CSC 310, Spring 2004 — Assignment #2

Due at **start** of lecture on March 15. Worth 10% of the course grade.

Note that this assignment is to be done by each student individually. You may discuss it in general terms with other students, but the work you hand in should be your own.

For this assignment, you will write and evaluate programs to compress black-and-white images using arithmetic coding in conjunction with various adaptive models.

An arithmetic coding package written in C is available for use in this assignment. You may obtain it from `/u/radford/310` on CDF or from the course web page. The file `Documentation` explains how to use the procedures in this package, and briefly describes some example programs. Note that for this assignment you will need only the procedures for encoding and decoding bits (procedures for larger alphabets are provided for other uses). You may use any programming language for the assignment that is able to call the procedures in this package, but using anything other than C or C++ would probably be perverse. You should also note that if you try doing this assignment on a Windows system you will encounter problems because the procedures provided are designed for Unix/Linux systems. In particular, by default, Windows files are in "text" mode, which causes some bytes to be translated into more than one byte. If you want to use Windows, you will have to change to using "binary" mode, which may require changing the procedures so they don't use standard input and standard output.

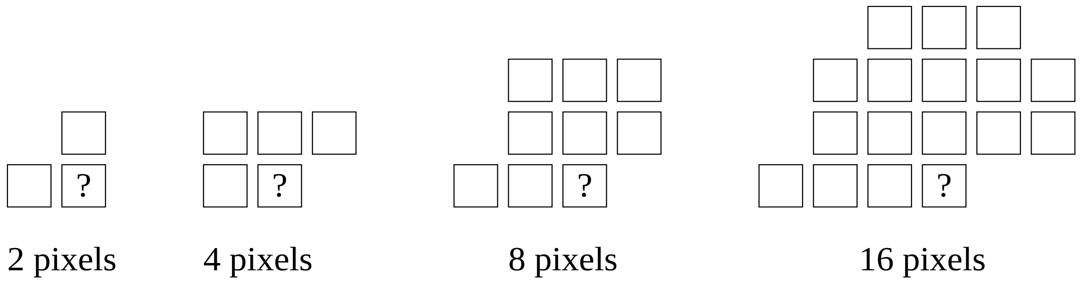
Five images produced by converting Postscript documents to bitmaps are also available on CDF (called `page1.mono` to `page5.mono`) or from the course web page. These images are stored as one bit per pixel (0=white, 1=black) in raster scan order, top to bottom, left to right (with the low-order bit of each byte coming first). All the images are 576 pixels in width and 700 pixels in height (you may fix these dimensions as constants in your program). You should evaluate your compression methods on these five images, which you may take to be typical of the ones which the compression program would be used for. You can display one of these images, or an image obtained by using your program to compress and decompress one of them, using the following command on CDF:

```
display -size 576x700 mono:image-file
```

where `image-file` is the name of the file containing the image (eg, `page1.mono`). The web page also has the five images in `.bmp` format so that you can view them in a browser. (Note, however, that these `.bmp` files are *not* the ones you should try to compress, since they contain extra header information in addition to the actual pixels.)

Programs called `encmono` and `decmono` are provided for compressing and decompressing these images using a simple adaptive model with a two-pixel context. You may refer to these programs to learn how to use the bit input/output and arithmetic coding procedures, but you will want to write your programs in a more general way in order to do this assignment.

Part 1. For the first part of this assignment, you should write programs for compressing and decompressing images using contexts of various sizes. The program discussed in lectures and the very similar `encmono` and `decmono` programs use a two-pixel context when predicting a pixel value, consisting of the pixel to the left and the pixel above. You should evaluate how much better (or worse) compression is obtained when using larger contexts. In particular, you should evaluate how well the following contexts work (the first being the one used by `encmono` and `decmono`):



Here, the box with the question mark represents the pixel currently being encoded or decoded, and the other boxes represent those pixels, which will already have been encoded or decoded, that make up the context of nearby pixels to be used in predicting the current pixel. The number of contexts is equal to the number of possible combinations of values for these nearby pixels, which is 4, 16, 256, and 65536 for the four contexts shown above. When encoding pixels near the top, left, or right edge of the image, you should pretend that any of these neighboring pixels that are outside the image are 0 (white).

For each context, you should adaptively estimate the probabilities for white and black pixels using the counts of how many white and black pixels occurred previously. To these counts, you should add some constant, and then divide by the total to obtain probabilities. You should try adding the constant one (the Laplace scheme discussed in class), and also try the alternative of adding the constant 0.1. Since the arithmetic coding procedures take integer counts, you should implement adding 0.1 by keeping counts that are ten times the actual counts — so you add 10 to the appropriate count each time you see a new pixel, and initialize the counts to 10 for the Laplace scheme and to 1 for the alternative scheme.

The four possible contexts and the two possible schemes for initializing counts produce a total of eight methods for compressing images. You should try out all these methods on each of the five images provided. Since that's forty runs, you won't want to do them all manually. The `testpages` command file shows how (for the `encmono` and `decmono` programs) the process of trying out a method on all five images can be automated. It compresses and decompresses each of these images, checks that the decompressed image is identical to the original (using the `cmp` command), and outputs the size (in bytes) of the compressed image. You should do something similar to automate your tests.

You should hand in the output of these tests (along with the command files used to run them), and a discussion of the results — for example, if one method is better for some images but not for other images, you should try to explain this in terms of what the images are like. You should also hand in a listing of your programs, which should be written in good programming style. You should *not* hand in eight different encode and decode programs, one for each combination of context and count scheme. You should instead write one encode program and one decode program, which take arguments specifying the context and count scheme to be used. You may write this program to be more general than required for this assignment if you wish. In particular, you may write a single program to handle both this part of the assignment and the next, though you may also write separate programs for the two parts, if you wish.

Part 2. For the second part of this assignment, you should try to find a way of getting better compression than you obtained with the methods in Part 1 by adjusting the size of the context according to how much data is available, similarly to how the PPM method discussed in lectures works. However, the specifics of PPM, such as the use of an "escape" symbol, may not be appropriate for this application, in which the alphabet is binary. You should try out various schemes that you think are promising, and evaluate them on the five images provided. Among the issues to consider are what smaller context to use if the largest context doesn't have enough data, what constitutes "enough" data, and how to update the counts in the smaller contexts (recall the two options for this discussed in lectures with regards to PPM).

You should hand in a description of the schemes you tried, the encode and decode programs that implement them, the results of your tests (and any command files used to run them), and a discussion of the results.