Project Documented Design

Overview Of The Code:

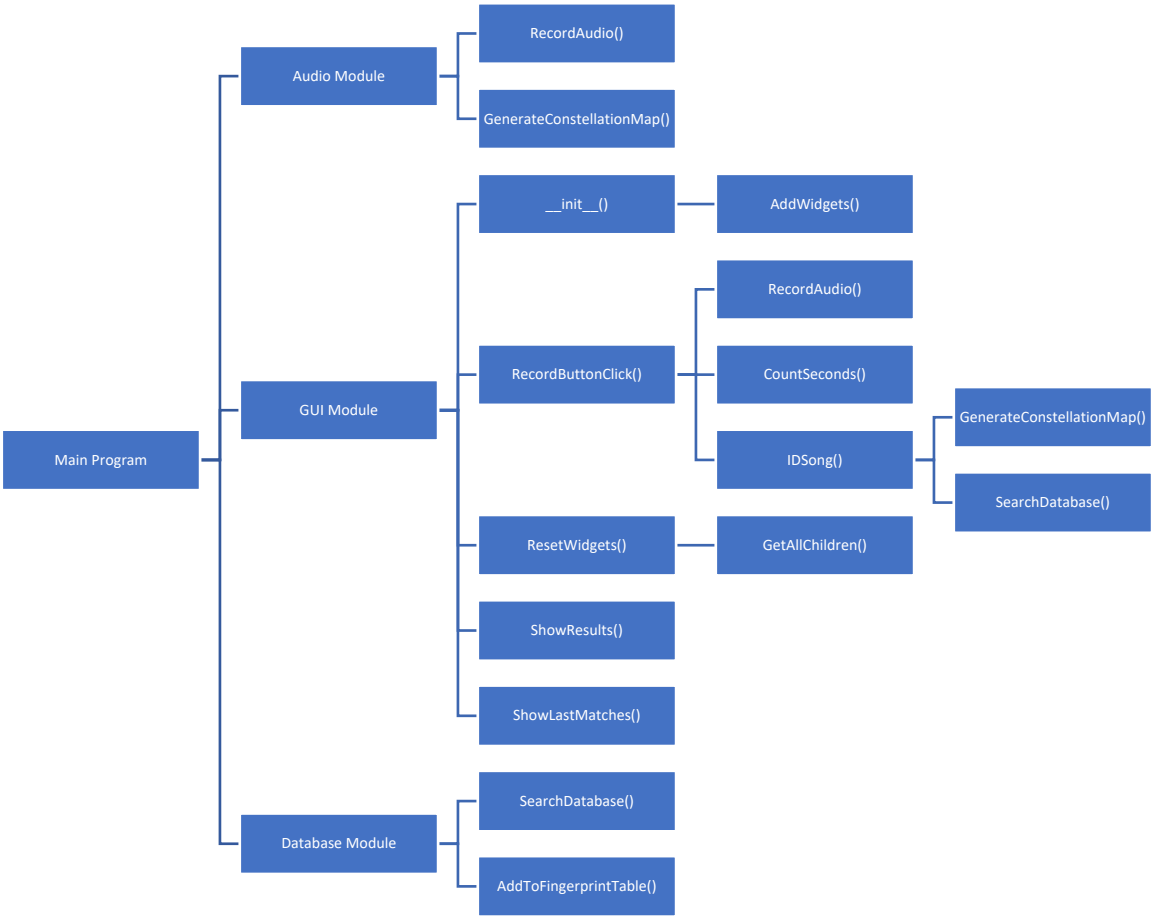
My software features three primary modules which each have a unique role in the process of the audio search. There is a fourth, called 'Main' but its purpose is simply to call all of the others.

The first is the Audio Module which handles all the algorithms required for producing a fingerprint of the recorded audio. Some of the most important algorithms in this module are the Fourier Transform, for separating audio into its spectrum, and a spectrogram band analysis algorithm for choosing the most dominant points in the spectrum.

