CMSC 330 Project #2 Fall 2012

Date due: Monday, October 22, 10:00:00 p.m.

1 Introduction

A common use of scripting languages is to write programs that read and analyze the output of other programs, often in order to perform various administrative and maintenance tasks. In this project you will write a Ruby program that will read the hypothetical output of a program producing information about users who log in to a machine, and produce a report with various results based upon it.

The project's purpose is to use some of Ruby's data structures, which include arrays, hashes, arrays of arrays, hashes of arrays, hashes of hashes, etc., and perhaps classes implementing user–defined data structures. There are various ways the information needed could be stored using Ruby's data structures; consider different options and choose one you think will work well. The most obvious data structure may not work the best. Although you should evaluate different data structures before beginning to code, don't be afraid to change your mind if your approach doesn't work well. Of course, altering the data structures used will be easier if you begin writing your program in a flexible, modular, and well–structured fashion. Writing classes and subroutines or methods to perform tasks are ways to make a program modular.

It may be possible to implement some steps in your program very easily using methods of Ruby's library classes, particularly in situations where it may be difficult or tedious to write your own code. It would be worthwhile to read about those that may be applicable, especially the `Hash` and `Array` class methods. Some of the `String` methods could prove useful as well, and in addition the `Time` class could be quite helpful.

To get the project files log into the Grace systems, execute `cp -r ~/330public/proj2 ~/330`, and cd to `~/330/proj2` to start working.

Remember that important questions may be raised and answered in Piazza. Be sure to read all messages there often. However, keep in mind that you **cannot** ask or explain how to implement any part of a project (even a small part). You are expected to read all messages before posting questions. See the project grading policy on the class webpage for full expectations and restrictions related to Piazza.

In this project we may **deduct credit** for students making more than fifteen submissions. This is a 300–level course that is taken after students have had a good deal of programming coursework. Students at this level should be able to test programs thoroughly themselves.

2 Problem description

Suppose a friend of yours has graduated and gotten a job as a system administrator at a great company. The system administrators are responsible for keeping the company's systems running properly, securely, and reliably for the programmers and developers who work on them, as well as the users of the company's services. As the systems are running they generate a large amount of logging data, and for a large system this information is quite voluminous. It would be impossible for the system administrators to read through the data manually to look for potential problems, so they have written a number of programs (that often run automatically at regular periods) to analyze the data collected and inform them when there's something that might need further investigation.

Assume there is a program that automatically keeps track of every time someone logs in to one of the company's systems. The information collected includes the IDs of the users who log in, the date and time they logged in, the duration of their login session, and where they logged in from.

The hypothetical system administrators are interested in finding out several things from this data:

- They want to know how many times each user logged into the systems during the time the login data covers.
- They want to find out how many times each group of users logs in. On a UNIX system users can be members of different groups, which allows giving access to files to sets of users. Every user is a member of a default group, which can be used to aggregate users by category or type. If the system administrators see that certain groups of users are using the machines more heavily it might be desirable to give them higher priority, or possibly to consider buying new systems for their dedicated use.

- Lastly, for security reasons,they want to know whether any user logged in at the same time from more than one location. While it might not represent a problem if a user was logged in at the same time from more than place (for example, a user might have left themselves logged in from home and then also logged into the system when they arrived at work), it might represent a situation where an intruder was able to crack someone's password or take advantage of some vulnerability to break into someone's account and gain access to the system some other way, and the systems administrators want to be alerted to any such situations so they can look into them more closely.

3 Program input and output

3.1 Input

In the discussion below "whitespace character" refers to a blank space or tab character, and "whitespace" refers to a sequence of one or more consecutive whitespace characters.

Your program will read zero or more lines of standard input, each ending with a newline, and will also be run with two arguments on its command line representing the names of two auxiliary input files. Validity of the input and error conditions are described in a later subsection, after processing of valid input is described.

3.1.1 Standard input

The standard input represents the data produced by the hypothetical logging program about the logins that is to be analyzed, where each line has the data for one login session, consisting of five groups of information. Note however that one group of information has internal whitespace, so there will actually be seven whitespace–separated fields on each line (an arbitrary amount of whitespace should separate the seven pieces of information on a line). Whitespace may also appear before the first field or following the last field on a line.

