CSc 372 — Comparative Programming Languages
Spring 2026 (McCann)

https://cs.arizona.edu/classes/cs372/spring26/

Program #1: Ruby

Due Date: February 18th, 2026, at the beginning of class

Overview: In this class, we study specific languages in some detail, but not all detail. Our purpose is to study the design of the languages. In the case of Ruby, we are focusing on the object–oriented characteristics. The best way to understand those features is to write programs that use them. Before you can do that, you also need to be able to use the basics of the language. That is why this assignment is a collection of programming exercises, ranging from the straight–forward to the more involved. Between the lecture material and this assignment, we hope that you will become comfortable enough with Ruby to continue to explore additional features of the language on your own, should you choose to do so.

Assignment: For each numbered task below, write a Ruby 3.2.3 (or earlier) program (named as stated) that accomplishes it.

1. **Hailstones**.

```
$ ruby hailstones.rb 7
The hailstone sequence starting with 7:

     7    22    11    34    17    52    26    13
    40    20    10     5    16     8     4     2
     1

There are 17 values; the largest is 52.
```

In 1937, Lothar Collatz, a German mathematician, suggested that the following result is always true: Start with a positive integer. If that integer is an even number, halve it. If it is odd, triple it and add one. Continuing this with the resulting integers will always cause you to reach one, sooner or later. What makes this process interesting is that no one has been able to prove that the sequence will reach one for every positive integer, but every number ever tested (over 5×10^{18} of them) has. This conjecture is known by many names, including the Collatz Conjecture and the Hailstone Sequence.

As an example, consider the integer 6. It's even, so we halve it to get 3. 3 is odd, so we triple it and add one, producing 10. Continuing produces this sequence of nine values:

$$6, 3, 10, 5, 16, 8, 4, 2, 1$$

You may be wondering why we stop at one. If we were to continue, we'd jump back to four ($3*1+1 = 4$), which starts a repeating sequence. Thus, there's no point in continuing past one.

Write a complete, well-documented Ruby program named `hailstones.rb` that accepts a positive integer from the command line and computes and displays the sequence resulting from it, starting with the given value, eight values per line in fields of size 6. The last line may have fewer than eight values. Then, display the total quantity of values in the sequence, and the largest value in the sequence (which might be the starting value). Use the output format shown above.

(Continued...)

2. **Wordify**.

```
$ cat data.txt
Some (made-up) data --
over 2.0 lines.
$ ruby wordify.rb < data.txt > words.txt
$ cat words.txt
Some
made-up
data
over
2
0
lines
```

Write a complete, well-documented Ruby program named `wordify.rb` that reads text from standard input (normally the keyboard, but could be the content of a text file, as demonstrated above), identifies the words in the input, and outputs them to standard output, one word per line. For this program, a word is defined to be a sequence of characters starting with a letter or a digit, and consisting of letters, digits, or hyphens.

There are many ways to do this sort of task in Ruby. Any of them will be acceptable, including ones using regular expressions, should you wish to read up on them and use them. (We don't anticipate spending much, if any, time on regular expressions in class.)

3. **Word Sort**.

```
$ cat words.txt
Some
made-up
data
over
2
0
lines
$ ruby wordsort.rb words.txt
$ cat words-sorted.txt
0
2
Some
data
lines
made-up
over
```

NOTE: The input to this program is in the same format as the output from `wordify.rb`, but is required to be in a file.

We hope that you learned Insertion Sort in an earlier class. (If you did not, check out the Wikipedia page on the algorithm; the URL is in the "Want to Learn More?" section of this handout.) A simple variation of Insertion Sort is Binary Insertion Sort, which uses binary search to find the proper location of the next item to be inserted into the growing sorted sequence.

Write a complete, well-documented Ruby program named `wordsort.rb` that accepts, as a command-line argument, the name of a file of words (one word per line); sorts those words using binary insertion sort; and saves the sorted list of words to a file, again one word per line, whose name is that of the input file with "-sorted" added after the name but ahead of the extension.

Additional requirement: You are to write stand-alone Ruby methods (that is, methods that are not explicitly added to a class) to perform the searching and the sorting. Thus, you will need to write two methods, though you may write additional helper methods if you wish.

(Continued...)

4. **Word Concordance**.

```
$ cat seuss.txt
One Fish
Two Fish
Red Fish
Blue Fish
$ ruby concordance.rb seuss.txt
Blue (4,1)
Fish (1,2) (2,2) (3,2) (4,2)
 One (1,1)
 Red (3,1)
 Two (2,1)
```

NOTE: The input to this program is in the same format as the input to `wordify.rb` — a text file of words, using the same definition of word.

You may be familiar with maps in Java and/or dictionaries in Python. Ruby's version of the *association list* concept is called a "Hash," but the idea is the same: A Hash is a collection of values of any type, indexed by keys of any type.

A *concordance* is an alphabetical list of words from a document in which the words are paired with their locations in the document.

Write a complete, well-documented Ruby program named `concordance.rb` that reads words from the given text file and uses a Hash to produce a sorted list of the words, one per line, with their locations listed to the right of the word in the form "(X,Y)", where X is the line number (the first line is line 1) and Y is the word number on that line (the first word is word 1). The example output above should clear up any confusion.

