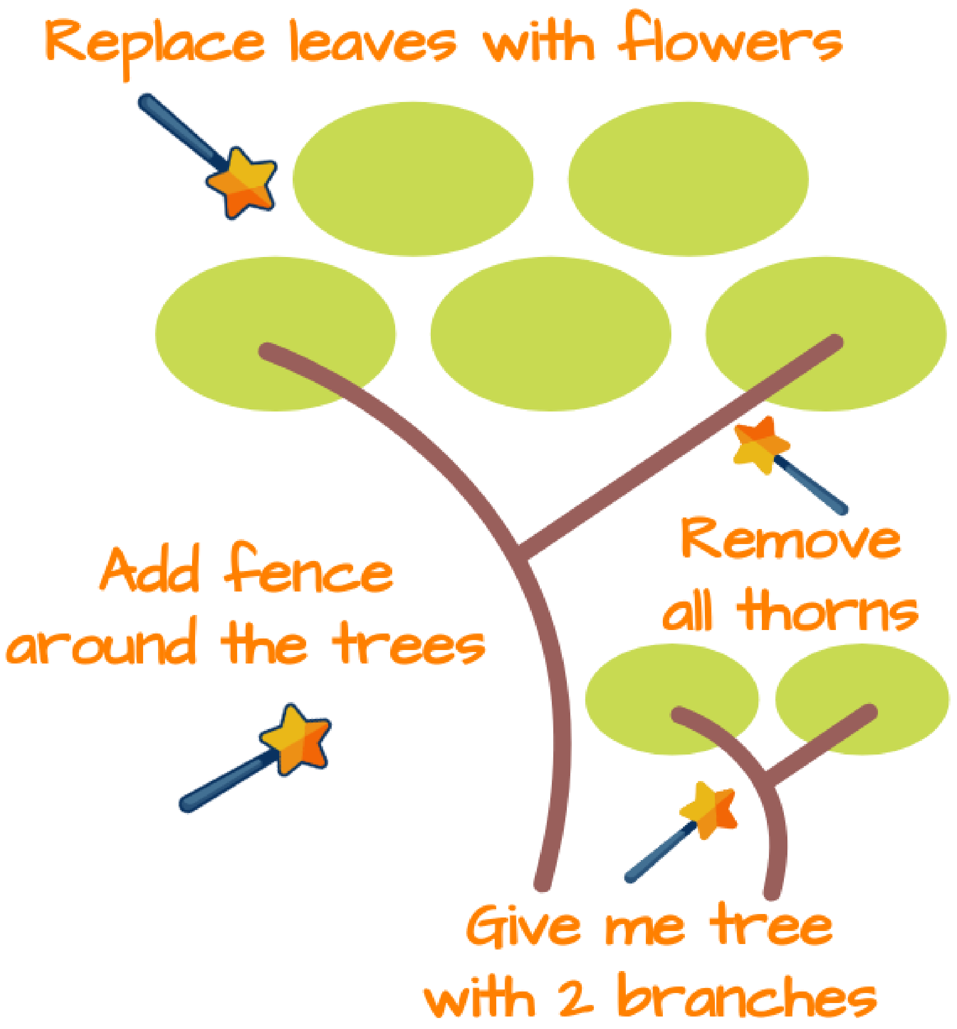
GNU SED

awesome stream editor


Replace leaves with flowers
Add fence around the trees
Remove all thorns
Give me tree with 2 branches

Sundeep Agarwal

Table of contents

Preface

You are likely to be familiar with the ”Find and Replace” dialog box from a text editor, word processor, IDE, etc to search for something and replace it with something else. `sed` is a command line tool that is similar, but much more versatile and feature-rich. Some of the GUI applications may also support **regular expressions**, a feature which helps to precisely define a matching criteria. You could consider regular expressions as a mini-programming language in itself, designed to solve various text processing needs.

The book heavily leans on examples to present options and features of `sed` one by one. Regular expressions will also be discussed in detail. However, commands to manipulate data buffers and multiline techniques will be discussed only briefly and some commands are skipped entirely.

It is recommended that you manually type each example and experiment with them. Understanding both the nature of sample input string and the output produced is essential. As an analogy, consider learning to drive a bike or a car — no matter how much you read about them or listen to explanations, you need to practice a lot and infer your own conclusions. Should you feel that copy-paste is ideal for you, code snippets are available chapter wise on GitHub.

