COMP 40 Assignment: Integer and Logical Operations

(Part C, the challenge problem, will be assigned immediately after the main part of the assignment, and will run for approximately one day.)

Contents

1 Purpose, overview, and instructions

The primary purpose of this assignment is to give you experience working with low level machine representations of numbers. You will write a program that takes a PPM image and compresses the image by transforming color spaces and discarding information not easily seen by the human eye (this is lossy compression). You will gain practice packing and unpacking binary data that puts multiple small values (both signed and unsigned) into a single larger integer (often referred to as a word). You'll work with two's-complement representations of signed data. You'll be exposed to some of the horrors of floating-point arithmetic. Finally, after your homework 4 submission is in, you'll do a challenge problem that will test the modularity of your code. Then you will breathe freely, as you will be well on your way to becoming a gifted *arith*metist.

As a minor side benefit, you'll also learn a little bit about how broadcast color TV works as well as the basic principle behind JPEG image compression.

Here's what you'll do:

- Write *and test* conversions between different representations of color image data: first a bijection between RGB and component video ($Y/P_R/P_B$) color spaces and then a discrete cosine transform.
- Write functions to put a small integer into a word or extract a small integer from a word. You'll work with both signed and unsigned integers.

There is a long story below about the representation of color and brightness and the use of techniques from linear algebra for image compression. The story is interesting and important, but the real reason you're doing this work is to give you a deep understanding of the capabilities and limitations of machine arithmetic. The amount of code you have to write is fairly small, certainly under 400 lines total. But to understand what code to write and how to put it together, you will have to analyze the problem.

Begin by running

```
git clone /comp/40/git/arith
```

to get files you will need for the assignment.

1.1 Problem-solving technique: stepwise refinement, analysis, and composition

In COMP 40, you practice *solving problems by writing programs*. You'll find problem-solving more difficult (and more satisfying) than simply writing a program someone has told you to write. To solve the problem of image compression, we recommend a technique called *stepwise refinement*.

When using stepwise refinement, one analyzes a problem by breaking the problem into parts, which in turn can be broken into subparts, and so on, until the individual sub-sub-parts are either already to be found in a library or are so easy as to be quickly solvable by simple code. Each individual subpart is solved by a function or by a collection of functions in a module. Each solution is written as another function, and so on, all the way up to the `main` function, which solves the whole problem. In other words, the solution to the main problem is composed of solutions to the individual parts. To design software systems successfully, you must master the techniques of analysis and composition.

Keep in mind these units of composition:

- The *function* should do one, simple job.
- The *interface to an abstract data type* packages an important abstraction in the world of ideas and makes it usable in a computer program. Such an interface *hides representation*, promoting reuse.
- Other *interfaces* can also promote reuse. Here are two useful design principles for interfaces:
  - *Package together* collections of functions that *operate in the same problem domain*. Examples might include statistical functions (mean, variance, covariance, and so on) or linear-algebra functions (inner and outer products, matrix multiply, matrix inversion, and so on).
  - *Package together* functions that *share a secret*. The idea is to hide the secret so you enable modular reasoning: the rest of the program doesn't know the secret, so it can depend only on the functions in the interface. A good example of this kind of interface is the `Pnm` interface, which hides the secret that each kind of PNM file has two different on-disk representations, as well as hiding the details of those representations.

In C, each interface is expressed in a `.h` file, and it normally is implemented by a single `.c` file. *We will evaluate your work according to how well you organize your solution into separate files.*

1.2 What we provide for you

All files we provide will be in `/comp/40/include` or `/comp/40/lib64`, or else you will acquire them using `git` (as discussed above). The files include

- The header file `bitpack.h` which you can compile against with the compiler option `-I/comp/40/include`. You will implement a corresponding `bitpack.c`.
- The header file `compress40.h` which you can compile against with the compiler option `-I/comp/40/include`. You will implement a corresponding `compress40.c`.

- The header file `arith40.h`, which you compile against with the `-I/comp/40/include` option, and whose implementation is in the library `libarith40.a`, which you link with the options `-L/comp/40/lib64 -larith40`[1]
- The file `40image.c`, which contains a `main` function that can handle the command line for you. You can acquire it by `git clone /comp/40/git/arith`.

1.3 What we expect from you

Your **design document**, to be submitted using `submit40-arith-design`, should *describe your overall design*. Your sections on the `Bitpack` module and on parts of the `40image` program can be relatively short, since these cases have had some of the design work done for you. But you should have a *detailed plan for testing* each of these components. Your design document may be a plain text document named `DESIGN` or a PDF named `design.pdf`.

Also, your `40image` program should not be implemented as a single component. Instead, you should design your solution as a combination of several components which should all be as modular as possible. This means reducing the degree to which each component must know implementation details about any of the other components.

The following elements of your design document will be *critical*:

- Documentation for:
  - The architecture of each major component
  - The overall architecture of your solution, i.e., how you plan to have these components interact with each other. Note that this does *not* mean that you should narrate the control flow of your solution.
- Architecture sections that identify modules, types, and functions *by name*. Choosing good names is valuable, so do it early. Formal *definitions* or *declarations* of your types and functions are not necessary at this stage; if you prefer not to write C interfaces, just sketch the types' definitions and functions' specifications in *concise*, informal English.
- *You must have a plan for testing each individual component in isolation*. If you don't have a good test plan, your compressor likely won't work. Your best bet is to write down invariants and write code to be sure that they hold on a variety of inputs. DON'T FORGET TO INCLUDE YOUR TESTING PLAN. In past years, many students have lost credit by failing to outline a sufficiently detailed test plan. The plan need not be complicated, but it must describe an effective approach to testing your implementation.

  An additional element that should help guide you toward a good design is an answer to the following question:
- How will your design enable you to do well on the challenge problem in Section 2.3 on page 12?

