Problem Set 1

Your answers will be graded by actual human beings (at least that's what we believe!), so don't limit your answers to machine-gradable responses. Some of the questions specifically ask for explanations; regardless, it's **always a good idea to provide a short explanation for your answer**. *Before* attempting the problems below, we encourage you to review Chapters 1 and 3, and solve the online practice problems on this material.

Problem 1: Information (2 points)

X is an unknown 8-bit binary number picked uniformly at random from the set of all possible 8-bit numbers. You are told the value of another 8-bit binary number, Y, and also told that X and Y differ in exactly 4 bits. How many bits of information about X do you now know? Explain your answer.

(points: 2)

Problem 2. Entropy when combining symbols. (1 point)

Suppose you are given a collection of $N \geq 1$ symbols, the i^{th} symbol occuring with probability p_i, for $1 \leq i \leq N$, with $p_1 + p_2 + ... + p_N = 1$. This distribution has entropy H. Now suppose we combine any two symbols, i and j, to a combined symbol with probability of occurrence $p_i + p_j$. All the other symbol probabilities remain the same. Let the entropy of this new distribution by H'. In general, is H' smaller than H, bigger than H, or the same as H, or is it not possible to tell for sure? Prove your answer.

(points: 1)

Problem 3. Green Eggs and Hamming. (8 points)

By writing Green Eggs and Ham, Dr. Seuss won a $50 bet with his publisher because he used only 50 distinct English words in the entire book of 778 words. The probabilities of occurrence of the most common words in the book are given in the table below, in decreasing order:

Rank	Word	Probability of occurrence of word in book
1	not	10.7%
2	I	9.1%
3	them	7.8%
4	a	7.6%
5	like	5.7%
6	in	5.1%
7	do	4.6%
8	you	4.4%
9--50	*all other words*	45.0%

A. I pick a secret word from the book.
The Bofa tells you that the secret word is one of the 8 most common words in the book.
Yertle tells you it is not the word *not*.
The Zlock tells you it is three letters long.

How many bits of information about the secret word have you learned from:

1. The Bofa alone?
2. Yertle alone?
3. The Bofa and the Zlock together?
4. All of them together?

Express your answers in $\log_2(100 / x)$ form, for suitable values of x in each case.

(points: 2)

B. The Lorax decides to compress Green Eggs and Ham using Huffman coding, treating each **word** as a distinct symbol, ignoring spaces and punctuation marks. He finds that the expected code length of the Huffman code is 4.92 bits. The average length of a word in this book is 3.14 English letters. Assume that in uncompressed form, each English letter requires 8 bits (ASCII encoding). Recall that the book has 778 total words (and 50 distinct ones).

1. What is the uncompressed (ASCII-encoded) length of the book? Show your calculations.

(points: 0.5)

2. What is the expected length of the Huffman-coded version of the book? Show your calculations.

(points: 0.5)

3. The words "if" and "they" are the two least popular words in the book. In the Huffman-coded format of the book, what is the Hamming distance between their codewords?

(points: 0.5)

C. The Lorax now applies Huffman coding to all of Dr. Seuss's works. He treats each word as a distinct symbol. There are *n* distinct words in all. Curiously, he finds that **the most popular word (symbol) is represented by the codeword 0 in the Huffman encoding.** Symbol *i* occurs with probability p_i; $p_1 \geq p_2 \geq p_3 \ldots \geq p_n$. Its length in the

Huffman code tree is L_i.

1. Given the conditions above, is it **True** or **False** that $p_1 \geq 1/3$? Explain.

(points: 1)

2. The Grinch removes the most-popular symbol (whose probability is p_1) and implements Huffman coding over the remaining symbols, retaining the same probabilities proportionally; i.e., the probability of symbol i (where $i > 1$) is now $p_i / (1 - p_1)$. What is the expected code length of the Grinch's code tree, in terms of L, the **expected code length of the <u>original</u> code tree**, as well as p_1? Explain.

