CSC 411 Assignment: Understanding and Using Interfaces

Design for part B only due Friday, January 26th at 11:59PM (parts A and C do not require a written design). Full assignment (parts A, B, and C) due Friday, February 2nd at 11:59PM.

Please read the entire assignment before starting work.

Purpose

This assignment has four goals:

1. To help you make the transition to programming in Rust, which we assume is a new language for you (we know some of you have some Rust experience already, but most do not).
2. To give you practice in identifying interfaces that can help you solve problems
3. To start you thinking about the interface as a unit of design
4. To introduce you to multiple representations of numbers

Pair programming

As will be discussed in class, pair programming is an effective way to more than double your programming productivity. Pair programming is strongly recommended for all assignments. Pair programming will help most with your design. It is up to you to find a partner (but approach the course staff if you have trouble).

Preliminaries

- You'll need a working Rust environment, which could be on your own computer, or on the homework server `homework.cs.uri.edu`. If you got through the pnmdata lab, then you're ready to go!
- The design document is due Friday night at 11:59PM
- You will submit the design document and your final code via Gradescope.

Part A: Brightness of a grayscale image

Please write a Rust program `brightness`, which should print the average brightness of a grayscale image. Every pixel in a grayscale image has a brightness between 0 and 1, where 0 is black and 1 is as bright as possible. Print the average brightness using decimal notation with exactly one digit before the decimal point and three digits after the decimal point. Print *only* the brightness.

The program takes at most one argument:

- If an argument is given, it should be the name of a portable graymap file (in `pgm` format). You will want to use the `csc411_image` crate just like we did in the `pnmdata` lab.
- If no argument is given, `brightness` reads from standard input, which should contain a portable graymap.
- If more than one argument is given, `brightness` halts with an error message (you can achieve this by using `assert()`).
- If a portable graymap is promised but not delivered, `brightness` should halt with some sort of error message on `stderr`. (Any message will do, but do print a message on standard error, and don't produce anything on standard output—especially not a senseless answer.)

For this part of the assignment, you are allowed (and expected) to use the `csc411_image` crate.

This means that you'll need to edit your `Cargo.toml` file to indicate the dependency on that crate:

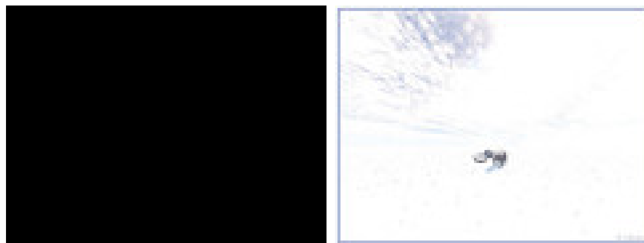
```
[dependencies]
csc411_image = "0.5"
```

Or run `cargo add csc411_image@0.5` from the command line.

Examples:

- Here are two photos:

Black cat in a coal cellar Polar bear in a snowstorm



If `cellar.pgm` is a picture of a black cat in a coal cellar at midnight, and if `bear.jpg` is a picture of a polar bear in an snowstorm, then output should look something like this:

```
homework{ndaniels}: target/release/brightness cellar.pgm
0.000
homework{ndaniels}: djpeg -grayscale bear.jpg | target/release/brightness
0.972
```

The first example takes its input from a file named on the command line, and the second example takes its input from standard input, as part of a Unix pipeline.

My solution to this problem in C took fewer than 35 lines.

My solution in Rust takes 7 lines. Welcome to a language for the 2020s.

Help with image files

You may find the documentation for the `csc411_image` crate at:

https://docs.rs/csc411_image/latest/csc411_image/

On a macOS or Linux environment (including WSL on Windows), with the `jpeg` toolkit installed, you can get images to play with by using one or more of the following programs:

- `djpeg` (use the `-grayscale` option)
- `pngtopnm`
- `pstopnm`
- `ppmtopgm`

Tip: if you are on macOS, install Homebrew from https://brew.sh (it also works on Linux, but you can likely use a native package manager like apt). Then `brew install jpeg`.

