Cosworth Electronics Challenge

Robert Hutchinson
Email: Robert.hutchinson.work@gmail.com
Contact Number: +44 7812520981

Part I

Restating the problem

- 32-bit `int` data is captured from channels C at frequencies f

$$C = C(f) f \in \{1, 2, 5, 10, 20, 50, 100, 200, 500, 1000\} Hz \tag{1}$$

- multiple channels per frequency
- Loggers read out data in tick intervals, with 1000 ticks (t) every second.

$$t \in \{0 : 999\} \tag{2}$$

- Data is read out on each tick for channel C under the following condition, where n is any integer.

$$t = \frac{n}{f} * 1000 \tag{3}$$

Part II

The Algorithm

- From equation 3 one can calculate which ticks t in every second will contain data from a channel given a certain frequency, where $t = 1000 \rightarrow 0$.
- If more than one channel has a data read out on the same ticker, the algorithm should expect data to come from channels with decreasing frequency and if they have the same frequency, ascending channel identifier.
- Channel numbers are not required in the logged data because, given a huge array of `int` data, simply knowing the frequency, start time and channel id of the channels, one get deduce which rows of the array correspond to which types of data.

1 Board Description

Broadly speaking the method of this algorithm is as follows.

1. The input is a set of `channels`, which contain a number a `channel_id` and `frequency` and a huge array of `int` data with a start time.
2. Group the given channels into *sets* which have the same frequency, and order those channels in each set by ascending `channel_id`.
3. Calculate the tick numbers t that trigger each *set*.
4. Calculate the offset between the block start time & the ticker number of the first element in the block. This is needed because the blocks won't necessarily, and probably don't start on the second.
5. By this point you know which ticks will result in data being recorded, and for each tick that results in data being recorded, how many bits of data are expected.
6. From this one can generate a pattern of `channel_id` that is written to, and the time in milliseconds of that value.

7. iterate over the data in the block and assign each row, a `channel_id` and a `time` in milliseconds of the event. Then fill *channel sets* accordingly with that data.
8. Using this algorithm, one can order data from any number of blocks, even in the blocks overlap in time. You just need to order the time series data after you have them in *channel sets*, then graph it.

2 More Detailed Description

This section gives more details about the algorithm along with snippets of code in Java. I will send a github link to all code.
Disclaimer : I don't have time to check that the code actually works for 2 reasons. Firstly, I wrote this all in VS code which doesn't compile projects (I'm working on a laptop that doesn't have eclipse and I didn't want to use time getting it setup). Secondly, it would require generating a dummy dataset to work on. I think this is probably outside the scope of the exercise. I also realised a little too late that my settings on LaTeX lstlistings makes java code look aweful. I will append the code to the end of the pdf.

- For the sake of completeness, if you are looking at the project code on Github. The main function is in `CosworthCode`.
- I've tried to comment the code as much as necessary so that you understand my logic.
- using separation of concerns there are a bunch of classes that simple handle passing data around and are not that interesting. The main classes that actually do stuff include:
  - `LoggingAlgorithm.java`
  - `Pattern.java`
  - `ChannelSet.java`
- In `LoggingAlgoirthm.java` the constructor simple generate some dummy channels and frequencies. The interesting part starts in `runAlgorithm()`.

```java
public void runAlgorithm(){
// Take the given set of channels that contain channelId and
    frequency and group them accordingly
sortChannelsIntoFrequencySets(channels, frequencySet);
// Calculate which ticks will fire the frequencySets
calculateTickValuesForCollections(frequencySet);
// Order the frequencySets in descending order
frequencySet =
    reOrderFrequencySetsInDescendingOrder(frequencySet);
// Now that the sets are ordered in descending order and we
    know which ticker values will fire each frequency set
// One can now calcualte a recurring pattern of channelIds
    that will be fired each second.
algPattern =
    calculateRecurrsiveTickPatternInData(frequencySet);
// ASSUMPTION = I will assume that the total number of
    channels in the complete system have
//           channelIds that are always consequtive and in
    ascending order from 0.
//           If this wasn't the case I would be tempted to
    make a HashMap<Integer, ChannelData>
// Where Integer would be the channelData channelId
for(Block block : blockList){
sortDataFromSingleBlock(block, orderedData, algPattern);
}
// DONE!
}
```

- `sortChannelsIntoFrequencySets()` does a simple enough job and groups together the provided list of `Channel` into groups depending on their `frequency`.