  Finally, here is a question that is not critical but that I would like you to answer in your design document:
- An image is compressed and then decompressed. Identify all the places where information could be lost. Then it's compressed and decompressed again. Could more information be lost? How?

  Your **implementation**, to be submitted using `submit40-arith`, should include
- File `bitpack.c`, which should implement the `Bitpack` interface.
- All the `.c` and `.h` files you create to implement the `40image` program.
- A `Makefile` file, which when run with `make`, compiles all your source code and produces both a `40image` executable binary and also a `bitpack.o` relocatable object file. File `bitpack.o` should contain the entire implementation of the `Bitpack` interface (and nothing else). You should create your `Makefile` script by adapting the ones used for earlier assignments.

[1] The linker translates the option `-lfoo` into a search for a file named `libfoo.so` or `libfoo.a`. It searches all directories named in a `-L` option as well as some built-in directories. You can see all the directories by running `gcc` with the `-v` option.

- A `README` file which
  - Identifies you and your programming partner by name
  - Acknowledges help you may have received from or collaborative work you may have undertaken with others
  - Identifies what has been correctly implemented and what has not
  - Explains the architecture of your solution
  - Says approximately how many hours you have spent *analyzing the problems posed in the assignment*
  - Says approximately how many hours you have spent *solving the problems after your analysis*

Descriptions of the image-compression and bit-packing problems follow, along with code, explanations, and advice.

2 Problems

2.1 Part A: Lossy image compression

Your goal is to convert between full-color portable pixmap images and compressed binary image files. Write a program `40image` which takes the option `-c` (for compress) or `-d` (for decompress) and also the name of the file to compress or decompress. The name of the file may be omitted, in which case you should compress or decompress standard input. If you're given something else on the command line, print the following on `stderr`:

```
Usage: 40image -d [filename]
       40image -c [filename]
```

A compressed image should be about three times smaller than the same image in PPM format. If not, you are doing something wrong.

I have designed a compressed-image format and a compression algorithm. The algorithm, which is inspired by JPEG, works on 2-by-2 blocks of pixels. Details appear below, but here is a sketch of the compression algorithm:

1. Read a PPM image from a file specified on the command line or from standard input. You can do this in the same manner that you did in the locality assignment (use either partner's code, or else the solution code once it's released). As with the locality assignment, you can let the supplied pnm reading code detect and raise exceptions for poorly formed input files; you do *not* need to catch such exceptions.

2. If necessary, trim the last row and/or column of the image so that the width and height of your image are even numbers.

3. Change to a floating-point representation, and transform each pixel from RGB color space into component video color space ($Y/P_B/P_R$).

4. Pack each 2-by-2 block into a 32-bit word as follows:

   (a) For the P_B and P_R (chroma) elements of the pixels, take the average value of the four pixels in the block (i.e., the DC component). We'll call these average values $\overline{P_B}$ and $\overline{P_R}$.

   (b) Convert the $\overline{P_B}$ and $\overline{P_R}$ elements to four-bit values using the function we provide you

   ```
   unsigned Arith40_index_of_chroma(float x);
   ```

   This function takes a chroma value between -0.5 and $+0.5$ and returns a 4-bit *quantized* representation of the chroma value.

   (c) Using a discrete cosine transform (DCT), transform the four Y (luminance/luma) values of the pixels into cosine coeffecients a, b, c, and d.

   (d) Convert the b, c, and d to five-bit signed values assuming that they lie between -0.3 and 0.3. Although these values can actually range from -0.5 to $+0.5$, a value outside the range ± 0.3 is quite rare. I'm willing to throw away information in the rare cases in order to get more precision for the common cases.

(e) Pack a, b, c, d, $\overline{P_B}$, and $\overline{P_R}$ into a 32-bit word as follows:

Value	Type	Width	LSB
a	Unsigned scaled integer	9 bits	23
b	Signed scaled integer	5 bits	18
c	Signed scaled integer	5 bits	13
d	Signed scaled integer	5 bits	8
$index(\overline{P_B})$	Unsigned index	4 bits	4
$index(\overline{P_R})$	Unsigned index	4 bits	0

The *index* operation is implemented by `Arith40_index_of_chroma`; it quantizes the chroma value and returns the index of the quantized value in an internal table.

To pack the codeword, you will use the `Bitpack` interface you will develop in Part B on page 9.

5. Write a compressed binary image to standard output. The header of the compressed binary image should be written by

```
printf("COMP40 Compressed image format 2\n%u %u", width, height);
```

This header should be **followed by a newline** and then a sequence of 32-bit code words, one for each 2-by-2 block of pixels. The `width` and `height` variables describe the dimensions of the original (decompressed) image, *after* trimming off any odd column or row.

- Each 32-bit code word should be written to disk in **big-endian** order, i.e., with the most significant byte first. You write a single byte using `putchar()`.
- Code words should be written in row-major order, i.e., first the code word for the 2-by-2 block containing pixel $(0, 0)$, then the block containing pixel $(2, 0)$, and so on.

Your decompressor will be the inverse of your compressor:

6. Read the header of the compressed file. The trick is to use `fscanf` with exactly the same string used to write the header:

5 ⟨*sketch of header reading code* 5⟩≡

```
unsigned height, width;
int read = fscanf(in, "COMP40 Compressed image format 2\n%u %u", &width, &height);
assert(read == 2);
int c = getc(in);
assert(c == '\n');
```

7. Allocate a 2D array of pixels of the given width and height. The `size` parameter should be the size of a colored pixel. Place the array, width, height, and the `denominator` of your choice in a local variable as follows:

```
struct Pnm_ppm pixmap = { .width = width, .height = height
                        , .denominator = denominator, .pixels = array
                        , .methods = methods
                        };
```

The PNM format allows some choice of denominator, but it should meet some constraints:

- The denominator should be large enough so that the resulting image does not show quantization artifacts.
- The denominator should be small enough that the resulting PPM file is not unduly large.
- The denominator may be at most 65535.