(points: 1)

D. The Cat in the Hat compresses Green Eggs and Ham with the LZW compression method described in 6.02 (codewords from 0 to 255 are initialized to the corresponding ASCII characters, which includes all the letters of the alphabet and the space character). The book begins with these lines:
I_am_Sam
I_am_Sam
Sam_I_am

We have replaced each space with an underscore (_) for clarity, and eliminated punctuation marks.

1. What are the strings corresponding to codewords 256, 257, and 258 in the string table?

(points: 0.5)

2. When compressed, the sequence of codewords starts with the codeword 73, which is the ASCII value of the character "I". The initial few codewords in this sequence will all be < 255, and then one codeword > 255 will appear. What **string** does that codeword correspond to?

(points: 0.5)

3. Cat finds that codeword 700 corresponds to the string "I_do_not_l". This string comes from the sentence "I_do_not_like_them_with_a_mouse" in the book. What are the first two letters of the codeword numbered 701 in the string table?

(points: 1)

4. Thanks to a stuck keyboard (or because Cat is an ABBA fan), the phrase "IdoIdoIdoIdoIdo" shows up at the input to the LZW compressor. The decompressor gets a codeword, already in its string table, and finds that it corresponds to the string "Ido". This codeword is followed immediately by a new codeword **not** in its string table. What string should the decompressor return for this new codeword?

(points: 0.5)

Problem 4. LZW compression. (2 points)

A. Suppose the sender adds two strings with corresponding codewords c_1 and c_2 in that order to its string table. Then, it *may* transmit c_2 for the first time **before** it transmits c_1. Explain whether this statement is True or False.

(points: 1)

B. Consider the LZW compression and decompression algorithms as described in 6.02. Assume that the scheme has an initial table with code words 0 through 255 corresponding to the 8-bit ASCII characters; character ``a" is 97 and ``b" is 98. The

receiver gets the following sequence of code words, each of which is 10 bits long:

```
97 97 98 98 257 256=
```

What was the original message sent by the sender?

(points: 1)

Zip archive of all required files for the programming tasks on this PS. Extract using unzip.

You'll need to install numpy and matplotlib (they should be available on the lab machines). Click here for instructions.

Python Task 1: Creating Huffman codes (3 points)

Useful download links:

PS1_tests.py -- test jigs for this assignment
PS1_1.py -- template file for this task

The process of creating a variable-length code starts with a list of message symbols and their probabilities of occurrence. As described in the lecture notes, our goal is to encode more probable symbols with shorter binary sequences, and less probable symbols with longer binary sequences. The Huffman algorithm builds the binary tree representing the variable-length code from the bottom up, starting with the least probable symbols.

Please complete the implementation of a Python function to build a Huffman code from a list of probabilities and symbols:

`encoding_dictionary` `= huffman(`*`plist`*`)=`

Given `plist`, a sequence of tuples `(prob,symbol)`, use the Huffman algorithm to construct and return a dictionary that maps symbols to their corresponding Huffman codes, which should be represented as *lists of binary digits*.

You will find it necessary to select the minimum element from pList. During the process, if you need to select the minimum element from a pList, you can use built-in Python functions such as sort() and sorted(). Python's heapq module is another alternative.

PS1_1.py is the template file for this problem.

The testing code in the template runs your code through several test cases. You should see something like the following print-out (your encodings may be slightly different, although the length of the encoding for each of the symbols should match that shown below):

```
Huffman encoding:=
   B = 00=
   D = 01=
   A = 10=
   C = 11=
  Expected length of encoding a choice = 2.00 bits
  Information content in a choice = 2.00 bits
Huffman encoding:
   A = 00=
   D = 010=
   C = 011=
   B = 1=
  Expected length of encoding a choice = 1.66 bits
  Information content in a choice = 1.61 bits
Huffman encoding:
   II = 000
   I = 0010
   III = 0011
   X = 010
   XVI = 011
   VI = 1
  Expected length of encoding a choice = 2.38 bits
  Information content in a choice = 2.30 bits
Huffman encoding:
   HHH = 0=
   HHT = 100=
   HTH = 101=
   THH = 110=
   HTT = 11100=
   THT = 11101=
   TTH = 11110=
   TTT = 11111=
  Expected length of encoding a choice = 1.60 bits
  Information content in a choice = 1.41 bits=
```

When you're ready, please submit the file with your code using the field below.

