# CSC 352, Fall 2015
# Assignment 2
# Due: Wednesday, September 9 at 23:59:59

**Some general comments about assignments**

I refer to assignments as "a*N*". The survey was `a1`; this assignment is `a2`. This document is the "`a2` write-up".

Assignment write-ups are a combination of education, specification, and guidance. My goal is to produce write-ups that need no further specification or clarification. That goal is rarely achieved. If you have questions you can either mail to `352f15` or post to Piazza using the appropriate folder. Be sure that you've read the full write-up for a problem before posting a question about it on Piazza.

Never post on Piazza code that is related to a problem on an assignment.

The write-ups for programming problems, be them the Java programs and bash scripts on this assignment or the C problems on later assignments, will usually include examples of command-line interaction. Unlike the examples in the slides, examples in write-up will have no vertical whitespace before the "`%`" in the prompt. Example:

```
% date
Sat Aug 29 21:56:10 MST 2015
% date -u
Sun Aug 30 04:56:11 UTC 2015
%
```

I assume that you've read every word of the syllabus but I recommend that you now read again the sections in the syllabus on *Assignments* and *Bug Bounties*.

**Recycling of problems from past semesters**

When I first started teaching here at The University of Arizona my aim was to come up with a great new set of problems for each assignment each time I taught a class. Maybe I just got old but I eventually found I couldn't keep up with that standard. I noticed that sometimes an old problem was simply better than a new one. I began to question my "all new problems every semester" goal. Is there an ideal set of problems to teach a set of concepts? How should a teacher's time be balanced between writing new problems and other responsibilities, like working with students and improving lecture material? If creative efforts fail is it better to recycle a good problem or go with a new one that's lesser just so that students won't be tempted to consult an old solution? The bottom line is that I sometimes do recycle problems from past semesters.

If you should come into possession of a solution from a past semester, whether it be from "homework help" site on the net, a friend who took the class, or some other source, let me encourage you to discard it! For one thing, I sometimes put *dunsels* in my solutions. (Google it!). I'd like to distribute solutions with perfectly clean and idiomatic code but I've found that an extra part that sticks out like a sore thumb (and that the lazy don't tend to notice!) help me eliminate students who find and reuse my solutions. If you notice a silly extra in my code, try removing it and see if things still work. If so, you've perhaps found a dunsel!

While I'm talking about disincentives for cheating I'll also mention that I keep all copies of all former students' submissions for a given problem. We use tools like MOSS to look for suspicious similarities

both in the set of this semester's submissions and in that set combined with submissions from any previous semesters when the problem was used.

**Remember my cheating policy: one strike and you're out!**

**Use the tester!**

The syllabus says,

> *For programming problems great emphasis will be placed on the ability to deliver code whose output exactly matches the specification. Failing to achieve that will typically result in large point deductions, sometimes the full value of the problem.*

I'll provide a tester that you can use to confirm that the output of each of your solutions for the programming problems exactly matches the specified output for a number of test cases.

**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! We'll be happy to help you with using the tester and understanding its output.

The tester is described in a separate document that will soon be available.

**Make reasonable assumptions about unspecified behavior**

I expect you to make reasonable assumptions about program behavior so that I can keep down clutter in the write-ups. For example, `sum.java` (below) outputs the sum of the integers read on standard input. I say to use `Integer.parseInt()` to parse input lines, so based on that you can assume that input values won't exceed the range of an `int`. I say nothing at all about the magnitude of the sum that might be accumulated, so a reasonable assumption is that an `int` will hold the sum, too. On the other hand, if an example showed a 1000-digit sum, you probably should assume you'll need to use `BigInteger`.

Nothing about command line arguments is mentioned in the write-up for `sum.java` so you should assume that behavior is undefined if it is run with command line arguments. The practical implication of behavior being undefined for some case is that we won't test with that case. And if we did, we couldn't complain if any of these things happened: (1) the program terminates immediately, (2) the arguments are silently ignored, (3) the program prints a warning on standard error but processes the data. Needless to say, "behavior is undefined" is never a green light for malicious behavior.

