Table of Contents

sed

While bash is a command interpreter that runs scripts, programs, etc. sed is a standalone text-processing utility/tool that modifies text using its own methods. bash can call sed, but sed doesn't need bash.
The formal definition is that sed is a stream editor allowing transformations of its input. It reads the stream input line by line, performs command/script actions, and outputs the modified line. Basically a substitution/editor tool. Because it makes only one pass through its input, sed is more efficient than an interactive editor (like ed).

Typically used to execute repetitive edits to >= 1 files, convert data, or select lines containing certain data.

sed can be performed directly in the command line or can be a script inside of a scriptFile.

Syntax:

```
sed options 'command' file1 #reads from stdin

sed options -f/or scriptFile file1 #reads from a file
```

The -f switch is used to specify a script file which allows for more complex capabilities.

☆ A pattern of any number of spaces/tabs in any order is done by '[\t]'

☆ In non-GNU versions of sed, you might have to enter a tab character instead of '\t'

Book: `sed [-n] program [fileList]`

Allows you to write simple, short sed programs without creating a separate file to hold the sed program.

Book: `sed [-n] -f programFile [fileList]`

The programFile is the pathname of a file containing a sed program. The fileList contains pathnames of the normal files that sed processes (input files). When you don't specify a fileList, sed takes its input from standard input.

---

sed processing steps for each input line:

1. Place the line in a buffer (pattern space)
2. Execute the command or commands on the buffer
3. Output the buffer to stdout

Three types of addressing modes in sed:

0 - no address -> operates on all lines of input
1 - one address -> operates on all matching lines
2 - address range -> operates on all lines within the range

Options: -- work under Linux/GNU sed only.
- work under Linux/GNU sed & OS X/BSD sed

--file programFile
-f programFile
Causes sed to read its program from the file named programFile instead of the command line. You can use this more than once on the command line.

--help
Summarizes how to use sed.

--inPlace[=suffix]
-i[suffix]
Edits files in place. Without this, sed sends output to standard output. With; sed replaces the file it's processing with its output. When you specify a suffix, sed makes a backup of the original file with the backup having the original filename + suffix appended.

--quiet or -n
--silent
Causes sed not to copy lines to standard output except as specified by the Print (p) instruction or flag.

Editor Basics

A sed program consists of >=1 lines with the syntax:

```
[address[,address]] instruction [argumentList]
```

- The addresses are optional; If omitted, sed processes all lines of input. The instruction is an editing instruction that modifies the text.
- The addresses select the line(s) the instruction part operates on.
- The number/kinds of arguments in the argumentList depend on the instruction. Use ';' to separate several sed commands on one line.

sed processing input steps:

1. Reads one line of input from fileList/standard input
2. Reads the 1st instruction from the program/programFile. If the address(es) select the input line, acts on the input line as the instruction specifies.
3. Reads the next instruction from the program/programFile. If the address(es) select the input line, acts on the input line as the instruction specifies.
4. Repeats step 3 until all instructions are executed in program/programFile.
5. Starts over with step 1 if there's another line of input; Otherwise finished.

Addresses

A line number is an address that selects a line.
Special case: '$' represents the last line of input.
A regular expression is an address that selects those lines containing a string that the regular expression matches.

Substitute

Syntax:

```
s/matchPattern/replaceValue/flags
```

- matchPattern is the pattern to find in a line
- replaceValue is the value used as a replacement for the matched value.
- flags:
  - g -> globally replace all occurrences in the line
  - p -> print the contents of the pattern space if successful
  - w -> write pattern space to a file if successful
  - *i* -> this is a number; replace the *i*th occurrence only.

Although examples show '/' as the delimiter between the matchPattern, replaceValue, and flags; You can specify a different delimiter after the "s". This can make it easier when you need a slash in your pattern or replacement value. Substitute simply uses the first character after the "s" as the delimiter. No extended ASCII though!