File to upload for Task 1: Browse...

(points: 3)

Python Task 2: Decoding Huffman-encoded messages (5 points)

Useful download links:

PS1_2.py -- template file for this task

Encoding a message is a one-liner using the encoding dictionary returned by the `huffman=` routine -- just use the dictionary to map each symbol in the message to its binary encoding and then concatenate the individual encodings to get the encoded message:

```
def encode(encoding_dict,message):
    return numpy.concatenate([encoding_dict[obj]=
```

```
for obj in message])=
```

Decoding, however, is a bit more work. Write a Python function to decode an encoded message using the supplied encoding dictionary:

`decoded_message` `= decode(`*`encoding_dict,encoded_message`*`)=`

> `encoded_message` is a numpy array of binary values, as returned by the `encode=` function shown above (this array can be indexed just like a list). `encoding_dict` is a dictionary mapping objects to lists of binary characters, as with the output of your `huffman` function from task 1.
>
> Your function should return (as a list) the sequence of symbols representing the decoded message.

PS1_2.py is the template file for this problem:

When you're ready, please submit the file with your code using the field below.

File to upload for Task 2: Browse...

(points: 5)

Python Task 3: Huffman codes in practice -- fax transmissions (5 points)

Useful download links:

> PS1_3.py -- Python file for this task
> PS1_fax_image.png -- fax image

A fax machine scans the page to be transmitted, producing row after row of pixels. Here's what our test text image looks like:

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam aliquet. Proin dapibus, lorem id interdum interdum, libero erat consequat risus, et vehicula eros lacus non nibh. Fusce suscipit, ipsum in porttitor tempor, odio purus tempor libero, vehicula feugiat nisl tellus eu ante. Maecenas euismod placerat lectus. Duis quis quam eu elit pellentesque varius. Etiam non pede a arcu euismod tempor. Etiam tincidunt egestas nunc. Fusce auctor semper tortor. Morbi dolor diam, condimentum id, volutpat a, sagittis a, sem. Praesent ac pede ac nisl aliquam varius. Vivamus lacinia, magna ut bibendum interdum, ligula eros posuere nisl, at eleifend sapien dui vel enim. Maecenas vitae pede. Praesent vestibulum elit.

Maecenas justo nisi, ullamcorper id, congue ac, convallis eget, purus. Fusce vel augue ac velit faucibus fringilla. Nulla quis purus sed urna cursus euismod. Nullam in leo. Sed aliquet nisi sit amet lectus. Phasellus blandit accumsan libero. Morbi eros augue, laoreet ut, blandit non, malesuada quis, purus. Morbi et elit eget elit consectetuer pretium. Nullam gravida sem vel urna. Fusce lacinia venenatis felis. Quisque tortor lorem, porttitor non, consequat eu, consequat et, massa.

Vestibulum nisl nisi, ultricies et, volutpat sit amet, tincidunt ac, diam. Nam vel dolor. Praesent ante neque, tincidunt eu, adipiscing eget, blandit ac, lacus. Nulla facilisi. In commodo semper mi. Aliquam erat volutpat. Aenean consectetuer arcu a arcu. Proin aliquet odio ut nunc. Phasellus vel sem. Nullam nec libero.

Instead of sending 1 bit per pixel, we can do a lot better if we think about transmitting the image in chunks, observing that in each chunk we have alternating runs of white and black pixels. What's your sense of the distribution of run lengths, for example when we arrange the pixels in one long linear array? Does it differ between white and black runs?

