CSC 411 Assignment: Integer and Logical Operations

Designs due Friday, November 3rd, at 11:59 PM. Full assignment due Friday, November 10th, at 11:59PM.

Purpose, overview, and instructions

The primary purpose of this assignment is to give you practice unpacking and repacking representations that put multiple integers (both signed and unsigned) into a word. You'll also learn to analyze two's-complement arithmetic so that you know how many bits are needed to store the results of calculations, and when not enough bits are available, you'll know how to adapt your code. You'll be exposed to some of the horrors of floating-point arithmetic.

As a minor side benefit, you'll also learn a little bit about how broadcast color TV works (or worked, now that it is obsolete) as well as the basic principle behind JPEG image compression (which is far from obsolete).

This assignment is inspired by Norman Ramsey at Tufts University.

Here's what you'll do:

- Write *and test* linear bijections: a discrete cosine transform and a bijection between RGB and component video ($Y/P_R/P_B$) color spaces.
- 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.
- Write a lossy image compressor that takes a PPM image and compresses the image by transforming color spaces and discarding information that is not easily seen by the human eye.

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.

Once your design document is complete and submitted, begin by running

```
mkdir arith
cd arith
git clone https://github.com/ndaniels/bitpack
cargo new --bin rpeg
```

You will now have the appropriate directory structure for this project: you start with the stub of `bitpack` that I created for you, and you create a new binary crate `rpeg` for image compression.

Dependencies

You will need the crates `csc411_arith` and `csc411_image`. Appropriate `use` statements will look like:

```rust
use csc411_image;
use csc411_arith;
```

Note that these might not belong in the same file.

Here we provide a reasonable starting point for `main` which you may use:

Contents of `main.rs`

```rust
use std::env;
use rpeg::codec::{compress, decompress};

fn main() {
    let args: Vec<String> = env::args().collect();
    let argnum = args.len();
    assert!(argnum == 2 || argnum == 3);
    let filename = args.iter().nth(2).unwrap();
    match args[1].as_str() {
        "-c" => compress(Some(filename)),
        "-d" => decompress(Some(filename)),
        _ => {
            eprintln!("Usage: rpeg -d [filename]\nrpeg -c [filename]")
        }
    }
}
```

You will also need a `lib.rs`:

```rust
pub mod codec;
```

And finally, a `codec.rs`:

```rust
use csc411_image;
pub fn compress(filename: Option<&str>) {
    todo!();
```

```rust
}

pub fn decompress(filename: Option<&str>) {
    todo!();
}
```

You'll also need to ***copy*** your `array2` directory from the `iii` project. If you do not have a working `array2`, please contact the course staff. If you do this, be sure to acknowledge the receipt of `array2` in your readme.

You *might* need to modify your `array2` or ours in able to provide the pixels to `csc411_image::Image::write()` in row-major order.

The `dependencies` section of your `Cargo.toml` should look like this:

```toml
[dependencies]
csc411_image = "0.5.0"
csc411_arith = "0.1.0"
array2 = { path = "../array2" }
bitpack = { path = "../bitpack" }
```

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

In CSC 411, 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 `csc411_image` 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 `.h` file, and it normally is implemented by a single `.c` file. In Rust, on the other hand, all code lives in `.rs` files, but functions and structs are made public with the `pub` keyword.

We will evaluate your work according to how well you organize your solution into separate files.

What we provide for you

All files we provide will be on `crates.io`, or else you will acquire them using `git` or create them with `cargo new` The files include:

- `bitpack.rs` as a stub for the bitpack crate. It has all ***public*** function signatures, but you must implement the function bodies. You may very well wish to write additional non-public functions as helpers within this crate; you should not need to write any additional public functions.
- The file `codec.rs`, whose initial contents are above. You will need to implement the `compress()` and `decompress()` functions.
- `lib.rs` within the `rpeg` crate, whose contents are given above.
- `Cargo.toml` within the `rpeg` crate, whose contents are given above.
- The crate `csc411_arith`, available via crates.io (so you need only put it as a dependency in your `Cargo.toml`).
- The crate `csc411_image`, available via crates.io (so you need only put it as a dependency in your `Cargo.toml`).
- The crate `csc411_rpegio`, available via crates.io (so you need only put it as a dependency in your `Cargo.toml`). Documentation is here

