Chunking Reference Manual

Version 3.3
Music Analysis and Algorithmic Composition in C++

Georg Boenn[1]

July 28, 2018

[1]https://github.com/gboenn/chunking

Contents

Preface

Background info on *chunking*. What it is there for and where it is coming from. Where is it going.

License

Chunking has been released under the GNU Public License, Version 3, which can be obtained from `https://www.gnu.org/licenses/gpl.html`.

Structure of the manual

Following this preface, instructions for building the software, as well as a brief overview about music analysis and algorithmic composition is presented in the introduction.

In chapter 2, the manual explains the main concepts that were implemented in `chunking`. Lilypond is used for score typesetting [1], Csound is used for sound synthesis [2]

`Chunking` makes use of a special shorthand notation for rhythm that serves as an intermediate data format to direct output to both backends, and in order to edit generated output, and to input it again into other functions that chunking is offering.

The Burrows-Wheeler Transform is originally a method for preparing files for compression. It is used also in bio-informatics to search for sequences within a genome. In `chunking`, it lies at the heart of many of the analysis methods offered for rhythms. It is not only applied to musical rhythms, but it can also be used for pitch sequences and ordinary text input.

The Christoffel words are part of an area in mathematics called *combinatorics on words.* Christoffel words are related to the so-called Euclidean rhythms, which received much attention in computer music in recent years. Chunking offers the generation of Christoffel rhythms, the search for a rhythmic pattern within Christoffel words, as well as a comparison between two Christoffel words in order to determine their degree of similarity.

[1]`http://lilypond.org`
[2]`http://csound.github.io`

Integer partitions offer powerful insights into musical rhythms and metres. In addition to the generation and filtering of integer partitions, `Chunking` offers an algorithm that produces the rhythmic structure of an entire musical sentence, with phrases and patterns, down to its constituent elements, the rhythmic chunks.

Finally, the Farey Sequence is offering access to the area of subdivisions, proportions and to the complex world of polyrhythms.

The third chapter of this manual gives you a reference to all the methods that can be called with `chunking` in alphabetical order. It includes many examples and detailed instructions on how to use these different functions.

Online Resources

To download the latest version, go to
`https://github.com/gboenn/chunking`.
For building instructions, see section 1.1.

Acknowledgements

The shorthand notation used by `chunking` is an extension of the work initiated by Peter Giger who invented so-called *rhythmoglyphs* to notate complex arrangements of simple base rhythms.

The partition algorithm in chunking uses a routine from John Burkardt's library SUBSET released under the GNU LGPL licence (`https://people.sc.fsu.edu/~jburkardt/cpp_src/subset/subset.html`).

Georg Boenn
`http://www.boenn.org/`

1

Introduction

1.1 How to build chunking from sources

Download `chunking` from https://github.com/gboenn/chunking or, from within the shell, type:
`git clone https://github.com/gboenn/chunking.git`

Then, do the following steps:

1. open terminal
2. cd into source code directory
3. make
4. If build succeeds: make install

With make install, `chunking` is copied into `/usr/local/bin`. Support data files in ascii text format are copied into `/usr/local/share/chunking` To uninstall `chunking` and its resources, use: `make uninstall`. The user needs write permissions for `make install` and `uninstall`.

1.2 Music Analysis

The functions for music analysis that are built into `chunking` use the following concepts.

1.2.1 The Burrows-Wheeler transform (BWT) and its inverse process

For example, the matrix of the inverse BWT (iBWT) contains all cyclic rotations of all substrings of an input phrase. In the case of the function `anaphrase`,this input phrase is a musical rhythm in shorthand notation. The iBWT generates matrices of all possible substrings of the input rhythm, plus, it generates also all of their cyclic rotations. The output can be translated into music notation via `lilypond`, and it can be synthesized with `csound`.

1.2.2 Rhythmic Impact Measurement (RMI)

The Rhythmic Impact Measurement (RMI) is a measurement of the level of complexity of a rhythmic pattern. If the length of a rhythmic pattern is set equal to 1, then the individual note onsets fall on ratios < 1 that, when in their lowest terms, are elements of a Farey Sequence n, where n denotes the length of the pattern in counts of small iso-chronic pulses. There exists a ranking of a natural integer in terms of the number of its prime factors and the relative size of the prime factors involved in the prime factorization of the integer. This ranking is achieved by applying the formula 1.1. It has been developed by Clarence Barlow for his composition system `Autobusk`. I have used and modified Barlow's formula in order to measure integer ratios that mark the onsets of notes within a cycle that is normalized between 0 and 1. According to formula 1.2, `chunking` sums together the ranking of the numerator and the ranking of the denominator.

.

$$\xi(N) = 2\sum_{r=1}^{\infty}\left\{\frac{n_r(p_r-1)^2}{p_r}\right\} \tag{1.1}$$

with $N = \prod_{r=1}^{\infty} p_r^{n_r}$, p_r is the rth prime number and n_r is its exponent in the prime factorization of N.

Table 1.1 shows the development of $\xi(N)$ for the first 17 integers.

After sorting according to increasing values of $\xi(N)$, the first 17 natural integers have the following order:

$$\{1, 2, 4, 3, 8, 6, 16, 12, 9, 5, 10, 15, 7, 14, 17\}$$

We use this measure to determine a weight C of an integer fraction. The fractions represent positions of onsets of a given rhythmic pattern within a metric cycle, or bar. For such ratios a/b in their lowest terms, with $a, b \in \mathbb{N}$ we calculate:

$$C(a/b) = \xi(a) + \xi(b) \tag{1.2}$$

with $\xi(n)$ defined in equation 1.1.