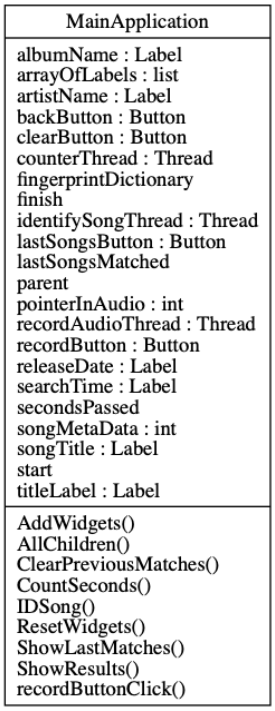
The second is the GUI Module which handles the construction and functionality of the GUI. I have primarily used a library called 'tkinter' to assist me with this. I have also made use of threading in this module, to call multiple functions simultaneously.

The final module is the Database Module dedicated to the searching of the database in order to find a fingerprint that matches the query. I have used multiple hash tables and thus a hashing function to help improve search efficiency, and I've also made use of writing and reading from text files for storage of fingerprints. The main algorithms here are for checking the number of matching notes per song, between the notes returned and those in the query, and another algorithm expands on this filtering of results by checking the order in time that the matching notes occur.

Here is a hierarchy chart, describing the structure of my program:



My code is not object-oriented however my GUI Module makes use of a class (called 'MainApplication') for organising the UI development. Here is the class diagram describing it:



Overview Of The Method

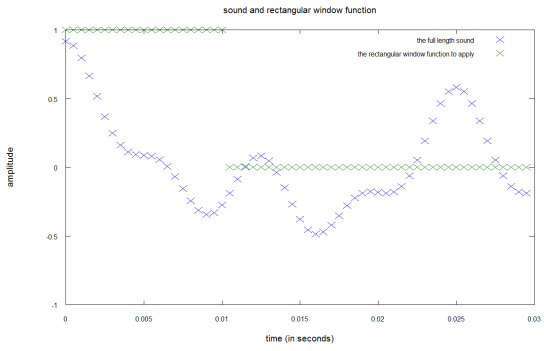
Generating the fingerprint:

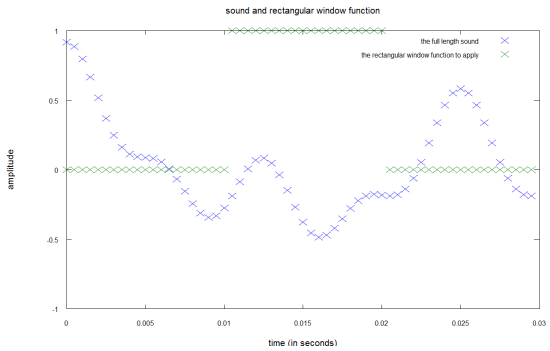
Audio is recorded through the device's microphone at 44100Hz in stereo. Before any further processing is done, I downsample the signal to a quarter of the original sample rate (ie. 11025Hz), but to avoid aliasing frequencies due to Nyquist Shannon[1] theory I have to apply a low pass filter which removes all frequencies over 5000Hz. The downsampling is required to reduce the amount of computation required later on.

Ultimately, I would like to apply a Fourier Transform to this audio to find the frequencies present over a given time interval of the audio, however doing this to all the audio at once would lose all time information. To avoid this, I use a windowing technique. As a technical implementation, a window function is an array containing the same number of elements as the input audio but every item is zero except across the section of audio we want to apply the Fourier Transform to. Here is an example of a rectangular window function which has the value of 1 across the audio we are operating on, and 0 elsewhere (green is the window function and blue is the audio signal).[2]

[1] 'Nyquist-Shannon sampling theorem', accessed 7th March 2019, <https://en.wikipedia.org/wiki/Nyquist–Shannon_sampling_theorem>

[2] 'How does Shazam work' 2015, Cristophe, accessed 27th February, <http://coding-geek.com/how-shazam-works/>

