UNIX Data Tools

Buffalo Chapter 7

Overview

In Chapter 3 we learned the basic operations within the Unix shell:

- standard out and standard error streams of data
- how to redirect our data streams
- how to efficiently run a series of commands using pipes
- how to manage command processes

Here, we'll learn a number of UNIX tools that will allow us to inspect and process data

Inspecting a data file for the first time:

- Use the command to navigate into the folder in the Buffalo online resources
- We can inspect a file by using the command to print its contents to the screen:

- That's a little unwieldly...perhaps we just want to see the first few lines of a file to see how it's formatted. Let's try:

- If we want to see less or more of a given file, we can specify the number of lines using the option:

Inspecting a data file for the first time: `  `

- Similar to `  `, you can use the `  ` command to inspect the end of a file:

[redacted]

- `  ` can also be useful for removing the header of a file; this is particularly useful when concatenating files for an analysis:

[redacted]

- And here's a handy trick for inspecting both the head and tail of a file simultaneously:

[redacted]

Additional uses of

- We can also use to inspect the first bit of output of a UNIX pipeline:

- When including head at the end of a complex UNIX pipeline, the pipeline will only run until it produces the number of lines dictated by
- Why is this important or useful? This dummy pipeline may help:

Inspecting files and pipes using

- is what is known as a "terminal pager"; it allows us to view large amounts of text in our terminal
- Whereas with the contents of our file flash before our eyes, with we can view and scroll through the file's contents
- Let's observe the difference between and using a file from the Buffalo Chapter 7 materials:

Try:

Then try:

- While viewing the file in try navigating with the space bar and the , , , , and keys. To exit the file, press

Using to highlight text matches and check pipes

- Highlighting text matches can allow us to search for potential problems in data
- For example, imagine we download useful Illumina data from another study and it's not clear from the documentation whether adapter sequence has been trimmed
- We can search for a known 3' adapter sequence using :

- can also be used to check the individual components of a pipe under construction:

- The commands will only run until a page of your terminal is full, limiting computation time

Inspecting files using the `` command

- The default of `` is to provide the number of lines, words, and bytes (characters) in a file:

- Each line of data entry in the .bed file should correspond to a single line of data entry in the .gtf file. Notice any problems?
- Using ``, see if you can inspect the two files and resolve this issue
- The discrepancy in the line numbers, may have been more clear had we only inspected the number of lines:

Inspecting file size using the `` and `` commands

- Before downloading or moving or running an analysis on a file, it is useful to know the file size
- There are a few ways we can extract this information
- First, we can use our old friend, the `` command with the `` and `` options:

[redacted]

- Or we can use the `` command, also with the ``, or "human readable" option:

[redacted]

- Personally, I prefer the less verbose format of ``, particularly when inspecting a large number of files

Inspecting the number of columns in a file with

- Another useful piece of information we may want to know about a file is its number of columns
- We could find this by visually inspecting the first line of the file, but this opens us up to human error:

- A better solution is to have our computers count the columns for us using an one-liner:

- is a bit different than some of the basic UNIX commands we've been learning…it is actually a simple programming language in itself…we'll come back to it in more depth later

Number of columns in files with headers

- Our handy script works well for the file, but what about for the ?
- We can get around this issue by employing the command we learned earlier:

- In the Buffalo book, this one-liner outputs that there are 16 columns…is this what you get?
- Thinking back to the first few chapters in Buffalo and our discussion regarding "robust" and "reproducible" code, why might this be considered a "brittle" solution?
- Can you think of a more robust solution?

- How might this be a brittle solution?

Using the `` command to extract specific columns

- On occasion, we will want to extract a subset of specific information from a file
- The `` command assumes tab delimitation and allows us to extract specific columns of a tab-delimited file
- For example, say we wanted just the start positions of the windows in our .bed file:

Using the `cut` command to extract specific columns

- The `-f` option allows us to specify columns in ranges (e.g., `-f 1-3`) and sets (e.g., `-f 1,3,5`) but ***DOES NOT*** allow us to order columns (e.g., `-f 3,1,5`)
- For example, we can extract chromosome, start site, and end site from our .gtf file by first removing the header and then cutting out the first, fourth, and fifth columns:

[redacted]

- We can also specify the delimiter in differently formatted files like .csv:

[redacted]

Tidying things up with

- Often times, when we inspect a tab-delimited file with , the results are fairly messy:

- This can make it difficult to understand file contents
- Fortunately, there's a UNIX program/option combination to tidy things up:

- should only be used for file inspection in the terminal, redirecting its standard out to a file will introduce variable numbers of spaces which could cause problems in downstream analysis
- can also be used with files with other delimiting characters:

: one of the most powerful UNIX tools

- Thus far we've only scratched the surface of the utility of
- In addition to being useful, is *fast*

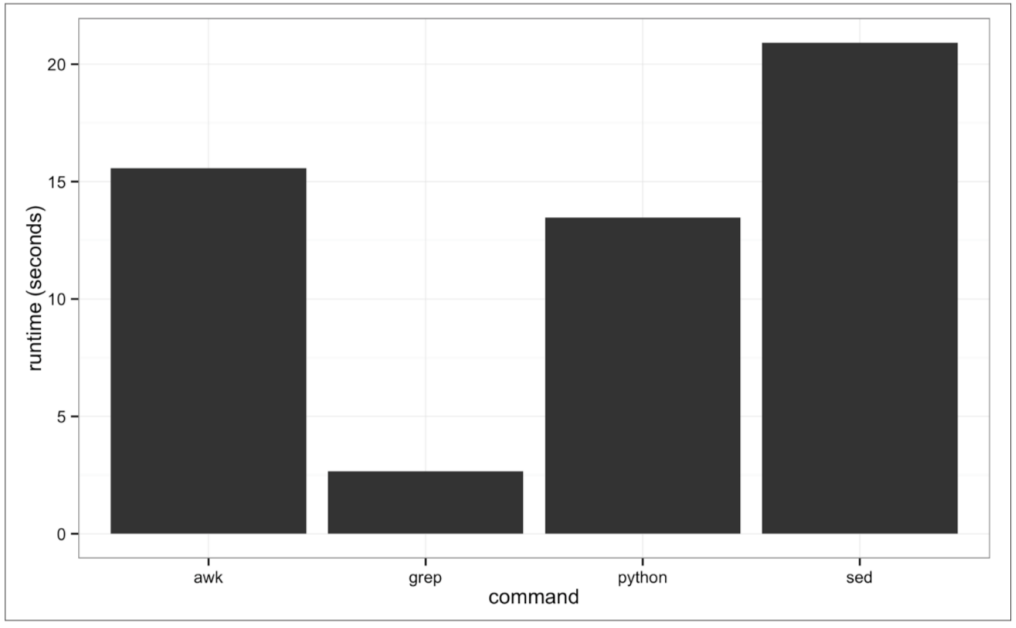


Figure 7-2. Benchmark of the time it takes to search the Maize genome for the exact string "AGATGCATG"

: one of the most powerful UNIX tools

- The program requires a *pattern* to search for and a *file* to search through:

- Quotes around the pattern prevent our shell from trying to interpret symbols in the pattern
- will also return partial matches:

- If a partial match is not desired, we can prevent this using the option which matches entire words
- For example, in the file we want to match everything but "bioinfo":

: one of the most powerful UNIX tools

- General rule: always be as restrictive as possible to avoid unintentional matches
- If the matching line itself does not provide enough context, the and options can be helpful:

[redacted]

- search patterns can also be made more flexible and powerful with *Basic Regular Expressions* (BRE) and *Extended Regular Expressions* (ERE)
- An example of a BRE:

[redacted]

- An example of an ERE:

[redacted]

Additional uses with various options

- Say we're interested in the number of small nuclear RNAs in our set of genes:

- Or perhaps we only want to extract the word matches to our search pattern, not the entire line:

Identification of non-ASCII files and characters

- In bioinformatics, many programs will assume that our input text files are encoded in ASCII
- Occasionally, often due to human manipulation of data files, our data can include include an invisible non-ASCII character that throws our program for a loop
- To easily determine whether a given file is encoded in something other than ASCII, the `` command can be quite useful:

Illustrating the trouble a non-ASCII character can cause

- To show how non-ASCII characters can cause problems, we'll install the program `` from github
- If you're working on hpc-class:

[redacted]

- If you're working on a directory on your own machine:

[redacted]

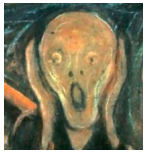
- Or if you've installed `` :

[redacted]

Illustrating the trouble a non-ASCII character can cause

- Now let's apply the following `` one-liner which should produce the reverse complement of our sequences:

- Shoot… `` choked on our second sequence…non-ASCII character!!



Sleuthing out our non-ASCII character with

- will identify the problematic character and the option will print the character as well:

Sorting plain-text data with

- Sorting plain text data can be necessary because:

[redacted]

- First, let's sort the file without options to see if we can figure out how the default program works:

[redacted]

- Options allow us to sort by specific columns in various orders and to tell that our data are numeric rather than alpha-numeric:

[redacted]

- Now see if you can figure out how to sort the file, first by chromosome, then by window start site

Additional features of

- Since sorting very large files can be computationally intensive, we may want to check whether a file is already sorted first using the option:

- We can also sort files in reverse order using the option:

- But how is this sorting?
- Can you think of a way to sort in reverse order based on both columns 1 and 2?
- What if we want to sort in forward order by column 1 and reverse order by column 2?