Don't assume needless limits. For example, `sum.java` should be able to process a googol of input lines as long the as the sum never exceeds the capacity of an `int`. A program like `args.java` shouldn't assume any limit on the number of arguments aside from all of them being able to fit into an array.

If a test run by the tester implies a significant behavior that's not mentioned in the write-up, that may be a bug in the write-up. For example, if a `sum.java` test produces a sum that requires a `long`, that would probably be worth a note to `352f15`.

Common sense can often produce reasonable assumptions about behavior. For example, `lengths.java`, which writes to standard output the length of each line read on standard input, simply produces no output if there's no input. On the other hand, if `sum.java` gets no input it still produces a sum: 0. We'll soon see that a write-up example that shows the result of redirecting standard input from `/dev/null`, "the null device", should immediately settle any question about behavior with an empty input.

**"Plain text file"**

Some problems, like `timeline.txt` and `questions.txt`, ask you to create a "plain text file". A plain text file contains no formatting data or markup of any sort. A plain text file can be created with editors like Emacs, Vim, Notepad++, Sublime, etc. Java source files are plain text files, for example.

If a plain text file is requested, do not submit a Word document, `.rtf` file, an HTML document, a PDF, etc.! Anything but a plain text file will earn a grade of zero.

**Create a symbolic link to reference the data files used in the examples**

An easy way to access the data files referenced in this write-up is to put a symlink (symbolic link) to `/cs/www/classes/cs352/fall15/a2` in your assignment 2 directory on lectura. We'll talk about symbolic links soon but I often describe a symbolic link as a Windows shortcut done right. Let's assume that in your home directory on lectura (`/home/YOURNETID`) you've made a `352` directory and in `352` you've made a directory `a2`. (You can do that with `mkdir -p ~/352/a2`. "`~/`" is a shorthand that bash recognizes as specifying your home directory.)

Go to your `352/a2` directory and make a symlink to `fall15/a2`:

```
% cd ~/352/a2
% ln -s /cs/www/classes/cs352/fall15/a2 .
```

Then take a look at what `ls` shows:

```
% ls -l a2
lrwxrwxrwx 1 whm whm 31 Aug 29 21:42 a2 -> /cs/www/classes/cs352/fall15/a2
```

That lowercase "L" at the start of the line indicates a symbolic link, and `a2 -> ...` shows what the symbolic link `a2` references. If you "`ls a2`", "`cd a2`", etc., you'll actually be operating on `/cs/www/classes/cs352/fall15/a2`.

Using "`.`" as the last argument for `ln -s` causes the symlink to have the same name as the target (`a2`) but you could name it something else. Omitting the last argument has the same effect.

**Hints**

Various problems have hints in `a2/`*`PROBLEM`*`-hints`. That collection of files, and the hints in each, may grow over time. I imagine that a number of third-semester CS students (students who have had only 127A and B) will need to make use of the hints but I encourage everyone to make some effort to solve each problem before looking at the hints. If you want a 300-level experience, don't look at the hints until you've got a version fully working.

I won't post updates on Piazza when adding hints, but you can use `ls -lt a2/*-hints` to see what's newest.

After your version is finished, whether you read the hints or not, I encourage you to look at the hints on the chance you might learn something from them.

If you would, to help me get a better picture of the class, add one of these comments to your Java code to indicate whether you looked at the hints before you had a fully working version.

```
// Hints: yes
```

```
// Hints: no
```

In bash, a # is comment to end of line, like // in Java, so use one of these in your bash scripts:

```
# Hints: yes
# Hints: no
```

**Beat the Coach!**

On some problems I'll include a "Beat-the-Coach time", which is how long it took me to develop my solution. Don't include the time it takes you to read and understand the problem, or the time to read man pages and experiment with commands.

# Java Problems

The following problems ask you to write some small Java programs. The purpose of these problems is to help you understand how Java fits into a UNIX environment, and to reinforce the basics of argument handling and I/O redirection in a language that you already know. (If you don't know Java, let me know.)

`lc.java`, on slide 47, is a good starting point for the programs that read standard input.