The method `bwtmatrix`, for example, outputs rhythmic impact measurements for each of the substrings that are created by the inverse Burrow-Wheeler Transform:

```
chunking -m bwtmatrix <string of rhythm in shorthand>
```

n	$\xi(n)$
1	0
2	1
3	2.66667
4	2
5	6.4
6	3.66667
7	10.2857
8	3
9	5.33333
10	7.4
11	18.1818
12	4.66667
13	22.1538
14	11.2857
15	9.06667
16	4
17	30.1176

Table 1.1: Table of the indigestibility of the first natural numbers.

1.2.3 Finding substrings in Christoffel words

A *word* is a finite or infinite sequence of letters taken from a finite alphabet. Christoffel words are constructed over the two-word alphabet $\{a, b\}$, with $a \prec b$. For the purpose of rhythm notation, resulting words are being mapped with $a \mapsto 1, b \mapsto 0$. We call the result of this mapping a Christoffel rhythm. The 1s represent note onsets, 0s represent silent pulses that fall in the inter-onset-times.

A Christoffel word can be constructed geometrically on the integer lattice $\mathbb{Z}^2$, see figure 1.1. First, one draws a line between the origin and the point (n, m); its slope is $\frac{m}{n}$ (rise over run). Because of the co-primality of m and n, $m \perp n$, the resulting diagonal does not touch any other points of the lattice. A Christoffel word is a path which follows this diagonal as close as possible. It encodes horizontal steps with the letter a, and vertical ones with b. Figure 1.1 shows the upper path of of $C(3, 5)$, which is *babaabaa*, as well as the lower path *aabaabab*. The strings represent the upper and lower Christoffel word. The only difference between the two is an exchange in the position of the first and the last letter. The reverse of the lower string is the upper string. We will always refer to the the *lower* Christoffel word with the notation $C(m, n)$.

With the `crhythm` method one can investigate the properties of a Christoffel rhythm. The arguments are the integers m and n that generate this

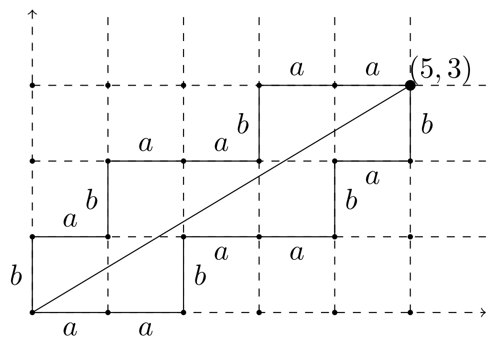

Figure 1.1: Geometric construction of the upper and lower Christoffel word $C(3,5)$ with slope $\frac{3}{5}$.

particular Christoffel word $Cm, n)$:

```
chunking -m crhythm <m: integer> <n: integer>
```

With a special method we can find out whether a rhythmic pattern that is written in shorthand notation is also a Christoffel word. And, it is possible to find the cases where an input pattern is a substring of a Christoffel word. This is possible with the method `lookup`, for example:

```
chunking -m lookup <shorthand pattern>
```

Apart from direct pattern matching, `chunking` also uses bit-transformations of the input pattern, because there is no reason of preferring the mapping-described above over other possible mappings. Moreover, these alternative mappings are also musically relevant.

1.2.4 Analysis of Polyrhythms with Farey Sequences

`chunking` performs an analysis of polyrhythms by using Farey sequences. Farey Sequences consist of all ratios in their lowest terms ranging between 0 and 1. They are of increasing order $n \in N$, which limits the size of the largest denominator. Here are a few examples:

$$\begin{aligned}
F_1 &= \{\tfrac{0}{1}, \tfrac{1}{1}\}\\
F_2 &= \{\tfrac{0}{1}, \tfrac{1}{2}, \tfrac{1}{1}\}\\
F_3 &= \{\tfrac{0}{1}, \tfrac{1}{3}, \tfrac{1}{2}, \tfrac{2}{3}, \tfrac{1}{1}\}\\
F_4 &= \{\tfrac{0}{1}, \tfrac{1}{4}, \tfrac{1}{3}, \tfrac{1}{2}, \tfrac{2}{3}, \tfrac{3}{4}, \tfrac{1}{1}\}\\
F_5 &= \{\tfrac{0}{1}, \tfrac{1}{5}, \tfrac{1}{4}, \tfrac{1}{3}, \tfrac{2}{5}, \tfrac{1}{2}, \tfrac{3}{5}, \tfrac{2}{3}, \tfrac{3}{4}, \tfrac{4}{5}, \tfrac{1}{1}\}\\
F_6 &= \{\tfrac{0}{1}, \tfrac{1}{6}, \tfrac{1}{5}, \tfrac{1}{4}, \tfrac{1}{3}, \tfrac{2}{5}, \tfrac{1}{2}, \tfrac{3}{5}, \tfrac{2}{3}, \tfrac{3}{4}, \tfrac{4}{5}, \tfrac{5}{6}, \tfrac{1}{1}\}
\end{aligned}$$