Problem analysis and advice

The main issues here are:

- You will have to do something sensible if somebody hands you input that is not a portable graymap. A panic *is* sensible.
- You can open a graymap like this:
  - `let img = GrayImage::read(input.as_deref()).unwrap();`
  - Where does `input` come from? Spend some time trying to figure that out.
- You will want to start off with `cargo new --bin brightness` to create a new Rust project.
- You'll have to deal with **two representations** of real numbers between 0 and 1.
  - A pgm file specifies a global denominator (it might be 255, or some other value – it's up to whoever creates the image) and then each pixel is represented as an integer between 0 and that denominator.
- Given an image, you can get the denominator pretty easily:
  - `let denom = img.denominator as u32;`

- And given that same image, you can iterate over the pixels:
  - `for pixel in img.pixels { ... }`

Part B: Finding fingerprint groups

In this problem you are to write a program `fgroups` (short for "fingerprint groups"), which when given a set of names with fingerprints will identify groups of names that share a fingerprint. The real object of the exercise is to familiarize you with some parts of the standard library.

Your input is always on standard input. Input is a sequence of lines where each line has the following format:

- The line begins with one or more non-whitespace characters. This sequence of characters is the fingerprint.
- The fingerprint is followed by one or more whitespace characters, not including newline.
- The name begins with the next non-whitespace character and continues through the next newline. A name may contain whitespace, but a name never contains a newline.
  - As an example of this, if a line of input is `fingerprint this is my name`, then the resulting *fingerprint* would be `fingerprint` and the resulting *name* would be `this is my name`.

For this part of the assignment, you may not use any external crates, but can use anything in the standard library (`std::`)

Finally, you may also assume that each name appears at most once.

You may assume that a fingerprint is at most 512 characters long (2048 bits represented in hexadecimal notation), but there is no a priori upper bound on the length of a name. So, it's easiest to allow fingerprints of arbitrary length.

What to do with good input

By the nature of the input, every fingerprint you read will be associated with at least one name.

- If a fingerprint is associated with exactly one name, ignore the fingerprint (and the name).
- If a fingerprint is associated with two or more names, those names constitute a **fingerprint group**. A fingerprint group always has at least two members.

Your program should print all the fingerprint groups in the following format:

- If there are no fingerprint groups, print nothing.
- If there is exactly one fingerprint group, print it.

- If there are multiple fingerprint groups, print them separated by newlines.

Groups may be printed in any order.

To print a fingerprint group, print each name in the group. Put a newline after each name; the `println!()` macro will help you here.

Names within the group may be printed in any order.

For example, if the input is

```
A    Ron
A    Poppy
B    Bill
A    Dubya
B    Barry
A    Orange
B    Joe
```

Then one possible output is

```
Ron
Dubya
Poppy
Orange

Bill
Barry
Joe
```

(This example output contains exactly one blank line.)

My solution to this problem in C took about 150 lines of C code. About 25 lines deal with input; about 25 lines are devoted to printing output; and about 50 lines are memory management and I/O. The actual algorithm is under 20 lines. The rest is comments, whitespace, `#include` directives, and the like.

My solution in Rust takes less than 35 lines.

What to do with bad input

- If you get an input line that is badly formed, write an error message to stderr, discard the badly formed line, and continue.
- If you get a fingerprint of more than 512 characters, you may choose to write an error message to stderr, discard the line, and continue—or you may handle the oversize fingerprint correctly. In other words, your program doesn't need to handle fingerprints larger than 512 characters, but you won't be penalized if it does. *In Rust, the simplest thing is to just handle fingerprints of any length, without truncating.*
- Your program should handle any name that can appear in the filesystem; if you get a name that is longer than you can handle, write a suitable

message to stderr, truncate the name, and use the truncated name with the given fingerprint. *In Rust, you should be able to handle anything that fits within memory, so don't worry about truncating anything.*
- If a name appears more than once, your program may silently give wrong answers, but it must not print an error, crash, or commit memory errors.

Performance target

Your `fgroups` program should perform well on inputs of up to **several hundred thousand lines.** On any reasonable laptop, it should be capable of processing at least a half a million lines per minute.

Problem analysis and advice

This problem boils down to simple string processing and standard data structures.

You may find the `splitn` string method to be quite useful: https://doc.rust-lang.org/std/str/struct.SplitN.html

Next, think about what existing data structures you can use to make this task easier. What did you learn about in CSC 211? You don't have to reinvent the wheel here, nor should you. Rust's standard library has `std::vec`, but there's a lot more in `std::collections`: https://doc.rust-lang.org/std/collections/index.html

Repeat: **the data structures are already built for you;** your job is to figure out which ones will be useful.

Getting started:

- Write the design document described in the next section.
- Create *another* new project, alongside your brightness project, with `cargo new --bin fgroups`. Don't do this in your `brightness` directory; do it one level up so that `fgroups` lives alongside `brightness`.