Prerequisites

Prior experience working with command line and `bash` shell, should know concepts like file redirection, command pipeline and so on. Knowing basics of `grep` will also help in understanding filtering features of `sed` .

If you are new to the world of command line, check out ryanstutorials or my GitHub repository on Linux Command Line before starting this book.

My Command Line Text Processing repository includes a chapter on `GNU sed` which has been edited and expanded to create this book.

Conventions

- The examples presented here have been tested on `GNU bash` shell with **GNU sed 4.7** and may include features not available in earlier versions
- Code snippets shown are copy pasted from `bash` shell and modified for presentation purposes. Some commands are preceded by comments to provide context and explanations. Blank lines to improve readability, only `real` time shown for speed comparisons, output skipped for commands like `wget` and so on
- Unless otherwise noted, all examples and explanations are meant for *ASCII* characters only
- `sed` would mean `GNU sed` , `grep` would mean `GNU grep` and so on unless otherwise specified
- External links are provided for further reading throughout the book. Not necessary to immediately visit them. They have been chosen with care and would help, especially during rereads
- The learn_gnused repo has all the files used in examples and exercises and other details related to the book. Click the **Clone or download** button to get the files

Acknowledgements

- GNU sed documentation — manual and examples
- stackoverflow and unix.stackexchange — for getting answers to pertinent questions on `bash` , `sed` and other commands
- tex.stackexchange — for help on `pandoc` and `tex` related questions
- Cover image
  - draw.io
  - tree icon by Gopi Doraisamy under Creative Commons Attribution 3.0 Unported
  - wand icon by roundicons.com
- softwareengineering.stackexchange and skolakoda for programming quotes
- Warning and Info icons by Amada44 under public domain
- arifmahmudrana for spotting an ambiguous explanation

Special thanks to all my friends and online acquaintances for their help, support and encouragement, especially during difficult times.

Feedback and Errata

I would highly appreciate if you'd let me know how you felt about this book, it would help to improve this book as well as my future attempts. Also, please do let me know if you spot any error or typo.

Issue Manager: https://github.com/learnbyexample/learn_gnused/issues

E-mail: learnbyexample.net@gmail.com

Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a freelance trainer, author and mentor. His previous experience includes working as a Design Engineer at Analog Devices for more than 5 years. You can find his other works, primarily focused on Linux command line, text processing, scripting languages and curated lists, at https://github.com/learnbyexample. He has also been a technical reviewer for Command Line Fundamentals book and video course published by Packt.

List of books: https://learnbyexample.github.io/books/

License

Book version

1.2

See Version_changes.md to track changes across book versions.

Introduction

The command name `sed` is derived from **s**tream **ed**itor. Here, stream refers to data being passed via shell pipes. Thus, the command's primary functionality is to act as a text editor for **stdin** data with **stdout** as output target. Over the years, functionality was added to edit file input and save the changes back to the same file.

This chapter will cover how to install/upgrade `sed` followed by details related to documentation. Then, you'll get an introduction to **substitute** command, which is the most commonly used `sed` feature. The chapters to follow will add more details to the substitute command, discuss other commands and command line options. Cheatsheet, summary and exercises are also included at the end of these chapters.

Installation

If you are on a Unix like system, you are most likely to already have some version of `sed` installed. This book is primarily for `GNU sed`. As there are syntax and feature differences between various implementations, please make sure to follow along with what is presented here. `GNU sed` is part of text creation and manipulation commands provided by `GNU` and comes by default on GNU/Linux. To install newer or particular version, visit gnu: software. Check release notes for an overview of changes between versions. See also bug list.

```
$ # use a dir, say ~/Downloads/sed_install before following the steps below
$ wget https://ftp.gnu.org/gnu/sed/sed-4.7.tar.xz
$ tar -Jxf sed-4.7.tar.xz
$ cd sed-4.7/
$ ./configure
$ make
$ sudo make install

$ type -a sed
sed is /usr/local/bin/sed
sed is /bin/sed
$ sed --version | head -n1
sed (GNU sed) 4.7
```

If you are not using a Linux distribution, you may be able to access `GNU sed` using below options:

- git-bash
- WSL
- brew

Documentation and options overview

It is always a good idea to know where to find the documentation. From command line, you can use `man sed` for a short manual and `info sed` for full documentation. For a better interface, visit online gnu sed manual.

```
$ man sed
SED(1)                           User Commands                          SED(1)

NAME
       sed - stream editor for filtering and transforming text

SYNOPSIS
       sed [OPTION]... {script-only-if-no-other-script} [input-file]...

DESCRIPTION
       Sed  is  a  stream  editor.  A stream editor is used to perform basic
       text transformations on an input stream (a file or input from a pipe-
       line).   While  in  some  ways  similar  to  an  editor which permits
       scripted edits (such as ed), sed works by making only one  pass  over
       the  input(s),  and  is consequently more efficient.  But it is sed's
       ability to filter text in a pipeline which particularly distinguishes
       it from other types of editors.
```

For a quick overview of all the available options, use `sed --help` from the command line. Most of them will be explained in the coming chapters.

```
$ # only partial output shown here
$ sed --help
  -n, --quiet, --silent
                 suppress automatic printing of pattern space
      --debug
                 annotate program execution
  -e script, --expression=script
                 add the script to the commands to be executed
  -f script-file, --file=script-file
                 add the contents of script-file to the commands to be executed
  --follow-symlinks
                 follow symlinks when processing in place
  -i[SUFFIX], --in-place[=SUFFIX]
                 edit files in place (makes backup if SUFFIX supplied)
  -l N, --line-length=N
                 specify the desired line-wrap length for the 'l' command
  --posix
                 disable all GNU extensions.
  -E, -r, --regexp-extended
                 use extended regular expressions in the script
                 (for portability use POSIX -E).
  -s, --separate
                 consider files as separate rather than as a single,
                 continuous long stream.
      --sandbox
                 operate in sandbox mode (disable e/r/w commands).
  -u, --unbuffered
                 load minimal amounts of data from the input files and flush
                 the output buffers more often
  -z, --null-data
                 separate lines by NUL characters
      --help     display this help and exit
      --version  output version information and exit

If no -e, --expression, -f, or --file option is given, then the first
non-option argument is taken as the sed script to interpret.  All
remaining arguments are names of input files; if no input files are
specified, then the standard input is read.
```

Editing standard input

`sed` has various commands to manipulate text input. **substitute** command is most commonly used, which will be briefly discussed in this chapter. It is used to replace matching text with something else. The syntax is `s/REGEXP/REPLACEMENT/FLAGS` where

- `s` stands for **substitute** command
- `/` is an idiomatic delimiter character to separate various portions of the command
- `REGEXP` stands for **regular expression**
- `REPLACEMENT` specifies the replacement string
- `FLAGS` are options to change default behavior of the command

For now, it is enough to know that `s` command is used for search and replace operation.

```
$ # sample command output for stream editing
$ printf '1,2,3,4\na,b,c,d\n'
1,2,3,4
a,b,c,d

$ # for each input line, change only first ',' to '-'
$ printf '1,2,3,4\na,b,c,d\n' | sed 's/,/-/'
1-2,3,4
a-b,c,d

$ # change all matches by adding 'g' flag
$ printf '1,2,3,4\na,b,c,d\n' | sed 's/,/-/g'
1-2-3-4
a-b-c-d
```

Here sample input is created using `printf` command to showcase stream editing. By default, `sed` processes input line by line. To determine a line, `sed` uses the `\n` newline character. The first `sed` command replaces only the first occurrence of `,` with `-` . The second command replaces all occurrences as `g` flag is also used (`g` stands for `global`).

⚠ If you have a file with a different line ending style, you'll need to preprocess it first. For example, a text file downloaded from internet or a file originating from Windows OS would typically have lines ending with `\r\n` (carriage return + line feed). Modern text editors, IDEs and word processors can handle both styles easily.

But every character matters when it comes to command line text processing. See stackoverflow: Why does my tool output overwrite itself and how do I fix it? for a detailed discussion and mitigation methods.

As a good practice, always use single quotes around the script to prevent shell interpretation. Other variations will be discussed later.

Editing file input

Although `sed` derives its name from *stream editing*, it is common to use `sed` for file editing. To do so, append one or more input filenames to the command. You can also specify `stdin` as a source by using `-` as filename. By default, output will go to `stdout` and the input files will not be modified. In-place file editing chapter will discuss how to apply the changes to source file.