If one sets the length of a polyrhythmic sequence equal to 1, then the elements of a Farey Sequence denote the positions of note onsets on the timeline. In order to generate a polyrhythm, one can filter a Farey Sequence n by using the coprime pair of integers $a : b$ by making sure that their product is identical to the order n of the Farey Sequence, such as $n = a \times b$. `chunking` uses the method `fpoly` to achieve this, for example:

```
chunking -m fpoly 30 5 6
```

creates an analysis of the polythryhm 5 : 6. The result displays the compound rhythm represented by whole integers as note lengths, as well as a two-voice representation of the onsets and how they map to both subdivisions:

```
5 1 4 2 3 3 2 4 1 5
1 0 0 0 0 1 1 0 0 0 1 0 1 0 0 1 0 0 1 0 1 0 0 0 1 1 0 0 0 0
1 0 0 0 0 6 5 0 0 0 3 0 5 0 0 2 0 0 5 0 3 0 0 0 5 6 0 0 0 0
```

1.3 Algorithmic Composition

1.3.1 Musical Sentences and Integer Partitions

`chunking` has an algorithm for the creation of musical sentences. It works on the basic idea of integer partitions. An integer partition of n are all sums of smaller integers that result in n. Integer partitions have an important role in Indian Classical music. However, the approach that `chunking` takes is different from the Indian techniques. The idea is that a stream of small pulsations is partitioned into smaller and smaller segments until one deals with the most basic chunks of 2 and 3 pulses in length. The process goes therefore from the top level of the sentence down to phrases, patterns and chunks. At each level, an algorithm was designed to select a particular kind of partition. That partition has certain properties. These are: First, all parts are unique; second, all parts should be as close together as possible. The third rule is that neighbouring parts should be coprime integers, which do not share a common divisor other than 1[1]. In the `sentence` algorithm the user defines the total length of the sentence in pulses, and then gives the maximum number of parts that should be generated at each level. The range of the input arguments, number of pulses n, and number of parts k, are $0 < n <= 120$, and $1 < k < 6$, respectively.

On the level of patterns, the algorithm picks a musical form out of a catalogue of seven musical categories. The musical categories are resistor, release, arch, catenary, alternating, growth, and decline. These forms are ordered sequences of 2s and 3s.

Each one of the resulting sequences is transcribed into rhythm shorthand notation (RSN). The 2s and 3s represent metric groups, or chunks. The chunks are being transcribed by randomly picking form all possible onset patterns within 2 or 3 pulses.

```
chunking -m sentence <n: int> <k: int>
```

[1]The co-prime rule is sometimes relaxed because of the re-ordering of the parts that is carried out as well.

1.3.2 Fragmentation

1.3.3 Patterns

1.3.4 Polyrhythms

2

Overview

2.1 Shorthand Notation

Chunking uses a shorthand notation for rhythm. Single ASCII characters are used to encode binary and ternary chunks of rhythms. It is possible to produce longer notes with *tie*s. Silences are being produced by surrounding one or more chunks with round brackets, see table 2.1.

2.2 Backends

2.2.1 printphrases

```
chunking -m printphrases <file1: shorthand notation> <file2: midi pitches>
```

2.2.2 Lilypond

2.2.3 Csound

2.2.4 Sqlite

2.3 Burrows-Wheeler Transform

2.3.1 anaphrases

```
chunking -m anaphrases <shorthand.txt>
chunking -m anaphrases <shorthand.txt> <min beats> <max beats>
```

2.3.2 bwt

```
chunking -m bwt <string>
```

Table 2.1: Set of symbols for rhythmic chunks.

Symbol	Transcription	Binary Form	Category
.	♪	1	unary
I	♩	10	binary
:	♪♪	11	
v	𝄾♪	01	
–	♩.	100	
<	𝄾♩	010	
w	𝄾𝄾♪	001	
X	♪♩	110	ternary
>	♩ ♪	101	
+	𝄾♪♪	011	
i	♪♪♪	111	
˜	e.g. I ˜ I = 𝅗𝅥	1000	tie
( )	e.g. (X) I = 𝄽.♩	00010	silence

2.3.3 bwtmatrix

```
chunking -m bwtmatrix <shorthand string>
chunking -m bwtmatrix <shorthand string> <integer: length>
```

2.3.4 bwtmel

```
chunking -m bwtmel <string of ascii notes>
<integer denoting semitones for transposition>
<character denoting a type of transformation>
```

Example: `chunking -m bwtmel 'c6ef' 0 g`

2.3.5 bwtpath

```
chunking -m bwtpath <shorthand string> <file name>
```

Two inputs: 1. A string of rhythm shorthand, for example: `'IXIIXIIIXX'` Internally, the algorithm converts the shorthand into a word over the alphabet a,b, where 'a' represents a note onset, and 'b' represents inter-onset pulses 2. A file containing a random list of row numbers of the inverse Burrows Wheeler (iBWT) matrix. The iBWT matrix has as many rows as the word has characters. An example file of row numbers may contain: $14, 10, 7, 5, 3, 1$

2.3.6 crhythm

```
chunking -m crhythm <m: integer> <n: integer>
```

Outputs a Christoffel word $C(m,n)$, its ancestors in the Stern-Brocot tree, and various other transformations, i.e. BWT, for example.

2.3.7 compc

```
chunking -m compc <m1: int> <n1: int> <m2: int> <n2: int>
```

Compares two Christoffel rhythms with each other in order to find intersections of their inverse Burrows-Wheeler matrix, i.e. comparing all of their possible cyclic substrings.

2.4 Christoffel Words

2.4.1 crhythm

```
chunking -m crhythm <m: integer> <n: integer>
```

Outputs a Christoffel word $C(m,n)$, its ancestors in the Stern-Brocot tree, and various other transformations. See also section on BWT.

