CSC 372, Spring 2015
Assignment 4
Due: Tuesday, March 10 at 23:59:59

Game plan for the Ruby assignments

Due to our mid-term exam on March 12 and Spring Break the following week, I'm spreading our Ruby work across three assignments, with the following due dates:

Assignment 4	Tuesday, March 10	(date is definite)
Assignment 5	Thursday, March 26	(date is tentative)
Assignment 6	Thursday, April 9	(date is tentative)

I think of this assignment as about a week's worth of work. It should perhaps be due on March 5 but to give you a little flexibility I'm letting it run five days longer, until March 10. I plan to issue Assignment 5 around March 5. You might want to use some of the slack time on this assignment to get a head start on Assignment 5.

Restrictions on `longest.rb` and `seqwords.rb`

Problems 1 and 2 have some restrictions that will hopefully lead to some creative thinking about string-based computations. I intend them to be a challenge and learning experience, not a frustration. I suggest that you start thinking about them as soon as possible and see what your subconscious comes up with. I believe everyone can solve these problems on their own but if you start to get frustrated, or the time you've budgeted starts to get short, ask me for some hints!

Problems 3 and 4, `mimax` and `xfield`, have no restrictions whatsoever.

Don't fall into the trap of thinking you must do these problems in sequence!

Use Ruby 1.9.3!

Ruby 1.9.3 is to be used for all Ruby assignments. Use "`rvm 1.9`" each time you login on lectura to set things for 1.9.3. You might get a blat about "`Warning! PATH is not properly set up...`" but if `ruby --version` shows 1.9.3, things should be fine. Here's what I see:

```
$ rvm 1.9
Warning! PATH is not properly set up
...lots more...
$ ruby --version
ruby 1.9.3p484 (2013-11-22 revision 43786) [x86_64-linux]
```

Use the tester!

Just like for assignment 3, there's a version of `tester` for this assignment. **Don't just "eyeball" your output—use the tester!** I won't have any sympathy for those who fail tests during grading simply because they didn't want to bother with the tester! I'll be happy to help you with using the tester and understanding its output.

I recommend making symbolic links for `a4` and `t` in your assignment 4 directory, for easy access to the tester and the data files.

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

Most of the programming problems on the Haskell assignments were functions that you tested inside `ghci`. All of the Ruby problems are this assignment require you to create programs that can be run from the command line, like `group.hs` and `avg.hs` on assignment 3.

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

```
$ date
Wed Feb 25 09:37:27 MST 2015
$ date | wc
      1       6      29
$ ls -ld .
drwxr-xr-x 266 whm staff 9044 Feb 25 09:19 .
$ (my prompt—bash is waiting for the next command)
```

In this write-up, however, output from command-line examples is followed by a blank line. Instead of the above, you'll see this:

```
$ date
Wed Feb 25 09:37:27 MST 2015

$ date | wc
      1       6      29

$ ls -ld .
drwxr-xr-x 266 whm staff 9044 Feb 25 09:19 .
```

Another UNIX tidbit: I produce the above spacing not by editing the output but by including a newline in my bash prompt, like this:

```
whm@lectura ~ 5001 % PS1="\n$ "

$ date
Wed Feb 25 09:49:17 MST 2015

$
```

Problem 1. (6 points) `longest.rb`

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 below.

Here are some examples. The files `lg.1` and `lg.2` can be found in the `a4` directory. (Make that `a4` symlink!)

```
$ cat lg.1
a test
for
the program
```

```
here

$ ruby longest.rb < lg.1
the program

$ cat lg.2
xx
a
yy
b
zz
$ ruby longest.rb < lg.2
xx
yy
zz
```

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

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

Let's use `longest` 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:

- **NO COMPARISONS, such as `<`, `==`, `!=`, `<=>`, `between?`, `eql?`, `empty?`, and `String#casecmp` may be used. (Any method that ends with `?` should be viewed with suspicion, as a rule.)**
- **The `case` statement may not be used.**
- **No arithmetic operations, such as addition, subtraction, negation, or the `pred` and `succ` methods may be used. Imagine that arithmetic was just never invented!**
- **The only types you may are use are `Fixnum`, `Bignum`, and `String`. In particular, you may not use arrays.**

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.

