PLAB-2 (TDT-4113) Project 3:
Data Compression

Purpose:

- Achieve a basic understanding of a few standard data-compression algorithms.
- Implement, from scratch, the compression algorithm that underlies ZIP files, the Lempel-Ziv procedure.
- Experiment with Lempel-Ziv compression alone (on image files) and in combination with two other coding approaches: ASCII and Huffman.

Practical Information:

- This project must be worked on individually.
- It must be solved using object-oriented Python.
- This project uses the module PythonLabs.Bitlab for Huffman Coding. To get PythonLabs, check the "Install PythonLabs" section of this site: http://ix.cs.uoregon.edu/ conery/eic/python/installation.html
- It also employs the file kdprims.py, which will be provided on the PLAB-2 Wiki.
- You must demonstrate a working version of this project by 2 pm (kl 14:00) on Wednesday, September 23, 2015.

1 Introduction

Computation is all about transferring and transforming data, which can be represented in many different ways. The same information may exist as an array of red-green-blue pixel values, or as a long string of integers, or as an even longer string of bits. Different modules in a computer system require different representations, so transformations are very commonplace inside the machine, and a sizeable fraction of the time that we spend programming is devoted to converting one representation into another.

In this project, you will examine a few different representations: text strings and bit strings, and you will investigate a few methods for transforming text to bits, bits to text, and bits to other sequences of bits. In many cases, these transformations constitute compressions: the output takes up less storage space than the input. Given the tremendous amounts of data processed by and sent between modern computers, it seems pertinent for every computer professional to have at least a basic familiarity with some of the more popular data compression methods. You will gain that familiarity by hands-on experience.

2 Three Coding Systems

In this project, you will define three different types of coding systems: ASCII, Huffman and Lempel-Ziv, each of which is described in this document (and the first two are covered in Chapter 8 of your textbook). Each system will correspond to a Python class, with methods for encoding and decoding strings.

At the input/output level, each of these classes will work with Python **strings**: each will take in a string to encode (as another string). The input may be a text string or a binary string (i.e., a string consisting only of 0's and 1's), while the output will, in all 3 cases, be a binary string. Since strings are the common denominator, it will be possible to pass the output of one coding system to the input of another (though not all combinations will be equally interesting). When the encodings involve compression, this makes it trivial to compress a string several times in succession. For example, the result of Huffman compression can be further compressed with the Lempel-Ziv algorithm.

Each of these three classes should inherit from a superclass called **Coder** (or whatever you want to call it). Coder should include at least the following two methods:

1. **gen_message_from_file(filepath):** This opens the file specified by the input argument, reads the entire file, and converts the contents into one large string. If you prefer to remove newlines from the contents, that is fine, but punctuation should be preserved, as should at least the bare minimum of spacing between symbol sequences. You are also free to convert all symbols to lower case. For example, it is fine to convert "BIG SALE on socks" to "big sale on socks", but not to "bigsaleonsocks": word boundaries need to be preserved. The large string produced by this method then constitutes the **message** that a coding system will encode.
2. **encode_decode_test(message):** This performs the test that is the cornerstone of all comparisons in this project. Given the input message (M), this method will encode it, producing E. Then it will decode E to produce, D, which hopefully matches M. Your code should **print (to the screen) ALL** the following information:
   - The complete contents of strings M, E and D. In some cases, these will be huge. Just be sure to print these out BEFORE the next three items so that those three are easy to find on your screen, i.e., they will be at the bottom.
   - A clear verification that D = M. Your code can simply test if "D == M" (Python handles the details of the string comparison) and print out an "equal" or "not equal" message.
   - The lengths of M, E and D. Python's **len** function should do the job.
   - The compression fraction, which is
   $$(1 - \frac{len(E)}{L}) \tag{1}$$
   where L is len(M) when you are encoding a binary string as another binary string, and L = 8*len(M) when you are encoding a text string as a binary string (since each text symbol would normally take up 8 bits when using ASCII coding).

   For example, when encode_decode_test is used for Lempel-Ziv compression of an input message consisting of twenty zeros, the screen output should look something like this:

```
Original Message:
 00000000000000000000
Encoded Message:
 0101001101000101
Decoded Message:
```

```
 00000000000000000000
The original and decoded messages are equal
Lengths of message, encoding and decoding:  20 16 20
Compression fraction: 0.19999999999999996
```

3 ASCII Coding

We use ASCII codes to convert symbols into numbers, but in a manner that involves no actual compression. Each symbol simply maps to an integer in the range 0 to 255. Thus one byte (8 bits) are sufficient to represent the integer, and thus, the symbol. ASCII is a common means of converting text files to binary files. Compression can then occur at the binary level by detecting recurring patterns of 0's and1's and devising binary representations that efficiently encode this repetition.