Feel free to use Eclipse or any other Java IDE you wish to develop your solutions, but I found Emacs to be plenty adequate for these simple problems. In most interviews for programming positions it's you, a whiteboard, and a marker, so I think there's some merit in doing simple coding without having an IDE do most of the typing for you.

**Problem 1. (1 point) `lengths.java`**

`lengths.java` reads lines from standard input and writes the length of each line to standard output. Note that these examples that assume the presence of the `a2` symlink mentioned above.

```
% cat a2/lengths.1
just
a
test

here
% java lengths < a2/lengths.1
4
1
4
0
4
%
```

If standard input is empty, `java lengths` produces no output. One way to supply an empty standard input is to redirect from `/dev/null`, "the null device". Another way is to pipe from `echo -n`, which produces no output. Observe:

```
% echo -n
% echo -n | wc -c
0
% wc -c < /dev/null
0
```

Here's how `lengths.java` behaves with no input:

```
% java lengths < /dev/null
% java lengths < /dev/null | wc -c
0
% echo -n | java lengths | wc -c
0
```

Note that either of the last two examples would be a definitive example for empty input. I show them both to build understanding.

**Problem 2. (1 point) `rev.java`**

`rev.java` reads lines from standard input and writes to standard output the characters of each line in reversed order. Examples:

```
% cat a2/rev.1
one
two
three
% java rev < a2/rev.1
eno
owt
eerht
% java rev < a2/rev.1 | java rev >out
% cat out
one
two
three
% java rev < /dev/null | wc -c
0
```

**Problem 3. (2 points) `sum.java`**

`sum.java` prints the sum of the integers it finds on standard input. In order for an integer to be included, the integer must be the only text on the line. (Call `Integer.parseInt()` with each input line. If `parseInt` produces a result, include that result in the sum.)

Example:

```
% java sum
50
x
10 20
    30
-5
^D (control-D)
45     (Note that the output may overwrite the echo of ^D — that's ok!)
%
```

Note that `"10 20"` is not included because there are two integers on the line. The following line, with `30`, is not included because of the leading spaces—`Integer.parseInt("  30")` throws an exception.

**Problem 4. (2 points) `iota.java`**

Write a Java program named `iota` (named after an analogous operation in APL) that accepts a single command line argument (`N`) that is assumed to be an integer greater than zero. `iota` then prints the integers from 1 through `N`, one per line.

If `iota` is run with no arguments, or more than one argument, it prints the error message "`usage: iota N`" on standard error (`System.err`, not `System.out`) and exits. Use `System.exit(1)` to terminate program execution with a status code of 1, which indicates an error.

Examples:

```
% java iota 3
1
2
3
% java iota 10000 | wc -l
10000
% java iota
usage: iota N
% java iota 10 20
usage: iota N
% java iota x
usage: iota N
% java iota 1000000 | java sum
500000500000
```

**Problem 5. (3 points) `box.java`**

`box.java` prints a box around whatever it reads from standard input.

```
% date | java box
+----------------------------+
|Sat Aug 29 17:31:33 MST 2015|
+----------------------------+
% cal -h | java box
+----------------------+
|    August 2015       |
|Su Mo Tu We Th Fr Sa  |
|                 1  |
| 2  3  4  5  6  7  8  |
| 9 10 11 12 13 14 15  |
|16 17 18 19 20 21 22  |
|23 24 25 26 27 28 29  |
|30 31                 |
+----------------------+
% cal -h | tr -d 2 | java box
+----------------------+
|    August 015        |
|Su Mo Tu We Th Fr Sa  |
|                 1  |
|   3  4  5  6  7  8   |
| 9 10 11 1 13 14 15   |
|16 17 18 19 0 1       |
|3 4 5 6 7 8 9         |
```

```
|30 31                  |
+----------------------+
%
% java box < /dev/null | wc -c
0
```

Implementation note: Change all occurrences of tabs (`'\t'`, ASCII character 9) in the input into a string of eight spaces. Yes, that's not a very good way to handle tabs in a program like this one but we'll live with it.