- `calculateTickValuesForCollections(frequencySet)` then calculates the tick values which will fire data being logged for this frequency set. The tick numbers are actually calculated in the `ChannelSet` class after the sets are re-ordered in *ascending channel identifier*, according to equation 3.

- `reOrderFrequenceSetsInDecendingOrder(List<ChannelSet>)` - this is a simple sort algorithm to order the frequency sets in decending order.

- `calculateRecurrsiveTickPatternInData(frequencySet)` - This is where it gets interesting. The idea is to create a pattern that occurs every second in the data which contains information on which channels are fired in what order.

```
private Pattern
    calculateRecurrsiveTickPatternInData(List<ChannelSet>
    sets){
Pattern pattern = new Pattern();
// loop over all ticks in a second
for(int t=0; t<SECOND ; t++){
// First look in each frequencyset and check if t appears in
    the tick array
for(ChannelSet set : sets){
if(!set.doesTickAppearInThisSet(t)){
// if current tick does not appear in the channelSet then
    move to next channelSet
continue;
}
// channelSet 'set' will read out on this tick value
// Therefore fill pattern with tick and channel values in
    ascending order over channelSet
for(Channel ch : set.getChannelList()){
// The Channel list will be in ascending order so simple read
    off the channelIds
pattern.add(new TickPattern(t, ch.getChannelId()));
}
}
}
// Pattern contains recurrsive list over a second
return pattern;
}
```

  - `Pattern` contains a list of objects that describe the `channelId` that was fired, and which `tick` fired it.
  - looping over all ticks in a second (0 - 999), for each second one has to calculate whether the tick will trigger a read out in descending frequencies.
  - If the `tick` will cause a readout then simple iterate over the channels in that frequency set and add them to the pattern.
  - As the lists are already ordered you simple need to read out the channels that will be fired by each tick in the second.

- After this is done, one just needs to iterate over the number of block they have and order the data into sets that are characterised by their `channelId`. This is done here:

```
for(Block block : blockList){
sortDataFromSingleBlock(block, orderedData, algPattern);
}
```

```java
/**
* This sorts out data for one block and adds the data to the
   orderedData list
*/
private void sortDataFromSingleBlock(Block block, List<ChannelData>
   data, Pattern pattern){
// Calculate offset of block in a second
final long blockStartTime = block.getStart();
// Find the index of the first tick that will be present in the data
   block
pattern.setStartIndex(blockStartTime);
// Now interate over all data in the block
for(int i=0 ; i<block.getLength();i++){
// Get information needed for current index
ChannelTime channelTime = pattern.getInfoForCurrentIndex();
// Add the data to the ordered sets - NOTE - THIS IS WHERE I USE THE
   ASSUMPTION ABOUT CONTINUOUS DATA
ChannelData channelData = orderedData.get(channelTime.getChannelId());
channelData.addDataToChannel(new DataPoint(block.getDataAtIndex(i),
   channelTime.getTime()));
//Increment index of pattern array
pattern.incrementIndex();
}
}
```

– Now because the start time of the block can occur any time in the second, we need to work out how far down the recurrent pattern we are in the beginning of the block.

```java
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
// will be from the next second at tick t=0.
return 0;
}
```

- The nearest second is saved because this is used later to calculate the raw time of each data point. This is stored as a `long` number milliseconds since the data logger started. If one knew this start time in global time system then one can work out the exact time of each data point. However I should probably note that this accuracy may tail off if left for a long time, depending on the ticker accuracy. When I was working with bluetooth data I had this problem whereby the data was given nanosecond timestamps which lost accuracy as you chained them together. Basically one could say to a fair degree of accuracy that successive values where 70 ish nanoseconds apart but once you compared values that were seconds apart without another reference, time started to drift.
- The `pattern index` is incremented inside the pattern class using the below code. This recorded the number of full seconds that elapsed whilst iterating through this pattern.

```
  public incrementIndex(){
this.index++;
if(this.index == finalPatternSize){
this.index = 0;
//every time it ticks over add a full second on to
    counter above
this.numberOfSecondsElapsedFromStartTime++;
}
}
```

