Implementing Several Lossless Compression Algorithms: Adaptive Arithmetic Coding (AC) and the LZW for C++

Erik Sven Vasconcelos Jansson
<erija578@student.liu.se>
Linköping University, Sweden

March 16, 2017

Abstract

Data compression provides essential methods when one wants to store or transfer information efficiently. In this short technical report, common techniques for modelling and two different coders are presented. By measuring the entropy of the Canterbury corpus, the limits of coding can be determined for a source. Both the *Lempel–Ziv–Welch* and *Arithmetic Coding* are briefly described and then implemented, where common practical problems & solutions are shown. Finally, measurements of both *rate* and *speed* for these methods are compared to the actual entropy.

Contents

Since this article is quite short, further details on *Arithmetic Coding*, *LZW*, *ANS* can be found below:

References

[1] M. Dipperstein. *Arithmetic Coding Discussion and Implementation.* 2014. [Online 2017-03-11].

[2] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp. The Use of Asymmetric Numeral Systems as an Accurate Replacement for Huffman Coding (rANS and tANS). In *Picture Coding Symposium (PCS), 2015*, pages 65–69. IEEE, 2015.

[3] P. G. Howard and J. S. Vitter. Practical Implementations of Arithmetic Coding. In *Image & Text Compression*, pages 85–112. Springer, 1992.

[4] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic Coding Revisited. *ACM Transactions on Information Systems (IS)*, 16(3):256–294, 1998.

[5] J. Nieminen. (Article) An Efficient LZW Implementation. `http://warp.povusers.org/EfficientLZW/`, 2007. Accessed: 2017-02-24.

[6] J. Rissanen and G. Langdon. Arithmetic Coding. *The IBM Research Journal*, 23(2):149–162, 1979.

[7] K. Sayood. *Introduction to Data Compression.* Morgan Kaufmann Series, third edition, 2005.

[8] T. A. Welch. A Technique for High-Performance Data Compression. *Computer*, 6(17):8–19, 1984.

[9] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, 30(6):520–540, 1987.

Source's Filename	Size in Bytes	H(S_n) in Shannons	H($S_n \vert S_{n-1}$)	H($S_n \vert S_{n-1}, S_{n-2}$)
alice29.txt	152089	4.567721306757149	3.418682225075357	2.48520469580814
asyoulik.txt	125179	4.808169990765448	3.417731826885349	2.53820221279907
bible.txt	4047392	4.342752822300183	3.269099339823096	2.47861800400018
cp.html	24603	5.229444511631613	3.467709563850320	1.73841659079717
E.coli	4638690	1.999821485762563	1.981417241665094	1.96323532309116
fields.c	11150	5.008337632132268	2.950926540872756	1.47064554512717
grammar.lsp	3721	4.633982742963610	2.806478773539829	1.28692014638656
kennedy.xls	1029744	3.573474997044099	2.777314597758521	1.71153686785860
lcet10.txt	426754	4.669132957659401	3.497027811530963	2.61231530515721
plrabn12.txt	481861	4.531375618032128	3.366065910049050	2.71692784135892
ptt5	513216	1.210175035218506	0.823658947303357	0.70519976712624
sum	38240	5.329193324866448	3.298284974946003	1.93087067303595
world192.txt	2473400	4.998316616349440	3.660472085185258	2.77064872349497
xargs.1	4227	4.900067215713765	3.196483510458247	1.55061392381553

Table 1: Entropy Estimations (0^{th}, 1^{st} and 2^{nd} Markov Orders) for the Canterbury Corpus Test Set

1 Source Modelling

Before *source coding* anything, formulation of the source's *statistical properties* needs to be *modelled*. Typically, ones which *predict future symbols* better. Below we define two useful models, which will be used to *estimate entropies* in the *Canterbury corpus*.

1.1 Stationary Source

Generates a *sequence* S of *symbols* $\{s_i\}$ within some *alphabet* $\mathcal{A}$. Symbols are *independent* of each other, knowing past symbols doesn't affect the current s_n:

$$p(s_n|s_{n-1}, \ldots, s_1) = p(s_n),\ s_i \in \mathcal{A}$$

1.2 Markov Information Source

However, in contrast, some sources *do* generate S's in which s_n *depends* on the k^{th} past symbols. Such a source has *memory* up to *Markov order* k. Formally:

$$p(s_n|s_{n-1}, \ldots, s_1) = p(s_n|s_{n-1}, \ldots, s_{n-k})$$

1.3 Estimating Source Entropy

Within *lossless data compression*, the best *rate of compression* achievable is the *Shannon entropy* [7]. Accurate modelling gives the source's *optimal rate*:

$$H(S) = -\sum_i p(s_i) \log_b p(s_i) \tag{1}$$

Finally, we attempt to *estimate* the entropies of the sources within the Canterbury corpus. First, we estimate the *joint probabilities* using Algorithm 1, which measures a symbol's frequency up to order k.

Algorithm 1 Estimating k^{th} Source Probabilities

Require: Sequence of $|S|$ symbols, where $s_i \in \mathcal{A}$.
  for $i \leftarrow 1$ **to** $|S|$ **do**
    for $j \leftarrow 0$ **to** k **do**
      state $\leftarrow (s_{i-j}, s_{i-j-1}, \ldots, s_i)$
      $f_{\text{state}} \leftarrow f_{\text{state}} + 1$
    end for
  end for
  for all state $\in f$ **do**
    total $\leftarrow |S| - |\text{state}| - 1$
    p(state) $\leftarrow f_{\text{state}} \div$ total
  end for

Having done this, we calculate the *joint entropy*, which is utilized in the *conditional entropy* since we have already calculated $H(S_n, \boldsymbol{S})$ and $H(\boldsymbol{S})$, where we define $\boldsymbol{S} \equiv S_{n-1}, \ldots, S_{n-k},\ \boldsymbol{s} \equiv s_{n-1}, \ldots, s_{n-k}$:

$$H(S_n, \boldsymbol{S}) = -\sum_n \cdots \sum_k p(s_n, \boldsymbol{s}) \log_b p(s_n, \boldsymbol{s}) \tag{2}$$

$$H(S_n|\boldsymbol{S}) = H(S_n, \boldsymbol{S}) + H(\boldsymbol{S}) \tag{3}$$

By using the implementation in Appendix A, we can produce Table 1 which has the $0^{th}, 1^{st}$ and 2^{nd} order conditional entropies of each source in the set.

Source's Filename	Size in Bytes	Rate in Bits/Symb	Encoding Millisec	Decoding Millisec
alice29.txt	70148	3.689839501870608	35.2187500000000	16.9062500000000
asyoulik.txt	62748	4.010129494563784	30.5937500000000	14.6562500000000
bible.txt	1501134	2.967113637621456	303.250000000000	118.062500000000
cp.html	14948	4.860545461935536	8.12500000000000	7.12500000000000
E.coli	1342680	2.315619280443400	318.437500000000	77.7500000000000
fields.c	7084	5.082690582959640	5.21875000000000	4.09375000000000
grammar.lsp	2818	6.058586401504968	3.56250000000000	3.31250000000000
kennedy.xls	332902	2.586289407852824	163.562500000000	111.468750000000
lcet10.txt	185158	3.471002029272128	84.3125000000000	79.4687500000000
plrabn12.txt	220976	3.668709441104384	102.718750000000	87.1875000000000
ptt5	70116	1.092966704077808	66.7812500000000	20.6875000000000
sum	25054	5.241422594142256	8.90625000000000	7.87500000000000
world192.txt	1075670	3.479162286730808	206.125000000000	83.1562500000000
xargs.1	3584	6.783061272770280	3.81250000000000	5.12500000000000

Table 2: Lempel-Ziv-Welch Coder ("Markov" Model-ish) Results in the Canterbury Corpus Test Set

2 Lempel–Ziv–Welch Coding

Another *dictionary coder* based off *Lempel–Ziv's 78* coding algorithm, introduced by *Welch* [8] in 1984. However, due to patent issues, it remained unused, and the *deflate coder* was usually a good alternative. Nowadays, these patents have expired, and it's used in e.g. the *Unix* `compress` tool, see *Nieminen* [5], but in practice, it's generally not used very much...