The basic conversion from text to ASCII is handled by the **ord** function in Python. For example, ord('a') $\rightarrow$ 97, ord('j') $\rightarrow$ 106, and ord('J') $\rightarrow$ 74 (it is case sensitive). It also applies to punctuation: ord('!') $\rightarrow$ 33, ord('.') $\rightarrow$ 46, and ord('@') $\rightarrow$ 64. To go the opposite direction, from ASCII codes to symbols, use the **chr** function: chr(97) $\rightarrow$ 'a', chr(74) $\rightarrow$ 'J', and chr(33) $\rightarrow$ '!'.

To encode a text message as an ASCII bit string, simply convert each message symbol to its ASCII code (using the ord function), and then convert that code to 8 bits. Then, append all of the 8-bit sequences into one (very long) bit string. That's your ASCII encoding. Since a single space also has an ASCII code, as does a new-line delimiter, you can encode relatively complex messages spanning many lines (without having to preprocess the message to, for example, remove whitespace).

To decode the ASCII bit string, pull off 8-bit sequences and convert them first to their corresponding integer and then to their symbol, using the chr function. When merged into one long string, these symbols will recreate the original message, including all punctuation, spaces and carriage returns.

3.1 Implementing ASCII Coding

To build a basic ASCII coder, define a Python class, **Asciicoder** (or whatever name you prefer), and define at least these two methods:

1. **encode(str):** This converts a message string (str) into a (possibly very long) string of ASCII bits.
2. **decode(bits):** This converts a bit string into a text message.

Since we will only be working with strings in this project, the Asciicoder will actually lead to an EXPANSION of your original message string: each symbol will be replaced by 8 binary values; and in a string, 0's, 1's and all other symbols take up the same amount of space. Essentially, every 0 and 1 in the string is taking up a byte (8 bits) by itself, since each 0/1 is actually being represented by **its** ASCII code: ord('0') $\rightarrow$ 48, and ord('1') $\rightarrow$ 49 in the string. If this were a real ASCII encoder, we would not be representing the encoded text as a string, but as real bits and bytes. Our main goals with ASCII are to to a) use it as a worst case compression algorithm as compared to Huffman and Lempel-Ziv encoding, and b) convert text into 1's and 0's so that compression algorithms that normally take binary input (e.g. Lempel-Ziv) become applicable.

To verify this expansion, simply create an Asciicoder object (ac) and then call ac.encode_decode_test('hello'). This should give an encoded bit string of length 40 and a compression fraction of 0.

4 Huffman Coding

In the ASCII codes used above, each symbol requires the same number of bits (8), and regardless of how often the same symbol (or sequence of symbols) appears – i.e. regardless of the message's **predictability** – there is no significant compression. Each letter in the source requires 8 bits in the target: no more, but, unfortunately, no less. This does make decoding quite easy – we just grab 8 bits at a time and convert them to their proper symbol – but we fail to exploit any known patterns of symbol usage (such as the fact that 'q' is almost always followed by 'u' in English) that might facilitate compression.

Frequency-based coding schemes, such as Huffman coding, take advantage of a very coarse level of predictability, namely the known frequency of occurence of each symbol, to support data compression.

As a simple example of this basic concept, consider a fictitious military scenario in which two battlefield regiments have very limited communication capabilities over a simple channel. The two basic signals are a short and a long pulse, denoted by 0 and 1, respectively; and the line must be used sparingly, for fear of both enemy detection and general wear-and-tear. It will presumably be used just once each day, at a previously-agreed-upon time.

Assume that conventional battlefield wisdom indicates that the only relevant messages in these situations are the five shown in Figure 1. In this example, each such message constitutes a *symbol*, e.g., A - E. The basic coding-theory problem is then: What combinations of short and long pulses should be used for each of the 5 symbols so as to minimize the total number of pulses sent over the channel in the course of the entire battle?

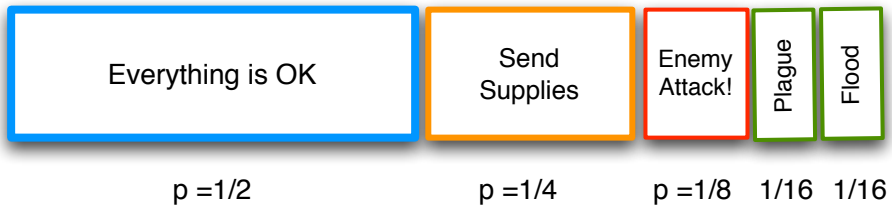

Figure 1: A simple set of military messages along with the probability (reflected in the width of each rectangle) of them being sent, i.e., of their underlying event actually occurring.

The probabilities of the underlying situations (attack, flood, plague, etc.) turn out to be of extreme importance in designing efficient codes, with the basic insight that less-frequent events should have longer codes. A naive approach ignores these probabilities and simply assigns 3 bits (pulses) to each symbol, since $\lceil log_2(5) \rceil = 3$. In contexts where many signals will be sent in sequence, the assumption of a common fixed length (k) for all symbols simplifies the decoding process, since the receiver knows to pull out and decode k bits at a time. Thus, there is no need for symbol-ending delimiter signals, which, of course, require extra bits.