Instead of: s/\/\/.*$//
You can use: s|//.*$||

Interjection of Notes From Professor's Lectures:

Typical uses of sed:
- Execute repetitive edits to >=1 file(s)
- Convert data
- Select lines containing certain data

By default: Prints to stdout
If you have one file, and it's 10 lines long; Your script will execute 10 times- Once for each and every line of input stream.

p - print
d - delete
a - append
i - insert
c - change
s - substitute -> most common
s/target/replacement/flags

Example:

```
s/cat//
$ echo "i have a cat" | sed 's/cat//'
i have a
$ echo "i have a cat" | sed 's/cat/XXX/'
i have a XXX
```

addresses -> line numbers and/or patterns

```
0 -> no address
#Basically means it's unconditional, this will replace cat on any
line.
s/cat/XXX/

1 -> single address
/dog/s/cat/XXX/ #looks for line that contains dog, if it contains that
                  #then it replaces cat with XXX
5s/cat/XXX/ #specifically on line 5 any instance of cat should be
replaced

2 -> two address mode (has a start and stop)
1,10 s/cat/XXX/ #want to replace cat with XXX on first 10 lines of a file
/START/,/STOP/ s/cat/XXX/ #if i only want to replace instances of
#'cat' between a line that starts with capital S->START until it finds
#another line that begins with capital S->STOP
```

Examples:

```
#only output to stdout all lines that contain "printf"
$ sed -n '/printf/p' cs1713p0.c
  // about the safety of scanf and printf
      printf("%-10s %-20s %10s %10s %10s %10s\n"
           printf("invalid input when reading student data, only %d
valid values. \n"
           printf("\tdata is %s\n", szInputBuffer);
      printf("%-10s %-20s %10.2f %10.2f %10.2f %10.2f\n"
#'/printf/p' means to print lines containing "printf"
#The -n means to not output lines normally.When used with the "p",
only the #results of the "p" are returned.
```

```
#create a new file2 from file2 replacing "Linux" with "Unix"
$ sed -i.txt 's/Unix/Linux/g' file1 > file2
$ diff file1 file2
1c1
< Although Unix is well known today, Unix was not known
-
> Although Linux is well known today, Linux was not known
5c5
< was a high school English teacher, if she knew Unix.
-
> was a high school English teacher, if she knew Linux.
#s -> substitute -> followed by a match pattern & replacement value
#g -> means globally change each occurrence on the line (doesn't mean
#globally in the file), every line is sent to file2 including both
#modified and unmodified lines.
```

```
#Modify file3 removing carriage returns; For safety make a backup.
$ sed -i.sav 's/\r/\n/g' file3
#-i -> edit the file "in place" aka modify the file. Creates a backup
#of "file3" using the suffix ".sav". This is useful if you need to
#modify multiple files.
#Replaces each carriage return with an empty string
#(deletes the return)
```

```
#modify file4 and file5 replacing only one occurrence of "cat" with
#"dog"
$ sed -i.cat '1,3 s/cat/dog/' file4 file5
#1,3 is the range of lines to apply the substitution
#The backup copies are named file4.cat and file5.cat
```

```
#Delete lines in line6 that begin with "#", producing file6.new
$ sed '/^#/d' file6 > file6.new
#The '/^#/d' is a pattern for deleting lines.
#"^" specifies that the pattern must be at the beginning of the line.
#d means delete
```

By default sed prints the patternspace to the screen; however by default it does not overwrite it. That's why we did file1 > file2 to write it to a new file. If you did 'sed -n 's/Unix/Linux/g' file1', think of that as no default output. That suppresses the default output. Rather than printing the line to the screen; it prints nothing. Only things that are explicitly printed will appear on the screen. "Poor man's way of doing grep"

$ **sed -n '/printf/ p' cs1713p0.c**
If you ever see the print command being used, it almost always implies that sed needs to be run with -n. If not, all the lines of the file would print; And the ones containing printf would print twice.

$ **sed -i.cat '1,3 s/cat/dog/' file4 file5**
-i does an in-place edit- rather than printing the changed lines to stdout, it actually modifies the original file.
Running this, nothing will print to the screen, but cat -n file1 will show that all instances of Unix have been replaced with Linux.