2.1 Encoding and Decoding

As with other *dictionary coders*, we seek *matches* in the sequence to be coded and the *dictionary*, which is built *adaptively* (as we go) with previous matches. Instead of coding the matching sequence, we write the *location* of the sequence within the dictionary.

Below follow the encoding & decoding algorithms:

Algorithm 2 Lempel–Ziv–Welch Encoding Steps

prefix $\leftarrow \epsilon$
dictionary$_i \leftarrow \epsilon s_i$, $\forall s_i \in \mathcal{A}$
while $s_k \leftarrow$ read(*input*) $\neq \varnothing$ **do**
  string $\leftarrow$ append(prefix, s_k)
  if index $\leftarrow$ find(string, dictionary) $= \varnothing$ **then**
    write(find(prefix, dictionary), *output*)
    dictionary$_j \leftarrow$ string ; prefix $\leftarrow s_k$
  else
    prefix $\leftarrow$ string
  end if
end while

Algorithm 3 Lempel–Ziv–Welch Decoding Steps

past $\leftarrow \epsilon$
dictionary$_i \leftarrow \epsilon s_i$, $\forall s_i \in \mathcal{A}$
while $c_k \leftarrow$ read(*input*) $\neq \varnothing$ **do**
  if string $\leftarrow$ exists(c_k, dictionary) $= \varnothing$ **then**
    dictionary$_j \leftarrow$ append(past, past$_k$)
    write(append(past, past$_k$), *output*)
  else
    dictionary$_j \leftarrow$ append(past, string$_k$)
    write(string, *output*) ; past $\leftarrow$ string
  end if
end while

2.2 Implementing Coder

Several practical details were left out above, these are covered in *Nieminen's* [5] excelled article. Main modifications are to *"chain together"* entries in the dictionary, meaning we avoid storing entire strings. Another detail is how to efficiently find a string in the dictionary, here we use a *binary search tree*, exploiting its structure, needs $\leq \lg 256$ comparisons.

See Appendix B for the implementation, we use 16-bit fixed codewords for the 2^{16} dictionary entries, requiring 576 KiB when full (then, resetting itself).

2.3 Rate and Speed

Comparing Table 1 and Table 2, we see that LZW performs poorly on small files. This is because the dictionary hasn't even had time to become half full. We also see it performs well on large natural texts.

Source's Filename	Size in Bytes	Rate in Bits/Symb	Encoding Millisec	Decoding Millisec
alice29.txt	87653	4.610616152384456	118.375000000000	111.312500000000
asyoulik.txt	75794	4.843879564463680	87.1562500000000	106.906250000000
bible.txt	2220355	4.388712533898368	867.375000000000	882.562500000000
cp.html	16306	5.302117627931552	42.5937500000000	49.0312500000000
E.coli	1173737	2.024255986065024	630.375000000000	633.000000000000
fields.c	7156	5.134349775784752	32.7812500000000	24.3437500000000
grammar.lsp	2297	4.938457403923672	12.7812500000000	12.7500000000000
kennedy.xls	482256	3.746608865892880	214.750000000000	217.937500000000
lcet10.txt	257791	4.832592078808864	185.812500000000	185.812500000000
plrabn12.txt	274853	4.563191459777816	197.531250000000	187.593750000000
ptt5	117098	1.825321112358144	116.906250000000	149.656250000000
sum	28393	5.939958158995808	64.4687500000000	56.3437500000000
world192.txt	1573387	5.088985202555184	588.937500000000	594.062500000000
xargs.1	2735	5.176247929973976	13.2812500000000	13.4687500000000

Table 3: Adaptive Arithmetic Coder (Stationary Model) Results in the Canterbury Corpus Test Set

3 Adaptive Arithmetic Coding

An *entropy encoder* attempts to assign shorter codewords to symbols that are more probable, leading to compression. *Arithmetic coding* was popularized by *Witten et al.* [9] in 1987, and has become central in both *lossless* and *lossy compression* because the coder separates modelling from the coding step and also because it provides rates coverging to entropy.

3.1 Encoding and Decoding

Conceptually, the idea is simple. Assign an interval $[l_k, u_k[$ which is unique to each symbol s_k, where $u_k - l_k$ is proportional to $p(s_k)$. This is usually done with the *cumulative probability*: $F(k) = \sum_{i=1}^{k} p(s_i)$, or if adaptive, the *empirical frequency* of the symbol.

Sequences are encoded by continuously "zooming" into the interval $[0, 1[$ with $F(k-1)$ and $F(k)$, by re-scaling the initial interval. Any number between the final interval can be chosen, usually: $(u + l) \div 2$.

Algorithm 4 Adaptive Arithmetic Encoding Steps

```
lower ← 0.0 ; upper ← 0.0
while s_k ← read(input) ≠ ∅ do
    range ← upper − lower
    upper ← lower + range · F(k)
    lower ← lower + range · F(k − 1)
end while
write((lower + upper) ÷ 2, output)
```

Algorithm 5 Adaptive Arithmetic Decoding Steps

```
c ← read(input)
while s_k ≠ ∅ do
    s_k ← s_k, F(k − 1) ≤ c ≤ F(k)
    upper ← F(k) ; lower ← F(k − 1)
    range ← upper − lower
    c ← (c − lower) ÷ range
    write(s_k, output)
end while
```

3.2 Implementing Coder

Unfortunately, there are several practical problems with this approach, *Howard et al.* [3], *Moffat et al.* [4] provide a nice summary of them. First, we want to have *integer arithmetic* for the *symbol frequencies* instead, and because *precision is limited*, we desire some way to handle *infinite sequences of bits* which should also be written in *small chunks* to a output.

See Appendix C for complete solutions, briefly, we *estimate symbol frequencies*, *re-scale* accodingly, and *shift out bits* we know never will change again. Arithmetic done in 32-bits, counts stored in 16-bits.

3.3 Rate and Speed

Looking at Tables 1,2,3, we see that, indeed, we approach the source's (0^{th} order) entropy with AC. It performs poorly against LZW because we assume a "Markov model" there. LZW is considerably faster. If we were to use PPM we should outperform LZW.