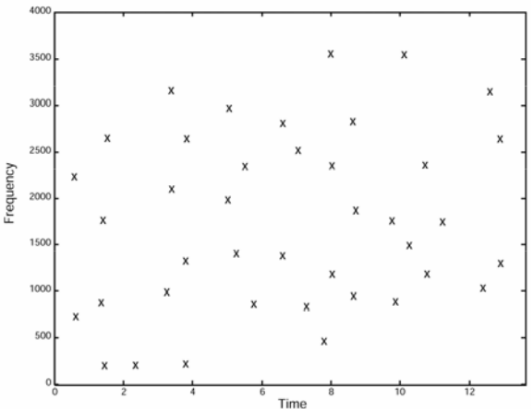




So having multiplied the window function with the audio we apply the Fourier Transform to the entirety of the resulting array. This will give us all the frequencies and their amplitudes for that particular section of the audio. We then slide the window over to the next part of the audio and apply the Fourier Transform to that. This is demonstrated in the second diagram above.

After doing this to all of the data I can gather information on the frequency, time and amplitude of every point in the audio. Now I need to find the frequencies which are the most powerful across all the data. Due to a variety of reasons we can't just pick the points with the highest amplitude. Hence we apply an algorithm to the data which I have described on page 15.

At this point I have all the information to generate a plot called a constellation map which will look something like this[3]:



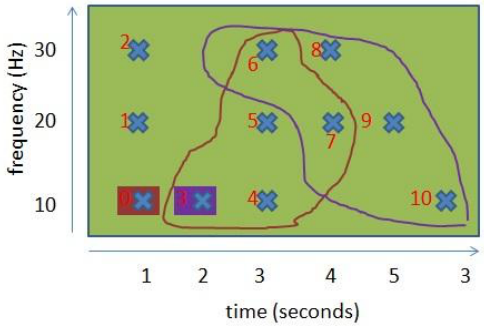
Each point represents a powerful frequency of the song. This is the basis of the audio fingerprint.

<u>Storing the fingerprint</u>

In order to do an efficient search later on, I need to store the fingerprint information in a particular way. Given a stream of audio points, I group them into 'target zones' which I implemented as a list where each element of the list is itself a list of five consecutive audio points – overlap is allowed between zones. So for example, points 1, 2, 3, 4, 5 would be the first target zone, points 2, 3, 4, 5, 6 would be the next, and so on. For each given target zone we also need to select an appropriate 'anchor point'. I used the point three before the first one in the target zone. So the group consisting of points 11, 12, 13, 14, 15 would have the point 8 as its anchor point. Here is a graphical explanation of this (Coloured circles show the

[3] 'An Industrial-Strength Audio Search Algorithm', Avery Li-Chun Wang, accessed 27th February, <https://www.ee.columbia.edu/~dpwe/papers/Wang03-shazam.pdf>

target zones, coloured boxes of corresponding colour show the anchor point of that target zone)[4]:



With all this information we can start to create the values that will actually be stored in the database.

An 'Address' for each point is made by this formula (where || means concatenation): ["frequency of the anchor point of the target zone it belongs to" || "frequency of the point in question" || "time difference between the anchor and the point"]

A 'Couple' for each point is made by this formula:
["Absolute time of the anchor in the song" || "Song ID"]

Each value here is also encoded into binary so when all the values are concatenated together, we can represent the addresses as 32-bit integers and the couples as 64-bit integers.

The database is made up of one enormous table where one column is the addresses and the other column is the couples - everything is in binary form. It's implemented as a hash table where the indexes are the addresses and the values for each index are a list of couples (only a list if multiple couples have the same address, hence we don't need to deal with hashing collisions).