Perhaps we can compress the image by using **run-length encoding**, where we send the lengths of the alternating white and black runs, instead of sending the pixel pattern directly. For example, consider the following representation of a 4x7 bit image (1=white, 0=black):

```
1 1 0 0 1 1 1=
1 1 1 0 0 1 1=
1 1 1 1 0 0 1=
1 1 1 1 1 1 1=
```

This bit image can be represented as a sequence of run lengths: [2,2,6,2,6,2,8]. If the receiver knows that runs alternate between white and black (with the first run being white) and that the width of the image is 7, it can easily reconstruct the original bit pattern.

It's not clear that it would take fewer bits to transmit the run lengths than to transmit the original image pixel-by-pixel -- that'll depend on how clever we are when we encode the lengths! If all run lengths are equally probable then a fixed-length encoding for the lengths (e.g., using 8 bits to transmit lengths between 0 and 255) is the best we can do. But if some run length values are more probable than others, we can use a variable-length Huffman code

to send the sequence of run lengths using fewer bits than can be achieved with a fixed-length code.

PS1_3.py runs several encoding experiments, trying different approaches to using Huffman encoding to get the greatest amount of compression. As is often the case with developing a compression scheme, one needs to experiment in order to gain the necessary insights about the most compressible representation of the message (in this case the text image).

Please run PS1_3.py, look at the output it generates, and then tackle the questions below.

Here are the alternative encodings we'll explore:

Baseline 0 -- Transmit the b/w pixels as individual bits

The raw image contains 250,000 black/white pixels (0 = black, 1 = white). We could obviously transmit the image using 250,000 bits, so this is the baseline against which we can measure the performance of all other encodings.

Baseline 1 -- Encode run lengths with fixed-length code

To explore run-length encoding, we've represented the image as a sequence of alternating white and black runs, with a maximum run size of 255. If a particular run is longer than 255, the conversion process outputs a run of length 255, followed by a run of length 0 of the opposite color, and then works on encoding the remainder of the run. Since each run length can be encoded in 8 bits, the total size of the fixed-length encoding is 8 times the number of runs.

Baseline 2 -- Lempel-Ziv compressed PNG file

The original image is stored in a PNG-format file. PNG offers lossless compression based on the Lempel-Ziv algorithm for adaptive variable-length encoding described in Chapter 3. We'd expect this baseline to be very good since adaptive variable-length coding is one of the most widely-used compression techniques.

Experiment 1 -- Huffman-encoding runs

As a first compression experiment, try using encoding run lengths using a Huffman code based on the probability of each possible run length. The experiment prints the 10 most-probable run lengths and their probabilities.

Experiment 2 -- Huffman-encoding runs by color

In this experiment, we try using separate Huffman codes for white runs and black runs. The experiment prints the 10 most-probable run lengths of each color.

Experiment 3 -- Huffman-encoding run pairs

Compression is always improved if you can take advantage of patterns in the message. In our run-length encoded image, the simplest pattern is a white run of some length (the space between characters) followed by a short black run (the black pixels of one row of the character).

Experiment 4 -- Huffman-encoding 4x4 image blocks

In this experiment, the image is split into 4x4 pixel blocks and the sixteen pixels in each block are taken to be a 16-bit binary number (i.e., a number in the range 0x0000 to

0xFFFF). A Huffman code is used to encode the sequence of 16-bit values. This encoding considers the two-dimensional nature of the image, rather than thinking of all the pixels as a linear array.

The questions below will ask you analyze the results. In each of the experiments, look closely at the top 10 symbols and their probabilities. When you see a small number of symbols that account for most of the message (i.e., their probabilities are high), that's when you'd expect to get good compression from a Huffman code.

The questions below include the results of running PS1_3.py using a particular implementation of `huffman`. Your results should be similar.

A.
```
Baseline 1:'=
  total number of runs: 37712'
  bits to encode runs with fixed-length code: 301696'=
```

Since 301696 > 250000, using an 8-bit fixed-length code to encode the run lengths uses more bits than encoding the image pixel-by-pixel. What does this tell you about the distribution of run length values? Hint: what happens when runs are longer than 8 bits?

