CSC 372, Spring 2018
Assignment 5
Due: Friday, March 16, 2018 at 23:59:59

Introduction

You should be able to complete this assignment using only the elements of Ruby presented on slides 1-154 and/or mentioned in this write-up. If you find yourself thinking that need to use other elements of the language, it probably indicates you don't realize the capabilities of what's been covered.

Restrictions

You'll see that two of the programs, xfield.rb and flash.rb, process command line arguments. On those problems you are not permitted to use any library code that parses/processes command line arguments, such as OptionParser. That's for two reasons: (1) the argument processing needed is a good exercise in plain old programming with Ruby, and (2) with things like OptionParser, the "cure" can be worse than the disease.

seqwords.rb, longest.rb, and jlongest.java have some detailed restrictions but note that the latter two are zero-point problems.

Ruby's String and Array classes

Ruby's String and Array classes are rich. (Maybe too rich!) Once you've read the assignment, browse the documentation for them to get a feel for what they offer. Slide 9 shows the pattern used by URLs for Ruby class documentation.

We'll study inheritance in Ruby later but like Java, Ruby classes inherit methods from their superclasses. Additionally, Ruby classes draw methods from "included" modules. When looking at http://ruby-doc.org/core-2.2.4/Array.html note the boxes for Parent, Methods, and Included Modules in the left sidebar. Array's parent, Object, isn't very interesting but the Enumerable module, which Array includes, has many useful methods.

Use Ruby 2.2.4!

Ruby 2.2.4 is to be used for all Ruby assignments. Use rvm, as shown on Ruby slides 13-14, to select version 2.2.4 on lectura.

The Usual Stuff

Make an a5 symlink that references /cs/www/classes/cs372/spring18/a5. Test using a5/tester (or a5/t).

Output from command-line examples is followed by a blank line

Many of the programming problems on the Haskell assignments were functions that you tested inside ghci. All of the Ruby problems on this assignment require you to create programs that can be run from the command line, like avg.hs on assignment 4.

A strong convention in the UNIX world is that programs do not output a trailing blank line. Observe these interactions, and the absence of blank lines:

```
$ date
Sat Mar  3 01:11:01 MST 2018
$ date | wc
  1       6      29
$ ls -ld .
drwx------ 7 whm whm 85 Mar  2 23:01 .
$ (my prompt—Bash is waiting for the next command)
```
However, for ease of reading in this write-up, output from command-line examples is followed by a blank line. Instead of interaction like the above, you'll see blank lines added, like this:

```
$ date
Sat Mar  3 01:11:01 MST 2018

$ date | wc
   1       6      29

$ ls -ld .
  drwx------ 7 whm whm 85 Mar  2 23:01 .

$
```

**Problem 1. (5 points) seqwords.rb**

For this problem you are to write a Ruby program that reads a series of words from standard input, one per line, and then prints lines with the words sequenced according to a series of specifications, also one per line and read from standard input.

**Don't overlook the restriction mentioned below.**

Here is an example with four words and three sequencing specifications, which are simply integers, one per line.

```
$ cat a5/sw.1
  one
two
  three
four
.
  1
  2
  .
  3
  2
  1
  1
  2
  3
  .
  4
  1

$ ruby seqwords.rb < a5/sw.1
  one two three
  three two one one two three
  four one
```

Note that lines containing only a period (.) end the word list and also separate specifications. For output, words are separated with a single blank. Here's another example:

```
$ cat a5/sw.2
tick
  .
  .
  1
  1

$ ruby seqwords.rb < a5/sw.2
```
Make these assumptions for seqwords.rb:

- There is at least one word and at least one sequencing specification.
- Each sequencing specification has at least one number.
- All entries in the sequencing specifications are integers and in range for the list of words.
- Because periods separate specifications, the input will never end with a period.
- The tester ignores trailing blanks in the output.

Here is a key simplification: Assume that words are between 1 and 100,000 characters in length, inclusive. (Note that 100,000 is a valid Fixnum literal in Ruby.) If your code doesn't make some significant use of that assumption, you'll be solving a harder problem than I intend. Mail to 372s18 or see us during office hours if you don't see how to leverage that limit.