- The `channelId` and data `time` is read off from the `pattern` using the below functions and this is past back to the `LoggingAlgorithm`.

```
private long calculateTimeOfDataPoint(int
    tickValue){
return nearestSecondToBlockStartTime +
    (numberOfSecondsElapsedFromStartTime * 1000) +
    tickValue;
}

public ChannelTime getInfoForCurrentIndex(){
TickPattern tickPattern = pattern.get(index);
return new ChannelTime(tickPattern.getChannelId(),
    calculateTimeOfDataPoint(tickPattern.getTick()));
}
```

- The `channelId`, `time` and `value` of the data point is then added to the `orderedData`.

```
ChannelData channelData =
    orderedData.get(channelTime.getChannelId());
channelData.addDataToChannel(new
    DataPoint(block.getDataAtIndex(i),
    channelTime.getTime()));
//Increment index of pattern array
pattern.incrementIndex();
```

- *An assumption is used here* I assume that the *channelId* will be in ascending order and continuous. This is so I get use the `channelId` as the index for `orderedData`.
- If this wasn't the case, I would be tempted to use a Hashmap, but there are probably better ways.

```
HashMap<Integer, ChannelData> hashmapData
```

Part III
General Questions

1. **What are the advantages of using an interleaved data format on a data logger ?**
   Mainly speed of data storage. To store the data in a more logical or human readable format would require constant computation, working out which data belonged in which column. By quickly storing all data in its raw form, yet in an organised way it allows you to quickly group the data for post processing. This saves time & money, in terms of cost of components since the data loggers don't need to do any expensive computation.
2. **How would you test the decoding algorithms that you have designed ?**
   I would do this in two ways.
   (a) Generate unit tests for each function so that I'm sure that each function is generating the correct output given dummy inputs.
   (b) Then generate a test sample, a number of data blocks that I know to be correct, and with it a collection of data that is ordered. Generating the data shouldn't be as difficult as organising it. Then feed the data in backwards and check the results.
3. **If you were to implement an application to do the above and to read the data over Ethernet, how would you structure the application to ensure maximum throughput at the times ?**
   TODO

```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){
```

```java
            // iterate over channels
            for(int i=0;i<channelSet.size();i++){
                boolean addedChannelToSet = false;
                Channel channelI = channels.get(i);
                for(int j=0;j<frequencySet.size(); j++){
                    List<ChannelSet> channelOfFrequency = frequencySet.get(j);
                    // For the list to exist it has at least one entry
                    if(channelOfFrequency.get(0).getFrequency() == channelI.getFrequency() ){
                        channelOfFrequency.add(channelI);
                        addedChannelToSet = true;
                        break; // breaking for loop going over FrequencySet
                    }
                }
                // check if channel was added to any ChannelSet organised in frequencySet
                if(!addedChannelToSet){
                    // if no channel was added then its part of a new set
                    frequencySet.add(new ChannelSet());
                    frequencySet.get(frequencySet.size()-1).add(channelI);
                }
            }
        }

        /**
         * Once the channels are grouped in collections of frequency in order than one needs to know
         * which ticks will fire these collections and for how many data points.
         */
        private void calculateTickValuesForCollections(List<ChannelSet> frequencySet){
            int frequencySetSize = frequencySet.size();
            for(int j=0;j<frequencySetSize;j++){
                ChannelSet set = frequencySet.get(j);
                // Calculate the tick values and frequency range of each set
                set.calculateOrderedTotal();
            }
        }

        /**
         * Some simple sort algorithm to get in descending ordered list of freqencies
         */
        private List<ChannelSet> reOrderFrequencySetsInDescendingOrder(List<ChannelSet> sets){
            int setLength = sets.size();
            List<ChannelSet> orderedList = new ArrayList<>();
            for (int i=0; i <setLength; i++){
                int highestFrequencyFound = 0;
                int highestFrequencyIndex = -1;
                for(int j=0; j< sets.size();j++){
                    ChannelSet channelSet = sets.get(j);
                    if(channelSet.getFrequencyOfSet() > highestFrequencyFound){
                        highestFrequencyFound = channelSet.getFrequencyOfSet();
                        highestFrequencyIndex = j;
                    }
                }
                if(highestFrequencyFound != 0 && highestFrequencyIndex >=0){
                    // Found highest frequency value set
                    orderedList.add(sets.get(highestFrequencyIndex));
                    // remove from parent set so that you dont iterate over needless values
                    sets.remove(highestFrequencyIndex);
                }
            }
            // now orderedList contains the frequencySet in ordered fashion
            return orderedList;
        }

        /**
         * This function calculates the pattern of data that will be observed every second. This data is stored in a list
         * where each element contains the tick number and the channelId that is fired.
         */
        private Pattern calculateRecurrsiveTickPatternInData(List<ChannelSet> sets){
            Pattern pattern = new Pattern();
```

```java
        // loop over all ticks in a second
        for(int t=0; t<SECOND ; t++){
            // First look in each frequencyset and check if t appears in the tick array
            for(ChannelSet set : sets){
                if(!set.doesTickAppearInThisSet(t)){
                    // if current tick does not appear in the channelSet then move to next channelSet
                    continue;
                }
                // channelSet 'set' will read out on this tick value
                // Therefore fill pattern with tick and channel values in ascending order over channelSet
                for(Channel ch : set.getChannelList()){
                    // The Channel list will be in ascending order so simple read off the channelIds
                    pattern.add(new TickPattern(t, ch.getChannelId()));
                }
            }
        }
        // Pattern contains recurrsive list over a second
        return pattern;
    }

    public void runAlgorithm(){
        // Take the given set of channels that contain channelId and frequency and group them accordingly
        sortChannelsIntoFrequencySets(channels, frequencySet);
        // Calculate which ticks will fire the frequencySets
        calculateTickValuesForCollections(frequencySet);
        // Order the frequencySets in descending order
        frequencySet = reOrderFrequencySetsInDescendingOrder(frequencySet);
        // Now that the sets are ordered in descending order and we know which ticker values will fire each frequency set
        // One can now calcualte a recurring pattern of channelIds that will be fired each second.
        algPattern = calculateRecurrsiveTickPatternInData(frequencySet);
        // ASSUMPTION = I will assume that the total number of channels in the complete system have
        //          channelIds that are always consequtive and in ascending order from 0.
        //          If this wasn't the case I would be tempted to make a HashMap<Integer, ChannelData>
        // Where Integer would be the channelData channelId
        for(Block block : blockList){
            sortDataFromSingleBlock(block, orderedData, algPattern);
        }
        // DONE!
    }

    /**
     * This sorts out data for one block and adds the data to the orderedData list
     */
    private void sortDataFromSingleBlock(Block block, List<ChannelData> data, Pattern pattern){
        // Calculate offset of block in a second
        final long blockStartTime = block.getStart();
        // Find the index of the first tick that will be present in the data block
        pattern.setStartIndex(blockStartTime);
        // Now interate over all data in the block
        for(int i=0 ; i<block.getLength();i++){
            // Get information needed for current index
            ChannelTime channelTime = pattern.getInfoForCurrentIndex();
            // Add the data to the ordered sets - NOTE - THIS IS WHERE I USE THE ASSUMPTION ABOUT CONTINUOUS DATA
            ChannelData channelData = orderedData.get(channelTime.getChannelId());
            channelData.addDataToChannel(new DataPoint(block.getDataAtIndex(i), channelTime.getTime()));
            //Increment index of pattern array
            pattern.incrementIndex();
        }
    }

