CSC 372, Spring 2014
Assignment 3 (with corrections as of March 3, 13:53)
Due: Friday, March 7 at 23:00

Problem 1. (7 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.

Examples:

```
% 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
% grep ^o /usr/share/dict/words | ruby longest.rb
oversimplification's
% ruby longest.rb < /dev/null
% ruby longest.rb < /dev/null | wc -c
0
%
```

Restrictions:

- **No comparisons, such as `<`, `==`, `!=`, `<=>`, `between?`, `eql?`, 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 and subtraction, may be used.**
- **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`).

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 specifications:

```
% 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 % is my shell prompt.  four one is the last line of sw.1)
```

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
%   (shell prompt)
```

```
% 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. Assume that words are between 1 and 1000 characters in length, inclusive. 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.

Problem 3. (10 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
```

A final example, **run on `lectura`**:

```
% 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
```

Although output is shown wrapped across several lines the first invocation above produces only two lines of output, demonstrated by piping the output into `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 write a solution that has no duplication of code.

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. (18 points) `xfield.rb`

For this problem you are to write a Ruby program that extracts columns of data from standard input. Command line arguments specify characters that delimit columns, the columns to extract, and strings to separate extracted columns in the final output.

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
%
```

By default, fields are considered to be delimited by one or more spaces. The English text and the real numbers could 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
%
```

Note that field numbering is 1-based—the first field is 1, not 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 used:

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

By default, output fields are separated by a single tab. (Use `"\t"`.) This default separator can be overridden with the `-s` flag. The separator must be at least one character in length.

```
% 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
%
```

A delimiter other than space can be specified with the `-d` option. To extract login ids and real names from `/etc/passwd`, one might use this:

```
% ruby xfield.rb -d: 1 5 < /etc/passwd | head
root    root
daemon  daemon
bin     bin
...more lines..
```

Note that the `-s` and `-d` options are single arguments—there's no space between `-s` or `-d` and the following string. The behavior of something like `xfield -s ... -d = 1` is undefined.

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 < 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 rule again:

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

Below are some more examples showing the rule in action. A blank line has been inserted in this write-up after the output of each command.

```
% 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. Trailing blank lines have been added for readability.

```
% 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
%
```

If no fields are specified, an error is printed and execution terminates

```
% ruby xfield.rb
xfield: no fields specified
% ruby xfield.rb -d.
xfield: no fields specified
%
```

To terminate execution, use code like this:

```
if ...no fields found... then
    puts "xfield: no fields specified"
    exit 1
end
```

The only error handling required for `xfield` is the "no fields" situation just mentioned. In all other odd situations, "the behavior is undefined", which means any behavior is ok—run-time errors, curious results, etc. are no problem if the user misuses `xfield`.

For example, `xfield -s -d:, 1`, is invalid for two reasons: (1) `-s` should be immediately followed by a separator string, like `-s,` or `-s///`. (2) `-d` should be followed by a single character, not two.

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 just `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`

It astounds me but the fact is 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     # three dots goes to length -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'
```

Quotes and backslashes specified on the command line are processed and fully consumed by the shell. In the usual case, 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\'
%
```

Additionally, the shell intercepts the < and > redirection operators—the program never sees them:

```
% ruby echo.rb 1 2 3 < lg.1
3 arguments:
argument 0 is '1'
argument 2 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 with the 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`, the variable that references an array of strings that holds the command line arguments. 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.

If the user specifies multiple `-s` and/or `-d` arguments, just let the last one "win". For example,

```
ruby xfield.rb -s1 -d: -d. -s2 -d, -s. 1
```

is equivalent to

```
ruby xfield.rb -d, -s. 1
```

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 that there is a combination of integers and strings.

Problem 5. Extra Credit `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 these:

```
Hours: 10
Hours: 12-15.5
Hours: 15+
```

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 `a3` to **submit a single zip file named `a3.zip` that contains all your work**. If you submit more than one `a3.zip`, we'll grade your final submission. Here's the full list of deliverables:

`longest.rb`
`seqwords.rb`
`minmax.rb`
`xfield.rb`
`observations.txt` (for extra credit)

DO NOT SUBMIT INDIVIDUAL FILES—submit a file named `a3.zip` that contains each of the above files.

Note that all characters in the file names are lowercase.

Have all the deliverables in the uppermost level of the zip. It's ok if your zip includes other files, too.

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 slides 1-93.

If you're worried about whether a solution meets the restrictions, mail it to me and Dennis or show it to us in person. Do not, repeat **do not**, send code via a private post on Piazza—it's just too easy to forget to make it private!

The output of your solutions should exactly match the output shown in this write-up. The data files used in the examples can be found in http://www.cs.arizona.edu/classes/cs372/spring14/a3

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 analog to `/* ... */` 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
```

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 us. Give us a chance to speed you up! **Our 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.)