2.4.2 compc

```
chunking -m compc <m1: int> <n1: int> <m2: int> <n2: int>
```

Compares two Christoffel rhythms with each other in order to find intersections of their inverse Burrows-Wheeler matrix, i.e. comparing all of their possible cyclic substrings.

2.4.3 lookup

```
chunking -m lookup <shorthand pattern>
```

Searches all Christoffel words $C(m,n)$, from $C(1,1)$ to $C(20,20)$, for a matching rhythmic pattern.

2.5 Integer Partitons

2.5.1 partition

```
chunking -m partition <n> <flag parts-must-be-prime> <flag parts-not-?1?>
            <max prime in parts> <max int in parts>
            <min int in parts>
            <int to add to all parts>
            <number of distinct parts, 0 := print all partitions>
            <flag for p. with all parts being equal>
```

2.5.2 sentence

```
chunking -m sentence n k
```

with n = number of pulses <= 120, and k = number of distinct parts <= 5.

2.5.3 getpart

```
chunking -m getpart <n: integer> <k: integer> <1>
```

returns the partition of n into k unique parts with the lowest standard deviation, with $n <= 120$ and $k <= 5$.
The output format is: $n, part_1, part_2, ..., part_k, mean, k$

2.6 Farey Sequence

2.6.1 farey2binary

```
chunking -m farey2binary <Farey Sequence n> <Digestibility threshold f>
<int flag activates smooth filter and ignores Digestibility threshold>
```

2.6.2 fpoly

```
chunking -m fpoly <Farey Seq. N> <first subdivision> <second subdivision>
```

2.6.3 printfarey

```
chunking -m printfarey <Farey Sequence n>
```

Prints all members of Farey Sequence n, with each ratio also interpreted as slope of Christoffel word, including bit pattern operations, BWT, and shorthand notations.

2.6.4 intstrings

```
chunking -m intstrings <file1> <file2>
```

Input: Two ascii text files with lines of comma-separated rhythms in integer distance notation.
Output: various distance measurements between all pairs of rhythms between file1 and file2

2.7 Processes using the SNMR

2.7.1 fragment

```
chunking -m fragment <shorthand string>
```

2.7.2 fragrotate

```
chunking -m fragrotate <shorthand string>
```

2.7.3 jump

```
chunking -m jump <shorthand string> <n_symbols int> <k_times int>
<from_start? 0 (no) or 1 (yes)>
```

2.7.4 mutate

```
chunking -m mutate <shorthand string> <k_times int>
```

2.7.5 reverse

```
chunking -m reverse <shorthand string>
```

2.7.6 rotate

```
chunking -m rotate <shorthand string>
```

2.7.7 silence

```
chunking -m silence <shorthand string> <from_pos int> <to_pos int>
```

2.7.8 shape

```
chunking -m shape <shorthand string> <flag int>
```

2.7.9 shortening

```
chunking -m shortening <shorthand string> <from_top? 0 (no) or 1 (yes)>
```

2.7.10 swap

```
chunking -m swap <shorthand string> <k_times int>
```

2.8 Database queries

2.8.1 db

```
chunking -m db <search string in shorthand notation>
```

The search string may contain '%' for extended search.

2.8.2 dbinsert

```
chunking -m dbinsert <filename> <name> <origin> <composer>
```

File containing shorthand notation. One line creates a new entry in the table rhythm of rhy.db.

2.9 Miscellaneous

2.9.1 notenames

```
chunking -m notenames <string>
```

Returns: ascii code for pitches as one character per pitch, useful for Burrows-Wheeler. Returns also a list of MIDI note numbers. Example:

```
chunking -m notenames 'C4 C#4 D4 D#4'
c6d7
60, 61, 62, 63
```

2.9.2 notes2midi

```
chunking -m notes2midi <file name>
```

Input: Name of a text file containing lines of note names, for example:

```
C4 C4 G4 G4 A4 A4 G4
F4 F4 E4 E4 D4 D4 C4
```

Output: Lines of comma-separated Midi note numbers, for example:

```
60, 60, 67, 67, 69, 69, 67
65, 65, 64, 64, 62, 62, 60
```

After being saved as a file, the output can be used together with printphrases to merge the pitches with rhythmic phrases.

2.9.3 propseries

```
chunking -m propseries <chunk to divide iteratively: float>
<number of iterations: int> <divisor: float>
```

2.9.4 permutations

```
chunking -m permutations
```

Outputs the Christoffel words and BWTs of ratios built by the first twenty-one 7-smooth numbers.

2.9.5 permutations2

```
chunking -m permutations2
```

2.9.6 divisors

```
chunking -m divisors
```

lists all divisors of n in $\{1, 2, 3, ..., 499\}$.

3

Chunking - Reference

$\boxed{\textbf{A}}$

3.1 anaphrases

```
chunking -m anaphrases <input file>
```

Input: a text file containing rhythm patterns line by line in shorthand notation
Output: generates BWT and iBWTs of input pattern line by line; output as lilypond file (*.ly) and as shorthand notation in ascii text format (*.txt)

```
chunking -m anaphrases text/a1.txt
```

Analyses the shorthand phrases in a1.txt. Each line corresponds to one phrase. BWT and inverse BWT (iBWT) matrix are calculated for each phrase. The iBWT matrix contains all cyclic rotations of all substrings of a phrase. Duplicate substrings are removed.
For each phrase, the phrase itself and the list of all its iBWT substrings (k-mers) are concatenated into a single score in lilypond format.

```
lilypond.sh phrase_iBWT_0.ly
```

executes lilypond on the first output file generated by anaphrases to produce a pdf file.

```
preview.sh phrase_iBWT_0.pdf
```

launches the pdf viewer to look at the printed score rendered by lilypond.

```
chunking -m anaphrases <input file> <min beats> <max beats>
```

Optional input: min beats, max beats are positive integers denoting the minimum and maximum (exclusive) number of beats printed in the result.

```
chunking -m anaphrases text/a1.txt 3 4
```

The two optional integer arguments denote the minimum of onsets in every substring, and the maximum of onsets (non inclusive). The example above will print all substrings for every phrase that have three onsets only.