(points: 1)

B.
```
Experiment 1:
  bits when Huffman-encoding runs: 111656
  Top 10 run lengths [probability]:
    1 [0.39]
    2 [0.19]
    3 [0.13]
    4 [0.12]
    5 [0.05]
    7 [0.03]
    6 [0.02]
    8 [0.01]
    0 [0.01]
    255 [0.01]=
```

How much compression did Huffman encoding achieve, expressed as the ratio of unencoded size to encoded size (aka the *compression ratio*)? Briefly explain why the Huffman code was able to achieve such good compression.

(points: .5)

Briefly explain why the probability of zero-length runs is roughly equal to the probability of runs of length 255.

(points: .5)

C.
```
Experiment 2:
  bits when Huffman-encoding runs by color: 95357
  Top 10 white run lengths [probability]:
    2 [0.25]
    4 [0.20]
    3 [0.19]
    5 [0.08]
    6 [0.05]
    7 [0.05]
    1 [0.04]
    8 [0.02]
    255 [0.02]
    10 [0.02]
  Top 10 black run lengths [probability]:
    1 [0.73]
    2 [0.13]
    3 [0.07]
    4 [0.03]
    0 [0.02]
    5 [0.01]
    7 [0.01]
    8 [0.00]
    9 [0.00]
    10 [0.00]=
```

Briefly explain why the compression ratio is better in Experiment 2 than in Experiment 1.

(points: 1)

D.
```
Experiment 3:
  bits when Huffman-encoding run pairs: 87310
  Top 10 run-length pairs [probability]:
    (2, 1) [0.20]
    (4, 1) [0.15]
    (3, 1) [0.12]
    (5, 1) [0.07]
    (3, 2) [0.04]
    (7, 1) [0.03]
    (2, 2) [0.03]=
```

```
(4, 2) [0.03]=
(1, 1) [0.03]
(6, 1) [0.03]=
```

Briefly explain why the compression ratio is better in Experiment 3 than in Experiments 1 and 2.

(points: 1)

E.
```
Experiment 4:
  bits when Huffman-encoding 4x4 image blocks: 71628
  Top 10 4x4 blocks [probability]:
    0xffff [0.55]
    0xbbbb [0.02]
    0xdddd [0.02]
    0xeeee [0.01]
    0x7777 [0.01]
    0x7fff [0.01]
    0xefff [0.01]
    0xfff7 [0.01]
    0xfffe [0.01]
    0x6666 [0.01]=
```

Using a Huffman code to encode 4x4 pixel blocks results in a better compression ratio than achieved even by PNG encoding. Briefly explain why. [Note that the number of bits reported for the Huffman-encoded 4x4 blocks does not include the cost of transmitting the custom Huffman code to the receiver, so the comparison is not really apples-to-apples. But ignore this for now -- one can still make a compelling argument as to why block-based encoding works better than sequential pixel encoding in the case of text images.]

(points: 1)

Python Task 4: LZW compression (7 points)

Useful download links:

PS1_lzw.py -- template file for this task

In this task, you will implement the compression and decompression methods using the LZW algorithm, as presented in class and in Chapter 3. (We won't repeat the description here.) Our

goal is to compress an input file to a binary file of codewords, and uncompress binary files (produced using `compress`) to retrieve the original file.

We will use a table size of 2^{16} entries, i.e., each codeword is 16 bits long. The first 256 of these, starting from 0000000000000000 to 0000000011111111 should be initialized to the corresponding ASCII character; i.e., entry *i* should be initialized to `chr(i)` in Python for $0 \leq$ `i` ≤ 255.

Your task is to write two functions: `compress` and `uncompress`:

`compress(filename)`

Takes a filename as input and creates an LZW-compressed file named filename.zl (i.e., append the string ".zl" to the input file name). The basic task is to correctly handle all files that require no more than 2^{16} codewords. If a file has more than this number, you may take one of two actions:

- terminate the program after printing the string "This file needs more than 2**16 entries" OR
- Over-writing previously used table entries in `compress()` and `uncompress=` (consistently), coming up with a design **of your own**. This part is **optional** and will not be graded, so please don't feel compelled to work on it. But if you have the time and inclination and want to design this part (so you can handle large files, for instance), you are welcome to do it and demonstrate it during your checkoff interview.

