# Assignment 2 (due 11:55pm, Sunday 10 June 2012)

## Synopsis

You are to create a program which losslessly compresses and decompresses arbitrary files using arithmetic coding plus a first-order Markov model to provide probability estimates for symbols. To be more precise, the method you will implement is PPMA with k=1 and use of lazy update exclusion.

The program is to be written in Java, C, C++ or C#. (However you are provided with sample code in only Java and C.)

## More Details

### Usage

The Java version of the program is to be invoked from the command-line using a command like the following to compress a file named `test1.txt`:

```
java ppmc test1.txt
```

Invoking the C version of the program would be similar

```
ppmc test1.txt
```

The program should create a compressed file named `test1.txt.ppm`. The uncompressed input file may have any name; the new compressed file has a name formed by appending the suffix '`.ppm`'.

The same program can be invoked on a compressed file, such as `test1.txt.ppm`, as follows:

```
java ppmc test1.txt.ppm
```

or (for the C version of the program)

```
ppmc test1.txt.ppm
```

to create a new uncompressed file named `new-test1.txt`. The uncompressed file should be identical to the original file that was compressed.

### The Supplied Code

You are provided with the source code for the Arithmetic Coding equivalent of Adaptive Huffman Coding, as covered in class. Both C and Java versions are provided. You will need to replace the code which generates and uses the data model (the cumulative symbol counting code) with new code as described below. For the C version, this implies replacing the files `achuffman.c` and `achuffman.h` with new files and modifying `main.c` to invoke your new code. For the Java version, you need to replace the `ACAdaptiveHuffman.java` file with a new file named `ppmc.java`.

You are encouraged to compile and run the supplied code and verify that it does achieve a modest degree of compression on most files.

### Pseudocode for Compression

The pseudocode uses three arrays `Count2[257][257]`, `Count1[257]` and `UnitVector[257]`. Their meanings are as follows.

- `Count2[x][y]` is the number of times that the pair of symbols x followed by y has appeared so far in the input.
- `Count1[x]` is the number of times that symbol x has appeared so far in the input.
- `UnitVector[x]` holds 1 for all values of x.

Symbols are integers in the range 0 to 256 inclusive. Symbol values from 0 to 255 are byte values read from the input. The symbol value 256 represents a special ESCAPE code.

The `Count2` and `Count1` arrays are initialized to zero, except that `Count2[x][ESCAPE]` holds 1 for all x, and `Count1[ESCAPE]` holds 1.

Some pseudocode for estimating probabilities and then compressing is as follows.

```
prev = ESCAPE
while not end of file
     curr = read next input byte
     if Count2[prev][curr] = 0 then
          encode(Count2[prev],ESCAPE)
          if Count1[curr] = 0 then
               encodeV(Count1,ESCAPE)
               encodeV(UnitVector,curr)
          else
               encodeV(Count1,curr)
          Count1[curr] = Count1[curr]+1
     else
          encodeV(Count2[prev],curr)
     Count2[prev][curr] = Count2[prev][curr]+1
     prev = curr
encodeV(Count2[prev],ESCAPE)
encodeV(Count1,ESCAPE)
encodeV(UnitVector,ESCAPE)
```

where `encodeV` is a method/function which takes a vector of 257 counts as its first argument and a number in the range 0 to 256 (the symbol to encode) as its second argument. Its pseudocode is as follows:

```
encodeV( counts, symbol ):
```

$$\texttt{total} = \sum_{i=0}^{256} \texttt{counts}_i$$

$$\texttt{cumulative} = \sum_{i=0}^{\texttt{symbol}-1} \texttt{counts}_i$$

```
encode(cumulative, cumulative+counts[symbol], total)
```

where `encode` is the Arithmetic Encoding function shown on the lecture slides and implemented in the supplied code.

## Pseudocode for Decoding

The decoding of symbols and the updating of the counts has to exactly mirror what the compression method does.

Assuming that the arrays are initialized exactly the same way as above, pseudocode for decompression is as follows.

```
prev = ESCAPE
while true
      curr = getSymbol(Count2[prev])
      if curr = ESCAPE then
            curr = getSymbol(Count1)
            if curr = ESCAPE then
                  curr = getSymbol(UnitVector)
                  if curr = ESCAPE then
                        return  // that was an end of file
            Count1[curr] = Count1[curr]+1
      Count2[prev][curr] = Count2[prev][curr]+1
      output byte curr
      prev = curr
```

where `getSymbol` has this pseudocode

```
getSymbol( counts ):
```

$$\texttt{total} = \sum_{i=0}^{256} \texttt{counts}_i$$

```
freq = decode_target(total)
```

$$\texttt{symbol} = \min \text{k such that} \quad \texttt{freq} < \sum_{i=0}^{k} \texttt{counts}_i$$

```
decode(cumulative, cumulative+counts[symbol], total)
return symbol
```

### Submission

- You must submit *all* your source code files. They must be combined into a single archive file in one of three standard formats. We consider zip, tar+gzip (*aka* tgz) and jar to be the only acceptable formats.
  On a Linux system, where we assume that you used C as the implementation language, one way to create a tar+gzip archive uses these two commands
  ```
  tar -cvf myAssignment2Code.tar *.h *.c
  gzip myAssignment2Code.tar
  ```
- Make sure that your student number appears in the comment at the beginning of every file you create or modify. (Your name is optional.)
- Use the conneX website in the Assignment 2 section to submit your archive file. *Although conneX allows you to submit several files, we do not want that. We only want to receive a single archive file.*
- If the TA cannot successfully compile and run your program you will lose many marks. You should double check that you did not omit a needed file by unpacking your archive file into an empty directory and trying to compile your program.
- Once again, poor coding style and gross inefficiency in program execution (using way too much time or way too much memory) will be penalized.

### Advice on Testing and Debugging

You can add debugging code to the `encodeV` function (shown above as pseudocode) so that it outputs the value of the symbol received as its argument followed by the three arguments passed to `encode`. Output to a logfile is recommended, because you will get a lot of output.

Similarly you can add debugging code to the `getSymbol` function to show the symbol it computes followed by the three arguments passed to decode. Again, the output should be sent to a logfile.

Then run your program on a very small test case which contains some repetition, such as a file which contains just the word “banana”. The two logfiles created by an encoding run and a decoding run should be identical – i.e. identical symbol numbers and identical values passedf to encode/decode at each step. Any discrepancy and there will be a problem.

Remember that you can use a linux program like `diff` to compare two logfiles and tell you whether they are identical.