Analysis of Next Generation Sequencing Data
Introduction to Unix continued

Luce Skrabanek

22 January, 2019

1 Regular expressions

The `egrep` command is used to find patterns, or 'regular expressions' (RE) in text. Regular expressions are patterns that can match a family of character strings. The simplest RE is an exact textual match of a word, but you can also look for more general patterns. These regular expressions can appear cryptic, but they are very powerful.

Let's use a PDB file to illustrate the `egrep` command. To show only the TITLE lines, we can search with a specific string:

```
egrep TITLE 1TAW.pdb
```

But searching for a specific string like this can get you into trouble. Say we wanted to count how many atoms there are in the 1TAW structure.

```
egrep ATOM 1TAW.pdb | wc -l
```

Congratulations: you just shot yourself in the foot. Why? Try:

```
egrep ATOM 1TAW.pdb | less
```

What we need to do is search for all of the lines that **begin** with ATOM (not just that contain ATOM anywhere on the line).

```
egrep '^ATOM' 1TAW.pdb | wc -l
egrep '^ATOM' 1TAW.pdb | less
egrep -v '^ATOM' 1TAW.pdb
```

The apostrophes ensure the entire expression is evaluated by `egrep` instead of by the user's shell.

The `egrep -v` option reverses the sense of the match, printing those lines that do not match the specified pattern. This is an important part of testing your regular expressions. It is just as important to know about the false negatives as it is to know about the false positives.

There are other characters that have special meanings within the context of the regular expression.

.	matches any single character.
()	group the enclosed items into a single item.
*	matches zero or more occurrences of the preceding item.
+	matches the preceding item one or more times.
?	matches the preceding item zero or one times.
^	matches at the beginning of a line only.
$	matches at the end of a line only.
[]	matches any one occurrence of the characters enclosed within the brackets. If a hyphen is included, this indicates a range. All of the above special characters lose their special meaning within the brackets. ^ takes on another special meaning within these brackets. If ^ is the first character within the brackets, this indicates that the string should match any character except those in brackets.
{m}	matches m occurrences of the preceding item.
{m,n}	matches anywhere between m and n occurrences of the preceding item.
\	makes all of the above special characters lose their special meaning (including itself). This is called 'escaping out' the special character. Its special effect is also lost within brackets, as above.

The regular expression will match the longest pattern it can in the piece of text that you are looking for the regular expression in.

```
egrep l.*g
# Matches along, algorithm, log, long, loving, leading.

egrep \.
# Matches a period

egrep e.$
# Matches 'He saw her in the tree', 'The man was me.', 'Well met', but
   not 'Did you know her?'

egrep [^a-m]nd
# Matches 'And then I saw her face' 'What a wonderful world it is'
```

Other common options for `egrep` include:

1. `egrep -i`
   Makes the regular expression case-insensitive.
2. `egrep -c`
   Prints the number of lines that match the search criteria.
3. `egrep -n`
   Prints out the line number on which each match is found.
4. `egrep -f`
   Read one or more newline separated patterns from file, and searches for each of them.

2 Text editors

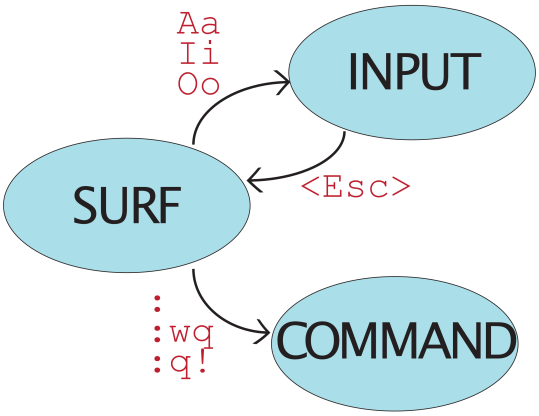
There are two text editors which are widely used: `vi` and `emacs`. Every Unix system has some version of vi installed, and most system administrators install emacs. Which one you use is a matter of preference (or any other that is available to you). Here, we will go into some detail in the use of vi.

2.1 vi (visual display editor)

There are three 'modes' in vi:

1. 'surfing' mode, in which you can move around the file and delete text
2. 'input' mode, where you can insert new text into the file
3. 'command' mode, where you can substitute text, search for text, delete text, access the shell, and save changes to the file.

From 'surfing' mode, you can access either 'input' mode, or 'command' mode. To exit input mode, pressing [Escape] brings you back to surfing mode. Command mode is only active for one command, after which you are returned to surfing mode.



To open an existing file myfile in vi, type `vi myfile`. The file will be opened in surfing mode. The cursor is initially set at line 1, character 1 of your text.

2.1.1 Surfing mode

In surfing mode, the most common movement keys are:

up	k, or up-arrow
down	j, or down-arrow
left	h, or left-arrow
right	l, or right-arrow
scroll up a page	Ctrl-b (Back)
scroll down a page	Ctrl-f (Forward)
last line of the file	Shift-g (G).
advance a word	w or W (can you spot the difference?)
go back a word	b or B
go to a pattern	/ or ?, followed by a regular expression. If the word appears more than once, you can go to each subsequent occurrence of the word by repeatedly entering `n`.