However, a statement such as

```
while x > 1 do         # NOT PERMITTED!
```

is not permitted—it contains a comparison (`x > 1`).

Implementation note

The examples above show various combinations of redirection and piping with bash to supply `longest.rb` with data on "standard input" but all you need to do in `longest.rb` is read lines with `line = gets` (or whatever variable you want to use.)

Problem 2. (7 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 restrictions below.

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

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

$ ruby seqwords.rb < 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 sw.2
tick
.
1
.
1
1

$ ruby seqwords.rb < sw.2
tick
tick tick
$
```

Assume that there is at least one word and at least one sequencing specification. Assume that each sequencing specification has at least one number. Assume that 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.

Restriction: The only types you may are use are `Fixnum`, `Bignum`, and `String`. In particular, you may not use arrays.

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!)

Problem 3. (8 points) `minmax.rb`

Write a Ruby program that reads lines from standard input and determines which line(s) are the longest and shortest lines in the file. The minimum and maximum lengths are output along with the line numbers of the line(s) having that length.

```
$ cat mm.1
just a
test
right here
x
$ ruby minmax.rb < mm.1
Min length: 1 (4)
Max length: 10 (3)
```

The output indicates that the shortest line is line 4; it is one character in length. The longest line is line 3; it is ten characters in length.

Another example:

```
$ cat mm.2
xxx
xx
yyy
yy
zzz
qqq
```

```
$ ruby minmax.rb < mm.2
Min length: 2 (2, 4)
Max length: 3 (1, 3, 5, 6)
```

In this case, lines 2 and 4 are tied for being the shortest line. Four lines are tied for maximum length.

If the input file is empty, the program should output a single line that states "Empty file":

```
$ ruby minmax.rb < /dev/null
Empty file
```

Some examples with `/usr/share/dict/words`:

```
$ ruby minmax.rb < /usr/share/dict/words
Min length: 1 (1, 1229, 2448, 3799, 4500, 5076, 5514, 6213, 6951,
7266, 7757, 8316, 9129, 10654, 11149, 11482, 12369, 12426, 13108,
14502, 15288, 15415, 15729, 16171, 16207, 16346, 16484, 21151,
26000, 34170, 39292, 42567, 46256, 49013, 52079, 55431, 56202,
56806, 59393, 63790, 65313, 67250, 74022, 74432, 79106, 89039,
93338, 95154, 96416, 98713, 98730, 99012)
Max length: 23 (39886)

$ ruby minmax.rb < /usr/share/dict/words | wc -l
2

$ grep ^q /usr/share/dict/words | ruby minmax.rb
Min length: 1 (1)
Max length: 15 (27, 49, 52, 86, 162, 251)
```

Although output is shown wrapped across several lines the first invocation above produces only two lines of output, demonstrated by piping that same output into `wc -l`. (I should have done `$ !! | wc -l`)

IMPORTANT: DO NOT assume any maximum length for input lines.

You might be inclined to have some repetitious code in your `minmax.rb`, such as an if-then-else that handles minimum lengths and a nearly identical if-then-else that handles maximum lengths. There are no extra points for it for but I challenge you to produce a solution in which you `DRY`.

I consider even something like the following, albeit short, to be repetitious:

```
mins = [1]
maxs = [1]
```

If you think your solution has no repetition, include the following comment and I'll see if I agree.

```
# Look! No repetition!
```

Problem 4. (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`.

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 its weaker ancestor, `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 `-s`*`SEPARATOR`* 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 an input stream of any length. (Hint: Don't do something like read all the input lines into memory and then process them—they might not fit!)

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 is an input file:

```
$ cat 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:

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

`xfield` can be used to reorder fields:

```
$ ruby xfield.rb 3 2 1 < 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:

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

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

If a field reference is out of bounds, the string `"<NONE>"` is output:

```
$ ruby xfield.rb 1 10 2 < 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 <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 `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 < 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.

<u>Non-numeric arguments other than the `-s` and `-d` flags are considered to be text to be included in each output line</u>. 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 < 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 xf.2
one two three four

$ ruby xfield.rb -s. 1 2 3 < xf.2
one.two.three

$ ruby xfield.rb -s. A 1 2 3 B C < xf.2
Aone.two.threeBC

$ ruby xfield.rb -s. A 1 B C 2 3 D  < xf.2
AoneBCtwo.threeD

$ ruby xfield.rb -s. A 1 B C 2 D 3 E  < xf.2
AoneBCtwoDthreeE
```

Below are some cases that bring all the elements into play.

