3F7 Laboratory
Data compression: build your own CamZIP…

Dr Jossy Sayir
`js851@cam.ac.uk`

Michaelmas 2018

1 Introduction

In this lab, you will be led through the implementation and use of various data compression algorithms that you've learned about in the lectures, in particular

- Shannon-Fano coding;
- Huffman's algorithm;
- arithmetic coding.

This lab assumes that you have understood the basics of these data compression algorithms. If they haven't fully been covered in the lectures yet, you can still attempt the lab but the theoretical basis for some of the data compression methods you will study will only become clear once the methods are covered in the lecture.

The lab is currently in Python. For those preferring to do the lab in Matlab/Octave, a version of the lab is also available and has been tested on those platforms. Please refer to previous year's handout (available on the moodle site) for specific instructions relating to the Matlab/Octave version.

Before you start this lab, you should download the relevant files to your home directory. These are available on the Moodle site as a package `3f7lab_py.tgz`. The following instructions assume that you are a novice to Linux and are working in the DPO. If one or both of these assumptions is incorrect, you may download the files however convenient.

1. In the DPO, log into a Linux workstation (NOT Windows!)
2. open a web browser, log into moodle (`http://vle.cam.ac.uk`), surf to the course page and click on the lab file package `3f7lab_py.tgz`. Depending on your browser, you will either be prompted to select a location for the file (pick your home directory) or the file will automatically be downloaded to a "Download" folder
3. open a terminal window[1], find the file (e.g. by using the commands `cd` to change directory to where the file is located and `ls` to list the content of each directory). If necessary, move it to your home directory (e.g. by using the command `mv 3f7lab_py.tgz ~`).

[1] You may also do this step with a file and folder browser of your choice

4. uncompress the file using the command `gunzip 3f7lab_py.tgz` It is ironic that you need to use data decompression software in order to learn about data compression. Who knows? By the end of this lab, you may have invented a data compression algorithm that beats `gzip` / `gunzip`...

5. The resulting `.tar` file (short for "tape archive") is an archive that contains all the files you need. You can unpack the archive using the command `tar xvf 3f7lab_py.tar`

6. There should now be a new directory `3f7lab_py` in your home directory containing all the relevant files for this lab. Move to this directory (`cd 3f7lab_py`) and start your favourite editor to view and edit the programs we will study (type `emacs &` at the command prompt if you don't have a favourite editor.)

In this lab, you will be guided through the use of a few Python functions we have written for you, and in some cases required to complete incomplete functions writing your own code. You are very welcome to implement the algorithms in this lab in the programming language of your choice if you prefer that to Python and we will support you in any language, but we currently do not offer pre-written code except in Python and in Matlab/Octave.

Before we get into the core of the matter, a word of warning is due: data compression is a low-level task in a computer system. Algorithms for compressing data are typically implemented directly in hardware, or programmed in a low-level programming language so that they run as efficiently and fast as possible. Python and Matlab/Octave are not such low-level languages. There are constructs in those languages that appear simple but shield the user from the actual complexity of what is happening at the machine level. They often rely on complex internal data structures and functions. We will try as much as possible to rely on elementary tasks and data structures when implementing data compression algorithms so as to convey an accurate impression of how these would be implemented efficiently at the machine level.

2 Trees and tables

The steps in this section are best done either using the iPython/Jupyter notebook provided or directly in the python interpreter. To use the python interpreter, type `python` a the shell prompt and then type `from trees import *` at the python prompt (in the rest of this section we will assume that you are using the interpreter, but following the same steps on the Jupyter notebooks is self-explanatory). Note however that if you do opt for the Jupyter notebooks, these need to be installed locally on your machine rather than copied into an online repository, because you will want to test the compression algorithms you will implement on all sorts of local files and this may not work if you run a Jupyter notebook on a server (we have not tested this.)

Some of the compression algorithms studied in 3F7 work by constructing a tree based on the probability distribution of the random variable you wish to encode. In order to implement these methods on hardware other than pen and paper, we need to learn how to represent a tree in computer memory. A tree according to graph theory is simply a graph with no cycles, or in other words a graph in which any two vertices are connected by exactly one path. There are various constructs and data structures in higher level programming languages for dealing with graphs and trees, but we will use a very low-level structure.

The “royal” way of storing trees in Python is as a recursive list, where each node is a list of its children. The outermost layer is the root and each of its elements corresponds to one of the root’s children, each of which is itself a list of their children, etc. For nodes whose children are leaves of the tree, these are represented by empty lists, possibly with a label giving the leaf’s identification. While this is very nice to work with in Python, it doesn’t translate well to other programming languages and relies on a vast overhead of internal constructs allowing Python to deal with lists of random non-typed objects that can themselves be lists of lists of lists. Hence, we will not use this approach (although if you are a Python geek you could of course as an exercise think about how all the data compression algorithms would work with such a data structure.)

We will store a tree as a simple list, where each entry corresponds to one node in the tree, and its value is the index of its parent. An entry of -1 a node without a parent, normally only the “root” node in a well-formed tree. This approach where elements in a list contain the index of other elements is what computer scientists call “pointers”: the value of the element is a pointer to another element rather than content that is of use in itself. The value -1 stands for the “null” pointer in this terminology, i.e., an empty pointer that doesn’t point to anything. For example `[-1 0 1 1 0]` is a tree with 5 nodes, where node 0 is the root, nodes 1 and 4 are children of the root, and nodes 2 and 3 are children of node 1.