Beat-the-coach time: 12 minutes

**Problem 6. (3 points) `eval.java`**

Write a Java program that evaluates simple integer arithmetic expressions that are specified as a sequence of command-line arguments. The result is printed on standard output. Examples:

```
% java eval 3 + 4 x 5
35
% java eval 1 x 23 x 456
10488
% java eval -3 x 7 + 4 - -100 / 3
27
% java eval 17 % 5
2
```

`eval` is **VERY** simple-minded. Repeat: `eval` is **VERY** simple-minded. It assumes the first argument is an integer, the second is an operand, the third is an integer, etc. **There is no precedence**—operators are simply evaluated from left to right.

The supported operators are `+`, `-`, `x`, `/`, and `%`. Note that `x` is used instead of `*` to indicate multiplication.

`eval` assumes that all operands and results can be represented with a `long`. Use the method `Long.parseLong()` to convert a `String` into a `long`, and keep in mind that it properly handles strings such as `"-100"`

`eval` does no error checking whatsoever; its behavior is undefined with non-numeric operands, invalid operators, and space-less expressions such as `2x3+5` (instead of `2 x 3 + 5`). Strictly FYI, here's what mine does on one undefined case:

```
% java eval 3 + 4 x
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
        at eval.main(eval.java:6)
```

Watch out for accidentally using `"*"` instead of `"x"` when testing! On the command line `*` is a *wildcard* that expands to the names of all the files in the current directory.

**Problem 7. (6 points) `mgrep.java`**

In this problem you are to write a micro-featured version of `fgrep` in Java. Here is a man page for `mgrep`:

```
NAME
      mgrep — print lines that contain a string
```

```
SYNOPSIS
     mgrep [-v] STRING [FILE1 ... FILEN]

DESCRIPTION
     mgrep searches each of the specified FILEs for lines that contain
     STRING.  Lines that match are printed on standard output,
     preceded by the name of the file containing the line.

     If no files are specified, mgrep reads standard input; the output
     is simply the lines that match, without any indication of their
     source.

     If a file does not exist, an error is printed on standard error
     but mgrep does not exit--it continues with the next file, if any.

OPTIONS
     -v   Inverts the search, printing non-matching lines.  If the
          first argument is not -v, it is assumed to be STRING.
          (java mgrep -x y looks for occurrences of "-x" in y.)
```

Note that `mgrep` doesn't have a lot of smarts about argument processing. Take advantage of that to keep the argument processing simple.

If no arguments are specified, `mgrep` prints a usage message on standard error (use `System.err`).

```
% java mgrep e a2/rev.1 a2/lengths.1
a2/rev.1:one
a2/rev.1:three
a2/lengths.1:test
a2/lengths.1:here
% java mgrep root < /etc/passwd
root:x:0:0:root:/root:/bin/bash
% java mgrep root /etc/gorp /etc/passwd
/etc/gorp: can't open
/etc/passwd:root:x:0:0:root:/root:/bin/bash
% java mgrep -v root < /etc/passwd | wc -l
3326        (NOTE: this number will vary as users are added or removed!)
% java mgrep
Usage: mgrep [-v] STRING [FILE1 ... FILEN]
%
```

Implementation notes: Use the `FileReader` class to create a `Reader` that in turn can be used to construct a `BufferedReader`. You may want to have a method that takes a `BufferedReader` as an argument and processes the lines produced by the reader. In a given run you might call that method once, with a `BufferedReader` for standard input; or you might call it many times, with a `BufferedReader` for each file in turn.

For a 300-level challenge, add a `-i` option, to make the search be case-insensitive. Handle all combination of options: `-v -i`, `-iv`, etc. Complain if an argument starts with "`-`" and isn't followed by "`i`" or "`v`".

If time permits, we'll make additional tests available for these "300-level challenges" but none will be provided intially.

# One-pipeline bash scripts

The following problems ask you to create one-pipeline bash scripts. Your solution must consist of a single pipeline, like this:

```
COMMAND ARGS | COMMAND ARGS | ... | COMMAND ARGS
```

In some cases, one command, not a pipeline, may be all you need. That's fine.

