1 MapReduce

For each problem below, write pseudocode to complete the implementations. Tips:

- The input to each MapReduce job is given by the signature of map().
- emit(key k, value v) outputs the key-value pair (k, v).
- for var in list can be used to iterate through Iterables or you can call the hasNext() and next() functions.
- Usable data types: int, float, String. You may also use lists and custom data types composed of the aforementioned types.
- intersection(list1, list2) returns a list of the common elements of list1, list2.

1.1 Given a set of coins and each coin’s owner in the form of a list of CoinPairs, compute the number of coins of each denomination that a person has.

CoinPair:
- String person
- String coinType

```
map(CoinPair pair):
    emit(pair, 1)
reduce(CoinPair pair, Iterable<int> count):
    total = 0
    for num in count:
        total += num
    emit(pair, total)
```

1.2 Using the output of the first MapReduce, compute each person’s amount of money.

```
map(tuple<CoinPair, int> output):
    pair, amount = output
    emit(pair.person, valueOfCoin(pair.coinType) * amount)
reduce(String person, Iterable<float> values):
    total = 0
    for amount in values:
        total += amount
    emit(person, total)
```
2 Spark

Resilient Distributed Datasets (RDD) are the primary abstraction of a distributed collection of items.

Transforms $RDD \rightarrow RDD$

- **map**($f$) Return a new transformed item formed by calling $f$ on a source element.
- **flatMap**($f$) Similar to map, but each input item can be mapped to 0 or more output items (so $f$ should return a sequence rather than a single item).
- **reduceByKey**($f$) When called on a dataset of $(K, V)$ pairs, returns a dataset of $(K, V)$ pairs where the values for each key are aggregated using the given reduce function $f$, which must be of type $(V, V) \rightarrow V$.

Actions $RDD \rightarrow Value$

- **reduce**($f$) Aggregate the elements of the dataset regardless of keys using a function $f$.

Call `sc.parallelize(data)` to parallelize a Python collection, `data`.

2.1 Given a set of coins and each coin’s owner, compute the number of coins of each denomination that a person has. Then, using the output of the first result, compute each person’s amount of money. Assume `valueOfCoin(coinType)` is defined and returns the dollar value of the coin.

The type of `coinPairs` is a tuple of (person, coinType) pairs.

```python
coinData = sc.parallelize(coinPairs)

out1 = coinData.map(lambda (k1, k2): ((k1, k2), 1))
                    .reduceByKey(lambda v1, v2: v1 + v2)

out2 = out1.map(lambda (k, v): (k[0], v * valueOfCoin(k[1])))
                    .reduceByKey(lambda v1, v2: v1 + v2)
```

2.2 Given a student’s name and course taken, output their name and total GPA.

CourseData:

```python
    int courseID
    float studentGrade // a number from 0-4
```

The type of `students` is a list of (studentName, courseData) pairs.

```python
studentsData = sc.parallelize(students)

out = studentsData.map(lambda (k, v): (k, (v.studentGrade, 1)))
                   .reduceByKey(lambda v1, v2: (v1[0] + v2[0], v1[1] + v2[1]))
                   .map(lambda (k, v): (k, v[0] / v[1]))
```
3 MapReduce/Spark Practice: Optimize the Friend Zone

3.1 You are given a list of tuples containing people’s unique int ID and a list of the IDs of their friends. Compute the list of mutual friends between each pair of friends in a social network. You have access to the intersection function, which takes in two lists finds the set of elements that appear in both lists.

FriendPair:
  int friendOne
  int friendTwo

map(tuple<int, list<int>> info):
  personID, friendIDs = info
  for fID in friendIDs:
    if (personID < fID):
      friendPair = (personID, fID)
    else:
      friendPair = (fID, personID)
  emit(friendPair, friendIDs)

reduce(FriendPair key, Iterable<list<int>> values):
  mutualFriends = intersection(values[0], values[1])
  emit(key, mutualFriends)

3.2 Solve the problem above using Spark.

The type of persons is a list of (personID, list(friendID)) pairs.

def genFriendPairAndValue(pair):
  pID, fIDs = pair
  return [((pID, fID), fIDs) if pID < fID else ((fID, pID), fIDs) for fID in fIDs]

def intersection(l1, l2):
  return [x for x in l1 if x in l2]

personsData = sc.parallelize(persons)

out = personsData.flatMap(genFriendPairAndValue).reduceByKey(intersection)

4 Warehouse-Scale Computing

Sources speculate Google has over 1 million servers. Assume each of the 1 million servers draw an average of 200W, the PUE is 1.5, and that Google pays an average of 6 cents per kilowatt-hour for datacenter electricity.

4.1 Estimate Google’s annual power bill for its datacenters.

\[1.5 \times 10^6 \text{ servers} \times 0.2\text{kW/server} \times 0.06\text{kW-hr} \times 8760\text{ hrs/yr} \approx 157.68 \text{ M/year}\]

4.2 Google reduced the PUE of a 50,000-machine datacenter from 1.5 to 1.25 without decreasing the power supplied to the servers. What’s the cost savings per year?
\[
PUE = \frac{\text{Total building power}}{\text{IT equipment power}} \implies \text{Savings} \propto (PUE_{old} - PUE_{new}) \times \text{IT equipment power}
\]

\[
(1.5 - 1.25) \times 50000 \text{ servers} \times 0.2 \text{kW/server} \times 0.06 \text{kW/hr} \times 8760 \text{hrs/yr} \approx \$1.314 \text{ M/year}
\]