What we expect from you

Your **design document**, called `DESIGN`, `DESIGN.pdf`, `DESIGN.txt`, or `DESIGN.md` (lowercase alternatives are also fine) to be submitted via Gradescope, should *describe your overall design*, and it should also include *separate descriptions of each component*. Your sections on the `bitpack` crate and on parts of the `rpeg` program can be relatively short, since in these cases we have done some of the design work for you. But you should have a *detailed plan for* *testing* each of these components.

Also, your `rpeg` program should not be implemented as a single component. Your design document should not only explain how `rpeg` is to be implemented by a combination of components, but should also present a ***separate design description of each component.***

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

- Separate documentation of the architecture of each major component as well as the overall architecture.
- 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 Rust code yet, just sketch the types' definitions and functions' specifications in concise, informal English.
- *You must have a plan for testing each individual component in isolation.* The testing can be simple, but if you don't do it, your compressor won't work. Your best bet is to write down universal laws and write code to be sure that they hold on a variety of inputs.

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 via Gradescope, should include:

```
arith/
    README.md {or .txt or .pdf}
    array2/
        Cargo.toml
        src/
            array2.rs
            lib.rs
    bitpack/
        Cargo.toml
        src/
            bitpack.rs
            lib.rs
    rpeg/
        Cargo.toml
        src/
            codec.rs
            lib.rs
            main.rs
            -- additional .rs files based on your design --
```

A reasonable `zip` command is:

```
zip -r arith.zip arith -x "arith/*/Cargo.lock" "arith/*/target/**" "arith/*/.git/**"
```

- The `README` file should:
  - Identify you and your programming partner by name
  - Acknowledge help you may have received from or collaborative work you may have undertaken with others
  - Identify what has been correctly implemented and what has not
  - Explain the architecture of your solution
  - Say approximately how many hours you have spent *analyzing the problems posed in the assignment*
  - Say 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.

Problems

Part A: Lossy image compression

Your goal is to convert between full-color portable pixmap images and compressed binary image files. Write a program `rpeg` 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:

```
Usage: rpeg -d [filename]
       rpeg -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.

We 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 will appear later in this assignment handout, 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.
2. If necessary, trim the last row, column, or both row and column of the image so that the width and height of your image are even numbers.
3. Change to a floating-point representation (think about the ppm format and its **`denominator`**)
4. transform each pixel from RGB color space into component video color space ($Y/P_B/P_R$)
5. Pack each 2-by-2 block into a 32-bit word as follows:

- For the P_B and P_R (chroma) elements of the pixels, take the average value of the four pixels in the block. We'll call these average values $\overline{P_B}$ and $\overline{P_R}$.
- Convert the $\overline{P_B}$ and $\overline{P_R}$ elements to four-bit values using the function we provide you:

```
csc411_arith::index_of_chroma(x: f32) -> usize
```

- This function takes a chroma value between -0.5 and $+0.5$ and returns a 4-bit *quantized* representation of the chroma value.
- Using a discrete cosine transform (DCT), transform the four Y (luminance/luma) values of the pixels into cosine coeffecients a, b, c, and 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.
  - 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 `csc411_arith::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` crate you will develop in Part B.
- Write a compressed binary image to standard output. The header of the compressed binary image should be written by

```
println!("Compressed image format 2\n{} {}", width, height);
```

- This header should be followed by 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.
- The Rust standard library functions `from_be_bytes` and `to_be_bytes` will be useful here:
- https://doc.rust-lang.org/std/primitive.u32.html#method.from_be_bytes You can write a single byte using `print!()`.
  - 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:

- Read the header of the compressed file to determine the width and height.
- Sadly, Rust lacks an equivalent to C's `fscanf`, but you may use the `scan_fmt` crate. The trick is to use exactly the same string as is printed in the header, as in this trivial example:

```
#[macro_use] use scan_fmt;

fn main() {
    let (w, h) = scan_fmt!("Compressed image format 2\n{} {}\n", u32, u32);
    println!("{} {}", w.unwrap(), h.unwrap());
}
```

You will also need to include it in your dependencies in `Cargo.toml`:

```
[dependencies]
scan_fmt = "^0"
```

- Allocate a 2D array of pixels of the given width and height.
- Use the new `csc411_image::write()` function to write out an image to `stdout`.

Changes to your array2 crate

- You're going to need to be able to deal with a ***mutable*** `array2` of `Rgb` pixels.
- You probably want a `get_mut()` method on your array2. You might alternatively want an `iter_row_major_mut()` method. It's up to you.
  - How? All that really changes is the type signatures. You tell the compiler the value returned should be mutable.
  - The resulting machine code is the same!
- We recommend using 255 as the denominator.
- Read the 32-bit code words in sequence, remembering that each word is stored in big-endian order, with the most significant byte first.
- For each code word, unpack the values a, b, c, d, and the coded $\overline{P_B}$ and $\overline{P_R}$ into local variables.
- Convert the four-bit chroma codes to $\overline{P_B}$ and $\overline{P_R}$ using the function we provide you:

```
csc411_arith::chroma_of_index(n: usize) -> f32;
```

- 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.
- 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 255, and create a row-major array of 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.)
- Once you have put all the pixels into your pixmap, you can write the uncompressed image to standard output by calling `csc411_image::write(None)`

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''Eclairage*. 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. 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 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:

$$\begin{pmatrix} Y \\ P_B \\ P_R \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}.$$

If your matrix arithmetic is rusty, the equation above is equivalent to

