(a) Common Friend

```
# Example: CommonFriends Template
  function main(){
        listOfFriendship = list of (Person, [List of Friends])
        # Write your pseudocode here.
        listOfCommonFriend = MapReduce(listOfFriendship)
          #Output Result
          Print contents of listOfCommonFriend on console
}

  function Map(<person>, <list of friends>){
        # Write your pseudocode here.
          Friends[] = [list of friends]
          for each friend in Friends {
                  Emit(<person, friend>, <Friends>)
          }
}

  function Reduce(<person, friend>, <List of FriendsLists>){
        # Write your pseudocode here.
          return (<person, friend>, <Intersection of List of FriendsLists>)
}
```

Explanation: For each friend in the list, the mapper emits as key a pair of the person along with the friend and the list of friends again. Take as an example the following person and its friends:
Amy -> [Betty, Charlie, Daniel, Eric]
The map phase now outputs the following pairs:
[Amy, Betty] -> [Betty, Charlie, Daniel, Eric]
[Amy, Charlie] -> [Betty, Charlie, Daniel, Eric]
[Amy, Daniel] -> [Betty, Charlie, Daniel, Eric]
…

The reducer takes as input the key/value pairs of the map phase. For each pair (key) it calculates the intersection of the corresponding friends list.
Suppose Charlie's friends list is as follows:
Charlie -> [Amy, Betty, Daniel, Jesse, Mike]
Then the relevant key/value pair of the map phase for calculating Amy's and Charlie's common friends is:
[Charlie, Amy] -> [Amy, Betty, Daniel, Jesse, Mike]
The intersection of these two lists is:
[Betty, Charlie, Daniel, Eric] inter [Amy, Betty, Daniel, Jesse, Mike = [Betty, Daniel]
In other words, Amy and Charlie have two mutual friends, Betty and Daniel.

(b)High School Days

```
# Example: HighSchool Template
  function main(){
        listOfFilenames = list of filenames containing
                  test scores of all students.
        # Write your pseudocode here.
        listOfScores = MapReduce(listOfFilenames)
        #Output Result
          for(i = 100; i>=0; i--) {
                  if( i in <scores> ) print (i) n times;
                  #if i is not in the <scores> then skip to the next i;
          }

}

  function Map(<filename>, <scorefileofOneClass>){
        # Write your pseudocode here.
          Scores[] = [scorefileofOneClass]
          for each score in Friends {
                  Emit(score, 1)
          }
}

  function Reduce(score, <listsofValues>){
        # Write your pseudocode here.
          return (score, length(listofValues))
}
```

making Reduce to build a dictionary of students' math score, if there is a score 99 one student earned, then 99 will appear in the key of dictionary, if there is a score forexample 30 that no student had such a low grade, then 30 will not appear in the key of dictionary, therefore, when looping to print out score in main(), scores not in dictionary will not be printed out.

(c) Good Old π

We can calculate π easily using a Monte Carlo experiment. Imagine you have a dart board with radius 1 meter (big dartboard). It's contained within a perfectly fitting square, thus it's 2m on a side. If we throw a dart, the probability that we hit the dart board, given that we hit within the square is the area of the board divided by the total area of the square, so π/4.

 # Example: Good Old pi Template

lets just make the input be [1,…,100]. And, say that each Mapper only gets a single input. The Mapper is where the MonteCarlo experiment actually happens. Let each mapper run 100 sims.

```
function Map(s) {
        hitCounter = 0
        for(int I = 0; I < 100; I ++) {
                x = rand()
                y = rand()
                if(x* x + y * y < 1) {
                        hitCounter ++
                }
        }
        Emit(1, hitCounter)
}

 function Reduce(key, values) {
        totalHits = 0
        for v in values {
                 totalHits += v
        }
        # each value is out of 100, to total is 100*|values|
        emit(key, totalHits / (100. * len(values))
}
```

Because there's only only one Reducer, there's only one answer from the whole experiment, the experimental average of the number of hits. This should converge to π/4 as the number of experiments increases.

(d)GaugeTheDistance

```
# Example: GaugeTheDistance Template
function main(){
    G[] = Adjacency list of the graph.
    # G[i] is a list of (neighbors, distance) tuple of node i.
    n = length(G)
    dist[] = list that will contain the distances
        from node 1 eventually. Initialize it arbitrarily.
    distUpdated[] = list that will contain distances from
            node 1 after each run of MapReduce in the following loop. Initialize it with (0,∞,∞,...,∞).

    while( NOT stoppingCriterion(dist, distUpdated)){
        dist = distUpdated
        # Clearly state the variables and fill in details.
        # key is the index of node i
        #value is a pair which constists current shortest distance noed i to node 1, and also G[i]
        key = i
        value = (dist[i], G[i])

        distUpdated = MapReduce(list of (<key>, <value>) pairs).
    }
    print the list dist[].
}
function stoppingCriterion(dist1, dist2){
    # Write your pseudocode here.
        if(dist1 == dist2) return true;
        else return false;
}

function Map(i, (dist[i], G[i])) {
    # Write your pseudocode here.
        for node j in G[i].neighbors {
            Emit(j, dist[i]+j.distance to i)
        }
}
```

#MapReduce collect all pairs (j, dist[i]+j.distance to i) and create a list of the (dist[i]+j.distance to i) → we call this list "listOfCombinedDistance"

```
function Reduce(node j, < listOfCombinedDistance >){
    # Write your pseudocode here.
        return (node j, Math.min(combined distance in < listOfCombinedDistance>))
}
```