However, these fixed-length codes waste bits compared to the most efficient codes, which manage to use variable-length symbol codes without introducing delimiters. This is achieved by *prefix coding*, wherein symbol codes are designed so that the full code for any symbol is never the beginning (prefix) of another

symbol. In decoding a message in prefix code, as soon as a variable-length segment matches the code for any symbol, the decoder can immediately pause and record that symbol; upon resuming reading from the channel, the decoder knows that the next signal is the beginning of a new symbol.

When there is no statistical relationship between neighboring symbols in a stream, the most efficient coding scheme turns out to be the **Huffman code**, a well-known prefix code designed by Donald Huffman at MIT in the early 1950's. Huffman codes are designed by an ingeniously simple algorithm that builds a labelled tree based on the symbol frequencies. In a nutshell, the algorithm clusters symbols based on frequency, with the least-frequent symbols clustering earliest (and thus ending up at the bottom of the tree). Frequent symbols cluster later, end up near the top of the tree, and thus have shorter codes.

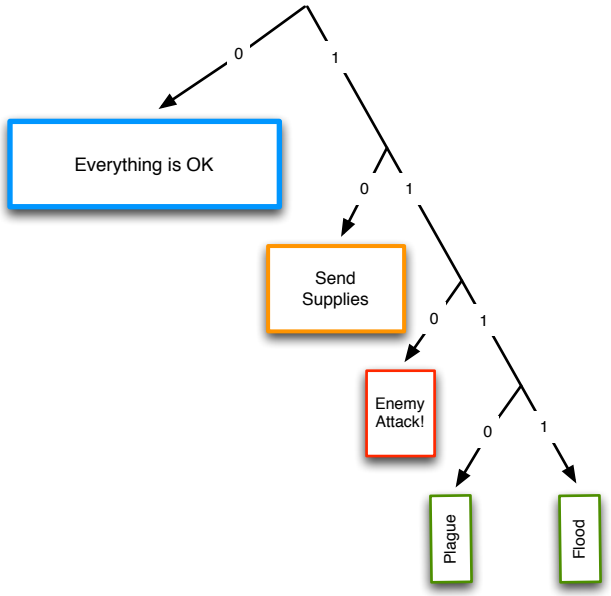

Figure 2: A Huffman tree (with codes read off along the branches) for the simple military example. Note that the most likely messages have the shortest codes and that no code is the prefix of another code.

Figure 2 presents the Huffman tree for the 5 military messages, and Figure 3 details the codes (read directly from the tree as the string of labels from the root to the particular event), the number of bits per code, and the *information content* of each symbol. As shown, the information content of a symbol equals the negative logarithm of its corresponding event's probability, which is also referred to as the *surprisal* of an event. Thus, the occurrence of rare events constitutes more information than that of common events. This information content also equals the number of bits needed to encode the symbol under an optimal (Huffman) protocol, as shown in Figure 3 . Both of these definitions of information content are used in the information-theory literature.

Given these codes, the military regiments would normally exchange a simple 1-bit message: "0", meaning everything is okay. If one outpost were responsible for receiving messages from several regiments, bundling them, and sending the summary to a high commander, a typical message might be 00110010, indicating that regiments 1 and 2 are ok (0,0), regiment 3 is under attack (110), regiment 4 is ok (0), and regiment 5 needs supplies (10). Thus, 5 messages require only 8 bits.

Using this same idea, we can devise more efficient codes for text files, thus improving on ASCII coding. Again, the key insight is to use short codes for frequently occurring symbols, such as 'e' and '." (period), and longer codes for symbols that (typically) only rarely appear, such as "z", "x", and "%". This requires

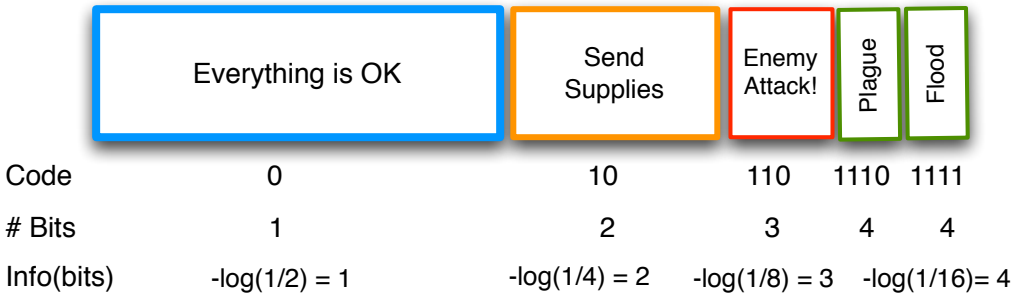

Figure 3: The information content of each symbol (defined as is $-log_2(p(s_i)) = log_2(\frac{1}{p(s_i)})$ is shown to be equivalent to the number of bits in the Huffman code for that symbol. Thus, the information in a symbol is directly proportional to its length (under an optimal encoding), which is inversely proportional to the probability of the event underlying the symbol.

a three-step preparatory process:

1. Calculate symbol frequencies based on a large and presumably "standard" body of text.
2. Build a Huffman tree based on these frequencies.
3. Read off the symbol codes from the Huffman tree and store them in a simple lookup table, e.g. a Python dictionary.