Visualising trees is the mainstay of phylogeneticists (people studying how an elephant is related to a mongoose) and genealogists (people studying how your brother-in-law is related to your great-uncle.) There are many websites and software packages specialising in representing complex trees. A standard tree description syntax has been developed by phylogenetics researchers and is known as the “Newick format” (named after a restaurant where it was first proposed). We have provided a function to convert between the list format and the Newick format. Type

```
t = [-1,0,1,1,0]
print(tree2newick(t))
```

then cut and paste the resulting string into an online tree visualiser in your browser. We suggest[2] “`http://phylo.io`”. `tree2newick` takes an optional argument `labels` for when the labels of the leaves and/or nodes do not correspond to the natural ordering $0, 1, 2, 3, \ldots$. We currently only support numerical labels. Try the command

```
print(tree2newick(t, ['root', 'child 0', 'grandchild 0', \
            'grandchild 1', 'child 1']))
```

and visualise again with an online tree visualiser. In `phylo.io` click on “Settings” and select “Branch Labels/Support” and see how the branches are now labeled. Note that if the number of labels is smaller than the length of the tree, the labels will be interpreted “leaves first” as can be seen by typing `print(tree2newick(t,’symbol 0’, ’symbol 1’, ’symbol 2’))`.

An alternative approach to storing a variable-length code is simply as a table of codewords. We store such a table as a dictionary of variable length lists, which is fortunately possible in Python. In other programming languages where “arrays” (the equivalent of lists) must all

[2] In the current version of `https://phylo.io` as of October 2018 there is a minor bug. The tree is only drawn if the string is terminated by a newline. If you press “Render” after pasting the string as is, it won’t draw anything. You need to place the cursor at the end of the string an tap “return” to add a newline, and then press “render.”

have the same number of elements per row, you can store such a code table as two lists: the first as a list of rows of length equal to the maximum codeword length, where each codeword is stored in the first positions of the row; and the second a list of codeword lengths specifying which portion of each row corresponds to the codeword.

We have provided functions `c = tree2code(t)` and `t = code2tree(c)` to convert between tree and table notation. Type

```
print(tree2code(t))
```

to see an example of the table notation. The function returns a dictionary of three codewords, $[0, 0]$, $[0, 1]$ and $[1]$, assigned to the source symbols '0', '1' and '2', and of lengths 2, 2, and 1, respectively. Now type

```
print(code2tree(tree2code(t)))
```

to persuade yourself that the original tree is recovered. Note that, while you should have recovered exactly the same tree if you used the functions we provided in this example, this will not always be the case. For example, try

```
print(code2tree(tree2code([3,3,4,4,-1])))
```

Can you understand why the result is $[-1, 0, 1, 1, 0]$? You may want to visualise the tree $[3, 3, 4, 4, -1]$ as above with `phylo.io` to clarify this, with labels `['grandchild 0', 'grandchild 1', 'child 0', 'child 1', 'root']`.

The code table representation of a code is more useful for example when encoding data, whereas the tree representation may be more convenient when used for decoding or implementing a tree construction algorithm such as Huffman's algorithm. While there is no problem with the fact that tree representation is not unique, equivalent code tables may present a problem. Consider the codes `c1 = [[0],[1,0],[1,1,0],[1,1,1]]` and `c2 = [[1],[0,1],[0,0,1],[0,0,0]]`. Both codes are prefix-free and have codeword lengths $\{1, 2, 3, 3\}$ and hence have the same performance when used to encode any source. However, if we use the first code to encode for example a DNA sequence with alphabet $\{A, C, G, T\}$, but the decoder uses the second code table for decoding, it will not re-consitute the same DNA sequence and the original information will be lost. Furthemore, if you try the command

```
print(tree2code(code2tree({'0':[1], '1':[0,1], '2':[0,0,1], '3':[0,0,0]})))
```

you will observe that the two codes map to the same tree via `code2tree` and that our code construction method in `tree2code` favours the first code. This is because the tree representation does not store the branch assignment order: by default, `tree2code` will label the children of a node with 0 or 1 in the order in which the children are listed in the node list. To remedy this, we introduce a third and final representation for a prefix- free code which we call the *extended tree* or `xtree`. In this format, each node in the tree contains the index of its parent, a list of indices of its children in the order corresponding to the coding alphabet, and a node label which, for leaf nodes, equals the symbol assigned to the codeword. For example, a node `[3, [0,2], '7']` indicates that the node's parent is node 3, and its children are nodes 0 and 2, where node 0 corresponds to an encoded "0" and node 2 corresponds to an encoded "1", and its label is '7'. Labels don't matter for intermediate (non-leaf) nodes except that they may help you indentify the node when you view it in a tree viewer such as `http://phylo.io`. A -1 value in the parent position indicates that the node is an orphan (only the root of the tree in a valid prefix-free code), while a -1 in a child list indicates that the node has no child corresponding to this symbol. For example, a node `[3, [-1,2], ’6’]` indicates a node labeled ’6’ with parent 3, child 2 corresponding to an encoded “1”, and no child corresponding to an encoded “0”. A node with no child correspondig to an encoded “1” on the other hand does not require a -1 entry, e.g., `[3, [0], ’8’]` has parent 3, child 0 corresponding to an encoded “0”, and no child corredponding to an encoded “1”. It is equivalent to `[3, [0,-1], ’8’]`. We have provided the functions `xt = code2xtree(c)`, `c = xtree2code(xt)`, `xtree2newick(xt)`, `t = xtree2tree(xt)`, `xt = tree2xtree(t)`. In the latter, children are assigned as per default. Repeat the example above converting from a code to a tree and back with `code2xtree` and `xtree2code` instead of `code2tree` and `tree2code` and persuade yourself that it solves the problem. `tree2xtree()` takes node labels as an optional second argument. If the list of labels is shorter than the tree, the labels will be assigned to leaves first. Note that internally and for coding purposes, we will only work with the extended tree format: the functions we initially played with, `tree2newick, tree2code` and `code2tree` internally call the corresponding extended tree functions via `tree2xtree` and `xtree2tree`. The only reason we introduced the simple tree format is that it was easier for you to understand the concept of a list of pointers without cluttering the format initially with children and labels.