```

```
}
```

```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
```

```java
            the next data point
            // will be from the next second at tick t=0.
            return 0;
        }

        public incrementIndex(){
            this.index++;
            if(this.index == finalPatternSize){
                this.index = 0;
                //every time it ticks over add a full second on to counter above
                this.numberOfSecondsElapsedFromStartTime++;
            }
        }

        public int getChannelIdForCurrentIndex(){
            TickPattern tickPattern = pattern.get(index);
            return tickPattern.getChannelId();
        }

        /**
         * The tick value is the millisecond value that the logger is fired on
         therefore knowing the block start time
         * and the number of secs elasped with the tick value you can calculate raw time
         */
        private long calculateTimeOfDataPoint(int tickValue){
           return nearestSecondToBlockStartTime + (numberOfSecondsElapsedFromStartTime *
            1000) + tickValue;
        }

        public ChannelTime getInfoForCurrentIndex(){
            TickPattern tickPattern = pattern.get(index);
            return new ChannelTime(tickPattern.getChannelId(), calculateTimeOfDataPoint(
            tickPattern.getTick()));
        }

        public DataPoint createDataPointFromPattern(int value){
            TickPattern tickPattern = pattern.get(index);
            DataPoint dataPoint = new DataPoint(value, time)
        }

        /*
        public List<TickPattern> getPatternForTick(int tick){
            // todo if needed
            return null;
        }
        */
    }
```

```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;
    }
}
```

```
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;
    }
}
```