Exercise 1

Suppose that we have a large web corpus. Its metadata file has lines of the form (URL, size, date, ...). For each host, find the total number of bytes, i.e. the sum of the page sizes for all URLs from that host.

1. Define the input and the output of the Map and Reduce functions.

Input:

Input: map(URL, size(in bytes)
Input: reduce((host, [s1,s2,s3]))

go.com/home, 5, _, go.com
go.com/hello, 5, _, go.com => 10
ga.com/home, 6, _, ga.com
ga.com/hello, 9, _, ga.com => 15

Output:

Output map - emit (hi, si)
Output Reduce - emit((hosti, Size))
host, sum of number of page size in bytes
go.com, 10
ga.com, 15

Step 1: find the host name, find the byte/number attached to that host name
Step 2: Send all the bytes associated with the same host-name at one place.
Step 3: sum up the bytes for each host name.
Step 4: and the output will be the sum of the bytes along with the appropriate hostname

2. Write Map and Reduce functions. (pseudo-code)

PSEUDO CODE:

```
map(URL, size(in bytes)) {
hi = extract the host (urli)
.emit (hi, si) // host and size of I }

Reduce ((host, [s1,s2,s3])) {
for size in [s1,s2,s3]{
size += size }
.emit ((hi, final s)) }
```

Mapper types { InputKey, InputValue, OutputKey, OutputValue }
Mapper <LongWritable, Text, Text, IntWritable>

Mapper: By definition of the framework:
MapperInputKey = *offset in the file from where we are reading.*
MapperInputValue = *Each line of the input file.*
MapperOutputKey of type text = *host name*
MapperOutputValue of type IntWritable = *size of the page*

Reducer types { InputKey, InputValue, OutputKey, OutputValue }
Reducer<Text, Iterable <IntWritable>, Text, IntWritable>

Reducer: By definition of the framework:
ReducerInputKey = *hostname*
ReducerInputValue = *Iterable <PageSizes>*

ReducerOutputKey = *hostname*
ReducerOutputValue = *total size*

3. Define the input and the output of the Map and Reduce functions. Assume that we have generated two matrices from the previous file; A (n x n) and B (n x n). One wants to calculate another matrix M (n x n), which the product of the first two.

MapInput = *LongWritable, Text*
MapOutput = *IntWritable, List<IntWritable>(Cell Number of output matrix, row of A or column of B)*

Map for A: Send the appropriate row from A to the reduce key that would need those.
Map for B: Send the appropriate column from B to the reduce key that needs those.

Reducer Input = *IntWritable, List<List<IntWritable>>*
Reducer output = *Combination = M(A()*B()) // each cell has its own function and brings it together in reducer*

Reduce: *we want to calculate the value at each cell of the output matrix with one call to the reduce function.*
Information required = *one row of A and one column of B.*

4. Write Map and Reduce functions of the matrix-matrix multiplication.

// would need two of these for A(n*n) and B(n*n)!!!!
// same with reducer

Concept:

```
Map(key, values){
  strA = a(a_row*a_column); // row -> [1,2,3] col -> [4,5,6]
  strB = b(b_row*b_column) // row ->  [6,5,4] col -> [3,2,1]
  key = b_row*a_column;
}
reduce(a_column, b_row){
  A*B // output -> 4*6+5*5+6*4
            // essentially
            for ( pos = 1 to N ) do
              {
                x = first value at position pos
                y = second value at position pos
                sum = sum + x*y;
              }
}

<!--
One row vector from matrix A
One column vector from matrix B
-->
```

5. What if the matrices A and B do NOT FIT in the mappers' memory? (How would you implement your Map-Reduce program?)

answer: Map reduce only needs to have one row and one column to fit into its memory. It wouldn't need the whole matrix to fit into memory.