$ **sed '/^#/d' file6 > file6.new**
'^' is called an anchor and specifies where the match is to take place, and reads as "beginning of line." In English -> In the beginning of the line, there should read a hash symbol.
$ **cat -n file6**
```
1 # This is a comment line one
2 # This is comment line two
3 x=y; # this comment isnt in column one
4 y = x / 100;
5 print x, y;
```
$ **sed '/^#/d' file6 > file6.new**
```
x=y # this comment isnt in column one
y = x / 100;
print x, y;
```
$ **sed '/#/d' file6 > file6.new**
```
y = x / 100;
print x,y;
#This deletes any line containing '#'
```

'd' -> delete changes the control flow of your script
Not only does it not print the current patternspace, it makes your script start over from the beginning, with a new line of input. So anything after the delete command, assuming it fired, isn't reachable for that particular cycle.
/printf/p
/^#/d
s/cat/dog/g
If 'd' is fired; everything below is unreachable. Like an implicit "continue"

global v. without

```
$ echo "cat cat cat" | sed 's/cat/XXX/'
XXX cat cat
$ echo "cat cat cat" | sed 's/cat/XXX/g'
XXX XXX XXX
$ echo "cat cat cat" | sed 's/cat/XXX/1'
XXX cat cat
$ echo "cat cat cat" | sed 's/cat/XXX/2'
cat XXX cat
$ echo "cat cat cat" | sed 's/cat/XXX/3'
cat cat XXX
```

Regex makes the longest leftmost match; they start on the lefthand side of the string and move to the right attempting to make a match.

Important Pattern Symbols:

These bear a strong resemblance to bash filename pattern meta-characters.
`extglob` -> adds pattern matching capabilities to bash

`[list]/[^list]` -> It is exactly/syntactically and semantically identical to the bash wildcard character class.

`'.'` -> is a standalone. It's analogous to the '?' in bash (matches any single character)

`'*'` -> only quantifier that's available in basic regular expression. It ascribes a quantity to the regex that immediately precedes it. In bash, the asterisk can stand alone; in regex it can't. Comparable to an exponent; You can't have an exponent without a base! Beware of using this; it's very bad at false-positives.

`'^'/'$'` -> Doesn't match a character-matches a position.

`'^'` -> For this match to be successful, it must occur at the beginning of the string. For '$' it must occur at the end of the string.

Example:

```
/[0-9]
my 123 string 456
/[0-9].
my 123 string 456
/[0-9].
my string 4
```

/[0-9]*
123 my string 142321312414142
Example of "longest leftmost match"- Not the longest it could possibly make- but it's the longest leftmost one.

/x*
123 my string 24531344213324
This will still make a match. There are no x's in my string, but because it says "0 or more x's" it found "0" x's and therefore it was a successful match.

/^[0-9][0-9]* #beginning of the line must have one or more digits
123 my string xxxxxxx

/^[0-9][0-9]* #no match anymore!
asfdgfhsa 123 my string xxxxxxx

/^[0-9][0-9]* #taking the anchor off matches the digits anywhere
asfdgfhsa 123 my string xxxxxxx

/[0-9][0-9]*$ #matches end of line
asfdgfhsa 123 my string xxxxxxx 423123

/[0-9][0-9]*$ #no longer matches
asfdgfhsa 123 my string xxxxxxx 423123 asfdgfg

Prof: "Pretty much everything is in extended regex besides grep and sed"

/^The/ -> Matches any line that begins with the word "The"
- - - - -

/[a-z]\{3\}[0-9\{3\)/ -> Matches a lowercase abc123 ID
- - - - -

/(.*)/ -> Matches an open parenthesis followed by any number of any
characters followed by a right parenthesis.
So (eiroweht82332), (243595), etc.
- - - - -

/[^abcd]\.$/ -> Any character other than lowercase a, b, c, and d.

[^a-d]\.$
asdf 123 my string xxx 42342 adasd2134.
Be careful because '.' will match literally anything if you don't escape it.
By adding the escape symbol you are saying to look for a literal '.'

```
/^[ \t]*$/ -> Beginning of line, do a character class with a
              space/tab, give me 0 of those, then end.
              This indicates it matches. White space or nothing.
```

- - - - -

```
pat1|pat2 -> Either/or
/cat|dog
my fav pet is my cat
my fav pet is my dog
my fav pet is my bat
```

- - - - -

```
+ -> similar to *, but one or more.
^[0-9][0-9]* <- basic mode
^[0-9]+ <- does the same thing
```