```
$ cat xf.3
xxxxxxxAxxxxxxxxxxBxC
DxExF
xG1xG2
xxxxHIxxxJKxxxLMNxxxOPQRSx

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

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

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

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

```
$ 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 produce the 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:

```
while line = STDIN.gets do
```

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

Delimiter-specific behavior in `String#split`

I'm astounded by the fact that `split` behaves differently when the delimiter is a space:

```
>> " a  b  c ".split(" ")
=> ["a", "b", "c"]

>> ".a..b..c.".split(".")
=> ["", "a", "", "b", "", "c"]
```

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:

```
printf("%d arguments:\n", ARGV.length)
for i in 0...ARGV.length do     # 0...N is 0..N-1
    printf("argument %d is '%s'\n", i, ARGV[i])
end
```

Execution:

```
$ 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 so that the program doesn't "see" them. Example:

```
$ 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='

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

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

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

<u>Additionally, the shell intercepts the < and > redirection operators—the program never sees those operators or their accompanying filename argument</u>:

```
$ 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'
$
```

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.

<u>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!</u>

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"` then 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.

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 integers and strings.

A lesson in LHtLaL

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 <xf.1 | ruby -n -e 'puts $_.inspect'
"one\t1.0\n"
"two\t2.0\n"
"three\t3.0\n"
"four\t4.0\n"
"twenty\t20.0\n"
```

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 `$_`.

MID-TERM HINT: I'll ask you what LHtLaL stands for. Answer: Learning How to Learn a Language.

Problem 5. (3 points) `hudak.txt`

In *The Haskell School of Expression* Paul Hudak writes,

> "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 agree or disagree with his claim that "most programming errors are manifested as typing errors"? For ths problem you 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`, will be needed for a thoughtful answer.

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

Problem 6. (<u>ZERO POINTS</u>) `support.txt`

If a language claims to support a particular paradigm then we expect an implementation of the language to be very good at supporting the techniques and methods that are espoused by the paradigm. For example, a computation like `sum [1..10000000]` demonstrates that Haskell is very good at recursion. How does Ruby fare with supporting recursive computations?

Problem 7. (<u>ZERO POINTS</u>) `errors.txt`

I've done a significant amount of work with over thirty languages and of them all I've found Haskell's type errors to be the most difficult type errors to understand. Is your experience similar to mine? What is it that makes Haskell's errors so difficult to understand? Do you have any ideas for improving this sad situation?

Problem 8. <u>Extra Credit</u> `observations.txt`

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

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three <u>examples</u>:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

Other 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 saying "Good!" as one point, "Interesting!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use the D2L Dropbox named `a4` to **<u>submit a single zip file named `a4.zip` that contains all your work</u>**. If you submit more than one `a4.zip`, your final submission will be graded. Here's the full list of deliverables:

`longest.rb`
`seqwords.rb`
`minmax.rb`
`xfield.rb`
`hudak.txt`
`support.txt` *(zero points!)*
`errors.txt` *(zero points!)*
`observations.txt` (for extra credit)

<u>DO NOT SUBMIT INDIVIDUAL FILES—submit a file named `a4.zip` that contains each of the above files.</u>

Note that all characters in the file names are lowercase.

Miscellaneous

Restrictions not withstanding, you can use any elements of Ruby that you desire, but the assignment is written with the intention that it can be completed easily using only the material presented on the slides prior to the *Iterators and blocks* section (around slide 130).

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

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds to 1% of your final grade in the course.

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.) A # 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
```

Stupid, huh? I agree!

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.

Keep in mind the point value of each problem; don't invest an inordinate amount of time in a problem or become incredibly frustrated before you ask for a hint or help. Remember that the purpose of the assignments is to build understanding of the course material by applying it to solve problems. If you reach the five-hour mark, regardless of whether you have specific questions, it's probably time to touch base with me. Give me a chance to speed you up! **My goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.**

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, and more. (See the syllabus for the details.)