- The first field represents the login ID of the user who logged in during this session, which should be a sequence of one to eight uppercase or lowercase letters or digits that begins with a letter.
- The second field contains the date when the user logged on, which is the day their login session began. This should consist of the common three–letter abbreviation of the day of the week the login began, the common three–letter abbreviation for the month it began, and a nonnegative integer for the day of the month the login began, which will obviously be between 1 and 31 (as above, all separated by whitespace). The first letters of the day and month names should be capitalized.
- The third field contains the starting time of the user's login session. Times are represented in 24–hour format using six digits, with colons between the hours and minutes and minutes and seconds, for example `16:07:00`, `05:19:10` or `00:21:20`.
- The fourth field represents the duration of the login session. For login sessions less than twenty–four hours in length this should be six digits inside parentheses consisting of the number of hours, minutes, and seconds separated by colons, as in `(03:20:14)` or `(00:18:27)`, but for login sessions of twenty–four hours or longer the number of days, (twenty–four hour periods) and a plus sign should precede the number of hours and minutes, for example `(2+03:04:53)`, indicating a login session that lasted for two days, three hours, four minutes, and fifty–three seconds. Notice that a user who logged in and logged out within the same second (a fast typist) would have a login duration that would appear as `(00:00:00)`).
- The last field represents the name of the machine the user logged in from, which should be a sequence of lowercase letters, digit characters, period characters, and dash characters beginning with a letter or a digit.

The lines of this file should appear in increasing chronological order by login start time. The input may span one or more month boundaries (so obviously it can span multiple days or weeks), but you may assume that the dates don't span a year boundary.

3.1.2 Password file

The first command–line argument should be the name of a file storing the login IDs of all of the users on the system, and the number of the primary group they belong to (each group on a UNIX system has a unique number). Login IDs are as described above, namely a sequence of one to eight uppercase or lowercase letters or digits beginning with a letter. The login ID on each line should be followed on the same line by a colon, an asterisk, another colon, the number of the user's default group, which should be a nonnegative integer, and a newline, so each user's ID and other data will appear on one line in this file. Note that multiple users may have the same primary group, and that no whitespace should appear other than the newline ending each line.

3.1.3 Group file

The second command–line argument should be the name of a file storing the names and information about the groups on the system. Group names consist of one or more lowercase letters. Each group name on a line should be followed by a colon, an asterisk, another colon, the number of the group, which as above should be a nonnegative integer, and a newline. Note that the name of a user's default group can be determined by finding their entry in the password file, extracting the group number that is the third field of that line, then looking in the group file for the line having that group number as its third field. As in the password file, terminating newlines are the only whitespace that should be present.

Note that group names and numbers should be unique in that no two lines of this file should have the same group name or number, but you cannot assume anything about the values of group numbers except that they are supposed to be nonnegative integers (for example, the group numbers may be in random order in the lines of this file and may not represent a consecutive sequence of integers). Note lastly that some groups in this file may be ones that no users have as their primary group.

3.2 Normal output

Your program should write all of its results to its standard output and (assuming there are no errors in the input) produce three sections of output.

3.2.1 Number of times users logged in

For the first section of output your program should print a line reading "Section #1:", (without the quotes), ending in a newline, spelled and punctuated exactly as shown, followed by zero or more lines of output each containing the login ID of a user followed by the number of times that user logged into the system. These lines must follow the **exact** format "`␣␣mickey:␣3␣time(s).`", without the quotes, where blank spaces are shown as ␣, for an example user with ID "mickey" who logged in three times. These lines are to be printed in increasing sorted order by login ID for the users who logged in at least once, where increasing refers to lexicographic or dictionary order, the same order that Ruby will compare strings using the `<=>` operator, or sort an array of strings using the `Array.sort()` method (which is done using `<=>`).

Each login session for a user is to be counted once, **except** if a login session spans a day boundary (begins **before** 00:00:00 and ends **at or after** 00:00:00) it is to be counted as **more than one time** that the user logged in. For example, if a user logged in at 23:30 on one day, stayed logged in for an hour, and logged out at 00:30 the next day, it would be counted as the user being logged in **twice**. Furthermore, a login session that lasts more than twenty–four hours could be counted as the user being logged in three or more times, if it spans the midnight boundary more than once. To determine the number of days a login session lasted, as well as to produce the results for the third section of output, you will have to add the login duration to the login start time on each line of the standard input.

A completely empty or blank line should separate this section of output from the next one.