Restriction: The only types you may use are Fixnum, Bignum, and String along with the values nil, true, and false (i.e., the values of the single-value classes NilClass, TrueClass, and FalseClass). In particular, you may not use Arrays.

Testing note

As an aid to help ensure that your solution doesn't violate the restriction, the file a5/check-seqwords.rb has Ruby code that disables various common operations on strings and arrays. Your seqwords.rb needs to explicitly load that file. The first non-comment/blank line of your seqwords.rb should be the following line, starting in column one:

```
load "a5/check-seqwords.rb"
```

If you don't have that line exactly right, you'll see a tester failure like this:

```
Error: No line with 'load "a5/check-seqwords.rb"' found in seqwords.rb
```

With that load in place, let's imagine we've got a version of seqwords.rb that creates an array and then adds an element to it with Array's << method—definitely not permitted! Here's what the failure will look like:

```
Test: 'ulimit -t 2; ruby seqwords.rb < a5/sw.1 | sed "s/ *$/\n"': FAILED
Differences (expected/actual):
*** a5/master/tester.out/seqwords.out.01 2016-02-22 20:20:03.000000000 -0700
--- tester.out/seqwords.out.01 2018-03-03 16:20:40.046552562 -0700
***************
*** 1,3 ****
! one two three
! three two one two three
! four one
--- 1,2 ----
! (eval):1:in `<<': oops -- can't use '<<' for that type! (RuntimeError)
! from seqwords.rb:8:in `
`
```

Note that "...oops -- can't use '<<' for that type!".

Remember: Loading a5/check-seqwords.rb will catch common violations of the restrictions but it's not foolproof. The "safe harbor" is to mail your code to 372s18 and ask for a manual inspection.

Problem 2. (16 points) xfield.rb

For this problem you are going to create your own version of a Ruby tool that I use every day: xfield.xfield was inspired by the UNIX cut(1) command but its behavior differs in various ways.

Here's a man-style description of xfield. Detailed examples follow.

SYNOPSIS:

```
xfield [-dC] [-sSEPARATOR] [FIELDNUM|TEXT]...
```
xfield extracts fields of data from standard input. Field numbers are one-based and may be negative to specify counting from the right. If a field number is out of bounds, "<NONE>" is output in place of actual data. Unlike cut(1), xfield allows fields to be specified in any order and appear more than once.

Fields are delimited by one or more spaces by default but an alternate character to delimit fields can be specified with -dc. Tabs separate output fields by default but -sSEPARATOR can be used to specify an alternate separator, which must be at least one character in length. There may be multiple -d and -s specifications, in any order, but they must appear before any field number or text specifications. If there are multiple specifications for either -d or -s, the last one of each "wins".

If a textual argument (not a number) falls between two field specifications (two numbers), that text is used instead of the separator.

Input lines are assumed to end with a newline. If there are no input lines, xfield produces no output.

If no fields are specified, the message "xfield: no fields specified" is printed, and xfield calls exit(1) to terminate execution. (exit is a Kernel method.)

xfield is able to handle any number of input lines. (Hint: Don't read all the input lines into memory and then process them—they might not fit! Just process lines one at a time.)

The behavior of xfield is undefined in cases that are not specifically addressed by this write-up or exercised with the tester. That means that any non-malicious behavior is ok—run-time errors, curious results, etc., are not a surprise if the user misuses xfield.