8. Read the 32-bit code words in sequence, remembering that each word is stored in big-endian order, with the most significant byte first. If the supplied file is too short (I.e. the number of codewords is too low for the stated width and height, or the last one is incomplete) fail with a Checked Runtime Error). You do not need to check for excess data following the last codeword.

9. For each code word, unpack the values a, b, c, d, and the coded $\overline{P_B}$ and $\overline{P_R}$ into local variables.

10. Convert the four-bit chroma codes to $\overline{P_B}$ and $\overline{P_R}$ using the function we provide you

    ```
    float Arith40_chroma_of_index(unsigned n);
    ```

11. Use the inverse of the discrete cosine transform to compute Y_1, Y_2, Y_3, and Y_4 from a, b, c, and d.

12. For each pixel in the current 2-by-2 block, you will now have a component-video representation of the color of that pixel, in the form $(Y_i, \overline{P_B}, \overline{P_R})$. Transform the pixel from component-video color to RGB color, quantize the RGB values to integers in the range 0 to xxx, where xxx is chosen to be appropriate for your denominator, and put the RGB values into `pixmap->pixels`. (Because repeated quantization can introduce significant errors into your computations, getting the RGB values into the right range is not as easy as it looks.)

13. Once you have put all the pixels into your pixmap, you can write the uncompressed image to standard output by calling `Pnm_ppmwrite(stdout, pixmap)`.

My compressor and decompressor total under 320 lines of new code, including many assertions and some testing code. (This total does not include the implementation of the chroma quantization in the `arith40` library.) I created five new modules (including chroma quantization) and reused a number of existing modules.

Conversion between RGB and component video

The CIE XYZ color space was created by the International Committee on Illumination in 1931. The committee is usually referred to as the CIE, which as an acronym for the French *Commission Internationale de l'Éclairage*. It is the international authority on standards for representation of light and color.

The XYZ system uses three so-called *tristimulus* values which are matched to the visual response of the three kinds of cone cells found in the human retina.[2] The Y value represents the *brightness* of a color; the X and Z values represent "chromaticity." Early black-and-white television transmitted only Y, or brightness. When color was added, analog engineers needed to make the color signal backward compatible with black-and-white TV sets. They came up with a brilliant hack: first, they made room for a little extra signal by reducing the refresh rate (number of frames per second) from 60Hz to 59.97Hz, and then they transmitted not the chromaticity, but the *differences* between the blue and red signals and the brightness. The black-and-white sets could ignore the color-difference signals, and everybody could watch TV.

The transformation is useful for compression because the human eye is more sensitive to brightness than to chromaticity, so we can use fewer bits to represent chromaticity.

There are multiple standards for both RGB and luminance/chromaticity representations. We will use component-video representation for gamma-corrected signals; this signal is a luminance Y together with two side channels P_B and P_R which transmit *color-difference* signals. P_B is proportional to $B - Y$ and P_R is proportional to $R - Y$. In each case, the constant of proportionality is chosen so that both P_B and P_R range from -0.5 to $+0.5$. The luminance Y is a real number between 0 and 1.

Given the RGB representation used by the portable pixmap (PPM) library, we can convert to component video by the following linear transformation:

```
y  =  0.299    * r + 0.587    * g + 0.114    * b;
pb = -0.168736 * r - 0.331264 * g + 0.5      * b;
pr =  0.5      * r - 0.418688 * g - 0.081312 * b;
```

[2] By contrast, the RGB system is matched to the red, green, and blue phosphors found on cathode-ray tube (CRT) computer screens. Despite the fact CRTs have largely been replaced by liquid-crystal displays, which have different color-response characteristics, computing standards remain wedded to the RGB format originally created for CRTs.

The inverse computation, to convert from component video back to RGB, is

```
r = 1.0 * y + 0.0      * pb + 1.402    * pr;
g = 1.0 * y - 0.344136 * pb - 0.714136 * pr;
b = 1.0 * y + 1.772    * pb + 0.0      * pr;
```

The discrete cosine transform

The discrete cosine transform (DCT), is a very important technique in data compression, but the details behind it are a bit beyond the scope of this course. The general idea behind the DCT as implemented in our compression algorithm here is that if we treat an image as 2 by 2 blocks of pixels and perform this transformation, we can represent the data about the brightness in the image in a way that facilitates more compression in later steps (well see more about that soon).

$$\begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix}$$

In cosine space, the 4 pixel block above gets encoded in 4 values called a, b, c, and d. Given the labeled 4 by 4 block of brightness values above, here are the equations giving the transformation to and from cosine space. (the interested reader could verify that the inverse transformation we've derived works by plugging in the formulas for a, b, c, and d into the inverse transformation):

Transforming from pixel space to DCT space:

$$\begin{aligned} a &= (Y_4 + Y_3 + Y_2 + Y_1)/4.0 \\ b &= (Y_4 + Y_3 - Y_2 - Y_1)/4.0 \\ c &= (Y_4 - Y_3 + Y_2 - Y_1)/4.0 \\ d &= (Y_4 - Y_3 - Y_2 + Y_1)/4.0 \end{aligned}$$

Transforming from DCT space to pixel space:

$$\begin{aligned} Y_1 &= a - b - c + d \\ Y_2 &= a - b + c - d \\ Y_3 &= a + b - c - d \\ Y_4 &= a + b + c + d \end{aligned}$$

When we consider the four-pixel array as an *image*, not just numbers, we can see that the new basis derived from cosine functions actually tells us something interesting about the image:

- The coefficient a is the average brightness of the image.
- The coefficient b represents the degree to which the image gets brighter as we move from top to bottom.
- The coefficient c represents the degree to which the image gets brighter as we move from left to right.
- The coefficient d represents the degree to which the pixels on one diagonal are brighter than the pixels on the other diagnoal.