You are encouraged to play with more elaborate examples of codes and trees. What happens if your code has unused leaves in the tree? What happens if you supply a code that is not prefix-free? Can these functions handle non-binary codes or do they only work with binary codes? Try a few code tables and trees, convert between them, and visualise them with `tree2newick` or `xtree2newick`.

You are only expected to learn to use the tools described in this section and do not necessarily need to study their implementations. Should you be curious about them, you are of course very welcome to examine the functions and ask questions or suggest improvements on the online forum.

Summary of tools covered in this section:

- `tree2newick` abd `xtree2newick` - tools we provided to convert to the Newick standard tree description syntax
- `http://phylo.io` - a Newick syntax tree visualisation website
- `tree2code` - a tool we provided to convert from the tree representation to a code table
- `code2tree` - a tool we provided to convert from the code table representation to a tree
- `code2xtree` - a tool we provided to convert from the code table representation to an extended tree
- `xtree2code` - a tool we provided to convert from the extended tree representation to a code table
- `tree2xtree` and `xtree2tree` - tools we provided to convert from tree to extended tree representation and vice versa

3 Shannon-Fano Coding

Your task in this section is to implement a function that constructs a Shannon-Fano code. There are two ways the Shannon-Fano code can be implemented:

1. as described in the course material, by computing the lengths $\lceil -\log_2 p_k \rceil$ and constructing the tree that has leaves at these depths.
2. Claude E. Shannon in his 1948 paper [Sha01] introducing information theory, which is available on the moodle site, describes a constructive method for the Shannon-Fano code on page 17.

Although simpler to describe, the first method is in fact quite tricky to implement in software. We will follow Shannon's method in this lab. If you are a confident programmer, you are welcome to also try your hand at implementing the approach described in the lecture notes.

Before you proceed with the implementation of Shannon's method, please read page 17 in Shannon's paper carefully. It describes a method based on intepreting the cumulative probabilities of the source symbols as binary sequences and using those as codewords. Make sure you understand the subtle argument Shannon makes to show that the resulting code is prefix-free (you will be asked a few questions about this proof in the quiz!)

Open the file `vl_codes.py` for viewing. This file contains the following functions:

- partial implementations of the Shannon Fano code and Huffman's code;
- functions to convert a string of bits to bytes and vice-versa; and
- functions to perform variable length encoding and decoding of source data given a tree or code table representation of a variable length code.

We will work through these functions in the remainder of this lab.

Your first task is to complete the implementation of the function `shannon_fano(p)`. This function takes a dictionary of probabilities `p`, eliminates any symbols with zero probability, and issues a code table description of the Shannon Fano code. If you examine the functions `vl_encode()` and `vl_decode()`, you will notice that the former requires a code table description of a variable length code, while the latter requires an extended tree description of the code to operate. As we have seen in the previous section, you can switch from one description to the other using `code2xtree()` and `xtree2code()`, so in our function `shannon_fano()` it suffices to construct only a code table. Note that if you do decide to also implement the tree construction algorithm discussed in the lectures, tha would obviously yield a tree rather than a code table.

We now give an outline of Shannon's approach. Please read the passage in Shannon's paper [Sha01] *before* reading this outline. First, you need to compute the *cumulative* probability function corresponding to the random variable, where source symbols are ordered in order of decreasing probabilities. You may either do this directly (as we suggest), or use the `numpy` function `cumsum(p)` to do so, but note that `cumsum()` sums its input vector from 0 to k to calculate its k-th component so that the first entry is `p(0)` and the last entry is 1 when `p` is a probability vector that sums to 1. Shannon requires that you sum it from 0 to $k-1$, so that the first entry is 0. Hence, you need to suitably alter the output of `cumsum()` to fit your purpose. Once you obtain the cumulative probabilities, Shannon's method consists in writing each cumulative probability as a binary number, and using that number without its leading

0 as the codeword, suitably truncated to the length $\lceil -\log_2 p_k \rceil$. Shannon shows in his paper [Sha01] that the resulting code is prefix-free. You may find the notion of a fractional binary number confusing, so the following example should hopefully clarify the notion:

$$0.43 = 4 \times 10^{-1} + 3 \times 10^{-2} \equiv 0.01101110\ldots = 2^{-2} + 2^{-3} + 2^{-5} + 2^{-6} + 2^{-7} + \ldots$$

Hence, in order to convert a fractional number to binary, all you need to do is to verify whether it is larger than 2^{-k} for increasing values of k, mark those positions at 1 in the binary sequence and substract the corresponding 2^{-k} from the number. You could also achieve the same effect by multiplying the number by 2 at every turn (scaling the range from $[0, 1]$ to $[0, 2]$) and verifying whether the resulting number exceeds 1. For example, supposing we want to convert a number x to binary, we could use the following steps:

1. set a counter k to zero
2. multiply x by 2, i.e., $x \leftarrow 2 * x$
3. if $x \geq 1$, set the k-th digit to 1, and replace $x \leftarrow x - 1$
4. if, on the other hand, $x < 1$, set the k-th digit to 0
5. repeat from step 2 until k reaches the required precision

It is your task to implement this operation in Python. Read the comments for further guidance on Python commands you want to use. Note that this whole section of code requires 7 commands in our implementation, so if you end up writing a huge program you are doing something wrong... The function is given to you with an excessive number of comments to make sure the less skilled programmers are able to get it as right as possible (we are not testing your programming skills here.) Once completed, we recommend that you delete the more mundane comments and keep only what it takes for you to remember what is happening, and you should have a short and sweet function `shannon_fano()`.