Once the codes are established, it is trivial to encode any new message: simply take each symbol and replace it by its Huffman code. Decoding is only slightly more work: walk down the target bit string until the current sequence represents the code for a symbol and then replace the bit sequence by the symbol. Huffman's is a prefix code, so, by definition, no symbol code is the prefix for another symbol code. So as soon as a bit sequence matches that of a symbol, it is guaranteed to be the correct symbol. This is the beauty of prefix codes!

4.1 Implementing Huffman Coding

For this project, you do not need to write the nuts and bolts of Huffman coding: frequency calculation and tree building. That code will be supplied. Your only job is to write a wrapper class (**Huffcoder**) with a few basic methods, thus providing a nice abstraction for the user of your system. All (s)he has to do is call the 'encode' and 'decode' methods that you implement, and all these have to do is call the supplied code.

First, to calculate symbol/character frequencies in a text file, use the function calc_char_freqs(filepath), which is defined in the file kdprims.py. This opens the given file, tabulates the frequency of occurence of each symbol, and returns a dictionary with entries of the form (symbol, frequency).

Next, to use this dictionary to produce a Huffman tree, we can take advantage of the "PriorityQueue" and "Node" class provided in John Conery's PythonLabs. To include these classes, add this line to your list of imports:

import PythonLabs.BitLab as btl

The BitLab file is large and reasonably complicated; you need not understand it in any detail. We will only be using a few pieces of it.

Now, to build the tree from a frequency dictionary, use the Python code in Algorithm 1, or something similar.

Algorithm 1 Building a Huffman tree when given a dictionary object (freqs) that houses the frequency of occurrence of each symbol. This exploits Conery's PriorityQueue object to connect tree nodes based on their frequency values. The nodes being popped from the queue are those with the lowest frequencies. Note that this saves the tree in the instance variable "tree" (self.tree), for later use during encoding and decoding of particular messages.

```
1: def build_tree(freqs):
2:                         pq = btl.init_queue(freqs)
3:                         while len(pq) > 1:
4:                                                 n1 = pq.pop()
5:                                                 n2 = pq.pop()
6:                                                 pq.insert(btl.Node(n1,n2))
7:                         self.tree = pq[0]
```

Now that the tree is built, you can use it to encode any string message (of any length) via this call:

btl.huffman_encode(msg,self.tree)

Here, msg is a string, and this returns a "Code" object (defined in BitLab), whose details you do not need to worry about. But, to get the string version of a Code object, use the "__repr__" method. So if x is the encoded message, then x.__repr__() returns this encoding as a string of bits.

Decoding is equally straightforward:

btl.huffman_decode(encoded_msg,self.tree)

Here, once again, encoded_msg is an instance of the class named "Code".

Again note that since we are using strings as our main representation, you will not see an actual reduction in string length in going from a text string to Huffman-encoded string. Rather, you will observe that each text symbol gets replaced by a short sequence of bits. In most cases, a text string of length L converts into a bit string of length k x L, where k is significantly less than 8 (e.g. between 3 and 4 for many of the sample files in this project) – whereas ASCII encoding gives k = 8. So although the bit string is longer than the original text string, if we were to convert the bit string to actual bytes and write it to a file, that file would take up less space (probably around 50% less) than the corresponding text file.

5 Lempel-Ziv Coding

As opposed to Huffman coding and other frequency-based methods – which analyze many cases, generate frequency distributions over symbols, and use those frequencies to produce efficient codes – the Lempel-Ziv approach requires no history of cases. Instead, it uses earlier portions of the current case in an indexed lookup table so as to more efficiently encode later segments of the same case. Each new segment is then represented as an old pattern plus one additional symbol, where the old pattern is not copied but represented by an index into the Lookup Table (LT). As the cases get longer, these indices become much shorter than the patterns themselves, thus yielding significant compression.

In contrast to the ASCII and Huffman codings, the Lempel-Ziv algorithm in this project will not work with text strings, only bit strings. So the input to Lempel-Ziv encoding will be a bit string, and the output will be another bit string. In some cases, particularly those involving images, the output can be quite a bit shorter than the input. To use Lempel-Ziv on a text string, you will first need to convert that string to an ASCII or Huffman-encoded bit string.

Figure 4 illustrates this basic relationship between the source string given to the Lempel-Ziv algorithm and the target string that it produces. It is important to notice that the lengths of the binary indices into LT become longer as we move rightward along the target string. This reflects the fact that LT begins as nearly empty and grows by one additional pattern at each iteration. So early in the encoding process, there are much fewer patterns in LT, thus requiring fewer bits to encode an index.