- - - - -

```
? -> zero or one.
[a-z]\.?$ #Lower case character, followed by a period, which is optional.
this is a test.
```

- - - - -

```
{i} -> "one quantifier to rule them all" basically very useful.
```

Variations:

```
x{n} -> precisely 'n' matches
[0-9]{3}-[0-9]{2}-[0-9]{4} #matching SSN number
x{n,} -> 'n' or more matches (greedy)
x{n,m} -> between 'n' and 'm' matches (inclusive)
* -> {0,}  #0 or more
+ -> {1,}  #1 or more
? -> {0,1} #0 or 1
x{2,4} #at least 2, at most 4
x
xx
xxx
xxxx
xxxxx
```

☆ Many of these quantifiers are greedy and will match as much as they can.

☆ *Basic* regex requires ‘-r’ in sed or ‘-E’ in grep to use natively.

! This is basically a repetition of above; Don't want to potentially miss any details.

. - Matches any character including newline
^ - The pattern that follows must begin at the first character
$ - The pattern that precedes the $ must match at the end of the line
* - Matches zero or more of the preceding character, group, or bracketed list.
[list] - Matches one character to any of the characters listed within the brackets. For convenience, range abbreviations can be used. (e.g., [a-f], [0-9])
[^list] - Matches one character if it's not listed within the brackets.

☆ *Extended* regular expressions require '-r' in sed or '-E' in grep to be used natively, without needing escape characters.

\(regex\) - Groups the inner regexp and allows it to be referenced later with \1, \2, etc.
\+ - Similar to *, but matches one or more
\{i\} - Matches i occurrences of the preceding character, group, or bracketed list.
\{i,i\} - Matches between i and j occurrences of the preceding character, group, or bracketed list.
\? - Matches zero or one instance of the preceding; Equivalent to \{0,1\}
pat1\|pat2 alternation - Matches either pat 1 or pat2

- Symbols representing characters to match ('.','[list',etc) are called **classes**.
- Symbols which dictate the amount of a preceding class ('*','+',etc) are called **quantifiers**.
- Groups can be referenced later with "\1", "\2", etc.

- To reference the characters: '.','^','$','*','[',']' in a regular expression, it must be escaped with a backslash.

- To reference characters '+', '{', '}', '[', ']' just use each normally. Their special (extended) meaning requires the use of a backslash when referenced in a basic regular expression.

sed also supports extended regular expressions if the '-E' or '-r' switch is specified. This changes some of the above symbols and adds additional ones.

Example:
Using sed regex; Pattern to match a phone number

Purely basic:
```
([0-9][0-9][0-9])[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]
```

Extended, but in basic mode:
```
([0-9]\{3\})[0-9]\{3\}-[0-9]\{4\}
```

Purely extended:
```
\([0-9]{3}\)[0-9]{3}-[0-9]{4}
```

Negation

We can negate an address (including a regex pattern) by following the pattern with a bang.
e.g Delete all lines in the file that don't say "linux" or "Linux"

```
$ sed '/[Ll]inux/! d' Linux
```

Append

a - Append inserts one or more lines of text after the matched lines.

```
$ sed 'line a\
$ sed '/PATTERN/ a\
both: user input: lineAppend' filename.[filetype]
```

```
$ sed '3 a\ #adds 'meow' after the 3rd line
> meow' example.txt
Linux Sysadmin
Databases - Oracle, etc.
Security
meow
Storage in Linux
```

```
$ sed '/Sysadmin/a \ #adds 'meow' after every line that matches the pattern "Sysadmin"
> meow' example.txt
Linux Sysadmin
meow
Databases - Oracle, etc.
Windows - Sysadmin, reboot, etc.
meow
```

```
$ sed '$ a\ #appends at the end of a file
```

Insert

i - Inserts one or more lines of text before the matched lines.

```
$ sed 'line i\
$ sed '/PATTERN/ i\
both: user input: lineAppend' filename.[filetype]
```

```
$ sed '3 i\ #adds 'meow' before the 3rd line
> meow' example.txt
Linux Sysadmin
Databases - Oracle, etc.
meow
Security
Storage in Linux

$ sed '/Sysadmin/i \
> meow' example.txt
meow
Linux Sysadmin
Databases - Oracle, etc.
meow
Windows - Sysadmin, reboot, etc.

$ sed '$ i\ #inserts a line before the last line of the file
```