A Source Code for the **entropy** Script

Listing 1 entropy.py

```
import os
import sys
import mmap
import math
import pprint
import argparse
import collections
import prettytable

class EntropyScript:
    SYMBOL_LENGTH = 1
    USAGE = "(-e | -p) [-w | -k INT]* SOURCE [SOURCE]*"
    DESCRIPTION = """Simple tool for estimating the #k
           entropy or the probabilities of a source"""

    def __enter__(self): return self
    def __exit__(self, etype,
                 value, etrace): pass

    def __init__(self):
        parser = argparse.ArgumentParser(description  =  self.DESCRIPTION,
                                         usage = "%(prog)s " + self.USAGE)
        argument = parser.add_argument # Shortcut for adding new argument.
        operation = parser.add_mutually_exclusive_group(required  =  True)
        operation = operation.add_argument # Mutually exclusive operation.

        operation("-e", "--estimates", dest = "estimates", action = "store_true",
                  help = """gives the empirical entropy for the sources in bits""")
        operation("-p", "--probabilities", dest = "probabilities", action = "store_true",
                  help = """retrieves the empirical probabilities of the source""")

        argument("-w", "--write-results", dest = "write", action = "store_true",
                                help = """instead of printing to stdout, create file for
    each""")
        argument("-k", "--order", dest = "order", action = "store", default = 0, metavar =
     "INT",
                                help = """builds Markov models of the order k for
    estimations""")
        argument("sources", metavar = "SOURCE", nargs = "+", # Assume user needs to give 1
     more.
                                help = """list of sources to estimate entropy and
    probability""")

        self.arguments = parser.parse_args()
        if (self.arguments.estimates and self.arguments.probabilities) or\
            not (self.arguments.estimates or self.arguments.probabilities):
            parser.print_help()
            sys.exit(-1)

    def execute(self, location = sys.argv[0]):
        k = int(self.arguments.order)
        print("Markov model k =",  k)
        for source in self.arguments.sources:
            with open(source, "r+b") as fd:
                print("Source: " + source)
                mm = mmap.mmap(fd.fileno(), 0)
                size = str(mm.size()) + " bytes"
```

```
53                print("Source length: " + size)
54                probs = self.probabilities(mm, k)
55                print("Source states:", len(probs))
56                mm.close() # Release the MM handle.
57            print("")
58
59            if self.arguments.probabilities:
60                self.print_probabilities(probs, source)
61            elif self.arguments.estimates:
62                entropy = self.estimates(probs)
63                self.print_estimates(entropy, source)
64            else: return 1 # Not reachable?
65            print("") # Separate next file.
66        return 0
67
68    def probabilities(self, source, k):
69        transitions = {}
70        maximum_length = source.size()
71        state = source.read(self.SYMBOL_LENGTH)
72        history = collections.deque(maxlen = k+1)
73        history.append(state) # Starting state.
74
75        while state: # Reads symbols until EOF.
76            for order in range(0, len(history)):
77                transition = [history[i] for i in
78                      range(order, len(history))]
79                transition = b"".join(transition)
80                if transition not in transitions:
81                    transitions[transition] = 0
82                transitions[transition] += 1
83            state = source.read(self.SYMBOL_LENGTH)
84            history.append(state) # Adds to window.
85
86        probabilities = {}
87        for state, frequency in transitions.items():
88            total = maximum_length - len(state) -  1
89            probabilities[state] = frequency / total
90        return probabilities # Joint probabilities.
91
92    def estimates(self, probabilities):
93        order = int(self.arguments.order)
94        entropy = 0.0 # Measured in bits and Sh,  for k.
95        previous_entropy = 0.0 # Joint entropy of k - 1.
96        for state, probability in probabilities.items():
97            information = -math.log(probability, 2)
98            if (len(state) - 1) == order: # For order k.
99                entropy   +=   probability * information
100           if (len(state) == order): # Previous entropy k-1.
101               previous_entropy += probability * information
102       # Follow the chain rule H(X|Y,Z) = H(X,Y,Z) - H(Y,Z).
103       return [entropy - previous_entropy, entropy] # J & C.
104
105   def print_probabilities(self, probabilities, source):
106       if self.arguments.write:
107           with open(source + ".probabilities", "w") as fd:
108               printer = pprint.PrettyPrinter(stream=fd)
109               printer.pprint(probabilities)
110       else: # Just print to the stdconsole.
111           pprint.pprint(probabilities)
112
113   def print_estimates(self, entropies, source):
114       table = prettytable.PrettyTable()
```

```
115         table.add_column("Markov Order", [self.arguments.order])
116         table.add_column("Conditional Entropy", [entropies[0]])
117         table.add_column("Conditional Compression", [entropies[0] / 8.0])
118         table.add_column("Joint Compression", [entropies[1] / 8.0])
119         table.add_column("Joint Entropy", [entropies[1]])
120         if self.arguments.write:
121             with open(source + ".entropies", "w") as fd:
122                 fd.write(table.get_string())
123         else: print(table)
124
125 INITIAL_ERROR_STATUS = -1
126 if __name__ == "__main__":
127     status = INITIAL_ERROR_STATUS
128     with EntropyScript() as tool:
129         status = tool.execute()
130     sys.exit(status)
```

B Source Code for the `liblzw` Library

Listing 2 lzwz.cc

```
1  #include <lzw/buffer.hh>
2  #include <lzw/dictionary.hh>
3  #include <lzw/encoder.hh>
4
5  #include <iostream>
6  #include <fstream>
7
8  void usage(const char* executable) {
9      std::cerr << "usage: " << executable << " "
10                << "<file-path> <compressed-path>"
11                << std::endl;
12 }
13
14 int main(int argc, char** argv) {
15     if (argc != 3) {
16         usage(argv[0]);
17         return 1;
18     }
19
20     std::ifstream file { argv[1], std::ios::binary };
21     if (!file) { // Failed to open file for reading.
22         std::cerr << "File '" << argv[1]
23                   << "' isn't valid."
24                   << std::endl;
25         return 1;
26     }
27
28     std::ofstream compressed { argv[2], std::ios::binary };
29     if (!compressed) { // Failed to open file for writing.
30         std::cerr << "File '" << argv[2]
31                   << "' isn't valid."
32                   << std::endl;
33         return 1;
34     }
35
36     lzw::DictionaryData dictionary_data;
37     lzw::CodeBufferData code_buffer_data;
```

```
38      lzw::Dictionary dictionary { dictionary_data };
39      lzw::CodeBuffer code_buffer { code_buffer_data };
40      lzw::Encoder word_encoder { dictionary };
41
42      char buffer[4096]; // Page size.
43      file.read(buffer, sizeof(buffer));
44      while (file.gcount() > 0) {
45          std::streamsize words { file.gcount() };
46          // Warning!: last element in buffer is the eof char!
47          for (std::streamsize i { 0 }; i < words; ++i) {
48              std::size_t codes { word_encoder.step(buffer[i], code_buffer) };
49              if (codes) compressed.write((char*)code_buffer.rdhead(),
50                                          codes * sizeof(lzw::Code));
51              code_buffer.reset(); // Set buffer head back to 0.
52          }
53
54          file.read(buffer, sizeof(buffer));
55      }
56
57      // There might be a prefixes or head element left
58      // in the buffer, therefore we flush the encoder.
59      if (word_encoder.finish(code_buffer)) {
60          compressed.write((char*)code_buffer.rdhead(),
61                           1 * sizeof(lzw::Code));
62      }
63
64      return 0;
65  }
```

