In [2]:

```
from music21 import converter, instrument, note, chord, stream
import glob
import pickle
import numpy as np
```

Read a Midi File

In [3]:

```
song1 = converter.parse("midi_songs/8.mid")
print(type(song1))
```

```
<class 'music21.stream.base.Score'>
```

In [4]:

```
song1
```

Out[4]:

```
<music21.stream.Score 0x172ce96bdf0>
```

In [5]:

```
# song1 --> object of stream.Score type
#       --> will contain music in form of notes and chords
song1.show('midi')
# This will show the song in playable format
```

In [6]:

```
# song1.show('text')
# This will show the song in text-format (notes & chords)
```

In [7]:

```
# So, the chords and notes are stored in nested forms of containers
# .. to simplify this, store all of them in a single list
# ==> Flatten the elements.
elements_of_song = song1.flat.notes
```

In [8]:

```
print(len(elements_of_song))
print(elements_of_song)
print(type(elements_of_song))
```

```
336
<music21.stream.iterator.StreamIterator for Score:0x172cd8bddc0 @:0>
<class 'music21.stream.iterator.StreamIterator'>
```

In [9]:

```
count = 0
print("Following are some elements of song :-")
for e in elements_of_song:
    print(e, e.offset, type(e))
    count += 1
    if count > 7:
        break
    # e.offset --> will tell the time-duration of element
```

```
Following are some elements of song :-
<music21.note.Note C> 0.0 <class 'music21.note.Note'>
<music21.chord.Chord C5 E4> 0.0 <class 'music21.chord.Chord'>
<music21.note.Note C> 0.0 <class 'music21.note.Note'>
<music21.note.Note C> 0.0 <class 'music21.note.Note'>
<music21.chord.Chord C5 E4> 0.0 <class 'music21.chord.Chord'>
<music21.note.Note C> 0.0 <class 'music21.note.Note'>
<music21.note.Note G> 1.5 <class 'music21.note.Note'>
<music21.note.Note G> 5/3 <class 'music21.note.Note'>
```

Get the Notes & Chords from the Song

In [10]:

```
elex = elements_of_song[0]
ele2 = elements_of_song[4]
# isinstance(element, classType)
# If the element and its class match with classType --> this returns True (else False)
flag1a = isinstance(elex, note.Note)
flag1b = isinstance(elex, chord.Chord)
flag2a = isinstance(ele2, note.Note)
flag2b = isinstance(ele2, chord.Chord)
print(flag1a, flag1b, flag2a, flag2b)
```

```
True False False True
```

Processing a Note :-

In [11]:

```
note1 = elements_of_song[3]
print(note1.pitch)
print(type(note1))
# This gives the note in form of a class
```

```
print(type(note1.pitch))
# Get the string from the class
currNote = str(note1.pitch)
print(currNote)
# This will recover the note-name from class
```

```
C5
<class 'music21.note.Note'>
<class 'music21.pitch.Pitch'>
C5
```

Processing a Chord :-

In [12]:

```
chord1 = elements_of_song[1]
print(chord1)
print(type(chord1))
# This is a chord, let's figure this out.. how to process this
print(chord1.normalOrder)
# chord.normalOrder --> Gives the list of nodes in it.
# 2 --> A4
# 6 --> D5
# 9 --> F#4
# (Following some pattern of indexing.. have to figure it out)
print(type(chord1.normalOrder))
# Convert the chord-list into a string, concatenated with "+"
currChord = "+".join(str(x) for x in chord1.normalOrder)
print(currChord)
```

```
<music21.chord.Chord C5 E4>
<class 'music21.chord.Chord'>
[0, 4]
<class 'list'>
0+4
```

Making a list, only of Notes (from Notes) OR (from Chords)