```
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;
```

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

Suppose we have a 2-by-2 block of pixels with brightnesses Y_1 through Y_4. From the point of view of linear algebra, these Y_i values form a vector, but to exploit our geometric intuition about 2-by-2 blocks, we write them as a matrix:

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

It is easy to see that we can compute this matrix as the sum of four standard matrices, each of which is multiplied by the brightness of a single pixel:

$$\begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} = Y_1 \cdot \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + Y_2 \cdot \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} + Y_3 \cdot \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} + Y_4 \cdot \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

This representation does not take advantage of the way the human eye works; the eye is better at seeing gradual shadings of color than at seeing fine spatial detail. We can therefore write the same pixels using a different *orthogonal basis*:[1]

The basis comes from taking cosines at discrete points; in fact, the four new matrices we use come from computing $\cos 0$, $\cos \pi y$, $\cos \pi x$, and $(\cos \pi x)(\cos \pi y)$. The values used for x and y are the coordinates of the four pixels, so x and y take on only the integer values 0 and 1, and the equation is

$$\begin{pmatrix} Y_1 & Y_2 \\ Y_3 & Y_4 \end{pmatrix} = a \cdot \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} + b \cdot \begin{pmatrix} -1 & -1 \\ 1 & 1 \end{pmatrix} + c \cdot \begin{pmatrix} -1 & 1 \\ -1 & 1 \end{pmatrix} + d \cdot \begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}.$$

[1] The words "orthogonal basis" will mean something to you only if you have studied linear algebra.

This is the famous discrete cosine transform (DCT), so called because we compute cosines at discrete points.

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 coefficent d represents the degree to which the pixels on one diagonal are brighter than the pixels on the other diagnoal.

The usual trick in image compression is to throw away coefficients with high spatial frequencies. With a 2-by-2 block this is a bit hard. The a coefficient has a spatial frequency of 0, whereas b, c, and d all have frequencies of 1. If we keep only a (which is what we're doing to the chroma), then we're really just averaging four pixels together, blurring the image. If we keep all four, we don't save any information; we might as well keep the original Y_1 through Y_4.

There are two clear ways forward:

- In real images, a has a wide range, but b, c, and d are all quite small. The right thing to do would be to use 9 bits to represent a and to pack b, c, and d into 5 bits apiece, using a nonlinear quantization function. **But** we're already doing nonlinear quantization with the chroma (see below), and I don't want you to have to do a lot of tedious re-implementation of the same idea.
- We'll go ahead and code a as an unsigned, 9-bit scaled integer, and for b, c, and d, we'll use a very simple nonlinear coding:
  - 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.

Here are the equations giving the transformation to and from cosine space. To transform from cosine space into pixels, we just read off the sum from the previous page; to get from cosine space back to pixel space, we perform the inverse transformation:

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

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

The process of converting from an arbitrary floating-point number to one of a small set of integers is known as *quantization*. In reducing average chroma to just 4 bits, we quantize *in the extreme*. The quantization works by considering the floating-point chroma value and finding the closest value in the set

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

As seen below, most chroma values are small, so we 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[2] than the linear quantization $n = \texttt{floor}(15 * (P_B + 0.5))$. 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.

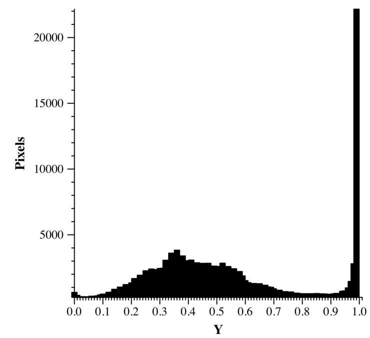
Quantization is implemented by sorting the values above into a 16-element array. To quantize a floating-point chroma value, we find the element of the array that most closely approximates the chroma, and I return that element's index.

Why it works

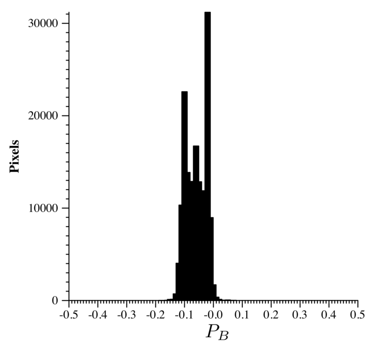
Here you can see a picture and three histograms, which tell how often each value of Y, P_B, and P_R occurs in the picture. The hump in Y values around 0.3 to 0.5 shows that the picture is somewhat dark; the big spike near 1.0 is the bright overcast sky in the background. The tremendous range in the available 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

[2] The quantization error is the difference between P_B and `chroma_of_index(index_of_chroma(`P_B`)`.

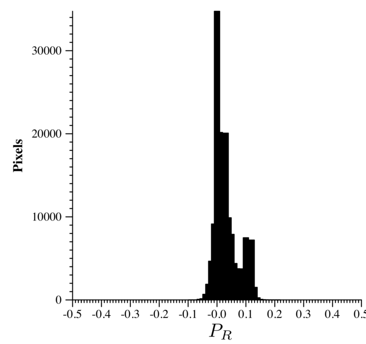
differences carry little information, so we are justified in using only 8 of 32 bits for color.



Y histogram for `tufts1.jpg`



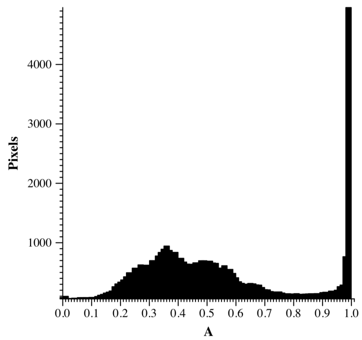
P_B histogram for `tufts1.jpg`



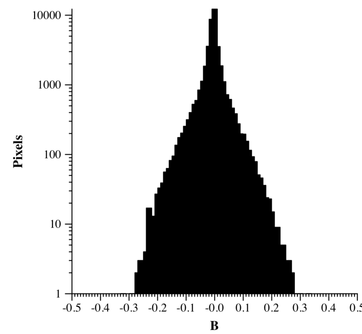
P_R histogram for `tufts1.jpg`

(Why Tufts? I previously taught this course at Tufts, and had this nice photo and set of histograms handy).

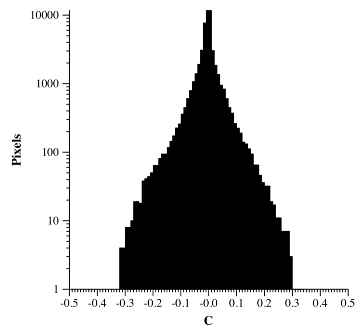
Here's a diagram that shows the results of the discrete cosine transform on Y:



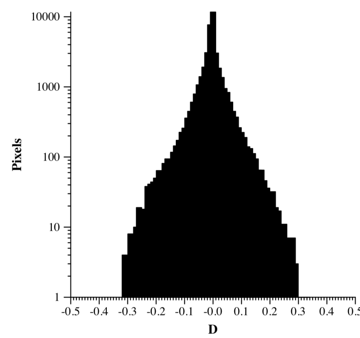
a histogram for `tufts1.jpg`



b histogram for `tufts1.jpg`



c histogram for `tufts1.jpg`



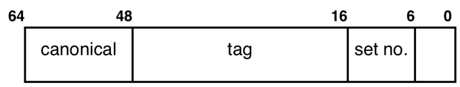
d histogram for `tufts1.jpg`

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

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. The binary representation of machine instructions uses such packings heavily, and the instruction-decode unit unpacks a sequence of bytes into fields that determine opcodes and operands. 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. For example, if a machine has a 2-way set-associative 128KB Level-1 cache with 64-byte cache lines, then 6 bits are required to address a byte within a line, and the line offset is a field 6 bits wide with its least significant bit at bit 0. There are $128K \div 64 = 1024 = 2^{10}$ sets, so the set number is a field 10 bits wide with its least significant bit at bit 6. Because bits 48–63 are canonical, the caching tag is 32 bits wide with its least significant bit at bit 16. All these fields are interpreted as unsigned integers.



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[3].

In this part of the assignment you will define bit-manipulation primitives as part of the `bitpack` crate:

```rust
use std::convert::TryInto;


/// Returns true iff the signed value `n` fits into `width` signed bits.
///
/// # Arguments:
/// * `n`: A signed integer value
/// * `width`: the width of a bit field
pub fn fitss(n: i64, width: u64) -> bool {
}

/// Returns true iff the unsigned value `n` fits into `width` unsigned bits.
///
/// # Arguments:
/// * `n`: An usigned integer value
/// * `width`: the width of a bit field
pub fn fitsu(n: u64, width: u64) -> bool {
}

/// Retrieve a signed value from `word`, represented by `width` bits
/// beginning at least-significant bit `lsb`.
///
/// # Arguments:
/// * `word`: An unsigned word
/// * `width`: the width of a bit field
/// * `lsb`: the least-significant bit of the bit field
pub fn gets(word: u64, width: u64, lsb: u64) -> i64 {
}

/// Retrieve an unsigned value from `word`, represented by `width` bits
/// beginning at least-significant bit `lsb`.
///
/// # Arguments:
/// * `word`: An unsigned word
```

[3] 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.

```
/// * `width`: the width of a bit field
/// * `lsb`: the least-significant bit of the bit field
pub fn getu(word: u64, width: u64, lsb: u64) -> u64 {
}

/// Return a modified version of the unsigned `word`,
/// which has been updated so that the `width` bits beginning at
/// least-significant bit `lsb` now contain the unsigned `value`.
/// Returns an `Option` which will be None iff the value does not fit
/// in `width` unsigned bits.
///
/// # Arguments:
/// * `word`: An unsigned word
/// * `width`: the width of a bit field
/// * `lsb`: the least-significant bit of the bit field
/// * `value`: the unsigned value to place into that bit field
pub fn newu(word: u64, width: u64, lsb: u64, value: u64) -> Option<u64> {
}

/// Return a modified version of the unsigned `word`,
/// which has been updated so that the `width` bits beginning at
/// least-significant bit `lsb` now contain the signed `value`.
/// Returns an `Option` which will be None iff the value does not fit
/// in `width` signed bits.
///
/// # Arguments:
/// * `word`: An unsigned word
/// * `width`: the width of a bit field
/// * `lsb`: the least-significant bit of the bit field
/// * `value`: the signed value to place into that bit field
pub fn news(word: u64, width: u64, lsb: u64, value: i64) -> Option<u64> {
}
```

You are to *implement this interface* in the file `bitpack.rs`.

Width-test functions

Your interface must be able to 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:

```
pub fn fitss(n: i64, width: u64) -> bool;
pub fn fitsu(n: u64, width: u64) -> bool;
```

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.