No temporary files or shell control structures (`if`, `while`, `for`, `case`, etc.) may be used.

We'll be happy to examine solutions for compliance; mail them to `352f15`, but don't cut those requests too close to the deadline; we might be swamped!

Needless to say, including a `# Hint:` comment doesn't violate the one-pipeline rule!

As example of what I'm looking for, here's the final pipeline from the slide 66 sequence packaged as a one-pipeline bash script:

```
% cat usercount
who | cut -f1 -d" " | sort | uniq | wc -l
```

Remember that as shown on slide 69 you've got to use `chmod +x` to make a script executable:

```
% chmod +x usercount
```

`chmod` needs to be done only once.

Also shown on slide 69 is that you may need to prepend dot-slash to the script's name to run it.

```
% ./usercount
27
```

Here's a one-pipeline script that uses a command-line argument to output the Nth line of standard input:

```
% cat nth
head -$1 | tail -1
% cal | ./nth 2
Su Mo Tu We Th Fr Sa
```

It's almost never practical to handle any errors in a one-pipeline script, so don't worry about any error conditions.

**Problem 8. (1 point) `amj`**

Write a one-pipeline bash script named `amj` that uses `cal` to display April, May, and June of 2016. Example:

```
% ./amj
      April                May                 June
Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa  Su Mo Tu We Th Fr Sa
                1  2   1  2  3  4  5  6  7            1  2  3  4
 3  4  5  6  7  8  9   8  9 10 11 12 13 14   5  6  7  8  9 10 11
10 11 12 13 14 15 16  15 16 17 18 19 20 21  12 13 14 15 16 17 18
```

```
17 18 19 20 21 22 23  22 23 24 25 26 27 28  19 20 21 22 23 24 25
24 25 26 27 28 29 30  29 30 31              26 27 28 29 30
% ./amj | wc -l
7
%
```

**Note that a general solution is not required, only one that works for 2016.**

Note: `cal` has some options that can trivialize this problem so here's a restriction: your pipeline may run `cal` only once, and like this: "`cal 2016`".

**Problem 9. (2 points) `revnum`**

Write a one-pipeline bash script named `revnum` that behaves like `cat -n` but instead of numbering in ascending order, it numbers in descending order:

```
% head -5 /etc/passwd | revnum
     5  root:x:0:0:root:/root:/bin/bash
     4  daemon:x:1:1:daemon:/usr/sbin:/bin/sh
     3  bin:x:2:2:bin:/bin:/bin/sh
     2  sys:x:3:3:sys:/dev:/bin/sh
     1  sync:x:4:65534:sync:/bin:/bin/sync
% seq 3 | revnum
     3  1
     2  2
     1  3
%
```

You'll find `tac` to be handy on this one.

**Problem 10. (2 points) `faclogins`**

Write a one-pipeline bash script that "scrapes" `cs.arizona.edu/personnel/faculty.html` and prints a numbered list of the logins of the CS faculty, in alphabetical order.

```
% ./faclogins
     1  alon
     2  bzhang
     3  collberg
    ...more lines...
    21  rts
```

To get started, run

```
curl -s http://www.cs.arizona.edu/personnel/faculty.html > x
```

and then look for `proebsting` in `x` using an editor of your choice. `x` will contain HTML. You don't need to know HTML but you do need to be able to see a pattern in the data.

Beat-the-Coach time: 73 seconds

**Problem 11. (1 points) `tomorrow`**

Write a one-pipeline bash script named `tomorrow` that prints the day of the week that tomorrow is.

```
% date
Sat Aug 29 19:10:35 MST 2015
% ./tomorrow
Sunday
```

**Problem 12. (3 points) `isleap`**

Write a one-pipeline bash script `isleap` that outputs `1` if its argument is a leap year and `0` if not.

```
% isleap 2015
0
% isleap 2016
1
```

My solution uses `cal`, `fgrep`, `tr`, and `wc`.