Once your Shannon-Fano code is complete, try to generate codebooks for various probability distributions. If you are still in the Python interpreter, type

```
from vl_codes import shannon_fano
from random import random
```

then for example try to apply your algorithm to a random probability distribution:

```
p = [random() for k in range(10)]
p = dict([(chr(k+ord('a')),p[k]/sum(p)) for k in range(len(p))])
print(f'Probability distribution: {p}\n')
c = shannon_fano(p)
print(f'Codebook: {c}\n')
xt = code2xtree(c)
print(f'Cut and paste for phylo.io: {xt2newick(xt)}')
```

and used `phylo.io` to visualise the resulting tree. Note how we define probability mass functions as Python dictionaries in this lab. If everything works, you should now see a tree with 10 leaves assigned to the symbols 'a', 'b', 'c', etc.

Test your function with various probability distributions. Does the function work if you include distributions with zero probabilities? If not, can you think of a simple remedy for this? As you will see, this is not necessary within the framework of the file compression we will use in this lab, but it may be useful to know how to deal with zero probabilities if you use your function in other contexts.

In the next section, you will have the opportunity to apply your Shannon-Fano code to actual data.

4 Compressing and de-compressing a file

We have provided a file `hamlet.txt` in the directory that contains the full text of Hamlet by Shakespeare. You can load the file using the commands

```
f = open('hamlet.txt', 'r')
hamlet = f.read()
f.close()
```

If the first command fails, check that you are in the directory where `hamlet.txt` is located[3]. Check that the file you accessed contains the text of Hamlet as expected using the command `print(hamlet[:294])` to print the first 294 symbols of Hamlet,which should give you the title and the list of the first 6 characters in the play.

The vector `hamlet` consists of ASCII codes of the characters in the file `hamlet.txt`. Find an ASCII code table on the internet and have it ready for the next sections.

You can now obtain the frequency counts of the ASCII symbols in Hamlet using the commands

```
from itertools import groupby
frequencies = dict([(key, len(list(group))) for key, group in groupby(sorted(hamlet))])
Nin = sum([frequencies[a] for a in frequencies])
p = dict([(a,frequencies[a]/Nin) for a in frequencies])
print(f'File length: {Nin}')
```

which will return a dictionary `frequencies` of frequencies of symbols present in Hamlet, and a dictionary `p` of normalised frequencies that can be interpreted as probabilities. The length of the file `Nin` will be displayed (it should be 207,039).

Note that the file only contains ASCII codes for the characters

```
!&'(),-.:;?ABCDEFGHIJKLMNOPQRSTUVWYZ[]abcdefghijklmnopqrstuvwxyz|
```

and carriage return as can be verified by typing `print(list(p))`.

You can now apply the Shannon-Fano algorithm to the probability vector you computed from the frequencies in Hamlet. Examine the code table returned by `c = shannon_fano(p)`. You can view the resulting tree using the command `print(xtree2newick(code2xtree(c)))`. On `phylo.io`, make sure "Branch Labels/Support" is still selected under "Settings", and right-click on the root node and select the option "expand all" to view all nodes and leaves.

[3] Check using commands `pwd` to display the name and location of the current directory, `cd ..` to move up one directory, `cd directory_name` to change to sub-directory `directory_name`, `ls` or `dir` to list the contents of the current directory.

You can zoom in and out of the tree (typically by using the scroll wheel or option on a mouse) and shift the tree right and left to examine various details of the tree. Which symbol is assigned the shortest and the longest codeword? Why? What are the frequencies of the symbols corresponding to the shortest and longest codewords? In what context does the rarest symbol with the longest codeword appear in the text of Hamlet? Note how the lower case letters mostly were assigned shorter codewords than the upper case letters. Does this make sense to you? Note also how the left and right square brackets and parentheses ended up as siblings on the tree.

Now, while we trust that you found it entertaining to examine code trees and play around with them, surely you are getting impatient and wondering when we will finally get to compress data rather than just examine code trees? We have provided a function `vl_encode(x)` that takes an input data vector `x` and a code table `c`, and outputs a binary vector of compressed data.

```
from vl_codes import vl_encode
hamlet_sf = vl_encode(hamlet,c);
print(f'Length of binary sequence: {len(hamlet_sf)}')
```

If you followed all instructions, `hamlet_sf` should be a binary vector of length 997,548. You may at first wonder why it should be considered a success to have compressed a file/vector of length `len(hamlet) = 207039` to a vector/file of length 997,548? Remember that the result you obtained is a binary vector. We have provided a function `bits2bytes(x)` that converts a binary vector `x` to a vector of bytes (groups of 8 bits) and a reverse operation `bytes2bits(x)` that converts back to the original binary vector. This is not as trivial as it sounds, because the number of bits may not be divisible by 8. If we simply stuff zeros at the end of the input vector to make it divisible by 8, we may be inadvertently adding codewords to the end of a compressed stream, resulting in extra symbols being decoded when the compressed file is uncompressed. To avoid this, `bits2byte(x)` adds a 3-bit prefix to the beginning of the binary input string `x`, specifying how many bits must be appended to the new string of length `length(x)+3` to make it divisible by 8. This will be a number between 0 and 7 and hence can be encoded in 3 bits. Try the following commands

```
from vl_codes import bytes2bits, bits2bytes
x = bits2bytes([0,1])
print([format(a, '08b') for a in x])
y = bytes2bits(x)
print(f'The original bits are: {y}')
```

The first command imports `bits2bytes` and `bytes2bits` from the libary. The second command converts the binary string `[0,1]` to bytes using the format described above. The third command allows you to visualise the contents of the variable `x`. The length of the input vector to `bits2bytes` is 2. The added 3 bit prefix brings the length up to 5. This means that 3 bits need to be appended to make up 8 bits or one byte. Hence, the value of the 3 bit prefix is the number 3 expressed in 3-bit binary format, `011`, followed by the input binary string `[0,1]`, followed by the 3 appended zeros. The last command recovers the original binary string of length 2 using the formatting described.