In [13]:

```
notes_of_song = []
# Empty array container for notes & chords
for ele in elements_of_song:
    # If element is a note, store it's pitch
    if(isinstance(ele, note.Note) == True):
        tempNote = str(ele.pitch)
        notes_of_song.append(tempNote)
    elif(isinstance(ele, chord.Chord) == True):
    # Else, element is a chord, split notes, and make string of them
        tempChord = "+".join(str(x) for x in ele.normalOrder)
        notes_of_song.append(tempChord)
```

In [14]:

```
print("No. of notes/chords =", len(notes_of_song))
print("Some elements of notes_of_song array are :-")
count = 0
for note1 in notes_of_song:
    print(note1)
    count += 1
    if count > 7:
        break
print("...")
```

```
No. of notes/chords = 336
Some elements of notes_of_song array are :-
C5
0+4
C2
C5
0+4
C2
G4
G4
...
```

Get All the Notes, from all the Midi Files

In [16]:

```
notes = []
count = 0
for file in glob.glob("midi_songs/*.mid"):
    midi = converter.parse(file) # Convert file into stream.Score Object
    if count < 10:
        print("parsing %s"%file)
    elements_to_parse = midi.flat.notes
    count += 1

    for elex in elements_to_parse:
        # If the element is a Note,  then store it's pitch
        if(isinstance(elex, chord.Chord) == True):
            notes.append("+".join(str(n) for n in elex.normalOrder))
        elif(isinstance(elex, note.Note) == True):
            noteString = str(elex.pitch)
            notes.append(noteString)
            # If the element is a Chord, split each note of chord and join them with +
print("...")
```

```
parsing midi_songs\0fithos.mid
parsing midi_songs\8.mid
parsing midi_songs\ahead_on_our_way_piano.mid
parsing midi_songs\AT.mid
parsing midi_songs\balamb.mid
```

```
parsing midi_songs\balamb.mid
parsing midi_songs\bcm.mid
parsing midi_songs\BlueStone_LastDungeon.mid
parsing midi_songs\braska.mid
parsing midi_songs\caitsith.mid
parsing midi_songs\Cids.mid
...
```

In [17]:

```
print("Length of notes-array = ", len(notes))
print("Some elements of note array :-")
count = 0
for n in notes:
    print(n)
    count += 1
    if count > 7:
        break
print("...")
```

```
Length of notes-array =  60764
Some elements of note array :-
4+9
E2
4+9
4+9
4+9
4+9
4+9
4+9
4+9
11+4
4+9
```

Saving the file, containing all Notes

In [18]:

```
import pickle

with open("notes", 'wb') as filepath:
    pickle.dump(notes, filepath)
```

In [19]:

```
# 'wb' --> Write-binary mode (to write data in a file)
# 'rb' --> Read-binary mode (to read data from a file)
with open("notes", 'rb') as f:
    notes = pickle.load(f)
    # This will load whole file-data to variable notes
```

Count of Unique Elements in Music :-

In [21]:

```
# In 'wb' and 'rb', same file needs to be referenced.
# Else, Will give error --> "Ran out of data".
print(len(set(notes)))
# This will print unique no. of elements.
# i.e. --> Unique notes/chords in all files.
numElements = len(set(notes))
```

```
398
```

Preparing Sequenctial Data for LSTM :-

In Markov chain, we have a window size. So choosing a sequence length. This length also states, how many elements are considered in a LSTM layer.

In [23]:

```
sequenceLength = 100
# Will give 100 elements to a layer, and will predict output for next layer using them.
uniqueNotes = sorted(set(notes))
countNodes = len(uniqueNotes)
print("No. of elements in uniqueNotes = ", len(uniqueNotes))
print("Some elements of uniqueNotes array are :-")
count = 0
for ele in uniqueNotes:
    print(ele)
    count += 1
    if count > 7:
        break
print("...")
```

```
Some elements of uniqueNotes array are :-
0
0+1
0+1+3
0+1+5
0+1+6
0+2
0+2+3+7
0+2+4+5
398
```

