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

 /**
     * Algorithm used to demonstrate parts of the coding challenge provided
     * Cosworth Electronics Ltd.
     *
     */
class CosworthCode{

    public static void main(String[] args){
        // Save a list of channels that the data logger will be
        LoggingAlgorithm loggingAlgorithm  = new LoggingAlgorithm();
        loggingAlgorithm.runAlgorithm();

    }



}
```

```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   }
```

```java
import java.util.ArrayList;
import java.util.List;

class Pattern{
    /**
     * This class contains the pattern of data that is given out over each second
     * The Pattern starts at the beginning of a second. This means that for blocks that
     * dont begin at the start of a second, we need to track the offset.
     */

     private List<TickPattern> pattern;
     private int finalPatternSize = -1; // I got into the habit of setting initial values to stuff I know shouldn't exist
     private int index = -1;

     // In order to correctly order the data from multiple blocks into human readable forms like graphs, the time of each
     // data point should be stored. To do this I will save the nearest time to the block start time and the number of seconds
     // That has elapsed since. Then from the tick number of the pattern one can calculate the ms time since the logger started that the
     // data point accured.
     // If one knows the global time at which the logger started then data from lots of loggers can be combined with many different channels
     // and many different blocks to give a human readable graph.
     private long nearestSecondToBlockStartTime = 0;
     private int numberOfSecondsElapsedFromStartTime = 0;

     public Pattern(){
         pattern = new ArrayList<>();
     }

     public List<TickPattern> getPattern(){
         return pattern;
     }

     public void add(TickPattern tickPattern){
         pattern.add(tickPattern);
     }

     /**
      * This function works out the starting index of the pattern in the block
      */
     public void setStartIndex(long blockStartTime){
         // number of milli seconds into second that the block began
        int offset = blockStartTime % 1000;
        nearestSecondToBlockStartTime = blockStartTime - (long)offset;
        //Calculate the number of positions left in the second before the pattern recurs
        // This is the number milliseconds left in the first non-complete second
        this.index = findIndexOfNearestTickToOffset(offset);
     }

     /**
      * Function to find the nearest tick to the offset provided by the block start time.
      */
    private int findIndexOfNearestTickToOffset(int offset){
     // pattern size should be fixed at this point
     this.finalPatternSize = pattern.size();
        for(int k=0; k < finalPatternSize ; k++ ){
            TickPattern tickPattern = pattern.get(k);
            if(tickPattern.getTick() >= offset){
                // Return the index as soon as the tick in the pattern is greater than
                //or equal to the offset provided by the block start time.
                return k;
            }
        }
        // If this function has not returned a value by this point it means that the offset in the second
        // was higher than any of the ticks in the pattern. If this is the case than
```

```
            the next data point
64          // will be from the next second at tick t=0.
65          return 0;
66      }
67
68      public incrementIndex(){
69          this.index++;
70          if(this.index == finalPatternSize){
71              this.index = 0;
72              //every time it ticks over add a full second on to counter above
73              this.numberOfSecondsElapsedFromStartTime++;
74          }
75      }
76
77      public int getChannelIdForCurrentIndex(){
78          TickPattern tickPattern = pattern.get(index);
79          return tickPattern.getChannelId();
80      }
81
82      /**
83       * The tick value is the millisecond value that the logger is fired on
          therefore knowing the block start time
84       * and the number of secs elasped with the tick value you can calculate raw time
85       */
86      private long calculateTimeOfDataPoint(int tickValue){
87          return nearestSecondToBlockStartTime + (numberOfSecondsElapsedFromStartTime *
              1000) + tickValue;
88      }
89
90      public ChannelTime getInfoForCurrentIndex(){
91          TickPattern tickPattern = pattern.get(index);
92          return new ChannelTime(tickPattern.getChannelId(), calculateTimeOfDataPoint(
              tickPattern.getTick()));
93      }
94
95      public DataPoint createDataPointFromPattern(int value){
96          TickPattern tickPattern = pattern.get(index);
97          DataPoint dataPoint = new DataPoint(value, time)
98      }
99
100     /*
101     public List<TickPattern> getPatternForTick(int tick){
102         // todo if needed
103         return null;
104     }
105     */
106 }
```