Sample input files used in examples are available from learn_gnused repo.

```
$ cat greeting.txt
Hi there
Have a nice day

$ # for each line, change first occurrence of 'day' with 'weekend'
$ sed 's/day/weekend/' greeting.txt
Hi there
Have a nice weekend

$ # change all 'e' to 'E' and save changed text to another file
$ sed 's/e/E/g' greeting.txt > out.txt
$ cat out.txt
Hi thErE
HavE a nicE day
```

In the previous section examples, all input lines had matched the search expression. The first `sed` command here searched for `day`, which did not match the first line of `greeting.txt` file input. By default, even if a line didn't satisfy the search expression, it will be part of the output. You'll see how to get only the modified lines in Print command section.

Cheatsheet and summary

Note	Description
`man sed`	brief manual
`sed --help`	brief description of all the command line options
`info sed`	comprehensive manual
online gnu sed manual	well formatted, easier to read and navigate
`s/REGEXP/REPLACEMENT/FLAGS`	syntax for substitute command
`sed 's/,/-/'`	replace first `,` with `-`
`sed 's/,/-/g'`	replace all `,` with `-`

This introductory chapter covered installation process, documentation and how to search and replace basic text using `sed` from the command line. In coming chapters, you'll learn many more commands and features that make `sed` an important tool when it comes to command line text processing. One such feature is editing files in-place, which will be discussed in the next chapter.

Exercises

Exercise related files are available from exercises folder of learn_gnused repo.

a) Replace `5` with `five` for the given stdin source.

```
$ echo 'They ate 5 apples' | sed ##### add your solution here
They ate five apples
```

b) Replace all occurrences of `0xA0` with `0x50` and `0xFF` with `0x7F` for the given input file.

```
$ cat hex.txt
start address: 0xA0, func1 address: 0xA0
end address: 0xFF, func2 address: 0xB0

$ sed ##### add your solution here
start address: 0x50, func1 address: 0x50
end address: 0x7F, func2 address: 0xB0
```

c) The substitute command searches and replaces sequences of characters. When you need to map one or more characters with another set of corresponding characters, you can use the `y` command. Quoting from the manual:

> **y/src/dst/**
>
> Transliterate any characters in the pattern space which match any of the source-chars with the corresponding character in dest-chars.

Use the `y` command to transform the given input string to get the output string as shown below.

```
$ echo 'goal new user sit eat dinner' | sed ##### add your solution here
gOAl nEw UsEr sIt EAt dInnEr
```

In-place file editing

In the examples presented in previous chapter, the output from `sed` was displayed on the terminal or redirected to another file. This chapter will discuss how to write back the changes to the input file(s) itself using the `-i` command line option. This option can be configured to make changes to the input file(s) with or without creating a backup of original contents. When backups are needed, the original filename can get a prefix or a suffix or both. And the backups can be placed in the same directory or some other directory as needed.

With backup

When an extension is provided as an argument to `-i` option, the original contents of the input file gets preserved as per the extension given. For example, if the input file is `ip.txt` and `-i.orig` is used, the backup file will be named as `ip.txt.orig`

```
$ cat colors.txt
deep blue
light orange
blue delight

$ # no output on terminal as -i option is used
$ # space is NOT allowed between -i and extension
$ sed -i.bkp 's/blue/green/' colors.txt
$ # output from sed is written back to 'colors.txt'
$ cat colors.txt
deep green
light orange
green delight

$ # original file is preserved in 'colors.txt.bkp'
$ cat colors.txt.bkp
deep blue
light orange
blue delight
```

Without backup

Sometimes backups are not desirable. Using `-i` option on its own will prevent creating backups. Be careful though, as changes made cannot be undone. In such cases, test the command with sample input before using `-i` option on actual file. You could also use the option with backup, compare the differences with a `diff` program and then delete the backup.

```
$ cat fruits.txt
banana
papaya
mango

$ sed -i 's/an/AN/g' fruits.txt
$ cat fruits.txt
bANANa
papaya
mANgo
```

Multiple files

Multiple input files are treated individually and the changes are written back to respective files.

```
$ cat f1.txt
have a nice day
bad morning
what a pleasant evening
$ cat f2.txt
worse than ever
too bad

$ sed -i.bkp 's/bad/good/' f1.txt f2.txt
$ ls f?.*
f1.txt  f1.txt.bkp  f2.txt  f2.txt.bkp

$ cat f1.txt
have a nice day
good morning
what a pleasant evening
$ cat f2.txt
worse than ever
too good
```

