```java
import java.io.*;
import java.lang.Math;
import java.nio.channels.Channels;
import java.util.*;

import sun.security.krb5.internal.crypto.dk.ArcFourCrypto;
class LoggingAlgorithm{

    // This is a list of channels that is known
    // these all have channelId and frequency
    private ChannelSet channels;

    // This is a list of a list of channels. EAch element of this
    // will be a list of channels with the same frequency and acsending order
    private List<ChannelSet> frequencySet;

    // This will eventually store the pattern of data that will be present in each second
    private Pattern algPattern;
    // Dummy example object of a block of data
    private Block exampleBlock;
    // So that algorithm can work on multiple blocks
    private List<Block> blockList;
    // To store orderedData at end of algorithm
    private List<ChannelData> orderedData;


    private static final int SECOND = 1000;

    public LoggingAlgorithm(){
        channels = new ChannelSet();
        frequencySet = new ArrayList<>();
        channels.add(new Channel(0, 1));
        channels.add(new Channel(1, 1));
        channels.add(new Channel(2, 20));
        channels.add(new Channel(3, 50));
        channels.add(new Channel(4, 20));
        channels.add(new Channel(5, 100));
        channels.add(new Channel(6, 200));
        channels.add(new Channel(7, 100));
        channels.add(new Channel(8, 20));
        channels.add(new Channel(9, 2));
        channels.add(new Channel(10, 2));

        // This is dummy data - todo if time randomly generate
        exampleBlock = new Block(257, 3000, new int[3000]);
        blockList = new ArrayList<>();
        blockList.add(exampleBlock);
         // Create objects to store orderedData
         orderedData = createStorageForOrderedData(channels);
    }

    private List<ChannelData> createStorageForOrderedData(ChannelSet originalSet){
        List<ChannelData> data = new ArrayList<>();
        for(Channel ch : originalSet.getChannelList()){
            data.add(new ChannelData(ch.getChannelId(), ch.getFrequency()));
            // ASSUMPTION : Under this current implementation the index of data will be the same as channelId
            // This will not be the case all the time, but for the sake of this challenge, due to time constraints
            // I will assume it is.
        }
        return data;
    }


    /**
     * After this function is ran frequncySet contains a list of lists of channels that are grouped by
     * Frequency and are in ascending order of channelId.
     */
    public void sortChannelsIntoFrequencySets(ChannelSet channelSet, List<ChannelSet> frequencySet){
```

```
69          // iterate over channels
70          for(int i=0;i<channelSet.size();i++){
71              boolean addedChannelToSet = false;
72              Channel channelI = channels.get(i);
73              for(int j=0;j<frequencySet.size(); j++){
74                  List<ChannelSet> channelOfFrequency = frequencySet.get(j);
75                  // For the list to exist it has at least one entry
76                  if(channelOfFrequency.get(0).getFrequency() == channelI.getFrequency
                    () ){
77                      channelOfFrequency.add(channelI);
78                      addedChannelToSet = true;
79                      break; // breaking for loop going over FrequencySet
80                  }
81              }
82              // check if channel was added to any ChannelSet organised in frequencySet
83              if(!addedChannelToSet){
84                  // if no channel was added then its part of a new set
85                  frequencySet.add(new ChannelSet());
86                  frequencySet.get(frequencySet.size()-1).add(channelI);
87              }
88          }
89      }
90
91      /**
92       * Once the channels are grouped in collections of frequency in order than one
          needs to know
93       * which ticks will fire these collections and for how many data points.