Some detailed examples of xfield in operation follow. Here's the input file we'll work with first:

```bash
$ cat a5/xf.1
one     1   1.0
two     2   2.0
three   3   3.0
four    4   4.0
twenty  20  20.0
```

The English text and the real numbers can be extracted like this:

```bash
$ ruby xfield.rb 1 3 < a5/xf.1
one     1.0
two     2.0
three   3.0
four    4.0
twenty  20.0
```

xfield can be used to reorder fields:

```bash
$ ruby xfield.rb 3 2 1 < a5/xf.1
1.0     1       one
2.0     2       two
3.0     3       three
4.0     4       four
20.0    20      twenty
```

xfield supports negative indexing, just like Ruby arrays:

```bash
$ ruby xfield.rb -1 1 < a5/xf.1
1.0     one
2.0     two
3.0     three
4.0     four
20.0    twenty
```

```bash
$ ruby xfield.rb -1 1 2 -2 < a5/xf.1
```
If a field reference is out of bounds, the string "<NONE>" is output:

```
$ ruby xfield.rb 1 10 2 < a5/xf.1
one     <NONE>  1
two     <NONE>  2
three   <NONE>  3
four    <NONE>  4
twenty  <NONE>  20
```

The `-s` flag specifies an output separator to use instead of tab.

```
$ ruby xfield.rb -s... 1 3 1 < a5/xf.1
one...1.0...one
two...2.0...two
three...3.0...three
four...4.0...four
twenty...20.0...twenty
```

To extract login ids and real names (and room/phone) from `a5/oldpasswd`, an excerpt from an ancient `/etc/passwd`, one might use `-d` to specify that a colon is the delimiter:

```
$ ruby xfield.rb -d: 1 5 < a5/oldpasswd
wnj     Bill Joy,457E,7780
dmr     Dennis Ritchie
ken     Ken Thompson
mike    Mike Karels
carl    Carl Smith,508-21E,6258
joshua  Josh Goldstein
```

Note that the `-s` and `-d` options are single arguments—there's no space between `-s` or `-d` and the following string.

**Non-numeric arguments other than the `-s` and `-d` flags are considered to be text to be included in each output line.** If a textual argument (not a number) falls between two field specifications (two numbers), that text is used instead of the separator:

```
$ ruby xfield.rb int= 2 ", real=" 3 ", english=" 1 < a5/xf.1
int=1, real=1.0, english=one
int=2, real=2.0, english=two
int=3, real=3.0, english=three
int=4, real=4.0, english=four
int=20, real=20.0, english=twenty
```

Note the use of quotation marks to form an argument that contains blanks. **The shell strips off the quotation marks so that the resulting arguments passed to the program do not have quotes.** See the Implementation notes for `xfield` section below for more on this.

Here's that text-argument-overrides-separator rule again:

**If a textual argument (not a number) falls between two field specifications (two numbers), that text is used instead of the separator.**

Here are some more examples showing that rule in action:

```
$ cat a5/xf.2
one two three four
```
Below are some cases that bring all the elements into play.

```bash
$ cat a5/xf.3
xxxxxxxxxAxxxxxxxxxxBxC
DxExF
xG1xG2
xxxxHIxxxxJKxxxxLMNxxxxOPQRSx
```

```bash
$ ruby xfield.rb -dx -s- 1 2 3 < a5/xf.3
A-B-C
D-E-F
G1-G2-<NONE>
HI-JK-LMN
```

```bash
$ ruby xfield.rb -dx -s- -1 ... -2 -3 @ < a5/xf.3
C...B-A@
F...E-D@
G2...G1-<NONE>@
OPQRS...LMN-JK@
```

```bash
$ ruby xfield.rb -s/ -de 1 2 < a5/xf.1
on/ 1 1.0
two 2 2.0/<NONE>
thr/ 3 3.0
four 4 4.0/<NONE>
twnty 20 20.0
```

If there are no input lines, xfield produces no output:

```bash
$ ruby xfield.rb -s/ -d: 1 x 2 3 < /dev/null
```

### Implementation notes for xfield

#### `gets` vs. `STDIN.gets`

The `gets` method does a little more than simply reading lines from standard input: If command line arguments are specified, `gets` will consider those arguments to be file names. It will then try to open those files and read lines from each in turn. That's really handy in some cases but it gets in the way for xfield. To avoid this behavior, don't use "line = gets" to read lines. Instead, do this:

```ruby
line = STDIN.gets
```

That limits `gets` to the contents of standard input.