```
pub fn gets(word: u64, width: u64, lsb: u64) -> i64;
pub fn getu(word: u64, width: u64, lsb: u64) -> u64;
```

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
```

To get the cache set number from a 64-bit address, you might use `bitpack::getu(address, 10, 6)`. It should be a **checked run-time error** to call `bitpack::getu` or `bitpack::gets` with a width w that does not satisfy $0 \le w \le 64$. 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 $w + \texttt{lsb} \le 64$.

Some machine designs, such as the late, unlamented **HP PA-RISC**, provided these operations using one machine instruction apiece. The Intel/AMD X64 architecture does not.

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:

```
pub fn newu(word: u64, width: u64, lsb: u64, value: u64) -> Option<u64>;
pub fn news(word: u64, width: u64, lsb: u64, value: i64) -> Option<u64>;
```

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 $0 \le w \le 64$. 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 $w + \texttt{lsb} \le 64$.

Further error handling If `bitpack::news` is given a `value` that does not fit in `width` signed bits, it must return `None`

Similarly, if `bitpack::newu` is given a `value` that does not fit in `width` unsigned bits, it must also return `None`.

If 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).unwrap(), 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[4]. You can also unit-test the `fits` functions 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.

Documentation

- For this assignment, **you are expected to write interface documentation that is compatible with `cargo doc`.**
- We want to see, for public functions, documentation of the functions' contracts. Specifically, what arguments are expected, what is returned, and how they relate. We don't need to see the types; Rust already provides that. We want the ***semantics***. Please also mention any failure modes or errors (e.g. if a function returns an `Option` type, under what circumstances does that `Option` evaluate to `None`?) as well as anything else the user of your code would need to know.
- There is documentation for how to write documentation here: https://doc.rust-lang.org/cargo/commands/cargo-doc.html
- But if you use the above example of `bitpack` as a model, you should do fine.

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

Supplementary material

Traps and pitfalls

The warnings always come after the spell

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, copied by Rust, 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.* We recommend that you define functions to do your shifting (you may want to use the `#[inline]` directive; this hints to the compiler that the functions should be *inlined* during optimization, so there is no calling cost). Your functions, unlike the hardware, should do something sensible when asked to shift by 64. (We expect you to figure out what might be sensible.)
- In Rust, you can create a literal value of a specific size with something like `0_u64`, `-1_i32`, and so on.
- You may be tempted to implement parts of the `bitpack` crate 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` (`f64` in Rust) contains only 64 bits of precision, and because some bits are used for sign and exponent, an `f64` cannot represent all 64-bit integers. An `f32` (`float` in C), 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.

- 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. We know of three ways to solve this problem:
  - A gifted arithmetist might find a way to do all the computations in rational numbers using only integer arithmetic, and might then find exactly the places in the code where quantization error can violate an invariant. In those places, values could be checked and adjusted.

- An engineer in a hurry might just do the floating-point arithmetic and then use an inline function to force each value into the interval where it belongs.
- An ambitious engineer writing a codec for a blu-ray player would probably take advantage of special machine instructions that do *saturating arithmetic*. Such machine instructions do standard addition and multiplication but then adjust the results if they would go outside the bounds of the representation. (When ordinary integer arithmetic overflows, the result "wraps around." You're doing arithmetic on integers modulo 2^{64} (64 is the word size).

We expect you to behave like an engineer in a hurry.

Detailed advice for `bitpack`

Here are some ideas to keep in mind when you approach the `bitpack` crate:

- The hardware provides three simple, powerful shift operations. But the C and Rust programming languages, which are 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 these problems by defining three inline 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 Rust code 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 (`|`).

Other helpful advice

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

- Implement `bitpack` *last*.
- 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.
  - Dealing with this sort of situation is common in scientific computing

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.

A useful main function

We provide a `main.rs` which provides a `main()` which handles command-line arguments; it is reproduced verbatim above. You should start with it when you have sorted out your design.

Common mistakes

The mistakes people typically make on this assigment 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