<u>Searching the database:</u>

When a 10 second long query is recorded, the same fingerprinting method is applied and we also generate addresses for each point using exactly the same method as before. Couples are slightly different however:

Query couple = ["absolute time of the anchor point of the point's target zone"]

For this query, we should now have a list of addresses with couples associated to each which we can use to search the database. Since the database table is implemented as a hash table (where the indexes are the addresses) we use the addresses of the query to directly access the values at that address (index) in the database. This will return the list of couples stored at each matching address in the database. We can count how many returned couples belong to which songs and then compute how many of these returned couples form a full target zone (ie. five specific points need to have been returned for it to count, rather than just one match. This reduces false results coming through). The song with the largest number of target zones returned is likely to be the correct match however we can't be sure of this just yet since it's possible that the notes have matched in an order differently to that which they occur in the song. Hence we must check time coherency.

---

[4] 'How does Shazam work' 2015, Cristophe, accessed 27th February, <http://coding-geek.com/how-shazam-works/>

I have described the algorithm for checking the time coherency between matched notes on page 16. Once we have established a song in the database which has a high number of target zones in common with the record and also has these target zones occur in the correct order, we can quite safely say that this song is the correct match.

Data

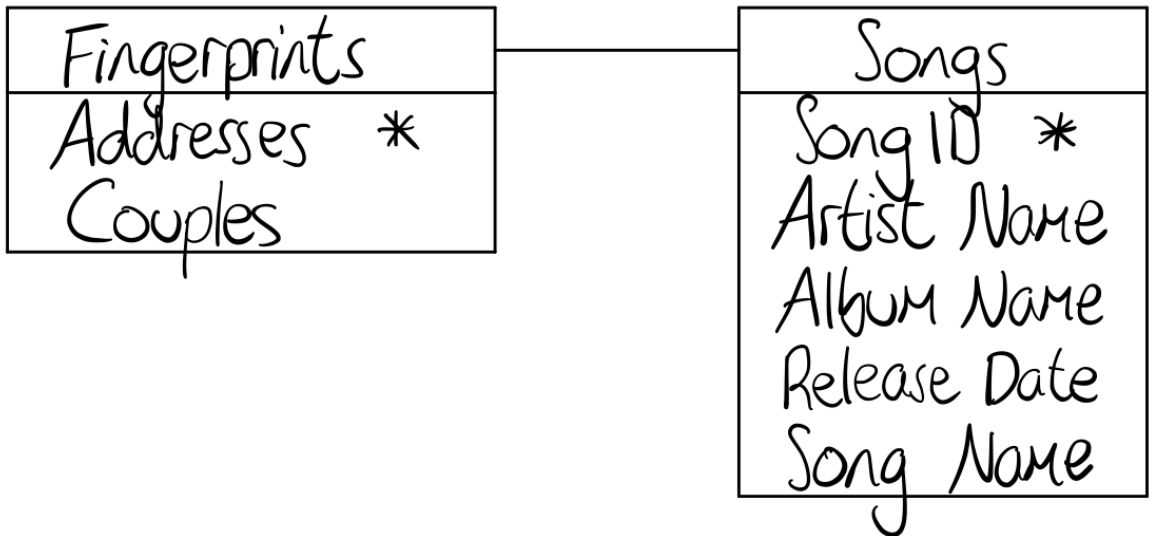
Here are various examples of some data types that are used in my software implementation

Data Structure	Example of use (these are variable names)
Hash Tables	fingerprintHashTable – This stores all of the precomputed fingerprints of every song in the database. The index is produced by a hashing function and is of the form of a 32-bit binary integer. The values stored in each index are 64-bit binary integers songMap – Maps a particular song ID to the metadata associated with that song. The index is a 32-bit binary integer representing the songID and the value is a list of various pieces of metadata about the song couplesHash – Counts the number of couples, belonging to particular songs, that are returned from a database search. The indexes are the song ID's and the values are the number of couples associated.
Arrays	audioData – The raw audio data captured by the microphone. It is an array of byte objects. downSampledAudio – Audio data with a sample rate 4 times lower than the originally recorded audio. The audio here is represented by unsigned integers now hammingWindow – A window function resembling a bell curve which is multiplied with each section of the audio individually before applying an FFT. It is an array of floats.
Lists	zeroPaddedAudio – Original audio data but with zeroes added to the end to make the total number of samples a power of two. windowedAudio – A list containing all the different sections of audio which have been multiplied by the Hamming window function. This is a list of lists of integers.