Listing 3 lzwx.cc

```
 1  #include <lzw/buffer.hh>
 2  #include <lzw/dictionary.hh>
 3  #include <lzw/decoder.hh>
 4
 5  #include <iostream>
 6  #include <fstream>
 7
 8  void usage(const char* executable) {
 9      std::cerr << "usage: " << executable << " "
10                << "<file-path> <compressed-path>"
11                << std::endl;
12  }
13
14  int main(int argc, char** argv) {
15      if (argc != 3) {
16          usage(argv[0]);
17          return 1;
18      }
19
20      std::ifstream file { argv[1], std::ios::binary };
21      if (!file) { // Failed to open file for reading.
22          std::cerr << "File '" << argv[1]
23                    << "' isn't valid."
24                    << std::endl;
25          return 1;
26      }
27
28      std::ofstream decompressed { argv[2], std::ios::binary };
29      if (!decompressed) { // Failed to open file for writing.
30          std::cerr << "File '" << argv[2]
```

```
31                  << "' isn't valid."
32                  << std::endl;
33          return 1;
34      }
35
36      lzw::DictionaryData dictionary_data;
37      lzw::WordBufferData word_buffer_data;
38      lzw::Dictionary dictionary { dictionary_data };
39      lzw::WordBuffer word_buffer { word_buffer_data };
40      lzw::Decoder code_decoder { dictionary };
41
42      char buffer[4096]; // Page size.
43      file.read(buffer, sizeof(buffer));
44      while (file.gcount() > 0) {
45          std::streamsize bytes { file.gcount() };
46          // Warning!: last element in buffer is (probably) the eof char?
47          for (std::streamsize i { 0 }; i < bytes; i += sizeof(lzw::Code)) {
48              lzw::Code code { *(lzw::Code*)(buffer + i) };
49              std::size_t words { code_decoder.step(code, word_buffer) };
50              if (words) decompressed.write((char*)word_buffer.rdhead(), words);
51              word_buffer.reset(); // Set the buffer head back to zero.
52          }
53
54          file.read(buffer, sizeof(buffer));
55      }
56
57      return 0;
58  }
```

Listing 4 lzw.hh

```
1  #ifndef LZW_LZW_HH
2  #define LZW_LZW_HH
3
4  #include "lzw/buffer.hh"
5  #include "lzw/definitions.hh"
6  #include "lzw/dictionary.hh"
7  #include "lzw/encoder.hh"
8  #include "lzw/decoder.hh"
9
10 #endif
```

Listing 5 definitions.hh

```
1  #ifndef LZW_DEFINITIONS_HH
2  #define LZW_DEFINITIONS_HH
3
4  #include <cstddef>
5  #include <climits>
6  #include <cstring>
7
8  namespace lzw {
9      // These are the default sizes of the word, and code buffers, which can still
10     // be modified by the user, carefully... Since one code can generate several,
11     // up to 65536 words maximum, we assume the worst case and allow only 1 code.
12     static constexpr std::size_t WORD_BUFFER_SIZE { 1 << 16 }, // The worst case.
13                                  CODE_BUFFER_SIZE { 1 }; // To be safe, only one.
14     static constexpr std::size_t DICTIONARY_ITEMS { 1 << 16 }; // 65536 elements.
15     static constexpr std::size_t BYTE_SIZE_IN_BITS { CHAR_BIT }, // Special case.
```

```
16                                  ALPHABET_SIZE  { 1 << BYTE_SIZE_IN_BITS }; // A.
17      using Byte = unsigned char; // Since the bitsize of characters isn't actually
18      // guaranteed to be always 8 bits in all platforms, we fail to compile these.
19      using Index = unsigned short; // According to the standard, shorts are always
20      // 16-bits long. Therefore, we don't need to make any checks for this type...
21      using String = Byte*; // Is used for storing the sequences in the dictionary.
22      static constexpr std::size_t ENTRY_SIZE { sizeof(Byte) + 4 * sizeof(Index) };
23      static_assert(BYTE_SIZE_IN_BITS == 8, "Your platform is an edge case. Sry.");
24      static constexpr Index EMPTY_STRING = static_cast<Index>(DICTIONARY_ITEMS-1);
25      using Word = Byte; using Code = Index; // Convenience for user's readability.
26      static constexpr Code UNKNOWN_WORD { '?' };static constexpr int LEEWAY { 2 };
27  }
28
29  #endif
```

Listing 6 buffer.hh

```
1   #ifndef LZW_BUFFER_HH
2   #define LZW_BUFFER_HH
3
4   #include "lzw/definitions.hh"
5   #include <type_traits>
6
7   namespace lzw {
8       template<typename T>
9       class BufferData final {
10      public:
11          ~BufferData() { delete[] data; }
12          BufferData(std::size_t size) : size { size } {
13              data = new T[size];
14          }
15
16          BufferData() { // Ugly hack for getting "default" size for a buffer.
17              if (std::is_same<T, Byte>::value)  size = lzw::WORD_BUFFER_SIZE;
18              if (std::is_same<T, Index>::value) size = lzw::CODE_BUFFER_SIZE;
19              data = new T[size];
20          }
21
22          T* data;
23          std::size_t size;
24      };
25
26      using ByteBufferData = BufferData<Byte>;
27      using IndexBufferData = BufferData<Index>;
28      using CodeBufferData  = IndexBufferData;
29      using WordBufferData  = ByteBufferData;
30
31      template<typename T>
32      class Buffer final {
33      public:
34          Buffer() = default;
35          Buffer(BufferData<T>& data)
36              : data { data.data},
37                size { data.size } {  }
38          Buffer(T* data,  std::size_t size)
39              : data { data }, size { size } {  }
40
41          // Utilities for manually controlling
42          // the buffer's head/datum behaviour.
43          void feed(T* data, std::size_t size);
44          void trim(std::size_t location);
```

```
45          void reset() { head = 0; }
46          void ff(std::size_t amount);
47          void rewind(std::size_t amount);
48          T* rdhead() { return data + head; }
49          T* rdbuf() { return data; }
50
51          // Quality of life functions for observing
52          // the behaviour of the byte/index buffer.
53          bool last() const { return head >= size; }
54          std::size_t length() const { return size; }
55          std::size_t left() const { return size - head; }
56          std::size_t location() const { return head; }
57
58          // The meat of the class, reads and writes bytes
59          // or indices. Provides simple abstractions when
60          // writing/reading large amount of data at once.
61          T read(); // Increments the tape head forward...
62          void write(T data); // Writes & increments head.
63          void write(const T* data, std::size_t size);
64          const T* read(std::size_t size);
65
66      protected:
67      private:
68          T* data { nullptr };
69          std::size_t size { 0 };
70          std::size_t head { 0 };
71      };
72
73      template<typename T>
74      void Buffer<T>::feed(T* data, std::size_t size) {
75          this->data = data;
76          this->size = size;
77          this->reset();
78      }
79
80      template<typename T>
81      void Buffer<T>::trim(std::size_t location) {
82          this->size = size;
83          if (head > size) {
84              head = size;
85          }
86      }
87
88      template<typename T> void Buffer<T>::ff(std::size_t amount) { head += amount; }
89      template<typename T> void Buffer<T>::rewind(std::size_t amount) { head -= amount; }
90
91      template<typename T> T Buffer<T>::read() { return data[head++]; }
92      template<typename T> void Buffer<T>::write(T data) { this->data[head++] = data; }
93
94      template<typename T>
95      const T* Buffer<T>::read(std::size_t size) {
96          const T* data { this->data + head };
97          ff(size); // fast-forward the head.
98          return data; // The user will here
99          // handle data reading "manually".
100     }
101
102     template<typename T>
103     void Buffer<T>::write(const T* data, std::size_t size) {
104         std::memcpy(head, data, size);
105         ff(size); // fast-forward!
106     }
```

```
107
108     using ByteBuffer  = Buffer<Byte>;
109     using IndexBuffer = Buffer<Index>;
110     using CodeBuffer  = IndexBuffer;
111     using WordBuffer  = ByteBuffer;
112 }
113
114 #endif
```