So ultimately you want something like this:

```
$ du -ah intro
4.0K    intro/brightness/Cargo.toml
 12K    intro/brightness/Cargo.lock
4.0K    intro/brightness/src/cursor.rs
4.0K    intro/brightness/src/main.rs
8.0K    intro/brightness/src
 33M    intro/brightness
4.0K    intro/fgroups/Cargo.toml
4.0K    intro/fgroups/Cargo.lock
4.0K    intro/fgroups/README.md
4.0K    intro/fgroups/src/main.rs
4.0K    intro/fgroups/src
224K    intro/fgroups
```

(your file sizes may vary).

Design document for fgroups

The heart of a software design lies in its representation of data. For this assignment, we are asking you to identify what **data structures you will need** to compute fingerprint groups, and **what each data structure will contain.**

For this design document, we need you to answer three questions:

1. What data structures will you use, and what concrete types will their type variables represent? Rust collection data structures are polymorphic, so you will have to explain what each type variable (those are the `T` and `V` and `S` and `K` you see in the documentation) will actually be. The answer here should be a concrete type, such as `u64` or `String`
2. Please **describe the invariant** that will hold when you are partway through reading input lines.
3. Based on these two explanations, explain briefly how you will compute the fingerprint groups once all input has been read.

Part C: Solving problems with fingerprint groups

The example above is not entirely facetious, but in this problem we ask you **list problems you could solve** with a working version of `fgroups`. Be imaginative. For one set of ideas, look up the man page for find, especially the -exec option. And it might help you to know that in the arcane jargon of computer science, a fingerprint is sometimes called a “message digest”. This information is especially useful if you also know how to use the apropos command.

Please include your answer to part C in your README file.

Organizing and submitting your solutions

By the first deadline, submit a brief design document that explains what data structures you will use to write fgroups and how you will use them. Your document should be a PDF document called `design.pdf`. Your submission will be through Gradescope.

In your final submission, please upload a zip archive that contains three items:

- A readme file (readme.txt, readme.md, readme.pdf, but not a Word doc)
- A folder called `brightness` that contains your brightness project
- A folder called `fgroups` that contains your fgroups project

To create this zip file, there are a few things to note: we want to *exclude* the rather large `target` directory from both brightness and fgroups; this is where

cargo builds its binaries, and it takes a lot of space. We also want to exclude the `.git` directory.

The following command does exactly what we want:

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

Note the quotes around the exclusion strings; the quotes are needed to prevent those wildcards (asterisks) from expanding prematurely.

The readme should:

- Identify you and your programming partner by name and
- Acknowledge help you may have received from or collaborative work you may have undertaken with classmates, programming partners, course staff, or others
- Identify what has been correctly implemented and what has not
- Tell me what kinds of problems you can solve using fgroups (part C of this assignment)
- Say approximately how many hours you have spent completing the assignment

When you get everything working, upload all files to Gradescope. If you are working on the homework server, this will likely require you to download the files from `homework` and upload them via a web browser. **This step is meant to be tedious** because you should **not** rely on Gradescope to test your code! Don't submit until you **know** it works.

Git

One nice thing about cargo is that when you run `cargo new`, it creates not only the Rust project, but initializes a git repository for you. You can use this to keep track of your progress; after you make some changes and verify that your code (for either project) compiles, then from within that project directory, type something (probably with a more useful commit message) like:

```
git commit -am "Replaced fgroups data structure with a black hole"
```

This allows you to see a log of your changes (`git log`) and even to revert back to older versions. If you have a github account, you can create a repository there and push your local repository up (please, please make sure you use ***private*** github repositories for this class!)

Avoid common mistakes

In this assignment, the most common mistakes involve producing output in a format other than what the specification calls for. You will learn to read specifications very carefully. Consider these questions:

- When a number is called for, how many digits of that number should be printed?

- Which parts of the assignment call for words to be printed, and which call for numbers?
- Does output go to standard output or go to a file?
- Does input come from standard intput or from a file? Or both?
- In the output, exactly where should you place blank lines?

Here are some other common mistakes:

- When proposing applications of fingerprint groups, it's easy to overlook that a "group" containing only one member might as well not exist.
- It's easy to think that you might have to implement several data structures. All the data structures you need are already implemented in the Rust standard library.
- It's possible to believe that you have to write code that reads images in the PNM formats. This work is already done for you; just call the functions in the `image` interface.