You won't want `less` in your final solution but to get started, type this command:

```
% cal 2015 | tr  " "  \\n | less
```

Beat-the-Coach time: 5 minutes and 22 seconds

**Problem 13. (2 points) `days`**

Write a one-pipeline bash script that shows how many users have been idle for more two or more days, according to `w(1)` output that `days` reads from standard input.

```
% w | ./days
22
% ./days < a2/days.1
22
```

A practical issue with this script is that the output of `w` is constantly changing. In order to have something stable to test against, `a2/days.1` is output redirected from `w`.

Perhaps on a future assignment we'll revisit `days` and add a `-t` option to run in test mode: Without `-t` it would use `w`'s output but with `-t` it would read from standard input.

`cut`'s handling of multiple consecutive spaces makes for some trouble with this problem. Start by piping `w`'s output through `tr -s " "`.

# Text-based problems

**Problem 14. (1 point) `timeline.txt`**

UNIX slide 4 and following cite various events on the UNIX timeline. For this problem you are to do a little research and write an entry to add to the UNIX timeline.

Submit your entry as a **plain text file** named `timeline.txt`. Use this format:

(1) The full entry should be a single line of text.
(2) Begin the entry with a year, then a space, followed by one or more sentences with whatever you want to say.

**We'll `cat` together all the submissions, run it through `sort -n`, and post the result on Piazza.**

As an example of the format I want you to use here's the first entry from slide 4 as a `timeline.txt` file. I'll use `cat` to display it and then `wc -l` to demonstrate it's only one line long.

```
% cat a2/timeline.txt
1965 Researchers from Bell Labs and other organizations begin work
on Multics, a state-of-the-art interactive, multi-user operating
system.
% wc -l a2/timeline.txt
1 a2/timeline.txt
%
```

**Feel free to use Google, Wikipedia, etc., for research on this question** but needless to say, no posts anywhere soliciting ideas.

You do not need to cite your source but the entry should be written in your own words.

The year must fall in the range `1970-2015`. The event must be related to UNIX or a UNIX-like system or environment (like Linux or Cygwin).

**Problem 15. (6 points) `questions.txt`**

`a2/questions.txt` is a plain text file with twelve free-response questions, each worth a half-point. Copy the file into your directory and edit your answers into it, leaving the questions intact. `a2/questions.txt` starts with two questions with answers, as examples. You'll see that answers should start with "`A: `" at the beginning of a line. Answers may be multi-line.

If you're unsure about the format for any of your answers, mail it to `352f15` and we'll take a look!

**Problem 16. 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 the following three examples:

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

If you want the one-point bonus, be sure to report your total (estimated) hours on a line that starts with "`Hours:`".

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, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

**Turning in your work**

Use the D2L Dropbox named `a2` to **submit a single zip file named `a2.zip` that contains all your work**. If you submit more than one `a2.zip`, your final submission will be graded.

`a2/delivs` (short for "deliverables") has the full list of files you are to submit, along with the optional `observations.txt`.

`zip(1)` gives me a headache so I usually use `jar` to make zips. Here's a command that uses *command substitution* to turn the output of `cat a2/delivs` into file name arguments for `jar`:

```
jar cvMf a2.zip $(cat a2/delivs)
```

To give you an idea about the sizes of my solutions, here's what I see as of press time.

```
% wc $(cat a2/delivs)
 12   31  332 lengths.java
 12   32  343 rev.java
 17   47  415 sum.java
 20   51  479 iota.java
 42  119 1205 box.java
 27   70  801 eval.java
 65  167 1820 mgrep.java
  1    8   30 amj
  1    6   19 revnum
  1   15  101 faclogins
  1    3   27 tomorrow
  1   19   60 isleap
  1   20   59 days
  1   18  141 timeline.txt
 71  518 2835 questions.txt
  0    0    0 observations.txt
273 1124 8667 total
```

There are no comments in my code. My `questions.txt` has been populated with my answers.

**Miscellaneous**

This assignment is based on the material on UNIX slides 1-102.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem would correspond 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.)

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 a student who has only taken CSC 127A and 127B but done well in them will need 10 to 12 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 eight-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.)