$\boxed{\textbf{B}}$

3.2 bwt

```
chunking -m bwt <shorthand string>
```

bwt performs the Burrows-Wheeler Transform (BWT) on the standard input string and calculates the matrix of the inverse BWT (iBWT), which produces all cyclic substrings via rotation and sort. Example:

```
chunking -m bwt 'IXIIX'
```

It automatically creates a text file bwt_phrases.txt, where each line represents one of the substrings generated by the iBWT. If the input string is a rhythm shorthand notation (RSN), then the output file can be used together with printphrases in order to generate a lilypond script and a csound score. The command line

```
chunking -m printphrases bwt_phrases.txt text/one.txt
```

will automatically generate the files print_phrases.ly and print_phrases.sco bwt also works with any other ascii string. The file bwt_phrases.txt is always generated, but it might not necessarily be a useful input for printphrases anymore because many ascii characters are not part of the shorthand code. The classic BWT example is:

```
chunking -m bwt 'banana$'
```

3.3 bwtmatrix

```
chunking -m bwtmatrix <string of rhythm in shorthand>
```

Example:

```
chunking -m bwtmatrix '-.IHI'
```

Input: A string of characters notating a rhythm in shorthand notation. Output: The complete inverse Burrows-Wheeler matrix line by line in the following form: <row #> <substring #> <rhythm string> <number of beats> <density> <absolute impact> <relative impact>

3.4 bwtmel

```
chunking -m bwtmel <string of ascii notes>
<integer denoting semitones for transposition>
<optional: character denoting a type of transformation>
```

Example:

```
chunking -m bwtmel 'deafgaa' 0
```

bwtmel creates a matrix of cyclic substrings based on the inverse BWT process. It takes at its input a string of ascii symbols as note names, a transposition in semitones (0 = no transposition, 12 = octave up, -7 = fifth down), and a character denoting one of three possible transformations of the input string before the BWT is performed: u for inversion (Inversion of all intervals between the original pitches; German: Umkehrung), k for retrograde (Going backwards starting with the last pitch; German: Krebs), and q for the retrograde of the inversion (German: Krebs der Umkehrung). If no character is given, then the input string is kept as it is (except for transposition). bwtmel automatically creates two text files: bwt_ones.txt and bwt_melody.txt. The latter contains the matrix of cyclic substrings encoded as comma-separated MIDI note values, with each line containing one element of each k-mer. The rhythms in bwt_ones.txt are a sequence of quarter notes. The two files can be used in order to print a lilypond score and to output a csound score with printphrases, for example:

```
chunking -m printphrases bwt_ones.txt bwt_melody.txt
```

Example of zero transposition and retrograde pitch transform:

```
chunking -m bwtmel 'deafga' 0 k
```

Example of zero transposition and inversion of pitches:

```
chunking -m bwtmel 'deafga' 0 u
```

Example of a transposition upward by a fifth and subsequent retrograde of inversion:

```
chunking -m bwtmel 'deafga' 7 q
```

Example of transposing the pitches an octave down and retrograde of inversion:

```
chunking -m bwtmel 'deafga' -12 q
```

The following list gives the ascii characters used by bwtmel to input all chromatic pitches between C3 and B4 (MIDI note 48 to 71): C, 1 ,D ,2 ,E ,F ,3 ,G ,4 ,A ,5 ,B ,c ,6 ,d ,7 ,e ,f ,8 ,g ,9 ,a ,0 ,b . bwtmel recognizes all notes within the range of the grand piano (A0 to C7). In order to convert from a list of note names (A0 to C7) to the program's ascii codes, one can use the command line:

```
chunking -m notenames <string of notenames>
```

Example:

```
chunking -m notenames 'C3 Db3 D3 B3'
```

3.5 bwtpath

```
chunking -m bwtpath <shorthand string> <text file>
```

Input: 1st argument: string of a rhythm in shorthand notation, 2nd argument: a text file containing a list of rows from which bwtpath extracts a substring. Output: lines of rhythm substrings according to both input arguments. The order of the lines corresponds to the order of the rows given in the text file as the 2nd argument. The ouput can be routed into a file, which can serve as the rhythmic input for printphrases at its 1st argument. Example of a workflow:

```
cat text/frag2.txt
15,11,11,9,7,7,5,5,5,3,3,2
chunking -m bwtpath 'I>IIIXII' text/frag2.txt > text/bwtfrag2.txt
chunking -m printphrases text/bwtfrag2.txt text/one.txt
lilypond.sh print_phrase.ly
preview.sh print_phrase.pdf
```

$\boxed{\text{C}}$

3.6 crhythm

```
chunking -m crhythm <m: integer> <n: integer>
```

Outputs a Christoffel word C(m,n), its ancestors in the Stern-Brocot tree, and various other transformations. Example:

```
./chunking -m crhythm 5 4
ancestors of 5/4 are:
L: 4/3
R: 1/1
```

```
ancestors of 4/3 are:
L: 3/2
R: 1/1
ancestors of 3/2 are:
L: 2/1
R: 1/1
ancestors of 2/1 are:
L: 1/0
R: 1/1
---------------------------
length: 9, slope: 5/4, Christoffel word: ababababb
ababababb
---------------------------
1,0,1,0,1,0,1,0,0,
---------------------------
0,1,0,1,0,1,0,1,1,
---------------------------
1,1,1,1,1,1,1,1,0,
---------------------------
0,0,0,0,0,0,0,0,1,
---------------------------
BWT-word: bbbbbaaaa
Etcetera
```

The method produces a text file crhythm_m_n.txt with rhythm shorthand notation. Each line in this file represents a cyclic substring (k-mer) of the inverse Burrows-Wheeler transform matrix (iBWT).

