Chapter 4

Stream editing

The Unix stream editor, `sed`, is a program that takes text as input, one line at a time, loads it into a buffer (called "pattern space" in the jargon and manual pages), applies some editing instructions to that one line and then either writes it out, deletes it or stores it in some internal memory ("hold space").

The applications of `sed` in this course will not use the hold space. Once your requirements get to the stage that you need the hold space, the author recommends that you switch to using `perl`.

The directory `/info/courses/text.tools.course/sed-examples` contains a file `fruit` that will be used for the examples in this section.

4.1 Search and replace

The most common use of `sed` is to find all instances of some string or regular expression and to replace it with some other string or expression based on the matched expression.

```
$ cat fruit
apples apples
bananas bananas
lemons limes
oranges oranges
pears pears

$ sed -e 's/apple/plum/' < fruit
plums apples
bananas bananas
lemons limes
oranges oranges
pears pears
```

If the results were wanted in a file, **newfruit** say, then the command to be given is sed -e 's/apple/plum/' ¡ fruit ¿ newfruit This command has taken the file `fruit` as input and, for each line line within it, replaced the first occurrence of "`apple`", if such exists, with "`plum`". Note that the line that has two instances of the word "`apples`" only had the *first* instance edited.

In the example above, `sed` was passed a command with the "`-e`" option. For a simple command like this, the quotes were not needed to protect the command from the shell, but later commands will require this protection and it does no harm so we will use quotes throughout.

In its simplest use, `sed` is passed a single command as an argument[1]. This command applies to a single line at a time. `sed` takes its standard input, one line at a time, applies the command and outputs the resulting line.

The "`s`" `sed` instruction means "*s*ubstitute". It takes two arguments which come between the slashes. The first is the string to be substituted and the second is the string to act as the replacement. Without any options, this will only replace the *first* occurrence of the target string.

```
$ sed -e 's/apple/plum/g' < fruit
plums plums
bananas bananas
lemons limes
oranges oranges
pears pears
```

The "`g`" at the end of "`s/apple/plum/`" acts as an option to the substitution. The "`g`" stands for "*g*lobal" and causes *all* instances of the target string "`apple`" to be replaced with "`plum`" in each line.

Exercise 17: Replace "`oranges`" with "`grapes`".

If the replacement string provides part of the replaced string, the newly formed string that matches the replacement is not considered for substitution. For example, consider the following rather contrived example:

```
$ echo baa | sed -e 's/ba/ab/'
aba
```

(Recall that the pipe character ("`|`") takes the standard output from the command on its left and "pipes" it into the standard input of the command on its right.) The result of the single substitution is to create a second opportunity for a substitution; the substitution acting on the first and second characters has caused the second and third characters to now match the pattern. It might be suspected that this newly formed string might be acted upon if the "`g`" option were applied. This is not the case:

```
$ echo baa | sed -e 's/ba/ab/g'
aba
```

[1]An "argument" is technical term simply meaning the thing or things that a function, command or option acts upon. For example, the copy instruction "`cp original copy`" has the command `cp` with two arguments, "`original`" and "`copy`".

Searching for further matches continues from the *end* of the substitution.

4.2 Multiple search and replace

Suppose we want to replace not only "`apple`" with "`plum`" but also "`orange`" with "`grape`". There are two ways to do this.

The first is simply to run `sed` twice:

```
$ cat fruit | sed -e 's/apple/plum/g' | sed -e 's/orange/grape/g'
plums plums
bananas bananas
lemons limes
grapes grapes
pears pears
```

and the second is to give two commands to a single `sed` with two uses of "`-e`":

```
$ cat fruit | sed -e 's/apple/plum/g' -e 's/orange/grape/g'
plums plums
bananas bananas
lemons limes
grapes grapes
pears pears
```

We will pursue the second strategy here.

Exercise 18: Turn the *second* instance of "`apples`" into "`plums`" on the first line.