Mapping Strings (unique-elements) to Integer values :-

In [24]:

```
# As ML models work with numerial data only, will map each string with a number.
```

```
noteMap = dict((ele, num) for num, ele in enumerate(uniqueNotes))
count = 0
for ele in noteMap:
    print(ele, " : ", noteMap[ele])
    count += 1
    if count > 7:
        break
print("...")
```

```
0  :  0
0+1  :  1
0+1+3  :  2
0+1+5  :  3
0+1+6  :  4
0+2  :  5
0+2+3+7  :  6
0+2+4+5  :  7
0+2+4+7  :  8
0+2+5  :  9
0+2+6  :  10
```

--> As sequenceLength is 100, will take first 100 data to input, and 101st data as output. --> For next iteration, take (2-101) data points as input, and 102nd data as output. --> So on... Sliding window (of size 100) as input, & next 1 data as output.

--> So, total we will get (len(notes) - sequenceLength) datapoints.

In [25]:

```
networkInput = [] # input-data
networkOutput = [] # will try to get output, using input
for i in range(len(notes) - sequenceLength):
    inputSeq = notes[i : i+sequenceLength] # 100 string-values
    outputSeq = notes[i + sequenceLength] # 1 string-value
    # Currently, inputSeq & outputSeq has strings.
    # Use map, to convert it to integer-values.
    # ..as ML-algorithm works only on numerical data.
    networkInput.append([noteMap[ch] for ch in inputSeq])
    networkOutput.append(noteMap[outputSeq])
```

In [26]:

```
print(len(networkInput))
print(len(networkOutput))
```

```
60664
60664
```

Create ready-data for Neural Network :-

In [27]:

```
import numpy as np
```

In [28]:

```
# n_patterns = int(len(networkInput)/100)
# No. of rows divided by 100.. as 100 columns, so Distributing data in 3-D format
n_patterns = len(networkInput)
networkInput = np.reshape(networkInput, (n_patterns, sequenceLength, 1))
# LSTM recieves input data in 3-dimensions
print(networkInput.shape)
```

```
(60664, 100, 1)
```

Normalize this data

In [29]:

```
# As the values are from 0 - uniqueNodes
# For better precision, converting data in range [0 - 1]
normNetworkInput = networkInput / float(numElements)
print("Some elements of normNetworkInput[0] array :-")
count = 0
for ele in normNetworkInput[0]:
    print(ele)
    count += 1
    if count > 10:
        break
print("...")
```

```
Some elements of normNetworkInput[0] array :-
[0.48743719]
[0.92713568]
[0.48743719]
[0.48743719]
[0.48743719]
[0.48743719]
[0.48743719]
[0.48743719]
[0.48743719]
[0.2638191]
[0.48743719]
```

In [30]:

```
# Now, values are in range [0 - 1]
# Network output are the classes, encoded into 1-vector
```

In [32]:

```
from keras.utils import np_utils
```

In [34]:

```
networkOutput = np_utils.to_categorical(networkOutput)
print(networkOutput.shape)
# This will convert output-data to a 2-D format
# In which each key(old-output value) has 229 categorical values
# And, the one which matches has some kind of flag marked to it.
```

```
(60664, 398)
```

Create Model

Download & Import Packages

In [35]:

```
from keras.models import Sequential
from keras.layers import *
from keras.callbacks import ModelCheckpoint, EarlyStopping
```

Creating a Sequential Model :-

In [38]:

```
model = Sequential()
```

Adding Layers to the Model :-

In [39]:

```
# And, this model has first layer as LSTM layer.
model.add(LSTM(units=512, input_shape=(normNetworkInput.shape[1], normNetworkInput.shape[2]), return_sequences=True))
# As this is the 1st layer, so we need to provide the input-shape (in argument)
# Here we are passing (100,1) as input_shape, as all data-points have shape (100,1)
# Also, we have to do return_sequences=True, as this isn't the last layer, also have further layers.

# After the 1st layer, adding a Dropout
model.add(Dropout(0.3))

# Also adding another LSTM layer.
model.add(LSTM(512, return_sequences=True))
# as this is also not the last layer.. return_sequences=True

# Again adding a Dropout
model.add(Dropout(0.3))

# And, now 1-more LSTM layer.
model.add(LSTM(512))

# And, adding a Dense-layer.
model.add(Dense(256))

# Again adding a Dropout.
model.add(Dropout(0.3))

# Now, the final layer.
#   (Adding dense layer with no. of neurons = countNodes)
#   (Also having an "softmax" activation function)
model.add(Dense(numElements, activation="softmax"))
```