3.2.2 Number of times members of groups logged in

In the second section of output your program should aggregate the number of times users logged in (for those who logged on at least once) by user groups instead of user IDs, using the user group information present in the password and group files. This section should consist of a line reading “Section #2:”, (without the quotes), ending in a newline, spelled and punctuated exactly as shown, followed by a sequence of zero or more lines, each containing a group name followed by the number of times that users who have that as their primary group logged in to the system. These lines must follow the exact format “`␣␣users␣group:␣3␣time(s).`”, without the quotes, where blank spaces are shown as ␣, for an example group with name “users” whose members logged in three times. The lines are to be printed in increasing order by group name.

Just as in the previous section, in this section logins crossing a day boundary are also to be counted as more than one login. A completely empty line should also separate this section of output from the next one.

3.2.3 Duplicate logins

For the third and last section of output your program should print a line reading “Section #3:”, (without the quotes), ending in a newline, spelled and punctuated exactly as shown, followed by zero or more pairs of lines. If you program detects that the same user logged in at any time while they were already logged in from **another** machine, it should print three lines in this section, followed by a completely empty line; these should be of the exact format illustrated by this example (where spaces are shown as ␣):

```
␣␣mickey:
␣␣␣␣Wed␣Sep␣12␣15:00:00␣03:20:00␣914-45a.umd.edu
␣␣␣␣Wed␣Sep␣12␣17:00:00␣04:01:00␣libwkmck1f2.umd.edu
```

In other words, this means that mickey logged in at 17:00:00 on Wednesday, September 12, from the host indicated in the second line, while already logged in starting at 15:00:00 on September 12 from the host indicated in the first line. If one login time ends at exactly the same second that another one begins they are considered to overlap.

The data that should appear on these lines should be evident from the example above, except that a duration of twenty–four hours or greater should be indicated with a number of days and a preceding plus sign, as for example `1+01:03:04`. Note that the hours, minutes, and seconds should be printed with exactly two digits, with a leading zero if less than ten, but not the days. The first line of the pair should be the login time that began earlier, which should be the same order these two sessions appeared in the standard input. If two logins began at exactly the same time then the one appearing earlier in the input should appear first. Each such pair should be **followed** by a blank line, so unless there are no overlapping logins at all the last such pair printed in this section will be followed by a blank line, and blank lines will separate all such pairs.

The order that these pairs of lines should be printed in, if there is more than one pair, is in increasing chronological order by the original login start time, with pairs for multiple duplicate login sessions that have the same starting time ordered by the start time of the overlapping or later login session. In other words, your program must scan through the login information in chronological order of start time (the order of the standard input), and for each login session that occurs scan through all the login sessions with later start times to see if they began while this one was still going on. Notice that these pairs of lines will not be sorted by login ID, but chronologically; if multiple users have multiple overlapping login sessions then their pairs of output lines may be separated by pairs of lines for other users.

Note that a pair of output lines is to be generated for **every** pair of overlapping logins. For example, if a user logged in for login session A, then logged in later from another host for login session B that overlapped chronologically with A, then logged in again from another host for login session C, from a host different from the two hosts in A and B, at a time that overlaps with both A and B, this section would have a pair of lines for sessions A and B, then a pair of lines for sessions A and C, then a pair of lines for sessions B and C.

There will be no lines in this section of output (other than the heading, terminated by a newline) if no users logged in at the same time from different hosts. Note that a user can be logged in multiple times from the **same** host concurrently, or at **nonoverlapping** times from different hosts, and no output should appear in this section as a result, it’s only logins at overlapping times from different hosts that are a concern.

3.2.4 Invalid input and error conditions

- Unless exactly two command–line arguments are present, or if either argument (or both) does not refer to a file that is present and can be successfully opened, your program should print a single line reading "Invalid argument." (without the quotes), ending in a newline, spelled and punctuated exactly as shown, and quit without producing any further output.
- Any lines in the password file or group file that are invalid (do not match the format described above) should be silently ignored (silently ignored means no error message should be printed). If any user ID appears in more than one line of the password file, the second and subsequent occurrences are just to be ignored; only the first line for a user ID is valid. Similarly, if the same group name or number appears in more than one line of the group file only the first one is valid and the rest should be silently ignored.