```java
class Channel{
    private int channelId;
    private int frequency;

    public Channel(int channelId, int frequency){
        this.channelId = channelId;
        this.frequency = frequency;
    }
    public int getChannelId(){
        return channelId;
    }
    public int getFrequency(){
        return frequency;
    }
}
```

```java
import java.util.ArrayList;

import com.sun.corba.se.impl.ior.FreezableList;

/**
 * this class contains the data output for a specific channelId
 */
class ChannelData{
    private int channelId;
    private int frequency;
    private List<DataPoint> data;

    // IT SHOULD BE NOTED - That by the point this object needs to be created, one could calulate how many bits of
    // data should be expected over the blocks provided to the program. If given more time I would add this in.

    public ChannelData(int channelId, int frequency){
        this.channelId = channelId;
        this.frequency = frequency;
        this.data = new ArrayList<>();
    }

    public void addDataToChannel(DataPoint dataPoint){
        data.add(dataPoint);
    }

    public int getChannelId(){
        return channelId;
    }

    public int getFrequency(){
        return frequency;
    }
}
```

```java
import java.util.ArrayList;

class ChannelSet{
    /**
     * This is a list of Channels that have the same Frequency
     */

    private List<Channels> channels;
    private int totalNumberOfChannelsAfterOrdering = -1; // If this value is -1 there has been a problem in algorithm.
    private int frequencyOfSet = -1;
    private int[] ticks;

    public ChannelSet(){
        channels = new ArrayList<>();
    }

    public void add(Channel channel){
        channels.add(channel);
    }

    /**
     * After this function is run the array ticks contains the t values of all ticks that have a data read out for this
     * Frequency
     */
    public void calculateOrderedTotal(){
        if(totalNumberOfChannelsAfterOrdering == -1){
            totalNumberOfChannelsAfterOrdering = channels.size();
        }
        if(totalNumberOfChannelsAfterOrdering > 0){
            frequencyOfSet = channels.get(0).getFrequency();
            ticks = new int[1/frequencyOfSet]; // calculates the number of ticks in a second thats fired by this frequency set
            // The fiddle factor offset here is to account for the fact that all frequencies are multiples of 1000
            // but they tick at t=0 not t=1000
            ticks[0] = 1;
            for(int i=0;i<ticks.length-1;i++){
                ticks[i+1] = (i*1000)/frequencyOfSet;
            }
        }
    }


    public int getFrequencyOfSet(){
        return frequencyOfSet;
    }

    public int[] getTicks(){
        return ticks;
    }

    public boolean doesTickAppearInThisSet(int tick){
        for(int t : ticks){
            if(t == tick){
                return true;
            }
        }
        return false;
    }

    public List<Channel> getChannelList(){
        return channels;
    }
}
```

```java
/**
 * Another simple class to help pass data from Pattern class to LoggingAlgorithm
 */

class ChannelTime{
    private int channelId;
    private long time;

    public ChannelTime(int channelId, long time){
        this.channelId = channelId;
        this.time = time;
    }

    public int getChannelId(){
        return channelId;
    }

    public long getTime(){
        return time;
    }
}
```

```java
/**
 * This is an example class of a block of data
 */

class Block{
    private long blockStartTime;
    private int blockLength;
    private int[] blockData;
    public Block(long startTime, int length, int[] data){
        this.blockStartTime = startTime;
        this.blockLength = length;
        this.blockData = data;
    }
    public long getStart(){
        return blockStartTime;
    }
    public int getLength(){
        return blockLength;
    }
    public int[] getData(){
        return blockData;
    }

    public int getDataAtIndex(int index){
        return blockData[index];
    }
}
```

```java
import com.sun.org.apache.regexp.internal.recompile;

/**
 * This object stores the tick number in the second and a channelid that is fired on that tick
 */

 class TickPattern{
     private int tick; // in range from 0->999
     private int channelId; // this is fired
     public TickPattern(int tick, int channelId){
         this.tick = tick;
         this.channelId = channelId;
     }

     public int getTick(){
         return tick;
     }

     public int getChannelId(){
         return channelId;
     }
 }
```

```java
/**
 * Simple class that stores the value and time of a datapoint
 */

class DataPoint{
    private int value;
    private long time;
    public DataPoint(int value, long time){
        this.value = value;
        this.time = time;
    }

    public int getValue(){
        return value;
    }
    public long getTime(){
        return time;
    }
}
```