You can also 'Cut', 'Copy' and 'Paste' text in surfing mode: (x refers to a single character, w to a word, and doubling the command refers to a line)

```
x   # delete the character under the cursor
4x  # delete the character under the cursor and the next three
    characters
dd  # delete the whole line that the cursor is on
dw  # delete to the end of the word
5dd # delete the current line, and the next four lines
D   # Shift-D; delete the line from the cursor position to the end
```

If a line is deleted like this, it is kept in memory until the next command, like a 'Cut' command in a word processor. To put the deleted line(s) somewhere else in the text, type `p`, and the line is inserted under the line you are currently on (like 'Paste' in a word processor). `P` [Shift-P] will insert the deleted line above the current line. If you deleted characters or words instead of lines, then the text will be placed just after your cursor position. To 'Copy' characters, words or lines, use `y` (yank) instead of `d`.

Other miscellaneous commands in surfing mode include:

```
r     # (replace) allows you to replace the character your cursor is
      # on (e.g., rt replaces the current character with t).
J     # Shift-j joins the line the cursor is on, and the line below it.
0     # moves to the beginning of the line
$     # moves to the end of the line
```


2.1.2 Input mode

There are three main ways to insert text:

`a`	(append) allows you to start typing after the cursor position
`A`	allows you to start typing at the end of the current line
`i`	(insert) allows you to start typing just before the cursor position
`I`	allows you to start typing at the beginning of the current line
`o`	(open line) makes a new line below the line you are currently on, and allows you to start typing at the beginning of that new line
`O`	opens a new line for typing above the line you are currently on

To get out of insert mode and back into surfing mode, hit [Escape].

2.1.3 Command mode

To enter command mode, type `:`. The command prompt will appear at the bottom of your editor screen. There are five commonly used commands:

```
# Moving to a specific line
:<number>          # takes you to that line in the file. You can find
   out what line the cursor is on using Ctrl-G in surfing mode.
# You can also use relative expressions:
:+9                # moves you 9 lines down
:-6                # moves you 6 lines up

# Inserting another file
:r <filename>      # copies filename into the file being edited, beneath
    the current line

# Deleting lines
:<line1>,<line2>d # deletes all lines from line1 to line2
:.,$d # deletes all lines from the current to the last

# Substituting text
:<line1>,<line2>s/<pattern1>/<pattern2>/[g] # substitutes pattern2 for
   pattern1 in the file between line1 and line2. Pattern1 can be a
   string or a regular expression. The (optional) `g' at the end
   indicates that this substitution should be done globally (per line)

# Saving and exiting from vi
:w                 # (write) saves the file
:w <filename>      # saves the file to filename
:wq                # (write and quit) saves the file and exits vi
:q!                # exits vi without saving any changes
```


Typing `u` (undo) will undo the last command.

There is also a `sed` command that will allow us to edit a file non-interactively (e.g., within a shell script).

```
sed 's/|/_/g' P04637.fasta # will change all occurrences of "|" to "_".
```

3 Starting shell scripting

Shell scripts are small programs. They let you automate multi-step processes, and give you the capability to use decision-making logic and repetitive loops. Most UNIX scripts are written in some variant of the Bourne shell; we will use `bash` here.

All scripts should begin with a 'shebang' (`#!`) with the name of the shell that will execute the commands in the file. Comments begin with a hash. You can use all of the UNIX commands you know in your scripts. The results of commands in backticks (upper-left of your keyboard) are substituted into commands. Use a `$` before variable names to use the value of that variable.

3.1 Variables

Variables are useful tools within shell scripts. Variables are names that have a value that can vary (hence 'variable'). Variables can be environment variables, or local variables. Some of the more common environmental variables are: `$HOME`, `$PATH`, `$USER`, `$SHELL`. Local variables are variables that are created by the user, usually only lasting within a particular session, or within a shell script. The value of a variable can change during the course of the execution of a shell script.

In the Bourne shell, we assign a variable like this:

```
variable=value
echo $variable
```

There are NO whitespace characters around the `=`. To use that variable, we can then type `$variable`, and the shell will substitute that with `value`. If the variable name is immediately followed by other characters that could be interpreted by the shell as being part of the variable name, the variable name itself is enclosed in curly brackets, e.g.

```
> verb=paint
> echo I like ${verb}ing

I like painting

> echo I like $verbing
I like
```

We can also assign the output of a command to a variable.

```
atomcount=`egrep -c "^ATOM" 1TAW.pdb`
atomcount=$(egrep -c "^ATOM" 1TAW.pdb)

echo "There are $atomcount atoms in the 1TAW.pdb file"
```


3.2 For loops

The `for` construct allows a list of commands to be executed several times, using a different value of a loop variable in each iteration. Note that we are using the result of the `ls` command to generate the list of things to loop over.

```
for file in `ls *.*`; do
  wc -l $file
done
```

For last week's homework, we had you download the chromosome size files for the three genome assemblies for *S. cerevisiae*. For three files, this is trivial to do individually, but it is possible to do this with a for-loop. Using a for-loop can make your intent clearer, and minimizes typos.

```
for assembly in 1 2 3; do
  wget http://hgdownload.cse.ucsc.edu/goldenPath/sacCer${assembly}/
     bigZips/sacCer${assembly}.chrom.sizes
done
```

3.3 Integer math

You can do basic integer arithmetic in your scripts, but using a double parentheses construct, and use variables inside these expressions.

```
echo $(( 5 + 9 ))
gccontentMin=$(( ( $gcMin * 100 ) / $total ))
```