Now try the following

```
hamlet_zipped = bits2bytes(hamlet_sf)
```

```
Nout = len(hamlet_zipped)
print(f'Length of compressed string: {Nout}')
```

You should obtain a length of 124,694, corresponding to a compression rate

$$R_{\text{zipped}} = \frac{117,553}{207,039} = 0.602.$$

Compression performance for data compression algorithms is more often measured in how many compressed bits are produced on average per input byte, giving the figure of

$$R_{\text{zipped}} = \frac{8 \times 124,694}{207,039} = 4.82 \text{ bits per byte.}$$

We need to compare this value to the entropy of the probability distribution we used to compress Hamlet. The following commands define a function `H()` that computes the entropy in bits and uses it to compute the entropy of the probability ditribution

```
from math import log2
H = lambda pr: -sum([pr[a]*log2(pr[a]) for a in pr])
print(f'Entropy: {H(p)}')
```

How far are we from the lower bound for compression? How large should the compressed file be if we were able to achieve the lower bound?

Before we can celebrate the implementation our first compression algorithm, we need a quick reality check: how do we decompress our file? The history of information theory is littered with bogus claims of stellar compression rates that later unravelled as "one way" compression algorithms where you can compress a file but never recover the file from its compressed version.

We have provided a function `vl_decode(x,xt)` that decodes a binary compressed data string. Try the commands

```
from vl_codes import vl_decode
xt = code2xtree(c)
hamlet_unzipped = vl_decode(hamlet_sf,xt)
print(f'Length of the unzipped file: {len(hamlet_unzipped)}')
```

If all went well, you should have obtained the same length as the length of the Hamlet file (`Lin`) you measured above. Note that we did not bother here to go from bits to bytes and back as we would if this were "live" and we wanted to store the compressed sequence into a file. You can now now verify that the decompressed file is still recognisable as Hamlet by using the following command:

```
print(''.join(hamlet_unzipped[:294]))
```

The reason for this syntax is that `hamlet_unzipped` is a list and would print as a collection of single-character strings if printed directly, whereas the `join` command joins all the strings together with an empty separator `''`.

Now that you've tested the encoding and decoding path succesfully, you may want to put it all together in one command. We've done this for you in a function `camzip`. This can be either called directly from the command shell as `python camzip.py method filename` or, remaining within the interpreter or Jupyter notebook, using the syntax

```
from camzip import camzip
camzip(method, filename)
```

The `method` argument specifies the compression method: currently, the only method you've implemented in Shannon-Fano compression, but we will also implement Huffman and arithmetic coding, so the method argument will eventually be one of `'shannon_fano'`, `'huffman'`, or `'arithmetic'`. The function reads a file specified by the `filename` argument, say for example `myfile.txt`. Depending on the method, it writes the compressed output into a file `myfile.txt.czs` (for Shannon- Fano compression), `myfile.txt.czh` (for Huffman compression) or `myfile.txt.cza` (for arithmetic coding). It also saves the source alphabet and measured frequencies of the original file in a file `myfile.txt.czp`. We also provided a function `camunzip` that can be called either from the command line as `python camunzip.py filename` or from the interpreter or Jupyter notebook using

```
from camunzip import camunzip
camunzip(filename)
```

`camunzip` does not need to be told the compression method since it can read it from the file extension, e.g., `camunzip('myfile.txt.czs')` will automatically assume a Shannon-Fano code for decoding. When decoding a file `myfile.txt.czx` for `x = s, h,` or `a`, `camunzip` will produce a file `myfile.txt.cuz`. For practical compression, we would probably just want an unzipped file without a different extension, but since we want to compare our original file to unzipped files to ensure that our algorithms are running as intended, we do not want to overwrite the original files. You can try `camzip('shannon_fano', 'hamlet.txt')` and `camunzip('hamlet.txt.cza')` and the use the following commands to get some statistics and verify that the decompressed file `hamlet.txt.cuz` is identical to the original file `hamlet.txt`:

```
from filecmp import cmp
from os import stat
from json import load

filename = 'hamlet.txt'
Nin = stat(filename).st_size
print(f'Length of original file: {Nin} bytes')
Nout = stat(filename + '.cz' + method[0]).st_size
print(f'Length of compressed file: {Nout} bytes')
print(f'Compression rate: {8.0*Nout/Nin} bits/byte')
with open(filename + '.czp', 'r') as fp:
  freq = load(fp)
pf = dict([(a, freq[a]/Nin) for a in freq])
print(f'Entropy: {H(pf)} bits per symbol')
if cmp(filename,filename+'.cuz'):
  print('The two files are the same')
else:
  print('The files are different')
```

Try compressing a range of files with the Shannon-Fano algorithm and see how well it does. Download a few interesting files from the internet. You could for example download files from the Canterbury Corpus (`http://corpus.canterbury.ac.nz/`), one of the standard file repositories for benchmarking compression algorithms. You may neglect the storage space required to store the code table (those doing an FTR will learn more about how this can be avoided.) What can you say about the compression ratios achieved? Can you explain why some do better than others? You may want to start recording the performance obseved in a table at this point as you will be asked to test several algorithms throughout the remaining sections of this lab and you will eventually want to compare their performance.