#### Delimiter-specific behavior in `String.split`

I'm astounded by it but the fact is that `split` behaves differently when the delimiter is a space:
Command line argument handling

The command line arguments specified when a Ruby program is run are made available as strings in ARGV, an array. Here is echo.rb, a Ruby program that prints the command line arguments:

```ruby
$ cat echo.rb
puts "#{ARGV.size} arguments:
for i in 0...ARGV.length do
  puts "argument #{i} is '#{ARGV[i]}'
end
```

Execution:

```ruby
$ ruby echo.rb -s -s2 -abc x y
5 arguments:
argument 0 is '-s'
argument 1 is '-s2'
argument 2 is '-abc'
argument 3 is 'x'
argument 4 is 'y'
```

Unescaped quotes and backslashes specified on the command line are processed and fully consumed by the shell; the program, echo.rb in this case, doesn't "see" them. Example:

```ruby
$ ruby echo.rb int= 2 ", real=" 3 ", english="
5 arguments:
argument 0 is 'int='
argument 1 is '2'
argument 2 is ', real='
argument 3 is '3'
argument 4 is ', english='
```

The shell does provide some mechanisms to allow quotes and backslashes to be transmitted in arguments:

```ruby
$ ruby echo.rb "" " x " \ \y\ ""
4 arguments:
argument 0 is "x"
argument 1 is "x"
argument 2 is "y"
argument 3 is ""
```

Additionally, the shell consumes I/O redirections with < and > — the program DOES NOT "see" those operators or their accompanying filename argument. Here's some evidence of that:

```ruby
$ ruby echo.rb 1 2 3 < lg.1
3 arguments:
argument 0 is '1'
argument 1 is '2'
argument 2 is '3'
```
$ ruby echo.rb 1 2 3 < lg.1 > out

$ cat out
3 arguments:
  argument 0 is '1'
  argument 1 is '2'
  argument 2 is '3'

Note that there is no trace of "<", ">", "lg.1", or "out" in the output.

The above examples were produced with a UNIX shell; but you'll see similar behavior when working on the Windows command line, although backslashes are handled differently.

**BOTTOM LINE:** Don't add code to your solution that attempts to process those shell metacharacters— that's the job of the shell, not your program!

An admonishment/HINT about argument handling

I've seen many students turn command line argument handling into an incredibly complicated mess. Don't do that! Here's an easy way to process arguments in xfield:

- Iterate over the elements in ARGV.
- If the argument starts with "-s" or "-d", save the rest of the string for later use.
- If argument.to_i produces something other than zero, then add the value (as an integer) to an array that specifies what's to be printed for each line.
- If argument.to_i produces zero, add argument to that same array.

That's about 15 lines of simple code.

Note that Ruby's ARGV is the counterpart to args in a Java declaration like void main(String args[]), but unlike Java, the name ARGV is fixed.

Remember that as stated in the Restrictions section at the top of the write-up, you aren't allowed to use argument-parsing libraries like OptionParser.

A HINT on handling the textual argument/separator rule

One way to handle the textual argument/separator rule is to simply make a pass over the argument array and if two consecutive numbers are encountered, put the separator between them, as if the user had done that in the first place. For example, if the separator is ",.", the specification

```
1 3 x 4 x -2 1
```

would be transformed into

```
1 . 3 x 4 x -2 . 1
```

A hint in a hint: Think about representing the above specification with this Ruby array:

```
[0, ",.", 2, "x", 3, "x", -2, ",.", 0]
```

Note the combination of Fixnums and Strings.

A lesson in LHtLaL (Learning How to Learn a Language)

Recall that xfield's output fields are separated by a single tab. (Use \t.) Let's demonstrate that by using a couple of ruby command line options:

```
% ruby xfield.rb 1 3 < a5/xf.1 | ruby -n -e 'puts $_.inspect'
"one\t1.0\n"
"two\t2.0\n"
"three\t3.0\n"
```
You can use `man ruby` to learn about those `-n` and `-e` options. `$ _` is a predefined global variable that holds "The last string read by the Kernel methods gets and readline." (from RPL) Here's the lesson in LHTLaL: I could have used `cat -A` to see those tabs but I chose to build my Ruby skills by learning about `-n`, `-e`, and `$ _`.