Prefix backup name

A `*` character in the argument to `-i` option is special. It will get replaced with input filename. This is helpful if you need to use a prefix instead of suffix for the backup filename. Or any other combination that may be needed.

```
$ ls *colors*
colors.txt  colors.txt.bkp

$ # single quotes is used here as * is a special shell character
$ sed -i'bkp.*' 's/green/yellow/' colors.txt
$ ls *colors*
bkp.colors.txt  colors.txt  colors.txt.bkp
```

Place backups in different directory

The `*` trick can also be used to place the backups in another directory instead of the parent directory of input files. The backup directory should already exist for this to work.

```
$ mkdir backups
$ sed -i'backups/*' 's/good/nice/' f1.txt f2.txt
$ ls backups/
f1.txt  f2.txt
```

Cheatsheet and summary

Note	Description
`-i`	after processing, write back changes to input file itself
	changes made cannot be undone, so use this option with caution
`-i.bkp`	in addition to in-place editing, preserve original contents to a file
	whose name is derived from input filename and `.bkp` as a suffix
`-i'bkp.*'`	`*` here gets replaced with input filename
	thus providing a way to add a prefix instead of a suffix
`-i'backups/*'`	this will place the back up copy in a different existing directory
	instead of source directory

This chapter discussed about the `-i` option which is useful when you need to edit a file in-place. This is particularly useful in automation scripts. But, do ensure that you have tested the `sed` command before applying to actual files if you need to use this option without creating backups. In the next chapter, you'll learn filtering features of `sed` and how that helps to apply commands to only certain input lines instead of all the lines.

Exercises

a) For the input file `text.txt`, replace all occurrences of `in` with `an` and write back the changes to `text.txt` itself. The original contents should get saved to `text.txt.orig`

```
$ cat text.txt
can ran want plant
tin fin fit mine line
$ sed ##### add your solution here

$ cat text.txt
can ran want plant
tan fan fit mane lane
$ cat text.txt.orig
can ran want plant
tin fin fit mine line
```

b) For the input file `text.txt`, replace all occurrences of `an` with `in` and write back the changes to `text.txt` itself. Do not create backups for this exercise. Note that you should have solved the previous exercise before starting this one.

```
$ cat text.txt
can ran want plant
tan fan fit mane lane
$ sed ##### add your solution here

$ cat text.txt
cin rin wint plint
tin fin fit mine line
$ diff text.txt text.txt.orig
1c1
< cin rin wint plint
---
> can ran want plant
```

c) For the input file `copyright.txt`, replace `copyright: 2018` with `copyright: 2019` and write back the changes to `copyright.txt` itself. The original contents should get saved to `2018_copyright.txt.bkp`

```
$ cat copyright.txt
bla bla 2015 bla
blah 2018 blah
bla bla bla
copyright: 2018
$ sed ##### add your solution here

$ cat copyright.txt
bla bla 2015 bla
blah 2018 blah
bla bla bla
copyright: 2019
$ cat 2018_copyright.txt.bkp
bla bla 2015 bla
blah 2018 blah
bla bla bla
copyright: 2018
```

d) In the code sample shown below, two files are created by redirecting output of `echo` command. Then a `sed` command is used to edit `b1.txt` in-place as well as create a backup named `bkp.b1.txt`. Will the `sed` command work as expected? If not, why?

```
$ echo '2 apples' > b1.txt
$ echo '5 bananas' > -ibkp.txt
$ sed -ibkp.* 's/2/two/' b1.txt
```

Selective editing

By default, `sed` acts on entire file. Many a times, you only want to act upon specific portions of file. To that end, `sed` has features to filter lines, similar to tools like `grep`, `head` and `tail`. `sed` can replicate most of `grep`'s filtering features without too much fuss. And has features like line number based filtering, selecting lines between two patterns, relative addressing, etc which isn't possible with `grep`. If you are familiar with functional programming, you would have come across **map, filter, reduce** paradigm. A typical task with `sed` involves filtering subset of input and then modifying (mapping) them. Sometimes, the subset is entire input file, as seen in the examples of previous chapters.

> A tool optimized for a particular functionality should be preferred where possible. `grep`, `head` and `tail` would be better performance wise compared to `sed` for equivalent line filtering solutions.

For some of the examples, equivalent commands will be shown as comments for learning purposes.