Summary of tools covered in this section:

- `bits2bytes, bytes2bits` - tools we provided for converting a stream of bits to bytes and vice versa
- `vl_encode, vl_decode` - tools we provided for encoding and decoding data using a variable length code/tree
- `H = lambda pr:  -sum([a*log2(a) for a in pr])` and then `H(p)` - a way to define entropy as an in-line function definition in Python
- `camzip, camunzip` - skeleton functions putting it all together to compress/uncompress a file directly to another file

5 Huffman's Algorithm

To implement Huffman's algorithm, we recommend using two data structures:

- an extended tree array `xt` initialised to have as many nodes as there are source symbols, where every node is initialised as an orphan (parent -1) with no children (empty list) and the corresponding source symbol as a label;
- a "labeled probability list" `p` , i.e., a list of tuples where the first element contains the index of the probability in the list of nodes above, and the second element contains the probability. Maintaining this list as a list of tuples means that we can now sort the list and still remember which probability corresponds to which symbol or, equivalently, which leaf in the tree.

Huffman's algorithm proceeds by selecting the two nodes in `p` with the smallest probabilities, creating a new parent node in `xt` and connecting those two nodes to the parent, then replacing the two nodes in `p` with the new parent node and assigning as its probability the sum of the probabilities of the deleted nodes. The process is repeated until there is only one node left in `p` and its probability is one. By this point, all nodes in `xt` should have been connected and the only node remaining with a -1 parent is the root. Note that for an alphabet size of N, this process has $N-1$ steps, where a new element of `xt` is created at every step, and two elements of `p` are replaced by one new element, so the extended tree will end up with $2N-1$ nodes of which $N-1$ are leaves.

We have provided a skeleton function `xt = huffman(p)`. Please complete the missing commands. The commands that need to be added are described in the comments. Once you have done this, test your algorithm and the resulting trees with a few simple probability distribution and visualise the tree to verify that the algorithm is running as intended. You can visualise the Huffman tree for Hamlet using the following commands:

```
from vl_codes import huffman
xt = huffman(p)
print(xtree2newick(xt))
```

Compress the same files you compressed in the previous section using `camzip` specifying the method as Huffman's algorithm and compare the results you obtain to the results obtained with the Shannon-Fano algorithm. Is the improvement significant?

An interesting question that you will need to answer in the quiz is what happens to Huffman decoding when one or several errors are introduced in the encoded sequence. You can do this using `camzip` and `camunzip` and viewing/editing the compressed file and decompressed file, or for more experimental control you can do it in the interpreter or Jupyter notebook with the following commands:

```
c = xtree2code(xt)
hamlet_huf = vl_encode(hamlet, c)
hamlet_decoded = vl_decode(hamlet_huf, xt)
print(''.join(hamlet_decoded[:294]))
```

and, to introduce an error in the compressed output, try the following:

```
hamlet_corrupted = hamlet_huf.copy()
hamlet_corrupted[400] ^= 1
hamlet_decoded = vl_decode(hamlet_corrupted, xt)
print(''.join(hamlet_decoded[:297]))
```

6 Arithmetic Coding

In order to implement an arithmetic encoder effectively, we begin by highlighting the limitations of the encoder described in the lecture notes. The procedure introduced in the lecture notes can be described as follows:

1. For a source string $(x_1, x_2, \ldots, x_n)$, determine an interval of length equal $P(x_1, \ldots, x_n)$ such that the intervals corresponding to all possible source strings are disjoint.

2. The interval can be computed recursively by subdividing the interval for a partial string $(x_1, \ldots, x_k)$ into sub-intervals of length proportional to the probability distribution of X_{k+1}, and picking the sub-interval corresponding to the value x_{k+1} in the string to obtain the interval for $(x_1, \ldots, x_k, x_{k+1})$.

3. The procedure above is repeated n times starting from the interval $[0.0, 1.0)$ to yield the interval for the sequence $(x_1, \ldots, x_n)$.

4. The codeword is chosen as the binary representation of the lower end-point of the largest dyadic interval $[\frac{j}{2^\ell}, \frac{j+1}{2^\ell})$ that fits within the interval computed for $(x_1, \ldots, x_n)$, where j, ℓ are integers.

Let us try this procedure in Python for a specific string. Load again the data vector hamlet from hamlet.txt and compute p the probability distribution inferred from its letter frequency counts as we did in the previous sections. We will initialise the interval end-points with lo = 0.0 and hi = 1.0 and compute the interval recursively for the first n symbols of hamlet for various values of n. Enter the following commands in the interpreter or Jupyter notebook:

```
f = [0.0]
for a in p:
  f.append(f[-1]+p[a])
f.pop()
f = dict([(a,f[k]) for a,k in zip(p,range(len(p)))])
```

This computes the cumulative probability distribution f. Now enter:

```
lo, hi = 0.0, 1.0
n = 4
for k in range(n):
  a = hamlet[k]
  lohi_range = hi - lo
  hi = lo + lohi_range * (f[a] + p[a])
  lo = lo + lohi_range * f[a]
print(f'lo = {lo}, hi = {hi}, hi-lo = {hi-lo}')
```

This executes the steps described in the lecture notes for $n = 4$ input symbols. Note how the new interval is defined as the scaled interval between the sum of the probabilities up to the value of the symbol at position k and the sum of the probabilities including the next value.

Now examine the value of lo and hi after the operation above and repeat the commands varying the value of n. What happens when you let n grow? What would happen if you tried to encode the whole file, setting n = 207,039? What would step 4 look like? You can visualise an approximation of the output of Step 4 after n iterations (where we denote ℓ as ell) using the command

```
from math import floor, ceil
ell = ceil(-log2(hi-lo))+2 if hi-lo > 0.0 else 96
print(bin(floor(lo*2**ell)))
```

What should ℓ ideally be if we ran the operation above on the whole file setting n = 207,039? Do you think that the file could be decoded from the value of the binary string obtained? Explain why?