Listing 7 dictionary.hh

```
 1 #ifndef LZW_DICTIONARY_HH
 2 #define LZW_DICTIONARY_HH
 3
 4 #include "lzw/definitions.hh"
 5 #include "lzw/buffer.hh"
 6
 7 namespace lzw {
 8     class DictionaryData final {
 9     public:
10         DictionaryData() : DictionaryData(lzw::DICTIONARY_ITEMS) {}
11         DictionaryData(std::size_t size) : size { size } {
12             heads = new Byte[size];
13             prefixes = new Index[size];
14             rights = new Index[size];
15             lefts = new Index[size];
16             roots = new Index[size];
17         }
18
19         ~DictionaryData() {
20             delete[] heads;
21             delete[] prefixes;
22             delete[] rights;
23             delete[] lefts;
24             delete[] roots;
25         }
26
27         Byte* heads; Index* prefixes;
28         Index *rights, *lefts, *roots;
29         std::size_t size;
30     };
31
32     class Dictionary final {
33     public:
34         Dictionary(DictionaryData& data)
35             : Dictionary(data.heads, data.prefixes,
36                          data.rights, data.lefts,
37                          data.roots, data.size) { }
38         Dictionary(Byte* heads, Index* prefixes,
39                    Index* rights, Index* lefts,
40                    Index* roots, std::size_t size)
41             : heads { heads }, prefixes { prefixes },
42               rights { rights }, lefts { lefts },
43               roots { roots }, size { size } {
44                   reset();
45               }
46
47         Index index() const { return head; } // Good for detecting if entry is added.
48         bool full() const { return head >= (EMPTY_STRING - LEEWAY); } // Resets dict.
49         void initialize(); // Assigns the standard 8-bit alphabet to the first items.
50         void reset(); // When dictionary is full, we want to clear/overwrite entires.
```

```
51          Index insert(Index prefix, Byte head); // Will *not* link the tree structure!
52          Index search(Index prefix, Byte head); // Searches AND inserts the entry only
53          // if it doesn't exist, giving either the existing index or the newly created
54          // one. However, this function will properly setup the binary tree structure.
55          Byte* traverse(Byte* buffer, Index index) const; // Populates one byte string
56          // produced after traversing the chain of prefixes and heads while outputting
57          // the results to the temporary buffer. This buffer should be large enough...
58          // NOTE: buffer will be the ending index of the buffer since we need to build
59          // it backwards. Luckily, the first element in the buffer will be returned...
60          bool exists(Index index) { return index < head; } // Assuming we write right.
61
62      protected:
63      private:
64          Index insert_root(Index prefix, Byte head);
65          Index insert_into_tree(Index prefix, Byte head, Index root);
66          Byte* heads { nullptr }; // Contains the parts after the <prefix>, the <head>.
67          Index* prefixes { nullptr }; // Now needed since we need to know parent index.
68          Index* rights { nullptr }, // Stores the right node of the binary search tree.
69                 *lefts { nullptr }, // Same deal here, but here we have the left nodes.
70                 *roots { nullptr }; // Finally, the root of the tree which is <prefix>.
71          std::size_t size { 0 },
72                      head { 0 }; // Enables easier insertion of new dictionary entires.
73      };
74  }
75
76  #endif
```

Listing 8 dictionary.cc

```
 1  #include "lzw/dictionary.hh"
 2
 3  void lzw::Dictionary::initialize() {
 4      // We cheat a bit here and don't construct the tree.
 5      for (std::size_t i { 0 };  i < ALPHABET_SIZE; ++i) {
 6          heads[i] = static_cast<Byte>(i);
 7          prefixes[i] = EMPTY_STRING;
 8          rights[i] = EMPTY_STRING;
 9          lefts[i] = EMPTY_STRING;
10          roots[i] = EMPTY_STRING;
11      }
12
13      heads[EMPTY_STRING] = UNKNOWN_WORD;
14      prefixes[EMPTY_STRING] = EMPTY_STRING;
15      rights[EMPTY_STRING] = EMPTY_STRING;
16      lefts[EMPTY_STRING] = EMPTY_STRING;
17      roots[EMPTY_STRING] = EMPTY_STRING;
18  }
19
20  void lzw::Dictionary::reset() {
21      head = ALPHABET_SIZE;
22      initialize();
23  }
24
25  lzw::Index lzw::Dictionary::insert(lzw::Index prefix, lzw::Byte head) {
26      heads[this->head] = head;
27      prefixes[this->head] = prefix;
28      rights[this->head] = EMPTY_STRING;
29      lefts[this->head] = EMPTY_STRING;
30      roots[this->head] = EMPTY_STRING;
31      return (this->head++);
32  }
```

```
33
34 lzw::Byte* lzw::Dictionary::traverse(lzw::Byte* buffer, Index index) const {
35     lzw::Index current { index };
36     while (current != EMPTY_STRING) {
37         *(--buffer) = heads[current];
38         current  = prefixes[current];
39     }
40
41     return buffer;
42 }
43
44 lzw::Index lzw::Dictionary::search(lzw::Index prefix, lzw::Byte head) {
45     if (prefix == EMPTY_STRING) return static_cast<Index>(head);
46     Index entry { EMPTY_STRING }; // Either inserted or existing entries.
47     if (roots[prefix] == EMPTY_STRING) entry = insert_root(prefix, head);
48     else entry = insert_into_tree(prefix, head, roots[prefix]);
49     return entry;
50 }
51
52 lzw::Index lzw::Dictionary::insert_root(lzw::Index prefix, lzw::Byte head) {
53     Index entry { insert(prefix, head) };
54     roots[prefix] = entry;
55     return entry;
56 }
57
58 lzw::Index lzw::Dictionary::insert_into_tree(lzw::Index prefix, lzw::Byte head, Index root
       ) {
59     lzw::Index current { root };
60     lzw::Index previous { root };
61     while (current != EMPTY_STRING) {
62         previous = current; // Needed in order to link tree.
63         if (head < heads[current]) current = lefts[current];
64         else if (head > heads[current]) current = rights[current];
65         else return current;
66     }
67
68     Index entry { insert(prefix, head) };
69     // Below we link the new entry to the binary tree.
70     if (head < heads[previous]) lefts[previous] = entry;
71     else rights[previous] = entry;
72     return entry;
73 }
```

Listing 9 encoder.hh

```
 1 #ifndef LZW_ENCODER_HH
 2 #define LZW_ENCODER_HH
 3
 4 #include "lzw/definitions.hh"
 5 #include "lzw/dictionary.hh"
 6 #include "lzw/buffer.hh"
 7
 8 namespace lzw {
 9     class Encoder final {
10     public:
11         Encoder(Dictionary& dictionary) : dictionary { dictionary } {  }
12         // Each iteration will consume 'amount' words from the word buffer and will
13         // then output a certain returned amount of codewords out to the code buff.
14         std::size_t step(Word word, CodeBuffer& code_buffer); // Step word encoder.
15         void restart() { word_prefix = EMPTY_STRING; current_word = UNKNOWN_WORD; }
16         std::size_t finish(CodeBuffer& code_buffer); // Writes if anything is left.
```

```
17
18      protected:
19      private:
20          Dictionary& dictionary;
21          Word current_word { UNKNOWN_WORD };
22          Index word_prefix { EMPTY_STRING };
23      };
24  }
25
26  #endif
```