When presented with two commands, `sed` takes one line at a time from its input, applies the first rule and then applies the second rule to the *resulting* line. This means that the second rule does apply itself to matching strings created by the application of the first rule. Compare and contrast the following two attempts to apply a rule twice to a line:

```
$ echo baa | sed -e 's/ba/ab/g'
aba
$ echo baa | sed -e 's/ba/ab/' -e 's/ba/ab/'
aab
```

4.3 Using regular expressions in `sed`

The string being searched for and replaced in this expression need not be a fixed string. It can be a regular expression, using exactly the same syntax as for `grep` described in chapter 3. However, the expressions used in `sed`'s regular expressions need to be thought out more carefully than `grep`'s for the following reason: if there is more than one possible match in a line, it did not matter to `grep` because it just displayed the line. It matters to `sed` because it may be going to substitute for whatever the match was.

For example, suppose we want to replace all the fruit that start with vowels with "`plums`". The regular expression "`\<[aeiou]`" will certainly match that. But looks what happens with the `sed` instruction "`s/\<[aeiou]/plums/`":

```
$ grep '\<[aeiou]' fruit
apples apples
oranges oranges

$ sed -e 's/\<[aeiou]/plums/' < fruit
plumspples apples
bananas bananas
lemons limes
plumsranges oranges
pears pears
```

Only the actual character that caused the match (the leading vowel) has been substituted. The regular expression must define the entire string to be replaced. In this case the words we are looking for begin with a vowel and are composed of lowercase letters, so the regular expression "`/\<[aeiou][a-z]*/`" might be expected to work.

```
$ sed -e 's/\<[aeiou][a-z]*/plums/' < fruit
plums apples
bananas bananas
lemons limes
plums oranges
pears pears
```

To do both words in a line, the `g` modifier is needed again:

```
$ sed -e 's/\<[aeiou][a-z]*/plums/g' < fruit
plums plums
bananas bananas
lemons limes
plums plums
pears pears
```

Exercise 19: Change every fruit whose *second* letter is a vowel into "`plum`".

4.4 Using matched expressions

Just as with `grep` it is possible in a `sed` regular expression to refer to the results of a previous match. The following example replaces all the lines consisting of the same fruit twice with the letter "`X`":

```
$ sed -e 's/^\([a-z]*\) \1$/X/' < fruit
X
X
lemons limes
X
X
```

Note the use of the anchors to stop the expression "`\([a-z]*\) \1`" from matching the blank in the middle of the "`lemons limes`" line. For this match, the "`[a-z]*`" matches *no* characters in the range "`a`" to "`z`" and "`\1`" has the value of the empty string. Try it yourself without the "`^`" and the "`$`".

With `sed` it is possible to go one step further than with `grep` and to use the matching string in the substitution. Just as "`\1`" means "whatever matched in the first "`\(...\)`" pair in the regular expression itself, it has the same meaning in the substitution as well:

```
$ sed -e 's/^\([a-z]*\) \1$/\1/' < fruit
apples
bananas
lemons limes
oranges
pears
```

Exercise 20: Swap the entries in every row. (This will only be visible in one line, of course.)

4.5 Files of `sed` commands

We have already seen in section 4.2 that `sed` can be passed multiple instructions with multiple "`-e`" options. This approach clearly becomes untenable once the number of commands becomes large. Instead, it is possible to place all the commands in a single file, one per line, and pass the name of the file to `sed` rather than each of the lines individually.

```
$ ls -l sedfile
-rw-r--r--   1 rjd4     uxsup         60 Feb 16 12:23 sedfile

$ cat sedfile
# Here is a comment line
#
s/apple/plum/g
s/orange/grape/g

$ sed -f sedfile < fruit
plums plums
bananas bananas
lemons limes
grapes grapes
pears pears
```

The "`-f`" option introduces a file of `sed` commands, just as the "`-e`" option introduces a single command.

Note that the quotes have been dropped from each command. Now that the `sed` commands are in a file they no longer need to be protected from the shell (which was the function of the quotes).

