1. (Solution and Grading by: <Le Xu>)

1st Mapper/Reducer

Map1 (key, value) // where key follows value
{
  Emit<key, value>
}

Reduce1 (key, list)
{
  if list.length >= 3
    For i from 0 to list.length
      For j from i to list.length
        For k from j to list.length
          Emit(sort(key, list[i], list[j], list[k]), key)
}

2nd Mapper/Reducer

Map2 (key, value) // key in the form of <k1, k2, k3, k4>, value in the form of a key
{
  Emit<key, value>
}
### Reduce2
(key, list) // key in the form of <k1, k2, k3, k4>, value in the form of a list

```java
If list.length == 4 Emit <key, list>
```

### 1st Mapper/Reducer

#### Map1
(key, value) // input (a, b) where a follows b

```java
if( b == @packers )
    emit ( @packers, a)
else
    emit ( sort(a,b), "") // the function sort(x,y) sorts “x” and “y” lexicographically
    // this ensures identical “key” for both (x,y) and (y,x)
```

### Reduce1
(key, value)

```java
List L = value.getList();
If(key == @packers) // “L” contains followers of @packers… e.g. “L” == <p,q,r>
{
    For each “unique” pair (a,b) in “L”
        emit (sort(a,b), 1);
        //hints: unique pairs in this example are <(p,q), (p,r), (q,r)>
}
Else
{
    emit (key, 0) // note that, “key” is the lexicographically sorted list of a and b
    // here, “at least” one of them follows the other
}
```

### 2nd Mapper/Reducer
Map2(key, value) // “key” is the sorted usernames (from Reduce 1), value is either 0 or 1
{ emits(key, value)
}

Reduce2(key, value)
{
    List L = value.getList();
    length = L.getLength();
    if (1 == length && 1 == value)
    {
        // (1 == length) ensures that the key was emitted from a single “source”
        // (1 == value) ensures that the “source” is the @packer’s follower list
        a = split(key)[0]; // get the first user from the lexicographically sorted list
        b = split(key)[1]; // get the second user from the lexicographically sorted list
        emit(a, b);
    }
}

3. (Solution and Grading by: <Beomyeol>)

1st Mapper/Reducer

map (k, v) { // v is an input line formatted (a, b) when a follows b.
    Parse an input line v = (a, b) // a follows b
    output (a, (b, ∅))
    output (b, (∅, a))
}

reduce (k, V) {
    followingSet ← ∅
    followerSet ← ∅
    foreach v = (a, b) ∈ V do
        followingSet ← followingSet ∪ {a}
        followerSet ← followerSet ∪ {b}
    end
    if @KingJames ∈ followingSet then
        output (k, #) // k follows @KingJames
    else if k != @KingJames ∧ |followerSet| > 10M then