We want the output to be tidy, so: Use the length of the longest word to position the word locations such that the first location of the longest word is separated from the word by one space, and the first locations of the rest of the words align with that list. All of the words should be aligned by their last letters (that is, right-justified). Again, this is shown in the output above.

5. **Clock and AlarmClock Classes**.

Seems like we should have you do something with creating classes and playing with OO language features, but at the same time we don't want to turn this into a 335 assignment, so we'll do something simple.

Write three complete, well-documented Ruby program files as detailed below. Note that objects of these classes just store time values, they don't 'keep' time like a real clock does.

(a) `clock.rb`: This file contains the definition of the class Clock. Clock uses Ruby's Time class to represent a 24-hour time (a.k.a. military time) in hours and minutes. (That is, 17 hours and 15 minutes is 5:15 p.m.) Its constructor accepts hours and minutes and uses those values to set the clock's time. Clock has a setter (`set_time(h,m)`) that also takes both hours and minutes, and two getters (`get_hour` and `get_minute`) to retrieve those components. The 'fun' method is `format_time`, which takes no arguments but returns a string that, when printed, shows the time as a digital clock does, in 12-hour time. For example, the string:

```
" _   _  _   _ \n|_|.|_||_| |_|\n|_|.|_||_| |\n"
```

will display as the (impossible, but illustrative) time:

```
 _  _  _   _
|_|.|_||_| |_|
|_|.|_||_| |
```

Of course, your version will need a fourth digit on the left. Note that the trailing character should be 'P' for PM or 'A' for AM, both of which can be formed from the seven bars available for forming digits. Each digit is three characters wide and three lines tall, and the A/P is separated from the last digit by a space, as shown. The 'colon' between the hour and minute is formed from a pair of periods.

(Continued...)

(b) `alarmclock.rb`: This file contains the definition of the class AlarmClock. AlarmClock inherits from Clock, and stores the time of the alarm, the setter `set_alarm(h,m)`, the getters `get_alarmhour` and `get_alarm_minute`, and the formatting method `format_alarm` (which produces the same sort of string that Clock's `format_time` does).

(c) `testclocks.rb`: This is to be a tester of your own design that exercises the methods of Clock and AlarmClock. The TA(s) will create their own tester for your classes, but a portion of your grade will come from their evaluation of the completeness of your tester.

This raises the question: "How much error-checking and error-handling do we need to do?" The answer is: Not much. If the given values for hours or minutes are outside of 24-hour ranges (0-23 for hours, 0-59 for minutes), set the instance variables to the closest legal value. For example, given an hours value of 26, set hours to 23.

Before anyone quibbles: The naming conventions for Ruby source code files aren't as well–defined as they are for internal names. Most people seem to use `alarm_clock.rb` rather than `alarmclock.rb` for a file containing the class `AlarmClock`. Your instructor has an aversion to underscores in all identifiers dating back to the days of dot matrix printers, when underscores often didn't get printed, which is why he's requiring you to name your files without them. He's stuck with accepting underscores in method names, as that's a well–defined Ruby naming convention.

Data: The input expectations for the programs are given with the program descriptions, above.

Output: Output details for each program are stated in the Assignment section, above. Please ask (preferably on Piazza) if you need additional details.

Hand In: You are required to submit your completed program files (your `.rb` files) using the `turnin` facility on lectura. The submission folder is `cs372p1`. Instructions are available from the document of submission instructions linked to the class web page. In particular, because we will be *grading* your program on lectura, it needs to *run* on lectura, so be sure to *test* it on lectura. Submit all files as-is, *without* packaging them into `.zip`, `.jar`, `.tar`, etc., files.

Want to Learn More?

- https://en.wikipedia.org/wiki/Collatz_conjecture — More info on the Hailstone sequence than you want to know!
- https://en.wikipedia.org/wiki/Insertion_sort — Wikipedia's description of Insertion Sort. Scroll down to the 'Variants' section to find Binary Insertion Sort.

Additional Hints, Reminders, and Requirements:

- There is a list of Ruby resources on the class web page, and other suggestions (irb, Googling) were mentioned in class. Working through statement syntax, execution errors, etc., are chores that we expect you to do on your own, as you would have to if you were learning a language on your own. But, if you get really stuck, the TA(s) and I are here to help.
- Worried about features of Ruby that we haven't (or won't) cover? You can either wait for those features to be covered, and/or you can explore them on your own. You're welcome to use features of Ruby beyond what we'll cover in class, so long as you are sticking to the expectations of the assignment. For example, if you get excited about creating a visualization of Hailstones, great, but make sure that your program also does what this assignment asks it to do.
- **AI Tool Usage:** You may use AI tools (LLMs, Generative AIs, IDE AI plug–ins, etc.) to help you write code on the programming assignments in this class. However, we *strongly* recommend that you use it to help you only when you get stuck, because you will be asked to write code on the exams. For that reason, write the code yourself, as much as possible, to help you learn the languages.
- **Start Early!** We don't expect that any of these programs will be conceptually difficult for you, but they will probably take some time to complete, due to your inexperience with the language.