	audioFrequencyDomain – The audio data after having an FFT applied to it. In this list the audio data is in the frequency domain rather than the time domain. It's a list of integers powerfulFrequencies – Contains all the most powerful frequencies picked out from the spectrum produced by the FFTs extractedList – Holds the fingerprint data read from a text file and stores each line of text in an index of the list encodedAddress – The addresses of each audio point but in binary form encodedCouples – The couples of each audio point but in binary form
Text File	fingerprintText – Holds the fingerprint associated with one particular song
Stack	lastSongsMatched – A stack which holds the last five song matches that the user did.

Database Layout:

Currently my database isn't an 'actual' database, but it is a combination of hash tables. One of which stores addresses as the indexes and a list of couples as the value of each index, the other stores information on the songs and their metadata. A possible ER diagram showing this design could be (where the asterisk shows the primary key):



External Files Used:

Fingerprints of every song stored in the database are also stored in a text file, as is the user recorded audio, temporarily, before every search. This text file features two main columns, time and frequency, and is laid out as shown below:

```
0.6564172335600907 742.8955078125
0.659501133786848 1108.9599609375
0.672108843537415 2605.517578125
0.6783673469387755 3348.4130859375
0.6848072562358276 4112.841796875
0.7492970521541951 742.8955078125
0.7523809523809524 1108.9599609375
0.7652607709750567 2637.8173828125
0.7712471655328799 3348.4130859375
0.7775963718820862 4102.0751953125
0.8414512471655329 656.7626953125
0.8442630385487528 990.52734375
0.8581405895691611 2637.8173828125
0.8603174603174604 2896.2158203125
0.8692970521541951 3962.109375
0.9343310657596372 656.7626953125
0.9371428571428572 990.52734375
0.9510204081632654 2637.8173828125
0.9538321995464853 2971.58203125
0.9649886621315193 4295.8740234375
1.0272108843537415 656.7626953125
1.0300226757369615 990.52734375
1.0439002267573696 2637.8173828125
1.0522448979591836 3628.3447265625
1.1182766439909297 441.4306640625
1.1229024943310657 990.52734375
1.1367800453514738 2637.8173828125
1.2116099773242632 495.263671875
1.2157823129251701 990.52734375
1.2296598639455782 2637.8173828125
```

I have written a function which can read this data into a two-dimensional array where each element is an individual line of the file, and within each element, there is a list containing the time and the frequency. Hence when back in python, this same text file will appear as:

```
[[0.6564172335600907, 742.8955078125], [0.659501133786848, 1108.9599609375], [0.672108843537415, 2605.517578125],
 [0.6783673469387755, 3348.4130859375], [0.6848072562358276, 4112.841796875], [0.7492970521541951, 742.8955078125],
 [0.7523809523809524, 1108.9599609375], [0.7652607709750567, 2637.8173828125], [0.7712471655328799, 3348.413085937
5], [0.7775963718820862, 4102.0751953125], [0.8414512471655329, 656.7626953125], [0.8442630385487528, 990.52734375],
 [0.8581405895691611, 2637.8173828125], [0.8603174603174604, 2896.2158203125], [0.8692970521541951, 3962.109375],
 [0.9343310657596372, 656.7626953125], [0.9371428571428572, 990.52734375], [0.9510204081632654, 2637.8173828125],
 [0.9538321995464853, 2971.58203125], [0.9649886621315193, 4295.8740234375], [1.0272108843537415, 656.7626953125],
 [1.0300226757369615, 990.52734375], [1.0439002267573696, 2637.8173828125], [1.0522448979591836, 3628.3447265625],
 [1.1182766439909297, 441.4306640625], [1.1229024943310657, 990.52734375], [1.1367800453...
```

There are two other files which are saved to disk however these are hash tables therefore I have made use of the ‘pickle’ library which saves them to a file as unreadable binary data. I can read this data back into a python hash table by using the inverse, ‘unpickle’, method.