94       */
95      private void calculateTickValuesForCollections(List<ChannelSet> frequencySet){
96          int frequencySetSize = frequencySet.size();
97          for(int j=0;j<frequencySetSize;j++){
98              ChannelSet set = frequencySet.get(j);
99              // Calculate the tick values and frequency range of each set
100             set.calculateOrderedTotal();
101         }
102     }
103
104     /**
105      * Some simple sort algorithm to get in descending ordered list of freqencies
106      */
107     private List<ChannelSet> reOrderFrequencySetsInDescendingOrder(List<ChannelSet>
        sets){
108         int setLength = sets.size();
109         List<ChannelSet> orderedList = new ArrayList<>();
110         for (int i=0; i <setLength; i++){
111             int highestFrequencyFound = 0;
112             int highestFrequencyIndex = -1;
113             for(int j=0; j< sets.size();j++){
114                 ChannelSet channelSet = sets.get(j);
115                 if(channelSet.getFrequencyOfSet() > highestFrequencyFound){
116                     highestFrequencyFound = channelSet.getFrequencyOfSet();
117                     highestFrequencyIndex = j;
118                 }
119             }
120             if(highestFrequencyFound != 0 && highestFrequencyIndex >=0){
121                 // Found highest frequency value set
122                 orderedList.add(sets.get(highestFrequencyIndex));
123                 // remove from parent set so that you dont iterate over needless
                    values
124                 sets.remove(highestFrequencyIndex);
125             }
126         }
127         // now orderedList contains the frequencySet in ordered fashion
128         return orderedList;
129     }
130
131     /**
132      * This function calculates the pattern of data that will be observed every
          second. This data is stored in a list
133      * where each element contains the tick number and the channelId that is fired.
134      */
135     private Pattern calculateRecurrsiveTickPatternInData(List<ChannelSet> sets){
136         Pattern pattern = new Pattern();
```

```
137             // loop over all ticks in a second
138             for(int t=0; t<SECOND ; t++){
139                 // First look in each frequencyset and check if t appears in the tick
                    array
140                 for(ChannelSet set : sets){
141                     if(!set.doesTickAppearInThisSet(t)){
142                         // if current tick does not appear in the channelSet then move
                            to next channelSet
143                         continue;
144                     }
145                     // channelSet 'set' will read out on this tick value
146                     // Therefore fill pattern with tick and channel values in ascending
                        order over channelSet
147                     for(Channel ch : set.getChannelList()){
148                         // The Channel list will be in ascending order so simple read
                            off the channelIds
149                         pattern.add(new TickPattern(t, ch.getChannelId()));
150                     }
151                 }
152             }
153             // Pattern contains recurrsive list over a second
154             return pattern;
155         }
156 
157         public void runAlgorithm(){
158             // Take the given set of channels that contain channelId and frequency and
                group them accordingly
159             sortChannelsIntoFrequencySets(channels, frequencySet);
160             // Calculate which ticks will fire the frequencySets
161             calculateTickValuesForCollections(frequencySet);
162             // Order the frequencySets in descending order
163             frequencySet = reOrderFrequencySetsInDescendingOrder(frequencySet);
164             // Now that the sets are ordered in descending order and we know which
                ticker values will fire each frequency set
165             // One can now calcualte a recurring pattern of channelIds that will be
                fired each second.
166             algPattern = calculateRecurrsiveTickPatternInData(frequencySet);
167             // ASSUMPTION = I will assume that the total number of channels in the
                complete system have
168             //          channelIds that are always consequtive and in ascending order
                from 0.
169             //          If this wasn't the case I would be tempted to make a
                HashMap<Integer, ChannelData>
170             // Where Integer would be the channelData channelId
171             for(Block block : blockList){
172                 sortDataFromSingleBlock(block, orderedData, algPattern);
173             }
174             // DONE!
175         }
176 
177         /**
178          * This sorts out data for one block and adds the data to the orderedData list
179          */
180         private void sortDataFromSingleBlock(Block block, List<ChannelData> data, Pattern
             pattern){
181             // Calculate offset of block in a second
182             final long blockStartTime = block.getStart();
183             // Find the index of the first tick that will be present in the data block
184             pattern.setStartIndex(blockStartTime);
185             // Now interate over all data in the block
186             for(int i=0 ; i<block.getLength();i++){
187                 // Get information needed for current index
188                 ChannelTime channelTime = pattern.getInfoForCurrentIndex();
189                 // Add the data to the ordered sets - NOTE - THIS IS WHERE I USE THE
                    ASSUMPTION ABOUT CONTINUOUS DATA
190                 ChannelData channelData = orderedData.get(channelTime.getChannelId());
191                 channelData.addDataToChannel(new DataPoint(block.getDataAtIndex(i),
                    channelTime.getTime()));
192                 //Increment index of pattern array
193                 pattern.incrementIndex();
194             }
195         }
196 
```

```
197
198   }
```