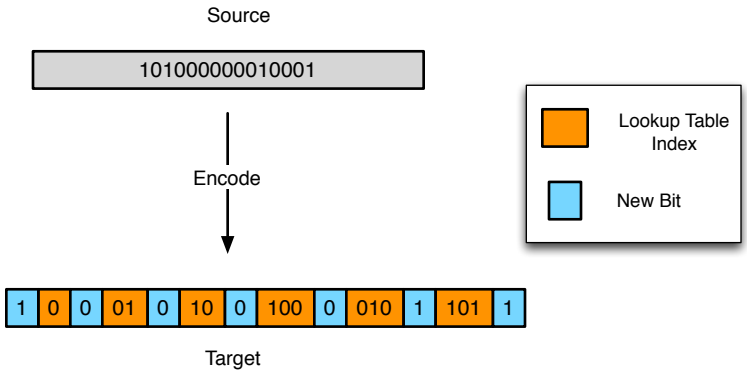

Figure 4: Result of encoding the source string as a target. Note that the number of index bits increases systematically in moving along the target string.

Also notice that in this simple example (which reappears several times in this document), the target's length **exceeds** that of the source. This is common for very small examples of Lempel-Ziv encoding; the compression only becomes evident with longer source strings in which binary indices into LT tend to be much shorter than the patterns themselves; initially, the indices are often longer than the patterns.

5.1 Lempel-Ziv Encoding

Figure 5 illustrates the basic encoding process, wherein segments of the source are first identified as **new patterns** (i.e., those that do not yet appear in LT), where each new pattern (S) – which has length k – has a length k-1 prefix (S*) that is an **old pattern** (i.e., one already existing in LT). The index of S* (converted to a binary vector) combines with the kth bit of S to produce the next segment of the target.

To find S, the algorithm must walk along the source from the current location and add each new bit to its current segment (C). With each bit addition, a new check of LT occurs. As soon as C becomes an invalid key into LT, S $\leftarrow$ C and the target can be expanded. On the next iteration, the current location of the source becomes the bit the follows S. Algorithm 2 sketches the basic encoding process, and a detailed trace of Lempel-Ziv encoding appears in the Appendix at the end of this document.

LT is a central data structure during both the encoding and decoding phases of Lempel-Ziv operations. During encoding, you will query LT with patterns, and it will return indices. To implement this, a Python dictionary is probably the wisest choice, since the keys into LT will be bit patterns (not simple integers); and dictionaries handle these types of queries very efficiently. During decoding, you will perform the opposite

(easier) operation: you will query LT with an index (k), and it will return the kth pattern. For that functionality, a standard vector of patterns will suffice. In this document, we will refer to the Lookup Table during both encoding and decoding as LT, but be aware that it will probably be a different data structure in each case (as indicated by the cartoon images of LT in Figures 5 and 6).

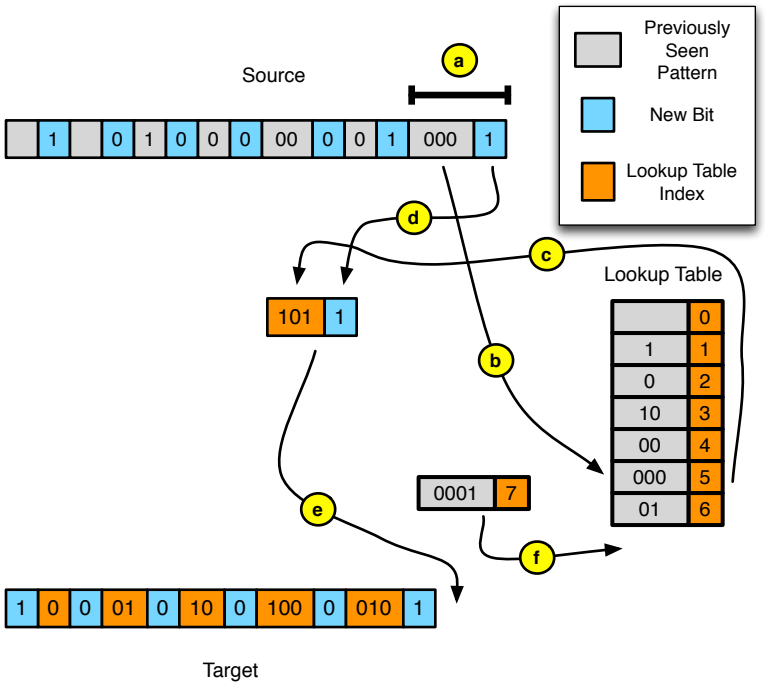

Figure 5: Encoding a source string as a Lempel-Ziv target string via the construction and use of a Lookup Table (LT) for previously-seen patterns. The process for encoding a source segment (denoted by a thick overbar) involves 6 steps: a) Find the proper segment by scanning the source rightward from the current location until the enclosed segment (S) does NOT exist in LT. (b) Given that S has length k, use S*, the length k-1 prefix of S as a key into the lookup table. (c) Convert the integer index associated with S* into a bit vector, and (d) combine this bit vector with the kth bit of S. (e) Append that combination (1011 in the figure) onto the Target. (f) Enter S into LT and associate it with the next available index.