Listing 10 encoder.cc

```
 1  #include "lzw/encoder.hh"
 2
 3  std::size_t lzw::Encoder::step(lzw::Word current_word, lzw::CodeBuffer& code_buffer) {
 4      Index freshest_string { dictionary.index() };
 5      Index current_string { dictionary.search(word_prefix, current_word) };
 6      bool string_exists { freshest_string == dictionary.index() };
 7      std::size_t codes_generated { 0 };
 8
 9      // Here we see if the dictionary search resulted
10      // in a new entry. If not we simply continue the
11      // reading process until we find a unique string
12      // and then inserting it to the dictionary. See:
13      if (string_exists) word_prefix = current_string;
14      else { // Below the string didn't exist in dict.
15          code_buffer.write(word_prefix);
16          word_prefix = current_word;
17          // We require a new context
18          // since our dictionary has
19          // limited storage, it also
20          // has the added benefit of
21          // clearing unused strings.
22          if (dictionary.full()) {
23              dictionary.reset();
24              this->restart();
25          }
26          codes_generated  +=  1;
27      }
28
29      // "Rewinds" the buffer for reading.
30      code_buffer.rewind(codes_generated);
31      return codes_generated;
32  }
33
34  std::size_t lzw::Encoder::finish(lzw::CodeBuffer& code_buffer) {
35      std::size_t codes_generated { 0 };
36      if (word_prefix != EMPTY_STRING) {
37          code_buffer.write(word_prefix);
38          codes_generated += 1;
39      }
40
41      // "Rewinds" the buffer for reading.
42      code_buffer.rewind(codes_generated);
43      return codes_generated;
44  }
```

Listing 11 decoder.hh

```
#ifndef LZW_DECODER_HH
#define LZW_DECODER_HH

#include "lzw/definitions.hh"
#include "lzw/dictionary.hh"
#include "lzw/buffer.hh"

namespace lzw {
    class Decoder final {
    public:
        Decoder(Dictionary& dictionary) : dictionary { dictionary } {  }
        // Each iteration will consume, usually, a single code from the code buffer
        // and will then output an arbitrary (max 2^16 - 1) amount of words to buf.
        std::size_t step(Code code, WordBuffer& word_buffer); // Step word encoder.
        void restart() { previous_index = EMPTY_STRING;
                         previous_index_head = UNKNOWN_WORD; }

    protected:
    private:
        Dictionary& dictionary;
        Index previous_index { EMPTY_STRING };
        Byte previous_index_head { UNKNOWN_WORD };
    };
}

#endif
```

Listing 12 decoder.cc

```
#include "lzw/decoder.hh"

#include <iostream>

std::size_t lzw::Decoder::step(lzw::Code current_code,
                               lzw::WordBuffer& word_buffer) {
    std::size_t decoded_words { 0 };
    if (previous_index == EMPTY_STRING) {
        word_buffer.write(current_code);
        previous_index_head = current_code;
        previous_index = current_code;
        decoded_words += 1;
    } else {
        if (dictionary.exists(current_code)) {
            lzw::Byte* buffer { word_buffer.rdbuf() };
            lzw::Byte* end_buffer { buffer + word_buffer.length() };
            lzw::Byte* start_buffer { dictionary.traverse(end_buffer, current_code) };

            long int buffer_size { end_buffer - start_buffer };
            word_buffer.ff(word_buffer.length());

            decoded_words += buffer_size;
            previous_index_head = *start_buffer;
            dictionary.insert(previous_index, *start_buffer);
        } else {
            lzw::Index new_entry_index { dictionary.insert(previous_index,
    previous_index_head) };

            lzw::Byte* buffer { word_buffer.rdbuf() };
            lzw::Byte* end_buffer { buffer + word_buffer.length() };
```

```
30              lzw::Byte* start_buffer { dictionary.traverse(end_buffer, new_entry_index) };
31
32              long int buffer_size { end_buffer - start_buffer };
33              word_buffer.ff(word_buffer.length());
34              decoded_words += buffer_size;
35              previous_index_head = *start_buffer;
36          }
37
38          previous_index = current_code;
39          if (dictionary.full()) {
40              dictionary.reset();
41              this->restart();
42          }
43      }
44
45      word_buffer.rewind(decoded_words);
46      return decoded_words;
47  }
```

C Source Code for the **libaac** Library

Listing 13 aacz.cc

```
1  #include "aac/encoder.hh"
2
3  #include <string>
4  #include <fstream>
5  #include <iostream>
6
7  int usage(const char* program) {
8      std::cerr << "usage: "
9                << program << " "
10               << "<input-file> "
11               << "<output-file>"
12               << std::endl;
13     return 1;
14 }
15
16 int main(int argc, char** argv) {
17     if (argc != 3) return usage(argv[0]);
18     std::string input_path  { argv[1] },
19                 output_path { argv[2] };
20
21     // Only allow opening of existing input
22     // files, but allow non-existing output.
23     std::ifstream input_file { input_path,   std::ios::binary};
24     if (!input_file)  return usage(argv[0]);
25     std::ofstream output_file { output_path, std::ios::binary};
26     if (!output_file) return usage(argv[0]);
27
28     aac::Statistics default_statistics; // Uniform.
29     aac::Encoder ascii_encoder { default_statistics };
30
31     aac::Byte byte { 0 };
32     std::size_t head { 0 };
33     aac::Byte bit_buffer[aac::MAX_PENDING];
34     input_file.read((char*)&byte, sizeof(aac::Byte));
35
```

```
36      while (input_file.gcount()) {
37          std::size_t bits_written { ascii_encoder.step(byte, bit_buffer, head) };
38          head += bits_written; // We might have to delay writing them until full.
39          if ((head >> 3) != 0) { // That is, until we have written at least a byte.
40              std::size_t bytes_written { head >> 3 };
41              std::size_t bits_remaining { head - (bytes_written << 3) };
42              output_file.write((char*)bit_buffer, bytes_written);
43
44              // Copy the remaining bits to head of the buffer.
45              for (std::size_t i { 0 }; i < bits_remaining; ++i) {
46                  int b { (bit_buffer[bytes_written] >> (7 - i)) & 0x01 };
47                  bit_buffer[0] &= ~(1 << (7 - i));
48                  bit_buffer[0] |= b << (7 - i);
49              }
50
51              head = bits_remaining; // Next bits will be put after.
52          }
53
54          input_file.read((char*)&byte, sizeof(aac::Byte));
55      }
56
57      ascii_encoder.exit(bit_buffer, head);
58      std::size_t bytes_written { head >> 3 };
59      output_file.write((char*)bit_buffer, bytes_written);
60      std::size_t bits_remaining { head - (bytes_written << 3) };
61
62      if (bits_remaining > 0) { // Still stuff to copy.
63          // Copy the remaining bits to head of the buffer.
64          for (std::size_t i { 0 }; i < bits_remaining; ++i) {
65              bool b { (bit_buffer[bytes_written] >> (7 - i)) & 0x01 };
66              bit_buffer[0] &= ~(1 << (7 - i));
67              bit_buffer[0] |= b << (7 - i);
68          }
69
70          // Finally, clear any padding bits added to bytes.
71          for (std::size_t i { bits_remaining }; i < 7; ++i)
72              bit_buffer[0] &= ~(1 << (7 - i));
73          output_file.write((char*)bit_buffer, 1);
74      }
75
76      return 0;
77  }
```