So far, we've just transformed our image data into a new space, but haven't really done much in the way of compressing the data. One of the fundamental ideas in lossy data compression is that we can save bits (space) by throwing away information that humans would have trouble noticing in everyday images. So, we try and encode values that are easily noticed and occur often with high accuracy at the expense of losing some information for values that occur very rarely.

Here, we "throw away information" through a process called *quantization*, which involves representing an entire range of values by one *quantized* value that lies somewhere in that range. When the set of quantized values is small, we can encode the quantized values as an integer index into the set of quantized values.

When you round numbers in your everyday life, you're really just quantizing. For instance, if someone asks you the time, you hardly ever communicate to them the current time in milliseconds, you round to the nearest minute or 5

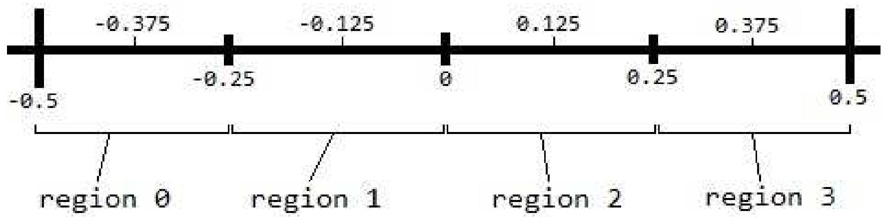


Figure 1: Illustration of quantization. values in region 0 can be represented by the integer zero, which stands for the value -0.375, values in region 1 by the integer 1 which stands for the value -0.125, etc.

minutes, knowing that saying "its 2:45PM" does a good job representing the true time 2:45:04:21 PM and does so in a much more compact way.

So, in our application, we will quantize b, c, and d as follows: (note that before, b, c, and d were likely stored in a C float which is 32 bits!)

- When $|b|$, $|c|$, and $|d|$ are small, which is to say at most 0.3, we'll code them as signed, 5-bit scaled integers.
- When $|b|$, $|c|$, or $|d|$ is not small, which is to say more than 0.3, we'll code it as if it were $+0.3$ or -0.3, whichever is closer. When b, c, and d have large magnitudes, this scheme leads to major coding errors, but in photographic images, such coefficients are rare.

Since Y_i is always in the range 0 to 1, we can see that a is also in the range 0 to 1, but b, c, and d lie in the range $-\frac{1}{2}$ to $\frac{1}{2}$. Thus, you can code a in nine unsigned bits if you multiply by 511 and round.

For coding b, c, and d, your objective is to code the floating-point interval $[-0.3, +0.3]$ into the signed-integer set $\{-15, -14, \ldots, -1, 0, 1, 2, \ldots, 15\}$. As noted above, any coefficient outside that interval should be coded as $+15$ or -15, depending on sign. There is more than one good way to do the calculation.

Quantization of chroma

Now that we know how to quantize the a, b, c, and d values, now we need to quantize chroma (color) data from the image. Empirically, we know that human eyes are better at distinguishing differences in brightness than they are in distinguishing differences in color, so we use only 4 bits for each of Pb and Pr.

As seen below, most chroma values are small, so we'll chose this set to be more densely populated in the range ± 0.10 than near the extrema of ± 0.50. By putting more information near zero, where most values are, this *nonlinear* quantization usually gives smaller quantization errors[3] than the linear quantization $n = \texttt{floor}(15 * (P_B + 0.5))$.

The quantization works by considering the floating-point chroma value and finding the closest value in the set

$$\{\pm 0.35, \pm 0.20, \pm 0.15, \pm 0.10, \pm 0.077, \pm 0.055, \pm 0.033, \pm 0.011\}.$$

Draw these numbers on a number line and look at the pattern!

But for those rare images that use saturated colors, when P_B or P_R is large, quantization errors will be larger than with a linear quantization. The net result is that when colors are more saturated, compression artifacts will be more visible. You probably won't notice artifacts if you compress an ordinary photograph, especially if it has already been compressed with JPEG. But if you try compressing and then decompressing a color-bar test pattern, you should notice artifacts.

Quantization is implemented by sorting the values above into a 16-element array. To quantize a floating-point chroma value, I find the element of the array that most closely approximates the chroma, and I return that element's index. For instance, in the example below, any number that lands in region 0 can represented by the integer 0, which maps to the floating point value -0.3.

[3]The quantization error is the difference between P_B and $\texttt{chroma\_of\_index}(\texttt{index\_of\_chroma}(P_B))$.

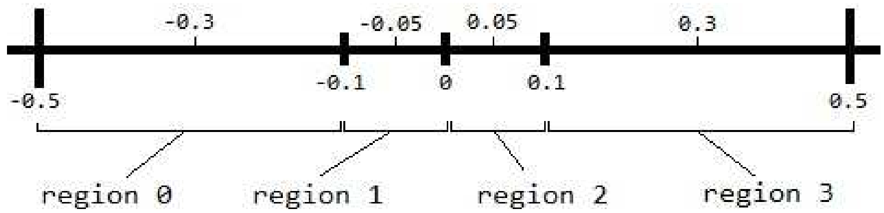


Figure 2: Example of nonlinear quantization: notice that the regions are differently sized!

2.2 Part B: Packing and unpacking integers

When programming at the machine level, it is common to pack multiple small values (sometimes called "fields") into a single byte or word. You will implement functions to perform these kinds of computations. It will take you longer to understand the specification than to write the code.

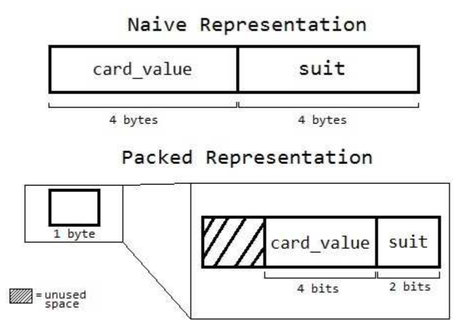
The best way to describe a field is to give its *width* and the *location of the least significant bit* within the larger byte or word.