  If after ignoring invalid lines from the password file there are no valid lines remaining in it, your program cannot produce correct results in all cases, so it should just print a single line reading "Invalid file.", (without the quotes), ending in a newline, spelled and punctuated exactly as shown, and quit without producing any further output. The same applies if there are no valid lines in the group file after ignoring invalid ones. Since some output may have already been generated when either of these situations is detected, any preceding output doesn't matter (this message should just be the last line of output in either of these cases).
- Your program cannot work correctly if any user's login ID does not appear anywhere in the password file or if any group number from the password file does not appear anywhere in the group file. It doesn't matter what results are produced in these cases.
- If any lines from the standard input are "syntactically" invalid in any way (wrong number of fields, or any field is not of the format described above) the line is also just to be silently ignored. Lines that are "semantically" invalid should also be ignored, meaning ones that have the right number of fields of the proper form, but their values are incorrect given the descriptions above. For example, lines with hours not in the range 0..23, minutes not in the range 0..59, incorrect day numbers for months (such as a line with Sep 31 in its start time), etc., should all be ignored. Since the input does not have an indication of the year, just assume it is not a leap year.

  If any line in the standard input has a date for its start time that is (strictly) earlier than the preceding line it is also to be silently ignored. It's not incorrect if two login sessions begin at exactly the same time.
- It's not an error if the program's standard input is empty (has zero valid lines), either because it has no lines, or because it has no valid lines (after ignoring invalid ones). In this case the output would just consist of the three section headings with two blank lines between them (assuming the password and group files were valid).
- Since the input does not have an indication of the year, your program should just assume that the date indicated actually does fall on the day of the week given. For example, if a line of the standard input contains Wed Sep 12 you just have to assume that September 12 was actually a Wednesday during whatever year this data was generated.
- Other than the fact that dates in the lines of the standard input must appear in nondecreasing chronological order, your program does not have to enforce any cross–line validity. For example, one input line could have "Mon Sep 10" in its login start date field, and the next line could have "Wed Sep 13", and although these dates could not possibly occur in the same year, you should just consider the dates valid. (Note that the day names are not used in producing any of the three output sections above, and have no effect other than possibly causing some input lines to be ignored if they are incorrect).

4 Example

The example below assumes that the first line of the Ruby program in the file "`proj2.rb`" contains the full pathname of the Ruby interpreter preceded by the characters `#!`, the UNIX command `chmod` has been used to make the file executable, and the UNIX prompt on this system is `grace2:~/330/proj2:`.

```
grace2:~/330/proj2: cat login-data
bugs     Wed  Sep  12  15:00:00  (04:00:00)  914-45a.umd.edu
bugs     Wed  Sep  12  17:00:00  (04:01:00)  libwkmck1f2.umd.edu
tweety   Wed  Sep  12  17:22:00  (07:08:00)  c-69-25-171.md.comcast.net
bugs     Wed  Sep  12  18:00:00  (00:01:00)  209.39.175.23
bugs     Wed  Sep  12  20:00:00  (00:02:00)  coredump.umd.edu
tweety   Wed  Sep  12  21:01:00  (03:29:00)  c-69-25-171.md.comcast.net
mickey   Wed  Sep  12  23:58:00  (00:01:00)  c-68-33-204.md.comcast.net
donald   Thu  Sep  13  10:11:00  (00:20:00)  wireless-20-16-67.umd.edu
tweety   Thu  Sep  13  12:20:00  (00:01:00)  ptx-dual-v1.net.umd.edu
donald   Thu  Sep  13  13:08:00  (00:11:00)  ptx-dual-v1.net.umd.edu
jessica  Thu  Sep  13  16:46:00  (00:05:00)  ptx-dual-v1.net.umd.edu
jessica  Fri  Sep  14  03:36:00  (00:00:00)  pool-151.west.verizon.net
goofy    Fri  Sep  14  14:06:00  (00:04:00)  mastercoder.student.umd.edu
mickey   Fri  Sep  14  17:31:00  (00:12:00)  ptx-dual-v1.net.umd.edu

grace2:~/330/proj2: cat passwd-file
bugs:*:3
donald:*:2
goofy:*:1
jessica:*:3
mickey:*:2
tweety:*:2

grace2:~/330/proj2: cat group-file
root:*:0
sysadmin:*:1
users:*:2
devel:*:3
webmaster:*:4

grace2:~/330/proj2: proj2.rb passwd-file group-file < login-data
Section #1:
  bugs: 4 time(s).
  donald: 2 time(s).
  goofy: 1 time(s).
  jessica: 2 time(s).
  mickey: 2 time(s).
  tweety: 5 time(s).

Section #2:
  devel group: 6 time(s).
  sysadmin group: 1 time(s).
  users group: 9 time(s).

Section #3:
  bugs:
    Wed Sep 12 15:00:00 04:00:00 914-45a.umd.edu
    Wed Sep 12 17:00:00 04:01:00 libwkmck1f2.umd.edu

  bugs:
    Wed Sep 12 15:00:00 04:00:00 914-45a.umd.edu
    Wed Sep 12 18:00:00 00:01:00 209.39.175.23

  bugs:
    Wed Sep 12 17:00:00 04:01:00 libwkmck1f2.umd.edu
    Wed Sep 12 18:00:00 00:01:00 209.39.175.23

  bugs:
    Wed Sep 12 17:00:00 04:01:00 libwkmck1f2.umd.edu
    Wed Sep 12 20:00:00 00:02:00 coredump.umd.edu
```