Listing 14 aacx.cc

```
 1  #include "aac/decoder.hh"
 2
 3  #include <string>
 4  #include <fstream>
 5  #include <iostream>
 6
 7  int usage(const char* program) {
 8      std::cerr << "usage: "
 9                << program << " "
10                << "<input-file> "
11                << "<output-file>"
12                << std::endl;
13      return 1;
14  }
15
16  int main(int argc, char** argv) {
```

```
17      if (argc != 3) return usage(argv[0]);
18      std::string input_path  { argv[1] },
19                  output_path { argv[2] };
20
21      // Only allow opening of existing input
22      // files, but allow non-existing output.
23      std::ifstream input_file { input_path,   std::ios::binary};
24      if (!input_file)  return usage(argv[0]);
25      std::ofstream output_file { output_path, std::ios::binary};
26      if (!output_file) return usage(argv[0]);
27
28      aac::Statistics default_statistics; // Uniform.
29      aac::Decoder ac_decoder { default_statistics };
30
31      int decoder_status;
32      aac::Byte byte { 0 };
33      std::size_t head { 0 };
34      aac::Byte bit_buffer[aac::MAX_PENDING];
35      input_file.read((char*)&byte, sizeof(aac::Byte));
36
37      do {
38          std::size_t head_byte { (head >> 3) };
39          while (ac_decoder.feed(byte, head - 8*head_byte))
40              input_file.read((char*)&byte, sizeof(aac::Byte));
41          decoder_status = ac_decoder.step(); // Feed the data!
42          output_file.write((char*)&decoder_status, 1);
43      } while (decoder_status != -1);
44
45      return 0;
46  }
```

Listing 15 aac.hh

```
1  #ifndef AAC_AAC_HH
2  #define AAC_AAC_HH
3
4  #include "aac/configs.hh"
5  #include "aac/encoder.hh"
6  #include "aac/decoder.hh"
7  #include "aac/statistics.hh"
8
9  #endif
```

Listing 16 configs.hh

```
1  #ifndef AAC_CONFIGS_HH
2  #define AAC_CONFIGS_HH
3
4  #include <utility>
5  #include <cstdint>
6  #include <cstddef>
7
8  namespace aac {
9      using Byte = unsigned char;
10     using Range = std::uint32_t;
11     using Bound = std::uint32_t;
12     using Symbol = unsigned char;
13     using Counter = std::uint16_t;
14     using Codeword = unsigned char;
```

```
15      using Interval = std::pair<Counter, Counter>;
16      static const std::size_t SYMBOLS { 257 };
17      static const std::size_t MAX_PENDING { 64 };
18      static const std::size_t END_OF_FILE { 256 };
19      static const Counter EOF_SYMBOL { 256 };
20
21      // We limit ourselves to 16-bit bounds because
22      // we are using multiplication between bounds.
23      static const Counter LOWER_BOUND {  0x0000  };
24      static const Counter UPPER_BOUND {  0xFFFF  };
25      static const Counter COUNT_BOUND {  0x3FFF  };
26      // Maximum bound for frequencies (2 bit less).
27      static const Counter SIGNIFICANT {  0x8000  };
28      static const Counter SECOND_MOST {  0x4000  };
29
30      // Here we define the bound limits for number convergance.
31      static const Counter ONE_QUARTER { (UPPER_BOUND + 1) >> 2 };
32      static const Counter TWO_QUARTERS { ONE_QUARTER << 1 };
33      static const Counter THREE_QUARTERS { ONE_QUARTER + TWO_QUARTERS };
34  }
35
36  #endif
```

Listing 17 statistics.hh

```
 1  #ifndef AAC_STATISTICS_HH
 2  #define AAC_STATISTICS_HH
 3
 4  #include "aac/configs.hh"
 5  #include <cstddef>
 6
 7  namespace aac {
 8      class Statistics final {
 9      public:
10          void clear(Counter*);
11          Statistics() { clear(nullptr); }
12          Statistics(Counter* bootstrap) {
13              clear(bootstrap);
14          }
15
16          Counter total() const;
17          Counter find(Counter) const;
18          Interval symbol(std::size_t) const;
19          void update(std::size_t);
20          bool frozen() const;
21
22      private:
23          // Cumulative frequency counts for
24          // the symbols in the alphabet+EOF.
25          Counter frequencies[SYMBOLS] {  };
26          bool frequencies_frozen { false };
27      };
28  }
29
30  #endif
```

Listing 18 statistics.cc

```
1  #include "aac/statistics.hh"
```

```
2
3  void aac::Statistics::clear(Counter* bootstrap) {
4      if (bootstrap == nullptr) {
5          for (std::size_t i { 0 }; i < SYMBOLS; ++i)
6              frequencies[i] = (i + 1); // Cumu freq.
7      } else {
8          frequencies[0] = bootstrap[0];
9          // Load in existing statistics on the data.
10         for (std::size_t i { 1 }; i < SYMBOLS; ++i)
11             frequencies[i] = bootstrap[i] + frequencies[i - 1];
12         frequencies_frozen = true;
13     }
14 }
15
16 aac::Counter aac::Statistics::find(Counter codeword) const {
17     if (codeword >= 0.0 && codeword < frequencies[0]) return 0;
18     for (std::size_t i { 1 }; i < SYMBOLS; ++i) {
19         if (codeword >= frequencies[i - 1]
20             && codeword < frequencies[i])
21             return i;
22     }
23
24     // Shouldn't happen?
25     return EOF_SYMBOL;
26 }
27
28 aac::Counter aac::Statistics::total() const {
29     return frequencies[END_OF_FILE];
30 }
31
32 aac::Interval aac::Statistics::symbol(std::size_t symbol) const {
33     if (symbol == 0) return aac::Interval { 0.0, frequencies[0] };
34     return aac::Interval { frequencies[symbol - 1],
35                            frequencies[symbol] };
36 }
37
38 void aac::Statistics::update(std::size_t symbol) {
39     if (frequencies_frozen) return;
40     // Update the cumulative frquencies.
41     for (; symbol < SYMBOLS; ++symbol) {
42         if (++frequencies[symbol] >= COUNT_BOUND)
43             frequencies_frozen = true;
44     }
45 }
46
47 bool aac::Statistics::frozen() const {
48     return frequencies_frozen;
49 }
```

Listing 19 encoder.hh

```
1  #ifndef AAC_ENCODER_HH
2  #define AAC_ENCODER_HH
3
4  #include "aac/configs.hh"
5  #include "aac/statistics.hh"
6  #include <cstddef>
7
8  namespace aac {
9      class Encoder final {
10     public:
```

```
11          Encoder() = default;
12          Encoder(Statistics statistics)
13              : statistics { statistics } {}
14
15          // Encodes the given char symbol
16          // and either produces some bits
17          // or nil; writes to the buffer.
18          // Returns the number of bits...
19          std::size_t step(Counter, Byte*,
20                           std::size_t b);
21          // Write out remaining bits ok??
22          std::size_t exit(Byte*   buffer,
23                           std::size_t b);
24
25      private:
26          std::size_t write(bool,   Byte*,
27                           std::size_t b);
28          Counter buffered_bits { 0 };
29          Statistics statistics;
30          Counter upper { UPPER_BOUND },
31                  lower { LOWER_BOUND };
32      };
33  }
34
35  #endif
```