**Problem 3. (20 points) flash.rb**

For this problem you are write a Ruby program `flash.rb`, a very simple flashcard-based study system.

Here's a sample input file for `flash.rb`:

```
$ cat a5/units.fc
mile : 5280 feet
day : 86400 seconds
acre : 43560 feet
section : 640 acres
furlong : 660 feet
```

Each line specifies two related pieces of information that the user wishes to learn. A colon separates the two pieces of the pair. Whitespace around each piece is ignored. (Remove it with `String.strip`.)

In its simplest mode of operation, `flash.rb` selects one of the pairs at random, and prompts the user with the left-hand piece. The user then enters their answer, which should match the right-hand piece, ignoring case. If the user answers correctly, the pair is removed from the current run. If not, the pair is retained and tested again at a random time later in the run, albeit possibly immediately. The process repeats until all pairs have been answered correctly.

A user may quit at any time by responding with "/u". ("Uncle!"). Entering "/u" does not count as a guess.

When all pairs have been answered or the user quits, some simple statistics are shown, along with any pairs that were never answered.

Here's a run:

```
$ ruby flash.rb a5/units.fc
section? 640 acres
Correct!

mile? 5280 feet
Correct!

acre? 43650 feet
Nope!

day? 86400 seconds
Correct!

furlong? 660 feet
Correct!

acre? 46350 feet
Nope!

acre? /u
Done!

4 correct in 6 guesses = 66.7%

Never answered:
acre: 43560 feet
```

http://www2.cs.arizona.edu/classes/cs372/spring18/a5.html
flash.rb supports a number of command-line arguments. Specifying -rt ("right") will cause the user to be prompted with the right-hand values of the pairs:

```
$ ruby flash.rb -rt a5/units.fc
5280 feet? mile
Correct!

660 feet? furlong
Correct!

640 acres? /u
Done!
```

2 correct in 2 guesses = 100.0%

Never answered:
day: 86400 seconds
acre: 43560 feet
section: 640 acres

The above case also demonstrates that the never-answered pairs are shown in the order they appear in the input file.

The -mix option causes a random selection of left and right prompts:

```
$ ruby flash.rb -mix a5/units.fc
day? 86400 seconds
Correct!

660 feet? furlong
Correct!
...
```

Behavior is undefined if both -rt and -mix are specified.

As a possible development aid, the -a option causes the answer to be shown with each prompt:

```
$ ruby flash.rb -a -mix a5/units.fc
day? (psst...'86400 seconds') 86400 seconds
Correct!

43560 feet? (psst...'acre') acre
Correct!
```

For learning purposes, if the user responds with "//", the answer is shown. The response "//" is counted as a guess.

```
$ ruby flash.rb a5/units.fc
furlong? //
Answer: 660 feet

acre? //
Answer: 43560 feet

section? //
Answer: 640 acres

mile? /u
Done!
```

0 correct in 3 guesses = 0.0%
Never answered:
mile: 5280 feet
day: 86400 seconds
acre: 43560 feet
section: 640 acres
furlong: 660 feet

When learning a set of things, it can be useful to focus on a subset. The \(-r\) option limits the pairs used to those in a specific range of line numbers. Let's work on United States presidents:

```
$ cat a5/presidents.fc
1:Washington
2:Adams
3:Jefferson
4:Madison
5:Monroe
[...and forty more...]
```

Let's focus on the tenth through the fifteenth presidents:

```
$ ruby flash.rb -r 10-15 -mix a5/presidents.fc
Buchanan? 15
Correct!

10? Tyler
Correct!

13? Fillmore
Correct!

Pierce? //
Answer: 14

12? Taylor
Correct!

11? Polk
Correct!

Pierce? 14
Correct!

Done!

6 correct in 7 guesses = 85.7%
```

Here's a new file to work with:

```
$ cat a5/hexlets.fc
A:10
B:11
C:12
D:13
E:14
F:15
```

Testing a program that does random selections poses some challenges. The \(-n\) option suppresses randomization as follows:

- Pairs are presented in the order they appear in the input file.
- If an answer is missed, the pair is requeued at the end.
If `-mix` is specified, prompts are alternated in a strict left-right-left-right pattern.