Algorithms

Fourier Transform

Every real-world sound is comprised of many different sine waves of varying frequencies. The Fourier Transform is an algorithm for splitting these sounds into their individual sine waves. The mathematical formula behind this algorithm is:

$$X(n) = \sum_{k=0}^{N-1} x[k] e^{-j(2\pi kn/N)}$$

Where N is the size of the window (the number of samples of the audio processed in one particular cycle of the loop), X(n) is the nth bin of frequencies returned, x[k] is the kth sample of the audio signal.

There are lots of different algorithmic implementations of this formula, but I settled on using the most common one which is the 'radix-2 DIT Cooley-Tukey FFT Algorithm'. It is recursively defined and runs much faster than just coding the formula exactly as it is presented (which I did initially). The algorithm goes as follows:

- Given a number of samples of audio for processing, divide them into two lists. One list holds all of the even index samples and the other holds the odd index samples.
- Recursively divide the resulting lists in the same way, ie. the even index list will itself be divided into odd and even index lists, and the same goes for the odd index list. The recursive base case is when the lists are of length 1.
- Now we apply the above formula to each of the odd and even lists individually and stick them together. This new list is now returned and we work back up the recursive stack computing the Fourier Transform of each new half-list and then gluing them together until we reach the top, where we will finally have the full Transform of the original audio.

The pseudocode for this algorithm would be as follows:

```
FourierTransform (audioSamples, numberOfSamples):
    IF numberOfSamples = 1 then
        RETURN audioSamples
    ELSE
        // NOTE: [::2] returns all even indexes in a list in python
        evenIndexes ⟵ FourierTransform(audioSamples[::2], numberOfSamples/2)
        oddIndexes ⟵ FourierTransform(audioSamples[1::2], numberOfSamples/2)

        FOR i ⟵ 1 to numberOfSamples/2
            append (evenIndexes[i] + exp(-2i*π/numberOfSamples) * oddIndexes[i]) to firstHalf
            append (evenIndexes[i] - exp(-2i*π/numberOfSamples) * oddIndexes[i]) to secondHalf
        ENDFOR
    ENDIF
    RETURN (concatenate firstHalf and secondHalf)
```

Spectrogram Analysis

This algorithm is used on the audio once we have applied the Fourier Transform which changes the audio into the frequency domain from the time domain. It's used to find the most powerful frequencies present in the spectrum. The algorithm is:

- Sort the frequency bins (Fourier Transform outputs discrete frequencies hence groups them into 'bins' of say: bin1 = 5-10Hz, bin2 = 10-15Hz, …), from each Fourier Transform result, into eight logarithmic bands.
- For each band, locate and store the strongest bin of frequencies
- Calculate the average value of the eight powerful frequency bins for every output in the entire song. Then average these averages to obtain just one mean number.
- Now keep only the bins from the eight powerful ones (per FFT result) that are above this mean value (multiplied by a coefficient)

Pseudocode for this algorithm:

```
LocatePowerfulFrequencies(audioData):
    frequencyBands ⟵ new list
    FOR j ⟵ 0 to length(audioData)
        FOR i ⟵ 0 to 8
            IF i = 0: append indexes 0 to 10 of audioData[j] to frequencyBands[j]
            IF i = 1: append indexes 10 to 20 of audioData[j] to frequencyBands[j]
            IF i = 2: append indexes 20 to 40 of audioData[j] to frequencyBands[j]
            IF i = 3: append indexes 40 to 80 of audioData[j] to frequencyBands[j]
            IF i = 4: append indexes 80 to 160 of audioData[j] to frequencyBands[j]
            IF i = 5: append indexes 160 to 250 of audioData[j] to frequencyBands[j]
            IF i = 6: append indexes 250 to 350 of audioData[j] to frequencyBands[j]
            ELSE: append indexes 350 to 512 of audioData[j] to frequencyBands[j]
        ENDFOR
    ENDFOR

    strongestBinPerBand ⟵ new list
    FOR j ⟵ 0 to length(frequencyBands)
        FOR i ⟵ 0 to length(frequencyBands[j])
            append (frequencyBands[j][i] sorted in descending order)[0] to strongestBinPerBand
        ENDFOR
    ENDFOR

    averageBinValue ⟵ sum(every element of strongestBinPerBand) / (length(strongestBinPerBand)*8)

    powerfulFrequencyBins ⟵ new list
    FOR j ⟵ 0 to length(strongestBinPerBand)
        tempList ⟵ new list
        FOR i ⟵ 0 to length(strongestBinPerBand[j])
            IF strongestBinPerBand[j][i][0] > averageBinValue
                append strongestBinPerBand[j][i] to tempList
        append tempList to powerfulFrequencyBins

    RETURN powerfulFrequencyBins
```