5.2 Lempel-Ziv Decoding

To decode a Lempel-Ziv target string, you simply need to walk down the target string pulling off index-new-bit pairs and then using the index to fetch a pattern from the LT. That pattern is combined with the new bit, with the combination being added to both the source string and LT. As a simple technical detail, LT is initialised to the 2-item list [ϕ, target[0]]: the empty string and the string containing only the initial element of the target string [1]. Again note that your code must account for the fact that the binary representations of LT indices get longer as decoding progresses. Figure 6 illustrates the key decoding processes, while Algorithm 4 provides pseudocode.

[1]Note that the data structure for the Lookup Table can be different in the encoding versus the decoding phase. In the former, the keys need to be patterns, while the values are indices. In Python, this is easily handled with a dictionary. During decoding, the key is just an index, while the value is a pattern. This can be implemented as a dictionary, but a simple one-dimensional array/list works just as well.

Algorithm 2 This converts a source string to a target string using Lempel-Ziv compression. The function INTEGER_TO_BITS(int,len) generates the binary representation of its integer argument (int). It returns a bit vector of length = len. This may require padding the original binary value with 0's in its higher-order bits. For example, the vector of length 6 for integer 13 is 001101, where the high-order bits are on the left. FIND_NEXT_SEGMENT is defined below in Algorithm 3.

```
1: procedure LZ-ENCODE(source)
2: seen ← length(source)
3: target[0] ← source[0]   First element of target always equals first of source.
4: LT ← { (φ : 0) (source[0] : 1) }   Initialize the Lookup Table
5: size = 2   Size of LT
6: currloc ← 1   current location in the source string; 0-based
7: While (currloc < slen):
8: [oldseg newbit] ← FIND_NEXT_SEGMENT(source,currloc,LT)
9: bitlen = ⌈log2(size)⌉
10: index = LT.lookup(oldseg)
11: index_bits = INTEGER_TO_BITS(index,bitlen)
12: target ← target.append(index_bits.append(newbit))
13: LT.add_new_pattern(oldseg.append(newbit))
14: currloc ← currloc + length(oldseg) + 1
15: size ← size + 1
16: End While
17: Return(target)
```

Algorithm 3 This walks through the source string until it finds a new segment. It then returns the new segment, but separated into two parts: the old (prefix) segment, and the new bit. At the very end of the source string, the algorithm may run out of bits and only find an old segment, in which case the returned new bit is simply the empty string, ϕ (as shown on line 5).

```
1: procedure FIND_NEXT_SEGMENT(source,loc,table)
2: seg ← oldseg ← φ
3: newbit ← φ
4: While (table.lookup(seg)):
5: If (loc ≥ length(source)): Return([seg,φ])
6: newbit = source[loc]
7: loc ← loc + 1
8: oldseg ← seg
9: seg ← seg.append(newbit)
10: End While
11: Return([oldseg, newbit])
```

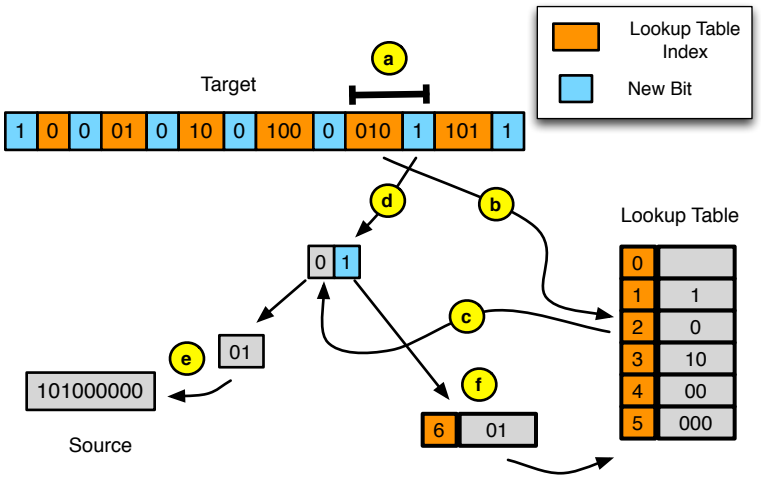

Figure 6: Decoding a Lempel-Ziv target involves building both a source string and a Lookup Table (LT). The process for decoding a target segment (denoted by the thick overbar) involves 6 steps: (a) Retrieve the next segment from the target. (b) Convert the index bits (orange) into an integer index (2) of LT. (c) Retrieve the indexed LT pattern (0), and (d) combine it with the new bit (1) from the target segment. This new pattern (01) is then (e) appended to the source string, and (f) added to LT.