Note also that the lines starting with a "`#`" are treated as comments by `sed` and that blank lines are ignored. We can use the fact the "`#`" is a comment character to create "`sed` scripts" in the same way that we have shell scripts. Consider the following file:

```
$ ls -l sedscript
-rwxr-xr-x   1 rjd4     uxsup         48 Feb 16 12:27 sedscript

$ cat sedscript
#!/bin/sed -f

s/apple/plum/g
s/orange/grape/g
```

The file `sedscript` is *executable* and *readable*. Because the first two characters of the file are "`#!`" it is treated specially by the operating system and the command

```
./sedscript < fruit
```

is transformed internally into

```
/bin/sed -f ./sedscript < fruit
```

where the "`/bin/sed -f`" component is the content of the first line after the "`#!`".

```
$ ./sedscript < fruit
plums plums
bananas bananas
lemons limes
grapes grapes
pears pears
```

The most important gain with `sed` scripts and files, other than simplicity for multiple commands, is that commands may now easily span more than one line.

Recall that the way to create an executable script is to create the file as a plain text file with your favourite editor and then apply `chmod` to it to make it executable. In this example, the editor used is `emacs`:

```
$ emacs myscript

$ ls -l myscript
-rw-r--r--   1 rjd4     uxsup          49 Apr 12 11:58 myscript

$ chmod a+x myscript

$ ls -l myscript
-rwxr-xr-x   1 rjd4     uxsup          49 Apr 12 11:58 myscript
```

Exercise 21: Create `sed` script equivalents of the answers to the exercises so far in this chapter.

4.6 Other `sed` commands

Of course, `sed` can do much more than just search and replace regular expressions. Here are a few more of its facilities, together with examples:

4.6.1 Appending new lines of text

Text can be added after a line containing a match in the following manner with the "a" `sed` instruction. Because the `sed` command spans lines, appending is better done with a `sed` script:

```
$ ls -l append
-rwxr-xr-x   1 rjd4     uxsup          42 Feb 16 14:58 append

$ cat append
#!/bin/sed -f
/lemons/a\
An appended line

$ ./append < fruit
apples apples
bananas bananas
lemons limes
An appended line
oranges oranges
pears pears
```

Note that, although the "`/lemons/`" expression matched the first half of the line "`lemons limes`", the new line was appended after the whole line containing the match.

Also note that it is important that no character (including whitespace) follow the backslash at the end of the line. The backslash acts to escape the newline character that marks the end of the line. By escaping it (with the "\" character), the newline is not regarded by `sed` as the end of its command, but is passed to the append instruction. Because the `sed` program then hasn't received a "real" newline, it carries on with the next line without treating it as a new command line.

It is possible to append more than one line in the following manner:

```
$ ls -l append2
-rwxr-xr-x   1 rjd4     uxsup         69 Mar 30 20:40 append2

$ cat append2
#!/bin/sed -f
/lemons/a\
An appended line\
And another appended line


$ ./append2 < fruit
apples apples
bananas bananas
lemons limes
An appended line
And another appended line
oranges oranges
pears pears
```

4.6.2 Inserting new lines of text

Text can be inserted before a matching line as in the following example, again using a script:

```
$ ls -l insert
-rwxr-xr-x   1 rjd4     uxsup         42 Feb 16 15:01 insert

$ cat insert
#!/bin/sed -f
/lemons/i\
An inserted line

$ ./insert < fruit
apples apples
bananas bananas
An inserted line
lemons limes
oranges oranges
pears pears
```

So insertion is similar to appending, except that the line precedes the matched line.

Exercise 22: Adapt the file `insert` so that it inserts more than one line.