The following example uses `a5/hexlets.fc`, limits choices to lines 2-5 (B-E), specifies mixing left and right prompts, and suppresses randomization. Study it closely!

```
$ ruby flash.rb -r 2-5 -mix -n a5/hexlets.fc
B? 11
Correct!

12? B
Nope!

D? 13
Correct!

14? F
Nope!

C? 12
Correct!

14? F
Nope!

E? 14
Correct!

Done!

4 correct in 7 guesses = 57.1%
```

**Argument handling summary**

- `-rt`: prompt with right-hand values
- `-mix`: prompt with a random mix of left- and right-hand values
- `-r FROM-THROUGH`: work with only the pairs found on line `FROM` through line `THROUGH`.
- `-a`: Show the answer along with the prompt.
- `-n`: No random behavior.

If an argument does not begin with a dash (`-`), it is assumed to be the name of the file of pairs to work with. The examples above use the suffix `.fc` for the pair files but that suffix is not expected nor assumed.

The behavior of `flash.rb` is undefined in cases that are not specifically addressed by this write-up or exercised with the tester. That means that any non-malicious behavior is ok—run-time errors, curious results, etc., are not a surprise if the user misuses the program or tries to trip it up.

**Implementation notes for flash.rb**

**DIY command line argument processing**

Remember that as stated in the Restrictions section at the top of the write-up, you aren't allowed to use argument-parsing libraries like `OptionParser`. You'll need to write some plain old Ruby code to do that processing yourself.

**Reading from a file**

Note that `flash.rb` reads a file whose name is specified as a command-line argument. The `open` method of
Kernel opens a file and returns an object whose `gets` method can be used to read lines. Example:

```ruby
$ cat fileio.rb
f = open(ARGV[0]) # first command-line argument

count = 0
while line = f.gets
  count += 1
end

f.close
puts "read #{count} lines"

$ ruby fileio.rb a5/presidents.fc
read 45 lines
```

Alternatively, you could use `f.readlines` to produce an array of all the lines in the file with a single call. Here's a quick test in `irb`:

```ruby
>> open("a5/presidents.fc").readlines.size
=> 45
```

An issue with automated testing of an interactive program

Here's a program that repeatedly prompts for a line and then prints the number of characters read:

```ruby
$ cat echoinput.rb
while true do
  print("Line? ")
  line = gets
  break if not line
  puts "read #{line.size} characters"
end

Usage:

$ ruby echoinput.rb
Line? just
read 5 characters
Line? testing
read 8 characters
Line? this
read 5 characters
Line? ^D
```

Note what we get when redirect those same lines from a file:

```ruby
$ cat lines
just
testing
this

$ ruby echoinput.rb < lines
Line? read 5 characters
Line? read 8 characters
Line? read 5 characters
Line? $
```

We see the output, but not the input. When trying to understand a diff, it's good to be able to see the input, too!

We can use `STDIN.TTY?` method to see if we're reading from the keyboard, which years ago, were teletypes.
("ttys"). The following Ruby invocations demonstrate that `STDIN.tty?` returns `true` iff standard input is the keyboard:

```ruby
$ ruby -e 'puts STDIN.tty?'
true

$ ruby -e 'puts STDIN.tty?' < /dev/null
false

$ echo testing | ruby -e 'puts STDIN.tty?'
false
```

You might try similar experiments with `STDOUT.tty?`.