Change

```
c - Changes one or more lines, replacing them with the specified text.
      $ sed 'ADDRESS c\
      $ sed '/PATTERN/ c\
      both: user input: lineAppend' filename.[filetype]

$ sed '1 c\ #Replaces first line of a file
> meow' example.txt
meow
Databases - Oracle, etc.
. .

$ sed '/Linux Sysadmin/c \
> Linux Cat' example.txt
Linux Cat
Databases - Oracle, etc.
```

Alternative to Scripts when needing multiple commands

The -e (lowercase) command argument can be used to provide multiple edits to a file without requiring a script file. This is very useful when you need variable values for different sed script steps since sed doesn't provide an easy way to pass variables as parameters.

Example:

```
$ sed -r -e 's/[0-9]+//' -e 's/[0-9]+//' -e 's/([0-9.]+).*$/\1/' inventory.txt
#removes 2nd, 3rd, and 5th logical columns.
#product IDs end in 3 numbers
```

Next

The next command can be used to skip lines without printing them.

- n -> Skips to the next line of input and overwrites the current line. Therefore, the current line and all changes made to it will not be printed.
- N -> Reads the next line of input and appends it (with a newline character preceding) to the end of the current line.

sed uses newline characters (\n) to separate one line from another and when a line is copied into the pattern space the newline character is removed. This means that using the commands we've seen so far, we would be unable to remove newline characters from a file. One solution to this is the N command.

Note that when printing a line of output sed prints a newline character provided there's at least one character in the buffer.
Because of these restrictions, sed isn't the recommended tool for removing newline characters. Instead use 'tr'.

Big Example:

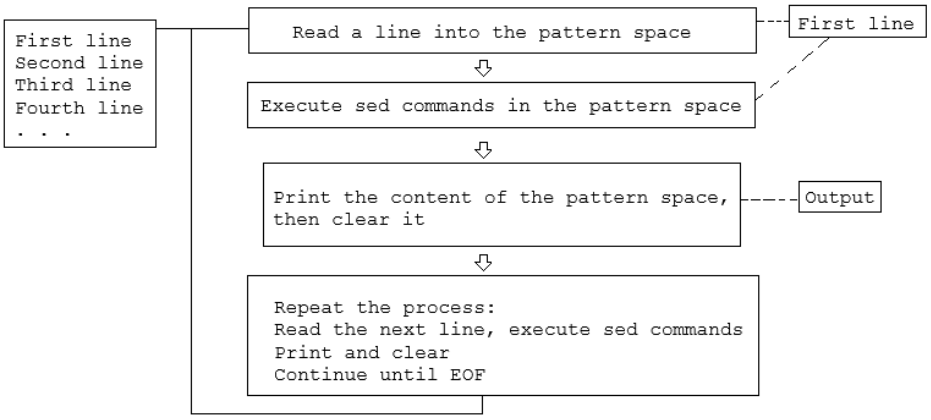
```
$ cat >example16
$ sed -E -f example16 < file1
Although Unix is well known today, Unix was not known
outside of technical circles in the 1980s. I worked
for a company that was hiring technical writers in
the mid 1980s. An interviewer asked a candidate, who
was a high school English teacher, if she knew Unix.
Having never heard of the operating system, but knowing
a word that was a homonym, she was unnerved.
$ cat >example16
/(Linux)|(Unix)/N
s/\n/\t/
CTRL-D
$ sed -E -f example16 < file1
Although Unix is well known today, Unix was not known outside of technical circles in
the 1980s. I
worked
for a company that was hiring technical writers in
the mid 1980s. An interviewer asked a candidate, who
was a high school English teacher, if she knew Unix. Having never heard of the
operating system,
but knowing
a word that was a homonym, she was unnerved
```

Input Buffers

The sed editor provides 2 buffers to use while editing a file: pattern space and hold space.
All commands we've used so far operate on the pattern space.
Immediately after finishing all commands in the file and before looping back, sed outputs the contents of the pattern space followed by a newline unless the -n flag is set. This is why all lines print by default.

sed stores line currently being processed in a temp buffer (pattern space)
-> finishes processing -> line in pattern space is sent to the screen

Although the example below shows only the pattern space being utilized, since the hold space remains unused unless explicitly accessed by commands, the hold space is like a warehouse; serving as a temporary storage area during data manipulation. Pattern space is more like an assembly line in manufacturing- data is processed sequentially here.