3.7 compc

```
chunking -m compc <m1: int> <n1: int> <m2: int> <n2: int>
```

Compares two Crhistoffel rhythms with each other to find intersections of their inverse Burrows-Wheeler matrix, i.e. comparing all of their possible cyclic substrings. Example:

```
chunking -m compc 5 6 2 3
```

Output:

```
length: 11, slope: 5/6, Christoffel word #1: aababababab
length: 5, slope: 2/3, Christoffel word #2: aabab
There are 22 unique elements in C(5/6) from sub-string size 2 to 5
There are 21 unique elements in C(2/3) from sub-string size 2 to 5
There are 20 common elements.
```

```
Ratio of the number of common elements
over the unique elements in C(5/6): 0.909091
Ratio of the number of common elements
over the unique elements in C(2/3): 0.952381
```

$\boxed{\textbf{D}}$

3.8 db

```
chunking -m db <search string in shorthand notation>
```

Queries the sqlite database 'rhy.db' that is part of the distribution of chunking. The search string may contain '%' for extended search. Example:

```
chunking -m db '%IX>%'
$ 19 sketchbook green Boenn 1
--iIIX>
$ 6/8 clave Latin America n/a 2
IIX>I
```

3.9 dbinsert

```
chunking -m dbinsert <filename> <name> <origin> <composer>
```

Creates new entries in the sqlite database 'rhy.db' that is used by chunking. Lines with shorthand notation are read from a file. Each line creates a new entry in the table 'rhythm' of 'rhy.db'. Name, origin, and composer are provided by the command-line and will be kept the same for each line of the input file.

3.10 divisors

```
chunking -m divisors
```

Input: none.
Output: A list of all divisors for integers $\{n = 1, n = 2, n = 3, ...n = 499\}$, excluding 1 and n.

```
...
Divisors of 4: 2
Divisors of 5:
Divisors of 6: 3 2
Divisors of 7:
Divisors of 8: 4 2
Divisors of 9: 3
Divisors of 10: 5 2
Divisors of 11:
Divisors of 12: 6 4 3 2
...
```

$\boxed{\textbf{F}}$

3.11 farey2binary

```
chunking -m farey2binary <Farey Sequence n>
<Digestibility threshold f>
<int flag activates smooth filter 2 3>
```

Example:

```
chunking -m farey2binary 7 4 0
```

Output:

```
Filtered Farey Sequence 7 according to max. Digestibility value 4 :
0/1 1/6 1/4 1/3 1/2 2/3 3/4 5/6 1/1
Filtered Farey Sequence with common denominator:
2/12 1/12 1/12 2/12 2/12 1/12 1/12 2/12
in integer distance notation:
2 1 1 2 2 1 1 2
```

Note: The binary output is in the source code, but not active.

3.12 fpoly

```
chunking -m fpoly <Farey Seq. N>
<first subdivision> <second subdivision>
```

Example:

```
chunking -m fpoly 20 5 4
F_20 filtered by subdivisions:
5 4
Filtered F_20 in integer distance notation:
4 1 3 2 2 3 1 4
and translated into binary rhythm:
1 0 0 0 1 1 0 0 1 0 1 0 1 0 0 1 1 0 0 0
1 0 0 0 5 4 0 0 5 0 2 0 5 0 0 4 5 0 0 0
```

The last line shows the occurrence of the denominators (subdivisions) on the timeline. To pick the right integer N for the Farey Sequence, calculate N = (first subdivision) x (second subdivision). The first and second subdivisions have to be co-prime.

3.13 fragment

```
chunking -m fragment <shorthand string>
```

Returns a random substring of the input string. Example:

```
chunking -m fragment 'IXI>:'
I>:
```

3.14 fragrotate

```
chunking -m fragrotate <shorthand string>
```

Returns a random rotation of a random substring of the input string. Example:

```
chunking -m fragrotate 'IXI>:'
XII
```

$\boxed{\textbf{G}}$

3.15 getpart

```
chunking -m getpart <n: integer> <k: integer> <1>
```

getpart returns the partition of n into k unique parts with the lowest standard deviation, with $n <= 120$ and $k <= 5$. The output format is:

```
n part_1 part_2 ... part_k mean k
```

Example:

```
chunking -m getpart 30 2 1
30 17 13 15 2
```

$\boxed{\textbf{I}}$

3.16 intstrings

```
chunking -m intstrings <file1> <file2>
```

Input: Two ascii text files with lines of comma-separated rhythms in integer distance notation.
Output: various distance measurements between all pairs of rhythms between file1 and file2
Example:

```
chunking -m intstrings integer_rhythm1.txt integer_rhythm2.txt
```

$\boxed{\textbf{J}}$

3.17 jump

```
chunking -m jump <shorthand string> <n_symbols int> <k_times int>
<from_start? 0 (no) or 1 (yes)>
```