Advanced sorting options in GNU

- The option can recognize numbers inside of strings…how might this be useful?
- Inspect the entire file:

[redacted]

- Why might we want to recognize numbers within a string here?

[redacted]

- In the event that you want to sort a truly enormous file, there are modifications to that can be applied to allocate more memory to the program:

[redacted]

Finding unique values using the program

- After first inspecting the entire file, run the program on this file and see if you can understand how this program works

- What do we need to do to get a truly unique list of letters?

- And what if we want unique values but still want want a count of each letter?

- And if you're still not convinced that this could be useful, try this:

Finding unique values using the program

- The output can also be sorted based on entry counts by piping to and using the option:

- What if you wanted these listed from most to least common in the file?
- We can also use the combination of and to gather information from multiple columns in a file:

- Or we can use these programs to process and inspect a subset of data from a file…for example, all the features associated with a particular gene:

Merging the contents of two files with the program

- The contents of two files can be merged by joining the files based on a common column
- In the following two files, what would be the common column to use for a join?

[redacted]

- In order to complete the join, we must first sort *both* files on the common column

[redacted]

- Let's talk through the following syntax to make sure it's clear:

[redacted]

Merging the contents of two files with the program

- Let's also look at the number of lines in our files to see if the join was complete:

- Now let's see what happens when there is not complete overlap in our common columns:

- Because chr3 is absent from the file, it is omitted entirely from the join
- The GNU option allows us to include these "unpairable" lines in our output file:

Processing data with the programming language

- Unlike the UNIX programs we've been learning, is a full-fledged programming language
- is simpler then and not built for complicated tasks, but it's great for quick data-processing tasks
- To learn we must understand how it:
  1. Processes records
  2. Uses pattern-action pairs
- processes data a record at a time and records are composed of fields
- assigns the entire record to variable , field 1 to , field 2 to , etc...
- In pattern-action pairs, first tries to match a specified pattern in a record or field and, if this is successful, the specified action is carried out

Processing data with the programming language

- We can mimic the program with by omitting the pattern component of a pattern-action pair:

[redacted]

- Similarly, we can also mimic :

[redacted]

- Standard arithmetic operators (+, -, *, /, etc...) can be used in the pattern component of pattern-action pairs
- For example, here our pattern is matching .bed file features that are at least 18bp long and the implicit action is to print matches to standard out:

[redacted]

Processing data with the programming language

- We can also link patterns in a chain to apply multiple conditions in our pattern using the (AND), (OR), and (NOT) operators
- For example, if we want the .bed features that are on chromosome 1 AND at least 10bp long:

[redacted]

- We can also include more explicit actions than just printing an entire record to standard out:

[redacted]

Additional functionality of the programming language

- The pattern-process tools we have learned thus far are very useful for processing files, but has many more useful tools
- The and commands can allow us to initialize variables before implementing our pattern-process across records ( ) and use this variable afterwards ( ):

- Here we initialize the variable and increment ( ) this variable by the length of each feature across all records and then divide this by ...what is ?
- can also be used to extract intermediate records (i.e., lines) in a file (the same process we discussed using and in a pipe):

Additional functionality of the programming language

- can also be used to convert a .gtf file into a .bed file:

- Note that the start site of features in the .bed file is 1 less than the start site of features in the .gtf file: .bed uses 0-indexing and .gtf uses 1-indexing

- Associative arrays (similar to dictionaries) can also be very useful in :

- Could this also be done with basic UNIX commands?

: functionality more tailored to bioinformatics

- is similar to but it can recognize common bioinformatics file formats (e.g., .bed, .sam, .vcf, .gff, .fastx) and includes useful programs for bioinformatics

- You could also use to convert a into a file:

- Or to print the number of sequences in a file, something you couldn't do with :

: functionality more tailored to bioinformatics

- Finally, the option can be very useful as it sets the field variables to the names given in a file header
- For example, take another look at the file:

- Let's use the option to find the markers where ind_A and ind_B have the same genotype:

Using the `sed` program to edit text in a stream

- In addition to many other functions, we can use `sed` to make simple "find and replace" edits to our files:

[redacted]

- If this file were many Gb in size, this stream editing would be much, much faster than opening the file and doing a find and replace in a text editor
- The above syntax only substitutes the first occurence of "chrom" on a line, to do this across all "chrom" values we'd need to use the global option of `sed`:

[redacted]