- g -> Copies contents of hold space into pattern space, overwriting the previous content of the pattern space.

- G -> Appends contents of hold space onto pattern space, followed by a newline(\n).

- h -> Copies contents of pattern space into hold space, overwriting the previous content of the hold space.

- H -> Appends contents of pattern space into hold space, followed by a newline(\n).

- h -> Deletes all lines in the pattern space and reads the next new line into the pattern space.

- H -> Deletes the first line in a multiline pattern space but doesn't read the next line.

- x -> Swaps the contents of the hold space and pattern space.

Examples:

```
#Add an empty line after each line
$ head -n5 file
hello
hello
hello
hello
hello
$ sed 'G' file | head -n5
hello

hello

hello
. . .

#Simulate tac function (reverse line order) using sed
$ sed '1!G;h;$1d' file
5
4
3
2
1
0
$ cat file
0
1
2
3
4
5
#1!G skip the first line, then append hold space content to
#pattern space for subsequent lines.
#!d delete all lines except the last one (building reverse
#order in hold space)

#Append matched lines to EOF
$ cat file
0
1 hello
2
3 hello
4
5
```

```
$ sed -e '/hello/H' -e '$G' file
0
1 hello
2
3 hello
4
5

1 hello
3 hello
$ sed -e '/hello/{H;d}' -e '$G' file
0
2
4
5

1 hello
3 hello
#/pattern/H append matched lines to hold space
#$G at EOF, append hold space content
#-e -> perform multiple edits
#-n -> suppress automatic output (use with p for explicit printing)

#Transpose rows to columns
$ sed -ne 'H;${x;s/\n/ /g;p}' file
0 1  2 3  4 5
#H -> Accumulate all lines into hold space
#$ -> at end of file:
      #x -> Swap pattern & hold spaces
      #s/\n/ /g -> replace newlines with spaces
      #p -> print the result

#Sum numbers from 1 - 100
$ seq 100|sed -ne 'H;${x;s/\n/+/g;s/^+//p}'
1+2+3+4+5+6+7+8+9+10+11+12+13+14+15+16+17+18+19+20+21+22+23+24+25+26+27+28+29
+30+31+32+33+34+35+36+37+38+39+40+41+42+43+44+45+46+47+48+49+50+51+52+53+54+5
5+56+57+58+59+60+61+62+63+64+65+66+67+68+69+70+71+72+73+74+75+76+77+78+79+80+
81+82+83+84+85+86+87+88+89+90+91+92+93+94+95+96+97+98+99+100
$ seq 100|sed -ne 'H;${x;s/\n/+/g;s/^+//p}' | bc
5050
#seq 100 generates vertical list 1 - 100
#bc (basic calc)
```

seq command syntax:

```
seq [option]. . .LAST/FIRST LAST/FIRST INCREMENT LAST
```

Quick Examples:

```
seq -s "+" 100 | bc #output:5050
seq 5 10 #prints 5 - 10 each on a new line
seq 0 2 10 #start at 0, increment by 2, until 10
seq -s "+" 1 2 10 | bc #sum odd numbers 1+3+4+7+9, output is 25

-w -> Equal width (pads with zeros)
$ seq -w 8 10
08
09
10
$ seq 2 2 5
2
4
$ sed -n 'p;n' file
1
3
5
$ sed -n 'n;p' file
2
4
6

$ sed 'n '1~2p' file #Address range -> prints every second line
1
3
5
$ sed -n '0~2' file #sed treats this as identical to 1~2 because line
2                   #numbers start at 1
4
6

#Moving a line to the end
$ cat >example17
/last/h
/last/d
G
CTRL-D
$ sed -f example17 hold_example.txt
First line
Second line here
Here is the third line
Almost at the final line.
This should be the last line.
```

Warning: Some implementations of sed restrict the max size stored in both the pattern space and hold space to 4000 bytes. GNU sed has no max buffer size provided it can malloc more memory