The lesson you have learned from the steps above is that the arithmetic encoding procedure as described in the lectures requires an infinite precision calculator. More precisely, it requires a calculator whose precision grows with n to yield sufficient significant digits in the expression of lo and hi so that their binary expansion is meaningful to the length of a binary string required for the encoding to be reversible. In basic Python, we do *not* have an infinite precision calculator. While arbitrary precision calculators are a topic of interest to computer scientists (see for example https://www.gnu.org/software/bc/), they are not in fact necessary for implementing an arithmetic code. There are a few tricks that can be used to remedy the problems that we observed in the steps above without resorting to an arbitrary precision calculator. They are:

1. You do not need to wait to end the interval computation step to start encoding bits. If the interval `[lo,hi)` is such that `hi > lo > 1/2`, then it is already clear that any dyadic interval within `[lo,hi)` will have a 1 in its first position. Similarly, if `lo < hi < 1/2`, there will be a 0 in the first position. In both cases, the interval can be rescaled. If `lo < hi < 1/2`, the interval end-points can simply be multiplied by 2. If `hi > lo > 1/2`, the interval end-points can be multiplied by 2 and shifted back to the $[0, 1)$ range by subtracting 1.

2. The procedure above works well when the data is such that the interval goes below or above the 1/2 boundary. However, you may be unlucky and the data may be such that the interval will become smaller while straddling the 1/2 boundary, thereby losing precision just as in the examples we tried above. This is not as bad as it sounds: if the interval is such that `1/4 < lo < hi < 3/4`, you know that when the straddling is finally resolved, any dyadic interval within the resulting interval will have either `10` at its beginning if the straddling is resolved above the 1/2 boundary, or `01` at its beginning if the straddling is resoved below the 1/2 boundary. Hence you can stretch the interval end-points by multiplying them by 2 and substracting 1/2 while keeping a count of how many times you've done that before straddling is resolved. If straddling is resolved above the 1/2 boundary after σ straddling counts, you need to output a 1 followed by σ zeros, while if it is resolved below the 1/2 boundary, you output a 0 followed by σ ones.

3. The lack of infinite precision in the interval calculation still has one minor pitfall that would no longer hit us for `n` larger than about 10 as it did in the examples we tried above, but would probably cause problems once we tackle source strings in the thousands: because the interval end-points are subject to automatic rounding by the finite-precision floating-point calculation, there is a danger that the intervals corresponding to all possible source strings will no longer be *disjoint*. In the lecture, it was shown that this is the condition for the code to remain prefix-free and hence uniquely decodable. In order for the code to remain prefix-free, we have to take control of the rounding of interval end-points and ensure that any rounding is always *inwards*, i.e., `lo` can be rounded up in calculations and `hi` can be rounded down, but never the other way around.

The easiest way to go about controling the automatic rounding of interval end-points is to revert to integer arithmetic. We can define a large integer `'one'` that we will consider to be the 1.0 boundary or initial interval high end-point, while integer 0 will remain the minimum boundary or initial interval low end-point. All arithmetic is done in full-precision floating point, but by rounding the resulting interval end-points to either the integer above (using `ceil()`) or the integer below (`using floor()`), we can control all the rounding that happens in our finite precision implementation.

We have provided a partial implementation of the procedure described above in a skeleton function `y = arithmetic.encode(x,p)`. This implementation is heavily inspired by the 1987 paper by Witten, Neal and Cleary [WNC87]. The function encodes an input string of bytes `x` to an output binary string `y` using a probability mass function described in a Python dictionary `p`. Your task is to complete the function as instructed.

Implementing an arithmetic encoder is a delicate task. There are a few subtly different ways it can be implemented and it is essential for the decoder to exactly mirror the operations of the encoder if you want compressed files to be decompressed successfully. We have provided a full arithmetic decoder. Examine the code for `arithmetic.decode(y,p,n)` and you'll find that it follows the same outline as the encoder. The function takes as its input the binary compressed string, the probability distribution and the length of the original encoded string. When completing the encoding function, make sure you either mirror the operations in the decoder exactly, or if you opted for slightly different operations in your encoder, make sure you modify the decoder supplied so it mirrors your operations in the encoder.

Once you've completed the function, try to encode `hamlet.txt` ensuring that you still have the appropriate probability distribution `p` for it. Calculate the compression rate in bits per byte and compare it to the entropy `H(p)`. What do you observe? Use the decoder and verify that the file was decoded properly by viewing its first 1000 or so characters and checking its length. You can do this using `camzip` or `camunzip` or in the interpreter or Jupyter notebook if you want to retain more control of the experiment, using the following commands:

```
import arithmetic as arith
arith_encoded = arith.encode(hamlet, p)
arith_decoded = arith.decode(arith_encoded, p, Nin)
print('\n'+''.join(arith_decoded[:294]))
```

and to test error resilience:

```
arith_corrupted = arith_encoded.copy()
arith_corrupted[399] ^= 1
arith_decoded = arith.decode(arith_corrupted, p, Nin)
print('\n'+''.join(arith_decoded[:294]))
```

Try various locations for the error bit or several error bits. What happens? Can you explain this?

Have a go at compressing the same files you compressed with your Shannon-Fano and your Huffman codes and add them to your comparison. Reflect on the compression rates achieved with arithmetic coding as compared to the other algorithms. Explain why the differences in performance are as they are. How could you improve on the performance of the arithmetic encoder?