Let's put `STDIN.tty?` into use with `echoinput2.rb`:

```ruby
$ cat echoinput2.rb
while true do
  print("Line? ")
  line = gets
  break if not line

  #
  # If we're *not* reading input from the console, "echo"
  # the line we just read.
  #
  puts \nInput: #{line.inspect}" if not STDIN.tty?
  puts "read #{line.size} characters"
end

$ ruby echoinput2.rb  < lines
Line?
Input: "just\n" read 5 characters
Line?
Input: "testing\n" read 8 characters
Line?
Input: "this\n" read 5 characters
Line? $
```

You'll need to use `STDIN.tty?` in your `flash.rb` to produce similar behavior when standard input is redirected, as the tester will do. Example:

```bash
$ cat a5/f.pres.1
Washington
2
Jefferson
//
Monroe
/u
$ ruby flash.rb -mix -n -r 1-6 a5/presidents.fc < a5/f.pres.1
1?
Response: 'Washington'
Correct!

Adams?
Response: '2'
Correct!
```
3?
Response: 'Jefferson'
Correct!

Madison?
Response: '//'  Answer: 4

5?
Response: 'Monroe'
Correct!

JQ Adams?
Response: '/u'
Done!

4 correct in 5 guesses = 80.0%

Never answered:
4: Madison
6: JQ Adams
$

Extra credit opportunity: For one point, create a flash.rb input file for some area that you find interesting. Name it flash-extra.txt. I'll publish a compilation of them, so don't submit anything you don't want everybody to see.

Problem 4. (3 points) hudak.txt

In The Haskell School of Expression the late Paul Hudak wrote,

"The best news is that Haskell's type system will tell you if your program is well-typed before you run it. This is a big advantage because most programming errors are manifested as typing errors."

Do you agree or disagree with his claim that "most programming errors are manifested as typing errors"? For this problem you are to present an argument based on your full experience as a programmer that either supports his claim or refutes it. (Do not argue both sides.) Take all your programming experience into account, not just 372.

As usual, I'm looking for quality, not quantity but as a ballpark figure I imagine 200-400 words, as reported by "wc -w hudak.txt", will be needed for a thoughtful answer.

As usual, the .txt suffix on hudak.txt should be enough to tell you that I'm wanting a plain text file, not a Word document, PDF, etc.

Problem 5. (ZERO points) longest.rb

Note: In several past semesters I've started the first Ruby assignment with this problem but I've seen too many students sink too much time into it, some of them becoming extremely frustrated. I think it's a fun problem but I've eliminated the burden of it by making it worth zero points. Try it if you wish!

Write a Ruby program that reads lines from standard input and upon end of file writes the longest line to standard output. If there are ties for the longest line, longest writes out all the lines that tie. If there is no input, longest produces no output.

Don't overlook the restrictions mentioned below.

Here are some examples.

$ cat a5/lg.1
a test
for

http://www2.cs.arizona.edu/classes/cs372/spring18/a5.html
the program
here

$ ruby longest.rb < a5/lg.1
the program

$ cat a5/lg.2
xx
a
yy
b
zz

$ ruby longest.rb < a5/lg.2
xx
yy
zz

Let's use the null device and "wc -c" to demonstrate that if longest.rb has no input, there's no output:

$ ruby longest.rb < /dev/null | wc -c
0

Let's use longest.rb and some greps to explore a list of words:

$ ruby longest.rb < /usr/share/dict/words
electroencephalograph's

$ grep ^q /usr/share/dict/words | ruby longest.rb
quadrilateral's
quadruplicate's
quadruplicating
qualification's
quartermaster's
questionnaire's

$ grep ^w /usr/share/dict/words | ruby longest.rb
whatchamacallit's
wrongheadedness's

$ grep ^wo /usr/share/dict/words | ruby longest.rb
woolgathering's
worthlessness's

Restrictions for longest.rb:

- NO COMPARISONS, such as but not limited to <,==, !=, <>, between?, eql?, empty?, and String.casecmp may be used. As a rule, any method that ends with ? should be viewed with suspicion. Additionally, any kind of sorting most likely involves comparisons.

- The case statement may not be used.