This is the "jumping needle" algorithm. It repeats n symbols of the input string k times from either the beginning (from_start = 1) or from the end of the input sting (from_start = 0), placing the fragments either at the beginning of the input string or at the end. Example:

```
chunking -m jump 'IXI>:' 2 3 1
$ jumping needle for 3 times the first 2 symbols.
IX
IX
IX
IXI>:

chunking -m jump 'IXI>:' 2 3 0
$ jumping needle for 3 times the last 2 symbols.
IXI>:
>:
>:
>:
```

$\boxed{\textbf{L}}$

3.18 lookup

```
chunking -m lookup <shorthand pattern>
```

Searches all Christoffel words, from C(1,1) to C(20,20), for a matching rhythmic pattern. The search includes bit transformations of the input pattern, and the inverse BWT matrix of all Christoffel words. Input: rhythm encoded in shorthand notation Ouput: corresponding binary pattern. All matches found. The output stops after a mximum of thirteen matches. Example:

```
chunking -m lookup 'IXI'
Shorthand pattern to analyse: IXI
1011010 7
[7 1, 0, 1, 1, 0, 1, 0, ]
1011010 is at pos. #2 in iBWT block #7 of C(3,4)
1011010 is at pos. #4 in iBWT block #7 of C(3,5)
1011010 is at pos. #2 in iBWT block #7 of C(4,5)
1011010 is at pos. #3 in iBWT block #7 of C(4,6)
...
1011010 is at pos. #2 in iBWT block #7 of C(6,7)
```

$\boxed{\textbf{M}}$

3.19 mutate

```
chunking -m mutate <shorthand string> <k_times int>
```

Randomly mutates the input string *k* times in a row. Example:

```
chunking -m mutate 'IXI>:' 2
$ mutations at positions: 2, 3,
IXv<:
```

$\boxed{\textbf{N}}$

3.20 notenames

```
chunking -m notenames <string of notenames>
```

Input: A string of notenames using 'b' for flat, '#' for sharp and octave numbers following the American conventions, i.e A4 = 440 Hz. Example:

```
chunking -m notenames 'C3 Db3 D3 B3'
Output:
C1DB
48, 49, 50, 59
```

The first line are the notes in ascii code useful for the *bwtmel* method. The second line are MIDI note numbers that are useful for the method *printphrases*. Workflow example:

```
chunking -m notenames 'C3 Db3 D3 B3'
chunking -m bwtmel C1DB 0 g
or:
chunking -m bwtmel C1DB 24 k
or:
chunking -m bwtmel C1DB 0 q
then:
chunking -m printphrases bwt_ones.txt bwt_melody.txt
lilypond.sh print_phrase.ly
preview.sh print_phrase.pdf
timidity print_phrase.midi
```

3.21 notes2midi

```
chunking -m notes2midi <file>
```

notes2midi converts lines of note names in <file> to strings of comma-separated MIDI note numbers. Example:

```
cat text/midi.txt
C3 C4 C5 C6
chunking -m notes2midi text/midi.txt
48, 60, 72, 84
```

The output is useful to write a file of pitches for printphrases.

P

3.22 partition

```
chunking -m partition <n> <parts-must-be-prime> <parts-not-'1'>
<max prime in parts> <max int in parts>
<min int in parts>
<int to add to all parts>
<number of distinct parts, 0 := print all partitions>
<flag for p. with all parts being equal>
```

Example: This example produces partitions of 90 into 4 distinct parts, all parts > 1:

```
chunking -m partition 90 0 1 101 90 1 0 4
90 = 81, 4, 3, 2, mean: 22.5 sigma: 33.7824  Coprime pairs.
90 = 80, 5, 3, 2, mean: 22.5 sigma: 33.2152
```

```
90 = 79, 6, 3, 2, mean: 22.5 sigma: 32.6535
90 = 79, 5, 4, 2, mean: 22.5 sigma: 32.6382
90 = 78, 7, 3, 2, mean: 22.5 sigma: 32.0975  Coprime pairs.
...
90 = 25, 24, 23, 18, mean: 22.5 sigma: 2.69258  Coprime pairs.
90 = 25, 24, 22, 19, mean: 22.5 sigma: 2.29129
90 = 25, 24, 21, 20, mean: 22.5 sigma: 2.06155
90 = 25, 23, 22, 20, mean: 22.5 sigma: 1.80278
90 = 24, 23, 22, 21, mean: 22.5 sigma: 1.11803  Coprime pairs.
```

3.23 permutations

```
chunking -m permutations
```

Outputs the Christoffel words and BWTs of ratios built by the first twenty-one 7-smooth numbers.

3.24 permutations2

```
chunking -m permutations2
```

Outputs filtered Farey Sequences. The filter is based on subdivisions taken from pairs of 7-smooth numbers. Transcriptions into integer sequences and into binary patterns are included. Example:

```
F_10 filtered by subdivisions:
2 5
Filtered F_10 in integer distance notation:
2 2 1 1 2 2
and translated into binary rhythm:
1 0 1 0 1 1 1 0 1 0
1 0 5 0 5 2 5 0 5 0
```