Conditional execution

As seen earlier, the syntax for substitute command is `s/REGEXP/REPLACEMENT/FLAGS`. The `/REGEXP/FLAGS` portion can be used as a conditional expression to allow commands to execute only for the lines matching the pattern.

```
$ # change commas to hyphens only if the input line contains '2'
$ # space between the filter and command is optional
$ printf '1,2,3,4\na,b,c,d\n' | sed '/2/ s/,/-/g'
1-2-3-4
a,b,c,d
```

Use `/REGEXP/FLAGS!` to act upon lines other than the matching ones.

```
$ # change commas to hyphens if the input line does NOT contain '2'
$ # space around ! is optional
$ printf '1,2,3,4\na,b,c,d\n' | sed '/2/! s/,/-/g'
1,2,3,4
a-b-c-d
```

`/REGEXP/` is one of the ways to define a filter in `sed`, termed as **address** in the manual. Others will be covered in sections to come in this chapter.

Delete command

To **d**elete the filtered lines, use the `d` command. Recall that all input lines are printed by default.

```
$ # same as: grep -v 'at'
$ printf 'sea\neat\ndrop\n' | sed '/at/d'
sea
drop
```

To get the default `grep` filtering, use `!d` combination. Sometimes, negative logic can get confusing to use. It boils down to personal preference, similar to choosing between `if` and `unless` conditionals in programming languages.

```
$ # same as: grep 'at'
$ printf 'sea\neat\ndrop\n' | sed '/at/!d'
eat
```

Using an **address** is optional. So, for example, `sed '!d' file` would be equivalent to `cat file` command.

Print command

To **p**rint the filtered lines, use the `p` command. But, recall that all input lines are printed by default. So, this command is typically used in combination with `-n` command line option, which would turn off the default printing.

```
$ cat programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan

Some people, when confronted with a problem, think - I know, I will
use regular expressions. Now they have two problems by Jamie Zawinski

A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick

$ # same as: grep 'twice' programming_quotes.txt
$ sed -n '/twice/p' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
$ # same as: grep 'e th' programming_quotes.txt
$ sed -n '/e th/p' programming_quotes.txt
Therefore, if you write the code as cleverly as possible, you are,
A language that does not affect the way you think about programming,
```

The substitute command provides `p` as a flag. In such a case, the modified line would be printed only if the substitution succeeded.

```
$ # same as: grep '1' programming_quotes.txt | sed 's/1/one/g'
$ sed -n 's/1/one/gp' programming_quotes.txt
naming things, and off-by-one errors by Leon Bambrick

$ # filter + substitution + p combination
$ # same as: grep 'not' programming_quotes.txt | sed 's/in/**/g'
$ sed -n '/not/ s/in/**/gp' programming_quotes.txt
by def**ition, not smart enough to debug it by Brian W. Kernighan
```

```
A language that does not affect the way you th**k about programm**g,
is not worth know**g by Alan Perlis
```

Using `!p` with `-n` option will be equivalent to using `d` command.

```
$ # same as: sed '/at/d'
$ printf 'sea\neat\ndrop\n' | sed -n '/at/!p'
sea
drop
```

Here's an example of using `p` command without the `-n` option.

```
$ # duplicate every line
$ seq 2 | sed 'p'
1
1
2
2
```

Quit commands

Using `q` command will exit `sed` immediately, without any further processing.

```
$ # quits after an input line containing 'if' is found
$ sed '/if/q' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
```

`Q` command is similar to `q` but won't print the matching line.

```
$ # matching line won't be printed
$ sed '/if/Q' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
```

Use `tac` to get all lines starting from last occurrence of the search string with respect to entire file content.

```
$ tac programming_quotes.txt | sed '/not/q' | tac
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

You can optionally provide an exit status (from `0` to `255`) along with the quit commands.

```
$ printf 'sea\neat\ndrop\n' | sed '/at/q2'
sea
eat
$ echo $?
2

$ printf 'sea\neat\ndrop\n' | sed '/at/Q3'
sea
```

```
$ echo $?
3
```

⚠ Be careful if you want to use `q` or `Q` commands with multiple files, as `sed` will stop even if there are other files to process. You could use a mixed address range as a workaround. See also unix.stackexchange: applying q to multiple files.

Multiple commands

Commands seen so far can be specified more than once by separating them using `;` or using the `-e` command line option. See sed manual: Multiple commands syntax for more details.

```
$ # print all input lines as well as modified lines
$ printf 'sea\neat\ndrop\n' | sed -n -e 'p' -e 's/at/AT/p'
sea
eat
eAT
drop

$ # equivalent command to above example using ; instead of -e
$ # space around ; is optional
$ printf 'sea\neat\ndrop\n' | sed -n 'p; s/at/AT/p'
sea
eat
eAT
drop
```

Another way is to separate the commands using a literal newline character. If more than 2-3 lines are needed, it is better to use a sed script instead.

