personA is a string representing the name of a person and personB is a string representing the name of one of personA's friends. Note that it may or may not be the case that personA is a friend of personB.

Reduce Output

The output should be a pair `(person, friend_list)` where person is a string and friend_list is a list of strings that includes all friends of that person, where each friend appears only once in the list.

You can test your solution to this problem using the data file friends.json:

```
$ python list_friends.py friends.json
```

You can verify your solution by comparing your result with the file list_friends.json.

Problem 3

This problem will use the output of the Problem 2 as the input to a new phase of MapReduce.

Create a python program called `pairs_of_friends.py` that implements a MapReduce algorithm to identify symmetric friendships in the input data. The program will output pairs of friends where personA is a friend of personB and personB is also a friend of personA.

If the friendship is asymmetric (only one person in the pair considers the other person a friend), do not emit any output for that pair.

Map Input

Use the output of problem 2 as input to problem 3. Each input record is a list `[person, friend_list]`, where person is a string representing the name of a person and friend_list is a list of strings, where each string is the name of a friend of that person.

Hint: Think about emitting a unique key associated with a pair of names.

Reduce Output

The output should be the pairs of friends whose friend relationships are symmetric: that is, (personA, personB), where, personB is on the list of friends for personA, and personA is on the list of friends for person.

Be careful to avoid producing duplicates.

You can test your solution to this problem using the solution file from the Problem 2 (list_friends.json) as the input:

```
$ python pair_of_friends.py list_friends.json
```

You can verify your solution by comparing your result with the file pair_of_friends.json.

Problem 4

For this problem, we will again use the output of the Problem 2 as the input to a new phase of MapReduce.

Create a python program called `mutual_friends.py` that implements a MapReduce algorithm to identify mutual friends for a pair of friends. The program will output a list of friends that personA and personB are both friends with.

Map Input

Use the output of problem 2 as input to problem 4. Each input record is a list `[person, friend_list]`, where person is a string representing the name of a person and friend_list is a list of strings, where each string is the name of a friend of that person.

Hint: Think about emitting a unique key associated with a pair of names.

Reduce Output

The output should be the list of friends who are mutual friends of a pair of friends: that is, (uniqueKey, list_of_mutual_friends), where personC is on the list_of_mutual_friends if personC is a friend of both personA and personB AND personA and personB have a symmetric friendship.

You can test your solution to this problem using the solution file from the Problem 2 (list_friends.json) as the input:

```
$ python mutual_friends.py list_friends.json
```

You can verify your solution by comparing your result with the file mutual_friends.json.

Problem 5

Consider two relations that represent user information names and user movie ratings, respectively. Implement a MapReduce algorithm to join the two relations based on the Movie ID/Rated Movie ID value.

MovieNames

Movie name	Movie ID
Citizen Kane	0253
Casablanca	1378
Key Largo	4872

MovieRatings

Rated Movie ID	User ID	User Rating
1378	12340987	3.0
4872	69238560	3.5
0253	17098775	2.0
1378	17098775	5.0
0253	69238560	4.0

Map Input

The input data is represented as tuples that include the relation name:

MovieNames, Citizen Kane, 0253
MovieNames, Casablanca, 1378

MovieNames, Key Largo, 4872
MovieRatings, 1378, 12340987, 3.0
MovieRatings, 4872, 69238560, 3.5
… etc. …

Reduce Output

The output from the reduce function will be of two types:

- For each key, emit output tuple(s) that represent the joined relation in the form: (Movie Name, MovieID, Rated Movie ID, User ID, User Rating)
- For each key, also emit the average user rating for each movie in the form: (Movie Name, AvgRating), where Movie Name is a string and AvgRating is a float.

You can test your solution to this problem using the data file movie_data.json:

```
$ python relational_join.py movie_data.json
```

You can verify your solution by comparing your result with the file relational_join.json.