4.6.3 Changing lines of text

A line of text matching a pattern can be replaced *in its entirety* with a “change” instruction:

```
$ ls -l change
-rwxr-xr-x   1 rjd4     uxsup         40 Mar  4 21:27 change

$ cat change
#!/bin/sed -f
/lemons/c\
A changed line


$ ./change < fruit
apples apples
bananas bananas
A changed line
oranges oranges
pears pears
```

Again, note that the *whole* line was changed, not just that part of it that matched.

4.6.4 Deleting lines

By default, `sed` prints out each line once it has finished with it. A line can also be deleted by a command so that it is not output:

```
$ sed -e '/lemon/d' < fruit
apples apples
bananas bananas
oranges oranges
pears pears
```

Exercise 23: Write a `sed` command or script to delete every line that has a repeated word.

4.6.5 Character substitution

It is possible to remap characters on a line with a `sed` instruction, “`y`”. For example, to uppercase all the vowels in a file we could do the following:

```
$ sed -e 'y/aeiou/AEIOU/' < fruit
ApplEs ApplEs
bAnAnAs bAnAnAs
lEmOns lImEs
OrAngEs OrAngEs
pEArs pEArs
```

Unfortunately, `sed`'s "`y`" instruction does not understand the "`A-Z`" range notation, so uppercasing everything with this command is rather unwieldy:[2]

```
$ sed -e 'y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/' < fruit
APPLES APPLES
BANANAS BANANAS
LEMONS LIMES
ORANGES ORANGES
PEARS PEARS
```

4.7 Further use of `sed`

If the information given so far is enough for your needs than you can skip this section. However, understanding this section will be of great help for the `awk` chapter following.

So far, individual `sed` commands have been given, in isolation from one another, to perform specific tasks. This section describes the way `sed` works and gives the general syntax of a `sed` command. All the discussion in this section will be in terms of `sed` scripts. They can all be run using `sed` files and the `-f` option, or even by multiple uses of the `-e` option.

As described at the start of this section, `sed` works by reading its input one line at a time into memory called "pattern space". Each line is run past the set of `sed` commands one command at a time. Each of these `sed` commands may or may not alter the line and when the input line has passed every `sed` line, the (possibly edited) line is output if it has not been deleted. Then `sed` reads the next input line into pattern space and repeats this until the input is exhausted.

As well as the pattern space, `sed` has a single other area of memory available to it, called the "hold space". Some of `sed`'s instructions access this single other variable in a variety of ways, but we will ignore it and the commands that access it. If the reader needs the use of the hold space then the author recommends the use of `perl` instead of `sed`.

The generic `sed` command: The generic `sed` command has the format:

`line-descriptor sed-instruction`

A "line descriptor" defines which lines in the input have the following instruction applied to them. If it is omitted, then *all* lines have the instruction applied. We have already seen examples of this structure, both with and without a line descriptor.

The command "`/lemon/d`" deletes all those lines with the string "`lemon`" in them. This `sed` command has a line descriptor "`/lemon/`" meaning "apply the following command to lines matching the regular expression '`lemon`'" and an instruction "`d`" meaning "*d*elete this line".

[2] The reader is advised to use the `tr` command for this sort of thing.

The command "`s/apple/plum/`" has no line descriptor as it starts with the `sed` instruction "`s`", meaning "*s*ubstitute". This substitution command is applied to *all* lines, but it has no effect on those lines that do not contain the string "`apple`".

Line descriptors: Line descriptors can take a variety of forms:

none: If no line descriptor is given, the following `sed` command is applied to all input lines. So this example deletes all the lines from a file:

```
$ sed -e 'd' < fruit
```

A number: A number specifies a line number. So this example just deletes line 2:

```
$ sed -e '2d' < fruit
apples apples
lemons limes
oranges oranges
pears pears
```

A regular expression: A regular expression specifies all those lines that match it.

```
$ sed -e '/^[aeiou]/d' < fruit
bananas bananas
lemons limes
pears pears
```

$: A dollar indicates the final line of the input.

```
$ sed -e '$d' < fruit
apples apples
bananas bananas
lemons limes
oranges oranges
```