To make the idea of bitpacking concrete, let's consider the following example: We might imagine that a naive implementation could use the following struct definition to keep track of the the suit and value of the card:

```
struct Card {
        int card_value;
        int card_suit;
};
```

This representation requires eight bytes. But we could notice that there are thirteen possible values (which can be represented using four bits with some extra space left over), and that there are four possible suits (which can be represented using only two bits).

So then we can see that we could fit a card into only a single byte. That single byte would have a field for the suit of width 2 with least significant bit 0, and a field for the value of width 4 with least significant bit 2. There are two unused bits left over starting at least significant bit 6.



When packing fields, you also have to deal with the question of whether an integer *fits* into a given number of bits. For example, the integer 17 cannot be represented in a 3-bit field.[4]

[4] A 3-bit field can be interpreted as signed or unsigned. When signed, it can represent integers in the range -4 to 3; when unsigned, it can represent integers in the range 0 to 7.

In this part of the assignment you will define bit-manipulation primitives to implement the `Bitpack` interface. Unlike the simple example above, which discusses packing bitfields into a single byte, the `bitpack` interface you will implement manipulates bit fields with a 64 bit (8 byte) value `word`.

10a ⟨*bitpack.h* 10a⟩≡

```
#ifndef BITPACK_INCLUDED
#define BITPACK_INCLUDED
#include <stdbool.h>
#include <stdint.h>
#include "except.h"
⟨exported macros, functions, and values 10b⟩
#endif
```

You are to *implement this interface* in file `bitpack.c`. Unless otherwise indicated below, your functions should use Hanson assertions (from `assert.h`) to ensure that shift values are for ≤ 64 bits, that widths are ≤ 64, and that bit fields to be accessed or updated fit entirely within the supplied 64 bit word.

Width-test functions

Your interface will test to see if an integer can be represented in k bits. The answer will depend on whether the k bits are interpreted as unsigned integers or as signed integers in the two's-complement representation. We will refer to these representations using the shorthand "k unsigned bits" and "k signed bits." Define these functions:

10b ⟨*exported macros, functions, and values* 10b⟩≡ (10a) 10c ▷

```
bool Bitpack_fitsu(uint64_t n, unsigned width);
bool Bitpack_fitss( int64_t n, unsigned width);
```

The functions tell whether the argument `n` can be represented in `width` bits. For example, 3 bits can represent unsigned integers in the range 0 to 7, so `Bitpack_fitsu(5, 3) == true`. But 3 bits can represent signed integers only in the range -4 to 3, so `Bitpack_fitss(5, 3) == false`.

Field-extraction functions

The next functions you are to define extract values from a word. Values extracted may be signed or unsigned, but by programming convention we use only unsigned types to represent words.

10c ⟨*exported macros, functions, and values* 10b⟩+≡ (10a) ◁ 10b 11a ▷

```
uint64_t Bitpack_getu(uint64_t word, unsigned width, unsigned lsb);
 int64_t Bitpack_gets(uint64_t word, unsigned width, unsigned lsb);
```

Each function extracts a field from a word given the width of the field and the location of the field's least significant bit. For example,

```
Bitpack_getu(0x3f4, 6, 2) == 61
Bitpack_gets(0x3f4, 6, 2) == -3
```

Fields of width zero are defined to contain the value zero.

Example of using field extraction: To get the `card_value` field from our Card representation, you might use `Bitpack_getu(card_word, 4, 2)`. It should be a checked run-time error to call `Bitpack_getu` or `Bitpack_gets` with a width w that does not satisfy $0 \leq w \leq 64$. Similarly, it should be a checked run-time error to call `Bitpack_getu` or `Bitpack_gets` with a width w and `lsb` that do not satisfy $w + \texttt{lsb} \leq 64$.

Some machine designs, such as the late, unlamented HP PA-RISC, provided these operations using one machine instruction apiece.

Field-update functions

If we're going to split a word into fields, we obviously want to be able to change a field as well as get one. In my design, I do not want to mess around with pointers, so "replacing" a field within a word does not mutate the original word but instead returns a new one:

11a ⟨*exported macros, functions, and values* 10b⟩+≡ (10a) ◁10c 11b▷

```
uint64_t Bitpack_newu(uint64_t word, unsigned width, unsigned lsb, uint64_t value);
uint64_t Bitpack_news(uint64_t word, unsigned width, unsigned lsb,  int64_t value);
```

Each of these functions should return a new word which is identical to the original `word`, except that the field of width `width` with least significant bit at `lsb` will have been replaced by a `width`-bit representation of `value`.

It should be a checked run-time error to call `Bitpack_newu` or `Bitpack_news` with a width w that does not satisfy $0 \leq w \leq 64$. Similarly, it should be a checked run-time error to call `Bitpack_newu` or `Bitpack_news` with a width w and `lsb` that do not satisfy $w + \texttt{lsb} \leq 64$.

If `Bitpack_news` is given a `value` that does not fit in `width` signed bits, it must raise the exception `Bitpack_Overflow` (using Hanson's `RAISE` macro from his `Except` interface). Similarly, if `Bitpack_newu` is given a `value` that does not fit in `width` unsigned bits, it must also raise the exception.