5 Development suggestions

- You may want to write your program to handle normal input first, then add checks for invalid input. This is because some checks for invalid input may require the data is already stored in some manner to be detected, and also because presumably more weight will be given to tests of normal cases than to error cases.
- You can perform date and time calculations more easily than you might think using Ruby's `Time` class. In particular, `Time.local()` creates a `Time` object and the `+` operator adds a number of seconds to a `Time`. Note that `Time`s can also be compared using the `<=>` comparison operator. If you print a `Time` object its `to_s()` method will be invoked, which (in the version of Ruby on the Grace machines) will result in a string of the form `2012-09-12 20:00:00 -0400`. However, the `Time` class `strftime()` method can be used to format times in a large variety of ways.
- As mentioned in discussion section, Ruby has an integrated debugger, which can be invoked by running Ruby with the `-rdebug` option before the name of your program (e.g., `ruby -rdebug proj2.rb`). The debugger's '`p`' command may be very helpful, as it prints the components of a variable or data structure (including arrays and hashes), in a formatted fashion, including any nested data structures. The '`var local`' command prints all of the local variables at the current point of execution. The chapter "When Trouble Strikes" of The Pragmatic Programmer's Guide linked to on the class webpage discusses the debugger in a bit more detail.

6 Project requirements and submitting your project

1. Your program should read all of its input from its standard input and the two files whose names appear on the command line, and write all of its output to its standard output. Of course in UNIX standard input may be redirected from a file, or standard output may be redirected to a file.
2. You may use any Ruby language features in your project that you would like, regardless of whether they were covered in class or not, so long as your program works successfully using the version of Ruby installed on the OIT Grace Cluster and on the CMSC project submission server.
3. Your submitted program file **must** be in a single file named "`proj2.rb`", otherwise the submit server will not recognize it, consequently its score will be zero.
4. To check that your output matches the expected output you can use the UNIX `diff` command, for example:

   ```
   proj2.rb passwd-file group-file < login-data > my-output
   diff -bB public1.output my-output
   ```

   If no output at all is produced by `diff`, your output matches the expected output and is correct. This `diff` command is exactly what the submit server is going to be using to check your program's output, so using it you can see yourself whether your output is right or not; if you don't get matching results when comparing your output this way then the submit server is not going say your results are right either.

   Two notes: the diff options `-bB` ignore differences that consist of only blank lines and amount of whitespace, since if all the data in your output matches the expected output we'll call your results right even if some whitespace doesn't exactly agree. Secondly, you may prefer `diff`'s unified output format, which you can see instead by using the options `-bBu`. In the unified format surrounding lines of the files being compared are shown (not just the lines that differ), with lines that should not be present in the first file preceded by `-` and lines that are not missing from in the first file preceded by `+`. (What is being compared is exactly the same whether the unified format is used or not, the results may just be easier to follow.)
5. You aren't required but are encouraged to use Ruby's `-w` option as it will assist in finding potential bugs.

6. Your program **must have** a comment near the top that contains your name, TerpConnect login ID (which is your directory ID), your university ID number, and your section number.

   The Campus Senate has adopted a policy asking students to include the following statement on each major assignment in every course: "I pledge on my honor that I have not given or received any unauthorized assistance on this assignment." Consequently you're requested to include this pledge in a comment near the top of your program source file. See the next section for important information regarding academic integrity.

7. Note that although you are not being graded on your source code or style (see the separate project grading handout), if the TAs cannot read or understand your code they cannot help should you have to come to office hours, until you return with a well–written and clear program.

8. As before, to submit your program just type the single command `submit` from the `proj2` directory that you copied above, where your program file `proj2.rb` should be. As mentioned above you may lose credit for having more than fifteen submissions.

7 Academic integrity

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus– please review it at this time.