Exercise 2

K-Means is a simple clustering algorithm that aims to partition a set of objects into k clusters in which each object belongs to the cluster with the nearest mean. K-Means algorithm has 4 steps:

• Step 1: Given k, partition objects into k nonempty subsets

For the maping method you need a = map(pointer_ID, Ci) : *look at question two as well to understand input*

a. What the mapping function will do for a K-Means clustering algorithm is go through all of all of the possible pointer_IDs, with their appropriate distances to the generated centroid
   i. (pid1, {distance(pid1, c2), c2})
   ii. (pid2, {distance(pid1, c2), c2}) … ect

K = number of clusters -> the output is // is k is 5 = you want 5 elements in the cluster
***essentially**:*
a. divide the numbers randomly
b. do this repeatedly till you find what → (output)
Other example Steps:
1. Get a text data with numbers: {426, 781, 216, 448, 859} // ie. K number of clusters to compute
2. divide the numbers randomly = EACH:.split(numbers)[0]

• Step 2: Compute seed points as the centroids of the clusters of the current partition. The centroid is the center (mean point) of the cluster

Lets say you have a graph of x*y with many cluster nodes in multiple places

iii. We need to GENERATE small Centroids in multiple places in the graph to gather the closest nodes in the cluster
   1. From mapper => emit (pointer_IDi, {distance (pointer_Idi, Ci)})
   2. Each closest node to the Centroid will be calculated by its distance to the Centroid

Other example Steps:
take number segments: <Centroid, Points>
a. {426} => (3, pointTo3, 426) // mapping
b. {781} => (2, pointTo2, 781)
c. {216} => (1, pointTo1, 216)
d. {448} => (2, pointTo1, 448)
e. {859} => (4, pointTo1, 859)

• Step 3: Assign each object to the cluster with the nearest seed point

After completing the first step of partitioning objects into k nonempty subsets, we need to group them with their correct nearest distance: usually with a for loop

a. which becomes:
   iv. (pidj, {distance(pid1, c2), c2})
   v. (pidj, {distance(pid1, c2), c2}) … ect for the biiiiiiig number of cluster nodes

Other example Steps:
i. After the mapping stage that includes the <Centroid, Point> you compute, combine and assign them
ii. // seed points = randomly assigned one
1. ie: (3, pointTo3, 426) => <(3, 1), 426> the 1 is number of pointsTo that are sumed up
2. (2, pointTo2, 781) => <(2, 2), 11 12 9> ***same as DI!!! 11 12 9 = adding each first number together***
3. (1, pointTo1, 216) => <(1, 1), 216>
4. (2, pointTo1, 448) ***=> ---------- look at d.***
5. (4, pointTo1, 859) => <(4, 1), 859>

• Step 4: Go back to Step 2 until no more new assignment Define the input

Mapper:

Map input: Map(key, value) // (Pointer_IDi, Cj);
Map Output: Map(Pointer_Idi, {distance(Pointer_Idi, Cj), Cj}) ;

Reducer:

Reducer Input: Reduce ((pidj, [{distance (pidj, C1), C1}, {distance (pidj, C2), C2)}]);
Reducer Output: objects that belong to the cluster with the nearest mean.

You would want to go back to some sort of Shuffler/Sort before coming back to rerunning this step again for more clusters

Steps:
1. Shuffling them, sorting and going back to step two:
   a. Shuffle => <(1,1) 216> && <(1,2) 131> // 131 means

Notes: question 3

<!—
Take input: (URL, size(in bytes), date, host)
host = url.split("/")[0]
Output(host, size)

Map => Shuffle/Sort Phase (no control over it/created by framework/same list for the same host => All outputs of the mapper with the SAME-KEY goes to one REDUCER

Reduce:
Input: Host, List<Size> (sorted)
Host, add all elements in the list<size>
Output : Host, size

--!>

Notes: question 2

<!--
class Map and reduce :
extends Mapper<URL, Size, Date>{
key is URL
host = extract host url
emit host
emit size
}
ReduceForWordCount extends Reducer<host and size>

{
for(value: URLS, Size, Date){
sum += value.getall(values)
}
sys.out (sum)
}
-->