3.25 printfarey

```
chunking -m printfarey <Farey Sequence n>
```

Prints all members of Farey Sequence n, with each ratio also interpreted as slope of Christoffel word, including bit pattern operations, BWT, and shorthand notations Example:

```
chunking -m printfarey 5
F(5) contains 1/2:
...
```

```
6  : 1/5  : aaaaab
aaaaab
---------------------------
1,1,1,1,1,0,
---------------------------
0,0,0,0,0,1,
---------------------------
1,0,0,0,0,1,
---------------------------
0,1,1,1,1,0,
---------------------------
Block #1
a,a,a,a,a,b,
Block #2
aa,aa,aa,aa,ab,ba,
Block #3
aaa,aaa,aaa,aab,aba,baa,
Block #4
aaaa,aaaa,aaab,aaba,abaa,baaa,
Block #5
aaaaa,aaaab,aaaba,aabaa,abaaa,baaaa,
Block #6
aaaaab,aaaaba,aaabaa,aabaaa,abaaaa,baaaaa,
...
```

3.26 printphrases

```
chunking -m printphrases <file1> <file2>
```

printphrases reads in a text file containing lines of phrases in shorthand notation (file1). The output is a single lilypond score file where each phrase is a single bar. In addition, a second text file (file2) is used to provide a list of pitches on each line corresponding to the rhythm phrase on the same line in the first text file. The pitches are provided in MIDI note number format separated by commas. If the number of pitches is smaller than the number of onsets in the rhythm, then the list of pitches is repeated in a loop. Any surplus of pitches is ignored. To transcribe the output into a score, use:

```
lilypond.sh print_phrase.ly
preview.sh print_phrase.pdf
```

3.27 propseries

```
chunking -m propseries <chunk to divide iteratively: float>
```

```
<number of iterations: int>
<divisor: float>
```

Output: series of progressive subdivisions of a number (similar to reading the Fibonacci series backwards) and the differences between them. The divisor can be any positive real > 0. For golden sections one could use 1.618 as a divisor, for example:

```
./chunking -m propseries 89 8 1.618
89
55.0062
33.9964
21.0114
12.986
8.02597
4.96042
3.06578
...
```

$\boxed{\text{R}}$

3.28 reverse

```
chunking -m reverse <shorthand string>
```

Returns the reverse of the input string. Example:

```
chunking -m reverse 'IXI>:'
:>IXI
```

3.29 rotate

```
chunking -m rotate <shorthand string>
```

Rotates the input string a random number of times. Example:

```
chunking -m rotate 'IXI>:'
I>:IX
```

$\boxed{\text{S}}$

3.30 sentence

```
chunking -m sentence <n: int> <k: int>
```

The sentence algorithm takes two integers as arguments. The first one represents the number of n equidistant pulses that measure the total length of the musical sentence. The second argument, k, is the maximum number of parts that will structure the length of pulses on the next lower level. For example, if k=2, then the sentence will be divided into two phrases of unequal lengths. These phrases in turn will be divided each into two patterns of unequal length. Finally, on the level of patterns the algorithm will pick a musical form out of a catalog with seven categories that matches the length of the pattern. The musical forms are resistor, release, arch, catenary, alternating, growth, and decline. These forms are ordered sequences of 2s and 3s. Each one of the resulting sequences will be transcribed into rhythm shorthand notation (RSN). The 2s and 3s represent metric groups, or chunks. The chunks are transcribed by randomly picking form all possible onset patterns within 2 or 3 pulses. The ouputs are: 1. A lilypond input script that can be rendered by lilypond into a pdf score:

```
lilypond.sh sentence_60_2.ly
preview.sh sentence_60_2.pdf
```

2. The output in shorthand notation, with one line per phrase for further editing and printing via printphrases 3. The output of a csound score that is used to call instrument events in csound with a series of parameters and envelopes. The ranges of the input arguments, number of pulses n, and number of parts k, are $0 < n <= 120$, and $1 < k < 6$, respectively.

3.31 shape

```
chunking -m shape <shorthand string> <flag int>
```

This algorithm was inspired by the Classical Indian practice of turning a rhythmic sequence into certain shapes that express a feature of growth or decline, or of both in sequence. Shape creates progressive shortening and/or lengthening on the basis of the input string. The names of the shapes are: hourglass (flag=0), tail (1), river (2), and barrell (3).[1]

```
chunking -m shape 'IX>:' 0
IX>:
IX>
IX
I
IX
IX>
IX>:
```

[1]The original Indian names are:

```
chunking -m shape 'IX>:' 1
IX>:
IX>
IX
I

chunking -m shape 'IX>:' 2
I
IX
IX>
IX>:

chunking -m shape 'IX>:' 3
I
IX
IX>
IX>:
IX>
IX
I
```

3.32 shortening

```
chunking -m shortening <shorthand string> <from_top? 0 (no) or 1 (yes)>
```

Performs the shape 'tail' (see the method *shapes*) on the input string by either reducing the string from the end (from_top=0), or from the head of the string (from_top=1). Example:

```
chunking -m shortening 'IX>:' 0
$ cycling through IX>:
IX>:
IX>
IX
I

chunking -m shortening 'IX>:' 1
$ from the front of IX>:
IX>:
X>:
>:
:
```

3.33 silence

```
chunking -m silence <shorthand string> <from_pos int> <to_pos int>
```

Converts the shorthand characters of the input string of the positions from_pos to_pos into silence. Note that the first character's position is '0', and that the to_pos position is excluded. Example:

```
chunking -m silence 'IXI>:' 1 3
I(XI)>:
0 12 34
```

3.34 swap

```
chunking -m swap <shorthand string> <k_times int>
```

Swaps a random pair of consecutive characters of the input string *k* times. Example:

```
chunking -m swap 'IX>:' 2
$ swapping at positions: 1, 0,
>IX:
```