7 Carrying on: adaptive and context-based compression

(This section only needs to be done for the Full Technical Report BUT we recommend reading it even if you aren't doing an FTR, because thinking about extensions is instructive in its own right.)

Before starting, you may want to compress some of the files you've compressed before using widely available software such as `gzip` and compare your results. Don't be disappointed... This should motivate you to do the FTR and beat those standard tools. It is possible! (particularly if speed of implementation is not your main focus.)

In this final section, we will not provide any function skeletons or tools and you are expected to write these by yourself. There are several ways you could go beyond the work in the 3F7 lab to bring it to FTR level:

1. Rather than measuring the frequencies in a file you want to compress and storing the resulting code or probability distribution in a separate file, you can calculate frequencies

on the fly as you encode. As long as you only use frequencies measured up to and not including the current symbol in the encoding operation, the decoder will have processed the same string and worked out the same frequencies, so can reconstruct the code without the need to supply it separately. This approach is called "adaptive" or sometimes "universal" coding[4].

2. In the lecture, you've already seen that compression can be based on the conditional probability distribution given a number of past symbols. The compression rates we achieved in the previous section are limited to an independent and identically distributed model of text. Text is obviously not independent and identically distributed (e.g. $P_{X_{k+1}|X_k}('u'|'q') \approx 1$) so the string probabilities we based our encoder on are not the true string probabilities of the text. You can improve performance a lot by measuring the conditional probability distribution of the next symbol based on a context.

For the first option, note that simply basing your probability model on the frequency counts of the symbols you've seen so far is not a good idea: if you have never seen an 'a' until now, the frequency of 'a' will be zero and your Huffman or arithmetic code will not have a codeword for it. As a result, if 'a' occurs, you will not have a codeword to "tell" the decoder that this new symbol needs to be added to its statistics. There are several remedies for that: the Predicition by Partial Matching (PPM) family of algorithms (see `http://en.wikipedia.org/wiki/Prediction_by_partial_matching`) adds an "escape" character to the alphabet in every context to indicate that a new symbol needs to be added, and encodes the new symbol using a block code. The same approach can be adopted in a non-contextual adaptive compression algorithm. Other approaches consist in giving every frequency count an initial bias of a set number ε so that none of them are ever zero. When $\varepsilon = 1$, the resulting probability estimator is called the Laplace estimator. You should modify `camzip` and `camunzip` to use an adaptive probability estimator of your choice, either using an escape symbol or a biased estimator. You may want to embed your probability estimator or escape mechanism into `arithmetic.encode` and `arithmetic.decode`, or, alternatively, write a separate function to estimate probabilities. If writing a separate function, it would be helpful to modify `arithmetic.encode` to process one symbol at a time rather than the whole file, and ditto for `arithmetic.decode`. This would allow you to supply a different probability model for every call of the function. You would want to make use of some Python mecanism to make the variables `lo, hi` and `straddle` persistent (pickle, shelf, or simply pass them as inputes and outputs to `encode` and `decode`) Your new `camzip` and `camunzip` should now be fully independent compression and decompression modules that map a file to a compressed file and back without the need for a separate code description. Compare the performance to the performance of the previous algorithms and explain your observations. This step completes the minimum requirements for an FTR and you may either stop here or pursue the optional steps below if you want to learn more about the topic.

You may also want to look at adaptive Huffman coding. The direct approach to this would be to run Huffman's algorithm at every encoding step on the current estimated probability distribution. The decoder can do the same assuming it's decoded all the past symbols correctly and has access to the same partial statistics as the encoder had at this point. However, there is a much more elegant way to implement an adaptive Huffman encoder based on the

[4] Note that the word "universal" implies a proof that the adaptive approach is optimal in the sense that it achieves the source entropy asymptotically.

observation that trees cannot change drastically from one step to the next assuming that only one frequency count will increase by 1. If you want to study and implement this algorithm, you could read the paper [Gal78] by Robert G. Gallager "Variations on a theme by Huffman". This paper is available for download from the 3F7 lab moodle page and it's easily one of the prettiest and most accessible scientific papers in the information theory literature. Sections I to III of the paper set out the mathematical basis for the algorithm, proving three "theorems" that establish useful properties of a Huffman code. Section IV describes the actual algorithm implementation.

Finally, if you want to "break the world compression record", you will need to look at contextual arithmetic coding. To visualise how probability models for compression are improved by taking context into account, we suggest that you play with the Dasher information-efficient text-entry interface (`http://www.inference.phy.cam.ac.uk/dasher/`). To embed context into your arithmetic encoder/decoder, try at first to estimate conditional distributions for every context of length κ as you did for the adaptive arithmetic encoder above. Try $\kappa = 1, 2, 3$. Measure the performance obtained in function of the length of context for files of different length. One "cheat" consists in taking *hamlet.txt* and concatenating 10 or 100 copies of it into one file, to see how the compression improves once the algorithm is able to accumulate more statistics. There are several ways to juggle the advantages of deep contexts (= better statistics in the long term) with its disadvantages (= more time/data needed for the model to converge). One way consists in using an escape symbol as hinted above. Another way consists in combining estimates from contexts at different lengths. A full study of optimal contextual statistical measurement is beyond the scope of this lab and you should refrain from spending too much time on this, but if you find the topic interesting, consider taking the Machine Learning module 4F13, the Speech and Language Processing module 4F11, in addition to the Advanced Information Theory module 4F5 which we trust that you will definitely take since your interest for 3F7 led you to do an FTR for this lab.

References

[Gal78] R. Gallager. Variations on a theme by huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, November 1978.

[Sha01] C. E. Shannon. A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55, January 2001.

[WNC87] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, June 1987.