11b ⟨*exported macros, functions, and values* 10b⟩+≡ (10a) ◁11a

```
extern Except_T Bitpack_Overflow;
```

You'll implement this exception as follows:

11c ⟨*code to be used in file* `bitpack.c` 11c⟩≡

```
#include "except.h"
Except_T Bitpack_Overflow = { "Overflow packing bits" };
```

If no exception is raised and no checked run-time error occurs, then `Bitpack_getu` and `Bitpack_newu` satisfy the mathematical laws you would expect, for example,

```
Bitpack_getu(Bitpack_newu(word, w, lsb, val), w, lsb) == val
```

A more subtle law is that if `lsb2 >= w + lsb`, then

```
getu(newu(word, w, lsb, val), w2, lsb2) == getu(word, w2, lsb2)
```

where in order to fit the law on one line, I've left off `Bitpack_` in the names of the functions. Similar laws apply to the signed `get` and `new` functions. Such laws make an excellent basis for unit testing.[5] You can also unit-test the `fits` functions by using the `TRY` construct in Hanson's exceptions chapter to ensure that the `new` functions correctly raise an exception on being presented with a value that is too large.

I'm aware of three *design alternatives* for the `Bitpack` module:

- Implement the signed functions using the unsigned functions
- Implement the unsigned functions using the signed functions
- Implement the signed functions and the unsigned functions independently, in such a way that neither is aware of the other

Any of these alternatives is acceptable.

Not counting the code shown here, my solution to this problem is about 90 lines of C, plus another 80 lines for testing.

[5] "Unit testing" means testing a solution to a subproblem before testing the solution to the whole problem.

2.3 Part C: The challenge problem

A few days after the main arith submission is due, I will announce a new format for the codeword in your image compressor. You will have **one day** to change your code to support the new format. You will be evaluated on the magnitude and scope of your changes (earlier bullets are more important):

- The more interfaces you have to change, the worse it is for you.
- The more modules you have to change, the worse it is for you—but a change in an implementation is not nearly as bad as a change in an interface.
- The more lines of code you have to change, the worse it is for you.
- For each change you make, the larger the source file containing that change, the worse it is for you.

The ideal code would allow you to adjust to a new codeword format by changing just a single line of a single file. I don't expect anyone to meet this ideal.

(I am posing this problem is not for its own sake, but to provoke you into thinking more deeply about your design. By thinking deeply, you are more likely to build a working compressor.)

(Continued on next page)

3 Supplementary material

3.1 Traps and pitfalls

Computations involving arithmetic are the most difficult to get right; a trivial typo can lead to a program that silently produces wrong answers, and finding it can be nearly impossible. The only helpful strategy is aggressive unit testing.

- A hellish property of C is that left and right shift are undefined when shifting by the word size. Worse, the Intel hardware does something very inconvenient: *if you shift a word left or right by 64 bits, nothing happens*. I recommend that you define helper functions to do your shifting. Your functions, unlike the hardware, should do something sensible when asked to shift by 64. (I expect you to figure out what might be sensible.)

- If you read the specification for the C language very carefully, you will notice that, even though `>>` on a signed value propagates the sign bit on our AMD 64 machines and indeed on almost all reasonable machines, that is not strictly required. There do exist a few computer models on which it does not work as you would expect, and the C standard allows for such variability. *Nonetheless, your solution may depend on the usual behavior of the >> operator on signed numbers; you may assume that it does indeed propage the sign bit.*

- C literals like `0`, `1`, `0l`, `1l`, `1ul`, and so on are *not* guaranteed to be 64 bits. Unadorned literals are considered to be of type `signed int`, which is 32 bits on our system. I recommend two safe ways make sure you are using all 64 bits when shifting, complementing, and doing other bitwise operations:

  - Explicitly cast the thing shifted or complemented (i. e., the constant) to `(uint64_t)` or `(int64_t)`
  - Shift or complement only variables, never literals, and declare each variable with type `uint64_t` or `int64_t`

  If you take our advice and use helper functions to do your shifting, and if you declare their arguments to have 64-bit types, the shifts will "just work™."

- You may be tempted to implement parts of the `Bitpack` interface by using a loop that does one iteration per bit. Don't! The `Bitpack` operations need to be implementable in one or two dozen instructions apiece. This is true not only to meet performance goals for code that we will rely on heavily, but to meet learning goals that you understand how to compute with shift, bitwise complement, and the other Boolean operations on bit vectors.

- You may be tempted to try to using the floating-point unit to compute powers of two. Don't! The problem is that at any given word size, a floating-point number reserves not only 1 bit for the sign bit, but a cluster of bits for an exponent. This means that a floating-point number always offers less precision than an integer of the same number of bits. In particular, an IEEE `double` contains only 64 bits of precision, and because some bits are used for sign and exponent, a `double` cannot represent all 64-bit integers. A `float`, whose representation is only 32 bits, is even worse.

  Once `n` is large enough, doing arithmetic with `pow(2, n)` *will* lead to serious error. Although you could possibly get away with computing `pow(2, n)` and then immediately converting to an integer, this idiom would be very confusing to any experienced C programmer you might work with; a C programmer expects you to compute 2^{n} using the expression `((uint64_t)1 << n)`, unless of course `n` is greater than or equal to the word size, in which case the number can't be represented anyway.

- Quantization error can drive values out of range. For example, when converted to floating-point component video, compressed, quantized, decompressed to floating-point component video, and finally converted back to floating-point color, colors may go negative. I know of three ways to solve this problem, but for this assignment, the best way to solve this problem is to behave like an engineer in a hurry and just do the floating-point arithmetic and then use a helper function to force each value into the interval where it belongs.

3.2 Detailed advice for `Bitpack`

Here are some ideas to keep in mind when you approach the `Bitpack` module:

- The hardware provides three simple, powerful shift operations. But the C programming language, which is usually so good at letting you get your hands on the hardware, tends to get in your way here:
  - It's too easy to confuse the two different right shifts.
  - It's too easy to get a shift that operates on only 32-bit values when you really want to operate on all 64 bits of a word. For example, the expression `2 << 60` does not do what you would hope (its value is not 2^{60}).