}
```java
foreach u ∈ followerSet do  
    output (u, *) // u follows k s.t. k is not @KingJames, doesn’t follow @KingJames, and has over 10M followers  
end

2nd Mapper/Reducer

map (k, v) { // k, v is an output of 1st Reducer above. k is username. v is either (true, false) or (false, true)  
    output (k, v)  
}

reduce (k, V) {  
    if * ∈ V ∧ # ∈ V then  
        output k  
    end  
}

4. (Solution and Grading by: Rui)  
   a. M=2k+1. The design could tolerate 2k simultaneous failures.  
      Consider a scenario where a certain process P is never selected into the random-k subset by any other process. P would only need to send heartbeat to its k predecessors and k successors. If these 2k processes fail simultaneously, the failure of P would not be detected by any active processes. Thus, this design does not provide completeness for 2k+1 simultaneous failures (2k + process P). When there are only 2k simultaneous failures, there is always at least one process alive among 2k neighbours, so the failure of P will always be detected.  
   b. No.  
      Heartbeat protocols are never 100% accurate in an asynchronous system as heartbeats can be lost.  
   c. Worst case: (N-1) heartbeats per second. Every process except for P itself chooses P to send heartbeat.  
      Best case: 2k heartbeats per second. Only the k predecessors and k successors of P choose P to send heartbeat (required).  
      Average case: 3k per second. Every process would ask 3k processes to send heartbeat, thus there are 3k * N heartbeats sent per period(second) in total. On
average, it is \((3k \ast N) / N = 3k\) heartbeats per second.

5. (Solution and Grading by: Rui)
   a. The protocol is eventually complete. A faulty member will be eventually chosen as a ping target at every non-faulty member, so it will be considered as faulty eventually.
   b. The probability of a process \(P\) is pinged every period is \(1 - \left(1 - \frac{m}{N}\right)^{N-1}\). The ping happens every \(T\) time units. Thus, the expected detection time is 
      \[
      \frac{T}{\left(1 - \left(1 - \frac{m}{N}\right)^{N-1}\right)}.
      \]
      
      \[
      \lim_{N \to \infty} \frac{T}{\left(1 - \left(1 - \frac{m}{N}\right)^{N-1}\right)} = \frac{T}{(1 - e^{-m})}.
      \]
   c. Thus, The expected detection time is still independent of \(N\) if \(N\) is large enough.

6. (Solution and Grading by: Zhuolun)
   Consider any node in the system. The probability it is chosen with full membership list is \(B/N\). The probability it is chosen with partial membership list is \(m/k \ast k/N = m/N\). When they are equal, \(B = m\).

7. (Solution and Grading by: Faria)
   a. The memory cost is the cost of storing the finger table. Although we have altered the algorithm used to populate the finger table, we have not added any more entries and so the size of the finger table should remain the same as that in the original algorithm = \(O(\log N)\).
   b. Again, the lookup cost is similar to the cost of Chord, as all the algorithm needs to work is the existence of any one finger in the range \(n+2^i\) and \(n+2^{i+1}\). By choosing a neighbor that is closest in terms of latency in the given interval, we achieve the same effect as Pastry does with its locality principle.
   c. Original TA answer: initial \(i\) means smaller values of \(i \rightarrow \) There are fewer choices in the initial hops as the value of \(i\) is small. Therefore, the algorithm makes big jumps in the initial hops and smaller jumps later, where larger values of \(i\) lead to more choices amongst neighbors.
      However, a lot of students have taken initial hops to mean initial hops \(\text{while routing}\), when \(i\) would be bigger. There, initial hops would be smaller as there are more choices and later hops would be bigger.
      Both answers are accepted (if the correct explanation is provided).
General comments:
1. Most students have failed to provide a comparison between earlier and later jumps.
2. Some students have presented an answer without an explanation. In this case, they will lose all points.

8. (Solution and Grading by: Rahul)

a. There are 20 nodes in the system, let’s say we start at node 1.
   Step 1:
   TTL = 3
   19,20,1,2,3
   Step 2:
   TTL = 2
   17,18,19,20,1,2,3,4,5
   Step 3:
   TTL = 1
   15,16,17,18,19,20,1,2,3,4,5,6,7
   Therefore **13 nodes** receive the query message.

b. A minimum **TTL = 5** is required for all nodes to receive the message.
   Step 1:
   TTL = 5
   19,20,1,2,3
   Step 2:
   TTL = 4
   17,18,19,20,1,2,3,4,5
   Step 3:
   TTL = 3
Step 4:
TTL = 2
13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9
Step 5:
TTL = 1
11,12,13,14,15,16,17,18,19,20,1,2,3,4,5,6,7,8,9,10

Let Node 21 (coach) be connected to all the nodes.
If Node 1 is sending the message it is connected to 21, so 21 receives the message and transmits it to every other node in 2 TTL. For **TTL = 2** we can be sure that all the nodes receive the query.

9. (Solution and Grading by: <Beomyeol>)
   a.

<table>
<thead>
<tr>
<th>i</th>
<th>f[i]</th>
<th>i</th>
<th>f[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2020</td>
<td>6</td>
<td>1992</td>
</tr>
<tr>
<td>1</td>
<td>2020</td>
<td>7</td>
<td>1992</td>
</tr>
<tr>
<td>2</td>
<td>2020</td>
<td>8</td>
<td>1992</td>
</tr>
<tr>
<td>3</td>
<td>2024</td>
<td>9</td>
<td>1992</td>
</tr>
<tr>
<td>4</td>
<td>1992</td>
<td>10</td>
<td>1992</td>
</tr>
<tr>
<td>5</td>
<td>1992</td>
<td>11</td>
<td>1992</td>
</tr>
</tbody>
</table>


10. (Solution and Grading by: Faria)
    We modify the reduce function to maintain partial results per key and update them as new values arrive for a given key.
    The barrier in the shuffle ensures that all values per key have arrived and are fully sorted before they are passed on to the reduce phase. This is why the reducer expects a list of values per key. If we break the barrier, we would be passing on records to the reducer as soon as they arrive from the mapper. This implies that the reducer must be modified.
to 1) expect a <key, value> pair as input (instead of <key, list of values> pair), and 2) maintain partial results per key and update the partial results as new values arrive for the given key. Partial results can be stored in any data structure. The developer would now have to specify how partial results should be updated per key as new values arrive. Note that the final result will be exactly as the same as original Hadoop.