Algorithm 4 The basic process of decoding a Lempel-Ziv target into a source string. BITS_TO_INTEGER converts a string of bits into an integer; it assumes that high-order bits come first in the string.

```
1: procedure LZ_DECODE(target)
2: tlen = length(target)
3: source = φ
4: LT ← [ φ , target[0] ]   Initialize the Lookup Table, a simple list
5: loc ← 1   Current location in the target string
6: size ← 2   size of LT
7: While loc < tlen:
8: bitlen = ⌈log2(size)⌉
9: index = BITS_TO_INTEGER(target[loc : loc+bitlen])   [ : ] behaves like Python's slice operator.
10: seg ← LT[index]
11: if loc + bitlen < tlen then :
12: seg ← seg.append(target[loc+bitlen])   Add the new bit
13: size ← size + 1
14: LT[size] ← seg   Update the lookup table
15: loc ← loc + 1
16: End If
17: source ← source.append(seg)
18: loc ← loc + bitlen
19: End While
20: Return(source)
```

6 Main Routines

Once your 4 classes (Coder, Asciicoder, Huffcoder and LZcoder) are in place, you will need to produce three top-level functions for performing the conversion and compression tests. These routines will primarily create coder objects, read messages from files (via Coder.gen_message_from_file(filepath)), and perform (one or two) encode-decode combinations (via Coder.encode_decode_test(msg)).

The three top-level functions are:

1. **Ascii_test(msg='Hello World',filepath=False,lz_flag=False)**
2. **Huff_test(msg='Hello World',filepath=False,lz_flag=False)**
3. **LZ_test(msg='00000000000000000000',filepath=False)**

For each of these functions, the message to be encoded will be given by the msg argument, unless the filepath argument has a value (other than False). If you specify neither a msg nor a filepath, 'Hello World' will be the msg for the first two tests, and twenty 0's will be the message for the third test. If you specify a filepath, then the message needs to be extracted from that file via the gen_message_from_file method, and this message from the file will override any message bound to the msg argument.

The lz_flag argument to the ASCII and Huffman tests provides the option for double encoding: first the input is converted to an ASCII or Huffman string, and then that string is encoded via Lempel-Ziv. So when lz_flag = True, two coders will be created, and two calls to encode_decode_test will be performed, with the results from both calls being printed to the screen.

To run Huff_test, you will need one additional file, **corpus1.txt**, which will be provided to you. This file should be the basis of the Huffman Tree and Huffman codes, which will then be used to encode the message provided to Huff_test, either via the msg or filepath argument.

You will need to be able to run Ascii_test and Huff_test on the following files (all provided to you):

1. sample1.txt
2. sample2.txt
3. sample3.txt

These runs should be possible with lz_flag set to True, and with lz_flag set to False. Does Lempel-Ziv give you any extra compression on these files? Can you explain why or why not?

When running with lz_flag = True, your high-level functions will call encode_decode_test twice, once for the ASCII or Huffman coder, and once for the Lempel-Ziv coder. Hence, your screen output will look something like the following run of Huff_test with msg = 'abba':

```
Original Message:
 abba
Encoded Message:
 10110110010110011011
```

```
Decoded Message:
 abba
The original and decoded messages are equal
Lengths of message, encoding and decoding:  4 20 4
Compression fraction: 0.375
Original Message:
 10110110010110011011
Encoded Message:
 10001110100101000011010011000
Decoded Message:
 10110110010110011011
The original and decoded messages are equal
Lengths of message, encoding and decoding:  20 29 20
Compression fraction: -0.44999999999999996
```

In the above example, note that the first set of messages and compression information pertains to the Huffman encoding of 'abba', while the second set refers to the Lempel-Ziv encoding of the Huffman encoding.

Your LZ_test function should handle the following three files (each of which are simply text files of bits, and each of which will be provided to you):

1. tumbler_bit.txt
2. rings_bit.txt
3. potus_bit.txt

These three files are essentially black-and-white versions of the following three GIF files: tumbler, rings, and potus (which you may want to check out). On which one do you get the best compression? Can you explain why?

Finally, with lz_flag = True, run Ascii_test and Huff_test on the following strings:

1. A string of 100 e's: 'eeeee....'
2. A string of 1000 e's
3. A string of 1000 'x's: 'xxxxxx....'
4. A string of 100 copies of 'ntnu': 'ntnuntnuntnu...'
5. A string of 1000 copies of 'ntnu'

You may want to use the function **n_strings** in kdprims.py to generate these repetitive strings.

Can you explain the differences in Lempel-Ziv compression fractions (LZF) that you get for different combinations of test functions and strings? For example, why is the LZF higher for the larger strings? When running Ascii_test, why do you get higher LZF on a string of 100 'ntnu's than on one of 100 'e's? Why does Huff_test give the opposite result: 100 'e's are more compressible than 100 'ntnu's? Comparisons like these should give you further insights into when and why LZ compression works well or poorly.