  I suggest working around the C problems by defining three helper functions, each of which gives you one hardware shift instruction. This way you can have a single point of truth where you answer the question

  > What C do I have to utter to get the hardware effect that I want?

  You can use this same point of truth to define a shift operation that is *better* than the hardware—one that does something sensible when asked to shift left or right by a full 64 bits.

- Once you can easily command the shift you want in the place you want it, the other part of the problem is figuring out which shifts to ask for. Here the best approach is to draw pictures. What does the word look like when you start? What do you want it look like when you finish? If you need intermediate words, what do they look like?

  A good way to draw pictures is to write `abcde` and so on for fields that you care about, and `xxxxxx` or `yyyyyy` for fields that you don't care about. Left and right shifts can move or eliminate fields, and if you have different words that contain fields in different positions, with zeroes elsewhere, you can compose them into a single word using bitwise or (single `|`).

3.3 Other helpful advice

In addition to avoiding the traps and pitfalls and defining your own shift operations, you might benefit from the following advice:

- Don't start with `Bitpack`. (But do get it started before the `Bitpack` lab.)
- The bit-packing functions obey a ton of algebraic laws. Discover them; code them; check them.
- Conversion from RGB to component video and back should be inverse functions; check both directions. Likewise for the cosine transform.
- Encoding and decoding a, b, c, and d into the codeword should be near-inverses, *provided* that b, c, and d have magnitude no larger than 0.3. Larger values of b, c, and d must be forced to $+0.3$ or -0.3 before encoding!
- When checking inverse properties, you will discover that the inexactitude of floating-point arithmetic means that your code only *approximately* satisfies the inverse laws. One way to deal with this is to say that x approximately equals y when

  $$\frac{(x-y)^2}{x^2+y^2}$$

  is small. However, this test can fail as well if x and y are both zero. In that case they are definitely approximately equal, but you have to check for it.

3.4 Testing

Plan to spend most of your time on this assignment creating and running unit tests. Once your unit tests all run, doing whole pictures should be pretty easy—the most likely mistakes are things like confusing width and height, and these can be observed pretty easily.

We will run unit tests against your code. A significant fraction of your grade for functionality will be based on the results of those unit tests.

3.5 A useful main function

For dealing with command-line options, consider such code as the following:

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "assert.h"
#include "compress40.h"

static void (*compress_or_decompress)(FILE *input) = compress40;

int main(int argc, char *argv[])
{
        int i;