It is also possible to specify a range of lines by giving a pair of line descriptors separated by a comma. The first descriptor specifies the first line of the range and the second the final line. As an easy example, consider the removal of the first three lines of a file, or everything after the first three lines:

```
$ sed -e '1,3d' < fruit
oranges oranges
pears pears

$ sed -e '4,$d' < fruit
apples apples
bananas bananas
lemons limes
```

Similar results can be produced by using a pair of regular expressions. Here, the `sed` instruction following the line descriptor is applied to the lines in each range that is bounded by the appropriately matching lines. Consider this example:

```
$ sed -e '/^b/,/^o/ s/a/A/g' < fruit
apples apples
bAnAnAs bAnAnAs
lemons limes
orAnges orAnges
pears pears
```

The `sed` command has line descriptor "`/^b/,/^o/`" and editing instruction "`s/a/A/g`" so it takes every line from one starting with a "`b`" to one beginning with an "`o`" and replaces all the instances of "`a`" with "`A`" in those lines.

There is an important difference between a pair of regular expressions for a line descriptor and a pair of line numbers. The pair of regular expressions might match more than one range of lines:

```
$ cat abc
abcde1
vwxyz1
ABCDE1
VWXYZ1
abcde2
vwxyz2
ABCDE2
VWXYZ2

$ sed -e '/a/,/A/ d' < abc
VWXYZ1
VWXYZ2
```

Two ranges of lines matched the line descriptor "`/a/,/A/`": lines 1–3 and lines 5–7. Both these ranges were deleted by the "`d`" instruction.

There is scope for confusion when the pattern to be matched at the start or stop of the line descriptor also occurs in the `sed` instruction being applied to the line. The rule to bear in mind is that *the tests for start and end of line descriptors are performed* before *the application of the corresponding instruction.* For example:

```
$ sed -e '/a/,/A/ y/aAvV/AaVv/' < abc
Abcde1
Vwxyz1
aBCDE1
VWXYZ1
Abcde2
Vwxyz2
aBCDE2
VWXYZ2
```

Here, the "`A`" in "`ABCDE1`" and "`ABCDE2`" turns off the character changing, even though it is changed to "`a`" itself by the rule. This is why the "`VWXYZ`" lines are untouched.

There is no reason why types of line descriptor cannot be mixed. The command "`1,/lemon/d`" will delete all lines upto and including the line containing "`lemon`".

We saw in section 4.4 that matches can be recorded in "`\(...\)`" pairs and referred to with the "`\1`" expressions. These "`\1`" expressions can be used in the second half of a substitution instruction that defined them with the corresponding "`\(...\)`" expression in the first half. However they *cannot* be used to span between regular expressions in a pair of line descriptors or between line descriptors and editing instructions. So the `sed` command "`/\([aeiou][aeiou]\)/,/\1/`" (an attempt to close a range of lines with a line contining the same pair of vowels that the opening line contained), will give an error message. Using the "`\(...\)`" and "`\1`" pair in the *same* line descriptor, though, is perfectly acceptable. The line descriptor "`/\([a-z]\)\1/,/\([a-z]\)\1/`" will start and end a range on a line with a double vowel, though *not necessarily the same double vowel.*

(nothing)	all lines
n	line *n*
/*reg. exp.*/	all lines matching the regular expression
`$`	the final line
a`,`*b*	the range of lines from *a* to *b* (inclusive)

Table 4.1: A summary of `sed` line descriptors

`a\` *line 1*`\` *line 2*`\` ... *line n*	*a*ppend lines 1 to *n* after current line
`c\` *line 1*`\` *line 2*`\` ... *line n*	*c*hange current line for lines 1 to *n*
`d`	*d*elete current line
`i\` *line 1*`\` *line 2*`\` ... *line n*	*i*nsert lines 1 to *n* before current line
`s/`*a*`/`*b*`/`	*s*ubstitute regular expression *a* for expression *b*

Table 4.2: A summary of `sed` editing instructions