Compiling the model :-

In [40]:

```
model.compile(loss="categorical_crossentropy", optimizer="adam")
# loss="categorical_crossentropy" --> since it has 229 classes.
# Not specifying any metrics (like accuracy), as it would not be a good metrics to evaluate.
```

This is our Model :-

In [42]:

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 100, 512)          1052672

 dropout (Dropout)           (None, 100, 512)          0

 lstm_1 (LSTM)               (None, 100, 512)          2099200

 dropout_1 (Dropout)         (None, 100, 512)          0

 lstm_2 (LSTM)               (None, 512)               2099200

 dense (Dense)               (None, 256)               131328

 dropout_2 (Dropout)         (None, 256)               0

 dense_1 (Dense)             (None, 398)               102286

=================================================================
Total params: 5,484,686
Trainable params: 5,484,686
Non-trainable params: 0
_________________________________________________________________
```

Training the Model :-

In [41]:

```
import tensorflow as tf
```

In [52]:

```
# (Entire code commented out, to prevent created model, from starting fit again, and old work getting wasted)
# Creating callbacks for fitting model.

checkpoint = ModelCheckpoint("model3.hdf5", monitor='loss', verbose=0, save_best_only=True, mode='min')
# 1st arg  -->  where the model will be saved
# 2nd arg  -->  We have to monitor the loss
# 5th arg  -->  As monitoring loss, so mode = "min", as loss should be minimum.

# We can also create an earlystopping callback, but lets only keep the checkpoint.

# Fitting the model :-

model_his = model.fit(normNetworkInput, networkOutput, epochs=10, batch_size=64, callbacks=[checkpoint])
# No. of epochs = 10 (trying for model3)
# batch size = 64
# No. of epochs = 100 (for model4 .. trained in google-colab)
# Then imported that model to this file...
```

```
Epoch 1/10
948/948 [==============================] - 4003s 4s/step - loss: 4.8030
Epoch 2/10
948/948 [==============================] - 3837s 4s/step - loss: 4.7745
Epoch 3/10
948/948 [==============================] - 3842s 4s/step - loss: 4.7721
Epoch 4/10
948/948 [==============================] - 3839s 4s/step - loss: 4.7687
Epoch 5/10
948/948 [==============================] - 12148s 13s/step - loss: 4.7669
Epoch 6/10
948/948 [==============================] - 3739s 4s/step - loss: 4.7658
Epoch 7/10
948/948 [==============================] - 3994s 4s/step - loss: 4.7650
Epoch 8/10
948/948 [==============================] - 3497s 4s/step - loss: 4.7646
Epoch 9/10
948/948 [==============================] - 3606s 4s/step - loss: 4.7646
Epoch 10/10
948/948 [==============================] - 3747s 4s/step - loss: 4.7644
```

Load Model :-

In [47]:

```
from keras.models import load_model
```

In [48]:

```
model = load_model("model4.hdf5")
```

Predictions :-

In [49]:

```
sequenceLength = 100
```

In [50]:

```
networkInput = [] # input-data
for i in range(len(notes) - sequenceLength):
    inputSeq = notes[i : i+sequenceLength] # 100 string-values
    # Currently, inputSeq & outputSeq has strings.
    # Use map, to convert it to integer-values.
    # ..as ML-algorithm works only on numerical data.
    networkInput.append([noteMap[ch] for ch in inputSeq])
```

In [84]:

```
print("Some elements of networkInput[0] array :-")
count = 0
for ele in networkInput[0]:
    print(ele)
    count += 1
    if count > 7:
        break
print("...")
```

```
Some elements of networkInput[0] array :-
194
369
194
194
194
194
194
194
...
```

In [52]:

```
print(len(networkInput[300]))
```

```
100
```

In [53]:

```
# Each data-point has 100-elements (in networkInput)
# We will give these 100-elements as input, & it will generate 1-output.
# Will add this 1-output in input, & discard oldest element from input. (again getting to 100 input-elements)
# This way, we will keep predicting 1-element each time.
```

In [54]:

```
startIdx = np.random.randint(len(networkInput)-1)
# This will get any random data-point-index from the input-data
# Data at each random data-point-index means --> 100 elements.
```

In [55]:

```
print(startIdx)
```

```
7121
```

In [56]:

```
networkInput[startIdx]
print("Some elements of networkInput[startIdx] array are :-")
count = 0
for ele in networkInput[startIdx]:
    print(ele)
    count += 1
    if count > 7:
        break
print("...")
```

```
Some elements of networkInput[startIdx] array are :-
389
371
256
256
274
371
274
371
...
```

In [57]:

```
# Above 100-element np-array, is the start sequence.
# Right now, we have :-
# element --> integer  mapping
# What is also required is :-
# integer --> element  mapping.
```

In [58]:

```
intNoteMap = dict((num,ele) for num,ele in enumerate(uniqueNotes))
# This will have  (integer --> element) mapping.
# uniqueNotes --> has all unique-elements
# noteMap --> has  (element --> integer) mapping.
# countNodes --> count of unique-elements.

# print(intNoteMap)
# Commented, as this is just "integer" --> "music-element" mapping.
```

Generate Input-music in playable format :-

In [59]:

```
# Taking the initial input-index pattern
pattern = networkInput[startIdx]
predictionOutput = []
```

In [60]:

```
inputMusicElements = []
inputMusic = []
inputMusic = (intNoteMap[ele] for ele in pattern)
```

In [61]:

```
offset = 0
# offset --> instance-time of particular element (note/chord)
# Have to iterate over all elements of predictionOutput
#   --> Checking whether is a note or chord ?

for element in inputMusic:
    # If element is a chord :-
    if('+' in element) or element.isdigit():
        # Possibilites are like '1+3' or '0'.
        notesInChord = element.split('+')
        # This will get all notes in chord
        tempNotes = []
        for currNote in notesInChord:
            # Creating note-object for each note in chord
            newNote = note.Note(int(currNote))
            # Set it's instrument
            newNote.storedInstrument = instrument.Piano()
            tempNotes.append(newNote)
        # This chord can have x-notes
        # Create a chord-object from list of notes
        newChord = chord.Chord(tempNotes)
        # Adding offset to chord
        newChord.offset = offset
        # Add this chord to music-elements
        inputMusicElements.append(newChord)
    # If element is a note :-
```

```
    else:
        # We know that this is a note
        newNote = note.Note(element)
        # Set off-set of note
        newNote.offset = offset
        # Set the instrument of note
        newNote.storedInstrument = instrument.Piano()
        # Add this note to music-elements
        inputMusicElements.append(newNote)
    offset += 0.5
    # Fixing the time-duration of all elements
```

In [62]:

```
# For playing them, have to create a stream-object
#     ..from the generated music-elements
midiInputStream = stream.Stream(inputMusicElements)
# Write this midiStream on a midi-file.
midiInputStream.write('midi', fp="testInput6.mid")
# 1st arg  -->  File-type
# 2nd arg  -->  File-name
```

Out[62]:

```
'testInput6.mid'
```

In [63]:

```
midiInputStream.show('midi')
```

Making Prediction :-

In [64]:

```
# Trying to generate (numIteration)-elements of music
numIteration = 1
# Try with different count variations, so named a variable
for noteIdx in range(numIteration):
    predictionInput = np.reshape(pattern, (1,len(pattern), 1))
    # reshaping into (1, 100, 1)
    # 1st argument --> count of data-points (batch-size)
    # As we have, 1-data of 100-length (2nd argument)
    # 3rd argument --> Because LSTM supports data in 3-dimension.
    # Also to predict over it, normalization is required (values between [0,1])
    predictionInput = predictionInput / float(countNodes)
    # Making prediction
    prediction = model.predict(predictionInput, verbose=0)
```

In [65]:

```
print("Some elements of prediction[0] are :-")
count = 0
for ele in prediction[0]:
    print(ele)
    count += 1
    if count > 7:
        break
print("...")
```

```
Some elements of prediction[0] are :-
0.00014231501
2.4924768e-07
6.4923205e-09
8.351073e-09
8.048707e-05
3.8892896e-08
5.966037e-08
1.9639047e-07
...
```

Analyzing Prediction :-

In [66]:

```
# Let's see, what our model has predicted
print("Measures of Dispersion of data :- \n")
print("Minimum value = ", np.amin(prediction))
print("Maximum value = ", np.amax(prediction))
print("Range of values = ", np.ptp(prediction))
print("Variance = ", np.var(prediction))
print("Standard Deviation = ", np.std(prediction))
print("Length of 1st Prediction-element = ", len(prediction[0]))
print("Count of unique elements = ", countNodes)
```

```
Measures of Dispersion of data :-

Minimum value =  2.7809166e-09
Maximum value =  0.273773
Range of values =  0.273773
Variance =  0.00040113291
Standard Deviation =  0.020028302
Length of 1st Prediction-element =  359
Count of unique elements =  398
```

In [67]:

```
# The values are in range [0,1].
# And, no. of values in 1st prediction are equal to the no. of unique elements we have.
# So --> it is clear that this has give the probabilities of all unique-elements.
# So, taking the element with max. probability
```

Again making prediction, with further processing

In [68]:

```
# Trying to generate (numIteration)-elements of music
numIteration = 200
# This time trying a larger no. of iterations.

for noteIdx in range(numIteration):
    predictionInput = np.reshape(pattern, (1,len(pattern), 1))
    predictionInput = predictionInput / float(countNodes)
    # Making prediction
    prediction = model.predict(predictionInput, verbose=0)
    # Taking the element with max. probability
    idx = np.argmax(prediction)
    # No. corresponding to max. probability element
    # The element corresponding to no. (idx) is :
    result = intNoteMap[idx]
    # Appeding this element to prediction-array
    predictionOutput.append(result)
    # Change input-sequence for further predictions
    # Add this into input, & discard the oldes one.
    pattern.append(idx)
    # slicing out the oldest element (0th index)
    pattern = pattern[1:]
    # Size of pattern remained constant at 100.
    #     (as added 1 element, & removed 1)
```

In [69]:

```
print("No. of elements in predictionOutput = ", len(predictionOutput))
print("Some elements of predictionOutput array are :-")
count = 0
for ele in predictionOutput:
    print(ele)
    count += 1
    if count > 7:
        break
print("...")
```

```
No. of elements in predictionOutput =  200
Some elements of predictionOutput array are :-
9+11+3
B-4
B3
D3
B-2
9+10+3
B-2
C7
...
```

Generate Music out of Predicted-data :

What required is to get a Midi File :-

In [71]:

```
outputMusicElements = []
# Array to store notes & chords.
```

Trying to Create a Note (from string) :-

In [72]:

```
tempStr = 'C4'
# Just copying from the predictionOutput display
# Creating a note-object (using note-package)
note.Note(tempStr)
```

Out[72]:

```
<music21.note.Note C>
```

In [73]:

```
# Music-note is generated.
# Similarly we can do for multiple elements.
newNote = note.Note(tempStr)
# Also, the note will have a off-set (timing)
# By default, offset was 0. (setting it manually here)
newNote.offset = 0
# And, the note will have an instrument
#    Can set, using storedInstrument package
newNote.storedInstrument = instrument.Piano()
# outputMusicElements.append(newNote)
# Above element is commented out, as it will get unwanted music like this random created note.
```

In [74]:

```
print(newNote)
```

```
<music21.note.Note C>
```

Creating Music-Elements from String-array :-

In [75]:

```
offset = 0
# offset --> instance-time of particular element (note/chord)
# Have to iterate over all elements of predictionOutput
#   --> Checking whether is a note or chord ?
```

```
for element in predictionOutput:
    # If element is a chord :-
    if('+' in element) or element.isdigit():
        # Possibilites are like '1+3' or '0'.
        notesInChord = element.split('+')
        # This will get all notes in chord
        tempNotes = []
        for currNote in notesInChord:
            # Creating note-object for each note in chord
            newNote = note.Note(int(currNote))
            # Set it's instrument
            newNote.storedInstrument = instrument.Piano()
            tempNotes.append(newNote)
        # This chord can have x-notes
        # Create a chord-object from list of notes
        newChord = chord.Chord(tempNotes)
        # Adding offset to chord
        newChord.offset = offset
        # Add this chord to music-elements
        outputMusicElements.append(newChord)
    # If element is a note :-
    else:
        # We know that this is a note
        newNote = note.Note(element)
        # Set off-set of note
        newNote.offset = offset
        # Set the instrument of note
        newNote.storedInstrument = instrument.Piano()
        # Add this note to music-elements
        outputMusicElements.append(newNote)
    offset += 0.5
    # Fixing the time-duration of all elements
```

In [76]:

```
print("No. of elements in outputMusicElements = ", len(outputMusicElements))
print("Some elements of outputMusicElement array are :-")
count = 0
for ele in outputMusicElements:
    print(ele)
    count += 1
    if count > 7:
        break
print("...")
```

```
No. of elements in outputMusicElements =  200
Some elements of outputMusicElement array are :-
<music21.chord.Chord A B E->
<music21.note.Note B->
<music21.note.Note B>
<music21.note.Note D>
<music21.note.Note B->
<music21.chord.Chord A B- E->
<music21.note.Note B->
<music21.note.Note C>
...
```

Trying to Play the Output Music :-

In [77]:

```
# For playing them, have to create a stream-object
#    ..from the generated music-elements
midiStream = stream.Stream(outputMusicElements)
# Write this midiStream on a midi-file.
midiStream.write('midi', fp="testOutput6.mid")
# 1st arg  -->  File-type , 2nd arg  -->  File-name
```

Out[77]:

```
'testOutput6.mid'
```

Loading the output-midi file :

In [78]:

```
midiStream.show('midi')
# Show the music in playable-format
```

In [79]:

```
outputMusic6 = converter.parse("testOutput6.mid")
print(type(outputMusic6))
```

```
<class 'music21.stream.base.Score'>
```

In [80]:

```
outputMusic6.show('midi')
# This will show the music in playable-format
```

Plotting inputMusicElements VS outputMusicElements :

In [81]:

```
import matplotlib.pyplot as plt
```

In [82]:

```
inputMusicNums = networkInput[startIdx]
```

```
print("Some elements of inputMusicNums :-")
count = 0
for ele in inputMusicNums:
    print(ele)
    count += 1
    if count > 7:
        break
print("...")
```

```
Some elements of inputMusicNums :-
389
371
256
256
274
371
274
371
...
```

In [83]:

```
outputMusicNums = []
for ele in predictionOutput:
    outputMusicNums.append(noteMap[ele])
print("Some elements of outputMusicNums are :-")
count = 0
for ele in outputMusicNums:
    print(ele)
    count += 1
    if count > 7:
        break
print("...")
```

```
Some elements of outputMusicNums are :-
320
333
338
357
331
311
331
354
...
```

Plot inputMusicNums VS outputMusicNums (prediction visualization) :-

In [ ]:

```

```