        for (i = 1; i < argc; i++) {
                if (strcmp(argv[i], "-c") == 0) {
                        compress_or_decompress = compress40;
                } else if (strcmp(argv[i], "-d") == 0) {
                        compress_or_decompress = decompress40;
                } else if (*argv[i] == '-') {
                        fprintf(stderr, "%s: unknown option '%s'\n",
                                argv[0], argv[i]);
                        exit(1);
                } else if (argc - i > 2) {
                        fprintf(stderr, "Usage: %s -d [filename]\n"
                                "       %s -c [filename]\n",
                                argv[0], argv[0]);
                        exit(1);
                } else {
                        break;
                }
        }
        assert(argc - i <= 1);    /* at most one file on command line */
        if (i < argc) {
                FILE *fp = fopen(argv[i], "r");
                assert(fp != NULL);
                compress_or_decompress(fp);
                fclose(fp);
        } else {
                compress_or_decompress(stdin);
        }
}
```

You can get this function using `git clone /comp/40/git/arith`.

4 Common mistakes

The mistakes people typically make on this assignment are covered above. To enumerate all the common mistakes would be to repeat much of the handout. Here are a half dozen carefully chosen ones:

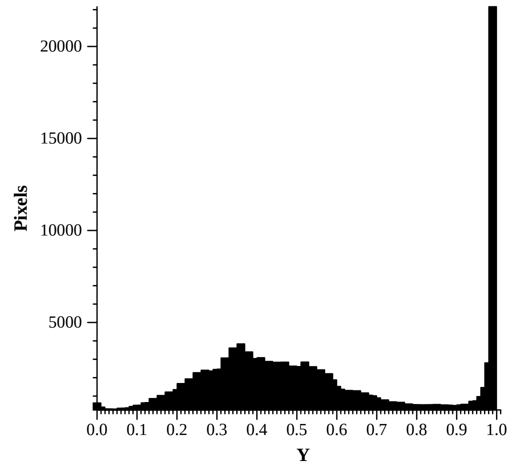
- Testing `Bitpack` without using all 64 bits
- Having different modules know the same secret
- Having one module know wildly unrelated secrets
- Forgetting what you know (or can look up) about the PPM specification
- Writing a codeword in some format other than big-endian binary format
- Getting the compressed image format right in concept but not right in practice

5 Appendix A: Why Quantization Works

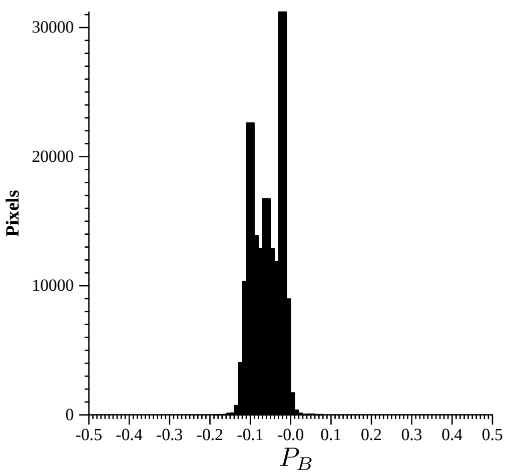
NOTE: the following information is absolutely NOT necessary for your completion of the project, but may be worth reading for people interested in data compression.

Here you can see a picture and three histograms, which tell how often each value of Y, P_B, and P_R occurs in the picture. The hump in Y values around 0.3 to 0.5 shows that the picture is somewhat dark; the big spike near 1.0 is the bright overcast sky in the background. The tremendous range in the available Y values shows that Y carries lots of information, so we are justified in using lots of bits (24 out of 32) to code it. As is typical, the chroma signals are mostly near zero; the blue chroma P_B is somewhat negative because of the lack of blue tones in the photograph; the red chroma P_R is somewhat positive, probably because of the red bricks. The narrow range of the actual chroma values shows that color differences carry little information, so we are justified in using only 8 of 32 bits for color.

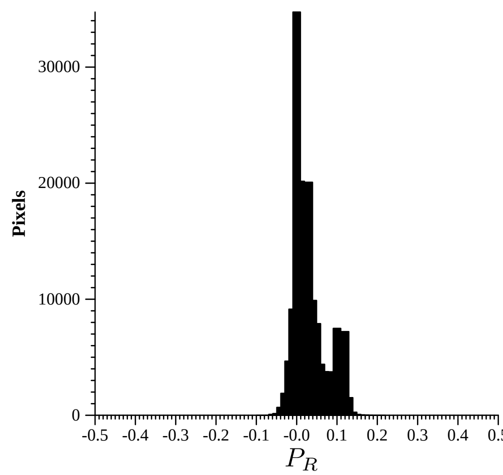
Image `tufts1.jpg`



Y histogram for `tufts1.jpg`

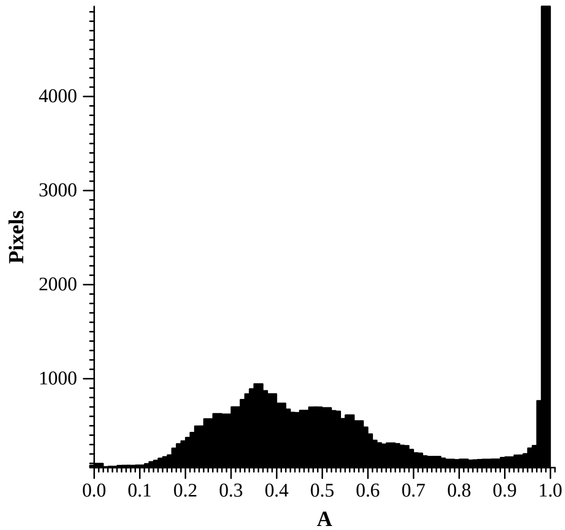


P_B histogram for `tufts1.jpg`

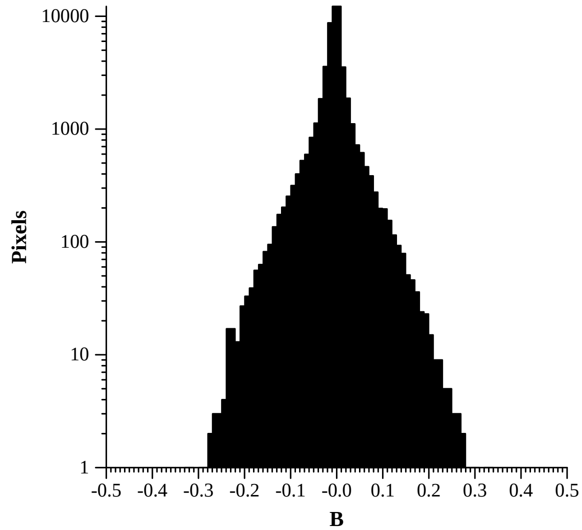


P_R histogram for `tufts1.jpg`

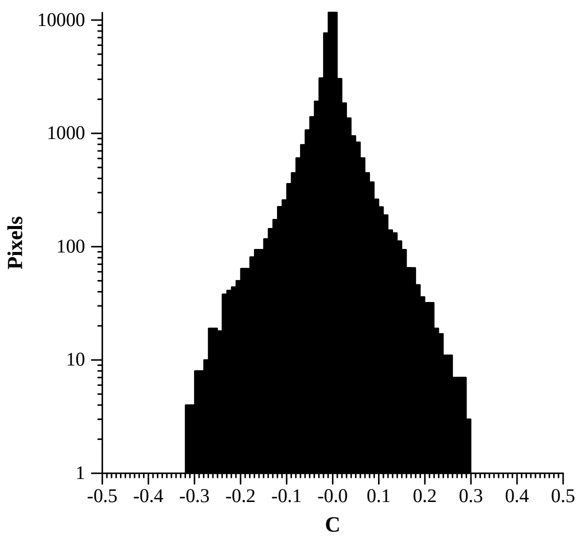
Here's a diagram that shows the results of the discrete cosine transform on Y:



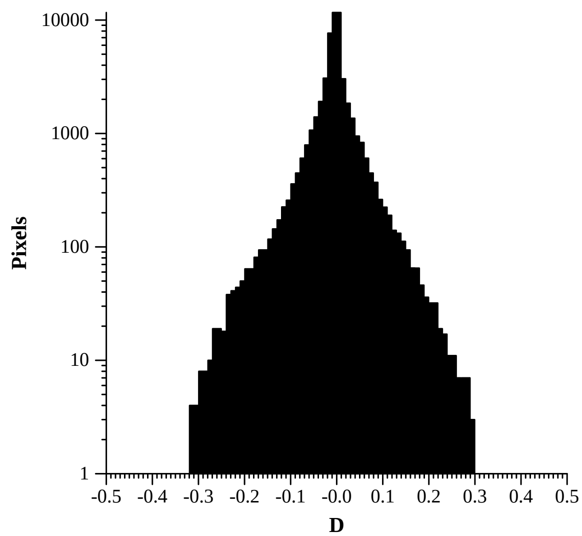
a histogram for `tufts1.jpg`



b histogram for `tufts1.jpg`



c histogram for `tufts1.jpg`



d histogram for `tufts1.jpg`

You see why it could be useful to quantize b, c, and d in a narrow range around 0.