Listing 20 encoder.cc

```
 1  #include "aac/encoder.hh"
 2
 3  std::size_t aac::Encoder::step(Counter symbol, Byte* buffer,
 4                                 std::size_t buffer_bitnum) {
 5      std::size_t bits_written { 0 };
 6      Range length { (Range)upper - (Range)lower + 1 };
 7      Counter total_symbols { statistics.total() };
 8      Interval symbol_interval { statistics.symbol(symbol) };
 9      statistics.update(symbol); // Increment frequencies.
10
11      Bound rescale { ((Bound)length * (Bound)symbol_interval.second) };
12      rescale /= (Bound)total_symbols;
13      upper = lower + rescale - 1;
14
15      rescale = ((Bound)length * (Bound)symbol_interval.first);
16      rescale /= (Bound)total_symbols;
17      lower = lower + rescale;
18
19      while (true) {
20          // Matching most significant bit for lower and upper.
21          if ((lower & SIGNIFICANT) == (upper & SIGNIFICANT)) {
22              bool set { (upper & SIGNIFICANT) != 0x0000 };
23              std::size_t head { buffer_bitnum+bits_written };
24              bits_written += write(set, buffer, head);
25          // Non-matching most significant bit and also non-matching
26          // second most significant... We will never converge here.
27          // Therefore, handle this case by letting some bit buffer.
28          } else if ((lower & SECOND_MOST) && !(upper & SECOND_MOST)) {
29              // Remove the SMSB from the stream...
30              lower = lower & ~(SIGNIFICANT | SECOND_MOST);
31              upper = upper | SECOND_MOST;
32              ++buffered_bits;
33          // Everything is ok.
```

```
34          } else break;
35
36          lower = lower << 1;
37          upper = upper << 1;
38          upper = upper | 1;
39      }
40
41      return bits_written;
42  }
43
44  std::size_t aac::Encoder::exit(Byte* buffer, std::size_t buffer_bit) {
45      std::size_t bits_written { step(EOF_SYMBOL, buffer, buffer_bit) };
46      std::size_t remaining { write((lower & SECOND_MOST) != 0, buffer,
47                                    buffer_bit + bits_written) };
48      return bits_written + remaining;
49  }
50
51  std::size_t aac::Encoder::write(bool state, Byte* buffer,
52                                  std::size_t buffer_bit) {
53      std::size_t bits_written { 0 };
54      std::size_t byte { buffer_bit >> 3 };
55      std::size_t bit_offset { buffer_bit - byte*8 };
56      std::size_t bit { 7 - bit_offset };
57
58      Byte data { buffer[byte] };
59      data = data & ~(1 << bit);
60      data = data | state << bit;
61      buffer[byte] = data;
62      ++bits_written;
63      ++buffer_bit;
64
65      while (buffered_bits--) {
66          byte = buffer_bit >> 3;
67          bit_offset = buffer_bit - byte*8;
68          bit = 7 - bit_offset;
69
70          data = buffer[byte];
71          data = data & ~(1 << bit);
72          data = data | !state << bit;
73          buffer[byte] = data;
74          ++bits_written;
75          ++buffer_bit;
76      }
77
78      buffered_bits = 0;
79      return bits_written;
80  }
```

Listing 21 decoder.hh

```
1  #ifndef AAC_DECODER_HH
2  #define AAC_DECODER_HH
3
4  #include "aac/configs.hh"
5  #include "aac/statistics.hh"
6
7  namespace aac {
8      class Decoder final {
9      public:
10         Decoder() = default;
11         Decoder(Statistics statistics)
```

```
12              : statistics { statistics } {
13              upper = statistics.total();
14          }
15
16          // Decodes the given byte codewo
17          // rd. Either produces some bits
18          // or nil; writes to the buffer.
19          // Returns the number of bits...
20          int  step(); // Use feed to load
21          // No need to terminate, we also
22          // encoded the end of file byte.
23          // However, we do need to locate
24          // the symbol for a given range.
25
26          // Feed some bits into the number.
27          std::size_t feed(Counter, std::size_t);
28          bool fed() { return number_bits == 0; }
29
30      private:
31          std::size_t write(bool,   Byte*,
32                            std::size_t b);
33          Counter buffered_bits { 0 };
34          Statistics statistics;
35          Counter number_bits { 16 };
36          Counter number { 0x0000 };
37          Counter upper { SYMBOLS },
38                  lower { 0 };
39      };
40  }
41
42  #endif
```

Listing 22 decoder.cc

```
1  #include "aac/decoder.hh"
2
3  #include <iostream>
4
5  int aac::Decoder::step() {
6      Range length { (Bound)upper - (Bound)lower + 1 };
7      Bound unscaled { (Bound)number - (Bound)lower + 1 };
8      unscaled *= statistics.total() - 1;
9      unscaled /= length;
10
11
12     Counter symbol { statistics.find(unscaled) };
13     if (symbol==END_OF_FILE) return -1;
14
15     Counter total_symbols { statistics.total() };
16     Interval symbol_interval { statistics.symbol(symbol) };
17     Bound rescale { ((Bound)length * (Bound)symbol_interval.second) };
18     rescale /= (Bound)total_symbols;
19     upper = lower + rescale - 1;
20
21     rescale = ((Bound)length * (Bound)symbol_interval.first);
22     rescale /= (Bound)total_symbols;
23     lower = lower + rescale;
24     statistics.update(symbol);
25
26
27     while (true) {
```

```
28          // Matching most significant bit for lower and upper.
29          if ((lower & SIGNIFICANT) == (upper & SIGNIFICANT)) {
30          // Non-matching most significant bit and also non-matching
31          // second most significant... We will never converge here.
32          // Therefore, handle this case by letting some bit buffer.
33          } else if ((lower & SECOND_MOST) && !(upper & SECOND_MOST)) {
34              lower = lower & ~(SIGNIFICANT | SECOND_MOST);
35              upper = upper | SECOND_MOST;
36              number ^= SECOND_MOST;
37          } else break;
38
39          ++number_bits;
40          number = number << 1;
41          lower = lower << 1;
42          upper = upper << 1;
43          upper = upper | 1;
44      }
45
46      return symbol;
47  }
48
49  std::size_t aac::Decoder::feed(Counter byte, std::size_t start) {
50      std::size_t length { 8 - start };
51      std::size_t pending;
52      if (length > number_bits) { pending = number_bits ; byte >>= (length - pending); }
53      else { pending = length ;  byte <<= (number_bits - pending);}
54      number |= byte;
55      number_bits -= pending;
56      return pending;
57  }
58
59  std::size_t aac::Decoder::write(bool state, Byte* buffer,
60                                  std::size_t buffer_bit) {
61      std::size_t bits_written { 0 };
62      std::size_t byte { buffer_bit >> 3 };
63      std::size_t bit_offset { buffer_bit - byte*8 };
64      std::size_t bit { 7 - bit_offset };
65
66      Byte data { buffer[byte] };
67      data = data & ~(1 << bit);
68      data = data | state << bit;
69      buffer[byte] = data;
70      ++bits_written;
71      ++buffer_bit;
72
73      while (buffered_bits--) {
74          byte = buffer_bit >> 3;
75          bit_offset = buffer_bit - byte*8;
76          bit = 7 - bit_offset;
77
78          data = buffer[byte];
79          data = data & ~(1 << bit);
80          data = data | !state << bit;
81          buffer[byte] = data;
82          ++bits_written;
83          ++buffer_bit;
84      }
85
86      buffered_bits = 0;
87      return bits_written;
88  }
```