- No arithmetic operations, such as addition, subtraction, negation (including negative literals such as -1), or the pred and succ methods may be used. Imagine that arithmetic was just never invented!

- The only types you may use are Fixnum, Bignum, String, along with the values nil, true, and false (i.e., the values of the single-value classes NilClass, TrueClass, and FalseClass). In particular, you may not use arrays.

Important: Regarding comparisons, you are permitted to employ the comparison that's implicit in control structures. For example, statements like
while x do ...  # OK
if f(x) then ...  # OK

are permitted.

Like for seqwords, you'll need a line at the top of your longest.rb that loads a checker (lobotimizer?) that disables a number of prohibited operations:

    load "a5/check-longest.rb"

Problem 6. (ZERO points) jlongest.java

This problem is another zero-pointer. It is simply a Java version of longest.rb, with an analogous set of restrictions: no comparisons, no sorting, no arithmetic, only int, boolean, and String values, etc. You may use Scanner to read lines. (Use new Scanner(System.in) to create an instance of Scanner to read from standard input.)

If you've never used Java on the command line, note that javac runs the Java compiler, producing a .class file. The .class file can then be run with the java command:

    $ javac jlongest.java
    $ ls -l jlongest.class
    -rw-rw---- 1 whm whm 1161 Mar  3 05:10 jlongest.class
    $ java jlongest < a5/lg.1
    the program

Compilation and execution can be combined with Bash's && operator:

    $ javac jlongest.java && java jlongest < a5/lg.2
    xx
    YY
    zz

The && directs Bash to run "java jlongest" iff compilation of jlongest.java is successful.

Problem 7. Extra Credit observations.txt

Submit a plain text file named observations.txt with...

(a) (1 point extra credit) An estimate of how many hours it took you to complete this assignment. Put that estimate on a line by itself, like this:

    Hours: 3.5

There should be only one "Hours:" line in observations.txt. (It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention that doesn't contain the string "Hours:" . Thanks!)

Feedback and comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me thinking "Good!" as one point, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use a5/turnin to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like aN.YYYYMD.DD.HHMMSS.tz. You can run a5/turnin as often as you want. We'll grade your final pre-deadline submission.

    a5/turnin -l shows your submissions.
To give you an idea about the size of my solutions, here's what I see as of press time:

```
$ wc $(grep -v txt a5/delivs)  # this command should work for you, too
 19   47  295 seqwords.rb
 60  150  1272 xfield.rb
114  291  2678 flash.rb
 13   36   237 longest.rb
 31   77   818 jlongest.java
237  601  5300 total
```

My code has few comments.

Note that each of the aN.*.tz files created in your directory by a5/turnin is essentially a time-stamped snapshot of your code. (If you need to recover a file and aren't familiar with tar, perhaps mail to 372s18—it's easy to accidentally overwrite your latest versions.)

**Miscellaneous**

Point values of problems correspond closely to the "assignment points" mentioned in the syllabus. For example, a 10-point problem would correspond to about 1% of your final grade in the course.

If you're worried about whether a solution meets the restrictions, mail it to 372s18—we'll be happy to look it over. But don't wait too long; there may be a crunch at the end.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.)

In Ruby, # is comment to end of line, unless in a string literal or regular expression. There's no Ruby analog for /* ... */ in Java and {- ... -} in Haskell but you can comment out multiple lines by making them an embedded document—lines bracketed with =begin/=end starting in column 1. Example:

```
=begin
    Just some
    comments here.
=end
```

Silly-looking, huh? I agree! (It looks like a construction that escaped from the 1960s.) RPL 2.1.1 has more on comments.

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 5 to 7 hours to complete this assignment.

Our goal is that everybody gets **100%** on this assignment AND gets it done in an amount of time that is reasonable for them.

Assignments are not take-home tests! We hope you'll make use of Piazza, email and office hours if you run into trouble. We're happy to help you get started with any or all of these problems but in any event, if you put five hours into this assignment and don't seem to be close to completing it, it's definitely time to touch base with us. Specifically mention that you've reached five hours. Give us a chance to speed you up!