Checking Time Coherency Of Matching Notes:

When we have found some notes that appear to match between the database song and the query it's important to then check whether these notes appear in the order that they do in the query, else it's probably the wrong song. The algorithm I have used for achieving this is:

- With every note stored in the database, we also know what time it occurs during the song. We can compute a list of delta times between when a matching note occurs in the database song with when it occurs in the query by simply doing 'time note occurs in database song – time note occurs in query'
- For every different value in the list of delta times, we count how many times it is present in the list.
- We keep the delta time with the maximum number of occurrences because this is telling us that for this particular delta time, we are getting the most notes that match between the query and the database.
- If this number of matching notes is above a certain threshold then this must be the correct song.

Pseudocode for the algorithm:

```
CheckTimeCoherency(listOfDeltaTimes):
        songIDHash ⟵ new dictionary
        FOR i ⟵ 1 to length(listOfDeltaTimes)
                FOR j ⟵ 1 to length(listOfDeltaTimes[i])
                        IF currentSongID is already a key in songIDHash then
                                append currentDeltaTime to songIDHash[currentSongID]
                        ELSE
                                songIDHash[currentSongID] = currentDeltaTime
                ENDFOR
        ENDFOR

        greatestDeltaTime ⟵ new dictionary
        FOR key in songIDHash
                highestFrequency ⟵ number of occurences of most prevalent item
                greatestDeltaTime[key] ⟵ highestFrequency
        ENDFOR

        FOR key in greatestDeltaTime
                IF overallHighestFrequencyValue < greatestDeltaTime[key]
                        overallHighestFrequencyValue ⟵ greatestDeltaTime[key]
                        IF overallHighestFrequencyValue > threshold
                                matchedSongID ⟵ key
                        ENDIF
                ENDIF
        ENDFOR
        RETURN matchedSongID
```

Merge Sort

I used a merge sort for ordering the frequency bins, from highest amplitude to lowest amplitude, after applying the Fourier Transform to the audio so that finding the frequencies with the greatest amplitude was easier.
Here is the pseudocode for my merge sort:

```
MergeSort(list):
        IF length(list) = 1
                RETURN list
        ENDIF

        firstHalf ⟵ list[:length(list)/2]
        secondHalf ⟵ list[length(list)/2:]

        firstHalf ⟵ MergeSort(firstHalf)
        secondHalf ⟵ MergeSort(secondHalf)

        L ⟵ new list
        WHILE (length(firstHalf) ≠ 0) and (length(secondHalf) ≠ 0)
                IF firstHalf[0] > secondHalf[0] then
                        append firstHalf[0] to L
                        delete index 0 from firstHalf
                ELSE
                        append secondHalf[0] to L
                        delete index 0 from secondHalf
                ENDIF
        ENDWHILE

        IF length(firstHalf) = 0 then
                FOR  i ⟵ 0 to length(secondHalf)
                        append secondHalf[i] to L
                ENDFOR
        ELSE
                FOR i ⟵ 0 to length(firstHalf)
                        append firstHalf[i] to L
                ENDFOR
        ENDIF

        RETURN L
```