`uncompress(filename)`

Takes a filename as input and creates an uncompressed file named filename.u (i.e., append the string "u" to the input file name). For example, if the input file is `test.zl`, your output file should be `test.zl.u`. If the file cannot be uncompressed successfully because it contains an invalid codeword, then terminate the program (don't worry about perfect error handling in this case; it's OK if the program terminates abruptly).

You can run the program as follows:

To compress: `python PS1_lzw.py -f <filename>'`
To uncompress: `python PS1_lzw.py -f <filename> -u'=`

You may find the following notes helpful for implementing these two functions in Python:

- The main thing to keep in mind that the compressed file is a **binary** file; you can't just write out strings corresponding to the codewords into the output file and expect that it will reduce space. The 16-bit number 0001010101011010, for example, should be stored in the file packed into two bytes, and not as a string of 16 characters (the latter would take up 8 times as much space). So remember to open the output file as a binary writeable file in `compress()` and remember to open the input file as a binary readable file in `uncompress()`.
- If you aren't familiar with how to open, read, and write files, check out the Python documentation on file I/O. For this task, code like the following should suffice:

```
outname = 'myoutputfilename'    # note: should be filename + '.zl'=
```

```
f = open(filename, 'rb')   # open file in read binary mode=
data = f.read()   # reads data from file f into a string
outfile = open(outname, 'wb') # open file for binary writing =
```

- To read and write binary data to/from files, you may find the `array` module useful. You may also find the `struct` module useful, depending on how you write your code.
- 16-bit codewords means that you can think of each codeword as an unsigned short integer in your Python code; the typecode format "H" for such data will be convenient to use.
- This is probably the simplest way of reading data into an array from a file :
    - When you are reading two bytes at a time ie when decompressing a file (a compressed file is the input)

```
=
     f = open(filename, 'rb')
     compressed = array.array("H", f.read())=
   =
```

    - When reading one byte at a time ie when compressing a file

```
     f = open(filename, 'rb')
     compressed = array.array("B", f.read())=
   =
```

- Your task is to implement both the compress and uncompress functions according to the method described in lecture and Chapter 3. To increase the likelihood that your software meets the intended specification according to Chapter 3, we have provided some test inputs and outputs, which you can download here (these only test the cases when the number of codewords does not exceed 2^{16}). Each test file (`a` and `rj`) has a corresponding .zl (`a.zl` and `rj.zl`) version. You may use these test files in debugging your implementation (but note that working correctly on these tests does not necessarily imply that your implementation is perfect). (We will use additional inputs to test your code, but again not exceeding 2^{16} codewords.)
- As an added consistency check, please verify for some of your own test files that running `compress` followed by `uncompress` produces the original file exactly (you can use `diff` on Linux or Mac machines to see if two files differ; you can use `diff` on Windows too if you install cygwin).

When you're ready, please submit the file with your code using the field below.

File to upload for LZW task: Browse...

(points: 7)

Questions for LZW task:

A. Download this zip archive and run unzip to extract three compressed (.zl) files, `g.zl`, `s.zl`, and `w.zl`. For these files, run your program and enter the following information:
    - The ratio of the compressed file size to the original uncompressed version. (*Optionally* (and only optionally), compare the effectiveness of your

compression to the UNIX `compress` and `gzip` utilities, if you want to see how close you get to these programs. Both programs are available on athena.)

- The number of entries are in the string table (maintained by both compress and uncompress). You can print the length of the table in `uncompress` to obtain this information.

(points: .5)

B. Using your `compress` implementation, try *compressing* `s.zl` (from the zip archive) even though it is already compressed. Does it further reduce the file size? Why or why not?

(points: .5)

6.02 Introduction to EECS II: Digital Communication Systems

Fall 2012