In addition to all of the above cases, you may be given simple messages to encode during the demo session if there is any doubt as to whether your system is behaving properly.

Note that for Huffman encoding, your original message should ONLY include symbols that appear in the file **corpus1.txt**. Otherwise, encoding may result in errors.

7 Demonstration

To receive a passing mark on this project, your system will need to handle several of the examples mentioned above, with the exact examples being chosen by the teacher(assistant) who monitors your demonstration. Also, your Huffman and ASCII encoder/decoder should run on any basic text message. So make sure that your system runs on ALL of the example strings and files listed above (and on a broad assortment of short text messages) before requesting to give a demonstration.

7.1 Appendix: A Detailed Example of Lempel-Ziv Encoding

What follows is a series of tables, each of which illustrates the state of the LZ-Encode algorithm on an iteration of the main While loop. The captions below each table explain the main activities.

Lookup Table	(ϕ : 0)
Source	101000000010001
(index,new bit)	
Target	

Table 1: Initial state of the Lempel-Ziv encoder; ϕ = empty string, whose index is 0 in the lookup table.

Lookup Table	(ϕ : 0) ('1' : 1)
Source	$\underline{1}$01000000010001
(index, new bit)	(, 1)
Target	$\underline{1}$

Table 2: Find the shortest substring that is not in the table (underlined in the source). Encode it as the combination of an index into the table plus an additional bit. Since the first symbol will always refer to the 0th table entry, ϕ, as the old pattern that it extends, the starting index (0) can be omitted. The new addition to the target string (also underlined.) is that (omitted)index plus the extending bit (1). Also, the lookup table is updated to include '1' as the entry with index = 1.

Lookup Table	(ϕ : 0) ('1' : 1) ('0' : 2)
Source	1$\underline{0}$1000000010001
(index,new bit)	(, 1) (0, 0)
Target	1$\underline{00}$

Table 3: The second bit (0) in the source is the next substring that has not yet been seen. It can be expressed as ϕ (index 0) plus the additional bit, 0. Thus, '00' is added to the target, and '0' is added to the lookup table with index = 2. Since the table now contains 3 entries, the next few portions of the target will require two bits to represent an index.

Lookup Table	(ϕ : 0) ('1' : 1) ('0' : 2) ('10': 3)
Source	10<u>10</u>00000010001
(index,new bit)	(, 1) (0, 0) (1,0)
Target	100<u>010</u>

Table 4: The next substring in the source that has not yet been recorded in the lookup table is '10'. This consists of the old pattern '1' (with index 1) plus an additional 0. At this point, two bits (01) are required to encode the index 1, so the new addition to the target is 010, i.e., the binary version of the index plus the new bit. Also, '10' becomes the next entry in the lookup table, with index = 3.

Lookup Table	(ϕ : 0) ('1' : 1) ('0' : 2) ('10': 3) ('00': 4)
Source	1010<u>00</u>000010001
(index, new bit)	(, 1) (0, 0) (1,0) (2,0)
Target	100010<u>100</u>

Table 5: The next new source substring is '00', which is a '0' (index = 2, or '10' in binary) plus an additional 0. Thus the new addition to the target is '100', and '00' is the next entry in the lookup table, which now has 5 entries and will thus require 3 index bits in the next round.

Lookup Table	(ϕ : 0) ('1' : 1) ('0' : 2) ('10': 3) ('00': 4) ('000': 5)
Source	101000<u>000</u>010001
(index, new bit)	(, 1) (0, 0) (1,0) (2,0) (4,0)
Target	1000101001<u>000</u>

Table 6: The next new source substring is '000', which is a '00' (index = 4, or '100' in binary) plus an additional 0. Thus the new addition to the target is '1000', and '000' is the next entry in the lookup table.

Lookup Table	(ϕ : 0) ('1' : 1) ('0' : 2) ('10': 3) ('00': 4) ('000': 5) ('01': 6)
Source	101000000<u>01</u>0001
(index, new bit)	(, 1) (0, 0) (1,0) (2,0) (4,0) (2, 1)
Target	10001010010000<u>101</u>

Table 7: The next new source substring is '01', which is a '0' (index = 2, or '010' in binary) plus an additional 1. Thus the new addition to the target is '0101', and '01' is the next entry in the lookup table.

Lookup Table	(ϕ : 0) ('1' : 1) ('0' : 2) ('10': 3) ('00': 4) ('000': 5) ('01': 6) ('0001' : 7)
Source	101000000010<u>0001</u>
(index, new bit)	(, 1) (0, 0) (1,0) (2,0) (4,0) (2, 1) (5,1)
Target	1000101001000010111<u>011</u>

Table 8: The next new source substring is '0001', which is a '000' (index = 5, or '101' in binary) plus an additional 1. Thus the new addition to the target is '1011', and '0001' is the next entry in the lookup table.