```
$ # here, each command is separated by literal newline character
$ # > at start of line indicates continuation of multiline shell command
$ sed -n '
> /not/ s/in/**/gp
> s/1/one/gp
> s/2/two/gp
> ' programming_quotes.txt
by def**ition, not smart enough to debug it by Brian W. Kernighan
A language that does not affect the way you th**k about programm**g,
is not worth know**g by Alan Perlis
There are two hard problems in computer science: cache invalidation,
naming things, and off-by-one errors by Leon Bambrick
```

⚠ Do not use multiple commands to construct conditional OR of multiple search strings, as you might get lines duplicated in the output. For example, check what output you get for `sed -ne '/use/p' -e '/two/p' programming_quotes.txt` command. You can use regular expression feature alternation for such cases.

To execute multiple commands for a common filter, use `{}` to group the commands. You can also nest them if needed.

```
$ # same as: sed -n 'p; s/at/AT/p'
$ printf 'sea\neat\ndrop\n' | sed '/at/{p; s/at/AT/}'
sea
eat
eAT
drop

$ # spaces around {} is optional
$ printf 'gates\nnot\nused\n' | sed '/e/{s/s/*/g; s/t/*/g}'
ga*e*
not
u*ed
```

Command grouping is an easy way to construct conditional AND of multiple search strings.

```
$ # same as: grep 'in' programming_quotes.txt | grep 'not'
$ sed -n '/in/{/not/p}' programming_quotes.txt
by definition, not smart enough to debug it by Brian W. Kernighan
A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

$ # same as: grep 'in' programming_quotes.txt | grep 'not' | grep 'you'
$ sed -n '/in/{/not/{/you/p}}' programming_quotes.txt
A language that does not affect the way you think about programming,

$ # same as: grep 'not' programming_quotes.txt | grep -v 'you'
$ sed -n '/not/{/you/!p}' programming_quotes.txt
by definition, not smart enough to debug it by Brian W. Kernighan
is not worth knowing by Alan Perlis
```

Other solutions using alternation feature of regular expressions and `sed`'s control structures will be discussed later.

Line addressing

Line numbers can also be used as filtering criteria.

```
$ # here, 3 represents the address for print command
$ # same as: head -n3 programming_quotes.txt | tail -n1 and sed '3!d'
$ sed -n '3p' programming_quotes.txt
by definition, not smart enough to debug it by Brian W. Kernighan

$ # print 2nd and 5th line
$ sed -n '2p; 5p' programming_quotes.txt
Therefore, if you write the code as cleverly as possible, you are,
Some people, when confronted with a problem, think - I know, I will

$ # substitution only on 2nd line
$ printf 'gates\nnot\nused\n' | sed '2 s/t/*/g'
gates
```

```
no*
used
```

As a special case, `$` indicates the last line of the input.

```
$ # same as: tail -n1 programming_quotes.txt
$ sed -n '$p' programming_quotes.txt
naming things, and off-by-1 errors by Leon Bambrick
```

For large input files, use `q` command to avoid processing unnecessary input lines.

```
$ seq 3542 4623452 | sed -n '2452{p; q}'
5993
$ seq 3542 4623452 | sed -n '250p; 2452{p; q}'
3791
5993

$ # here is a sample time comparison
$ time seq 3542 4623452 | sed -n '2452{p; q}' > f1
real    0m0.003s
$ time seq 3542 4623452 | sed -n '2452p' > f2
real    0m0.256s
```

Mimicking `head` command using line addressing and `q` command.

```
$ # same as: seq 23 45 | head -n5
$ seq 23 45 | sed '5q'
23
24
25
26
27
```

Print only line number

The `=` command will display the line numbers of matching lines.

```
$ # gives both line number and matching line
$ grep -n 'not' programming_quotes.txt
3:by definition, not smart enough to debug it by Brian W. Kernighan
8:A language that does not affect the way you think about programming,
9:is not worth knowing by Alan Perlis

$ # gives only line number of matching line
$ # note the use of -n option to avoid default printing
$ sed -n '/not/=' programming_quotes.txt
3
8
9
```

If needed, matching line can also be printed. But there will be a newline character between the matching line and line number.

```
$ sed -n '/off/{=; p}' programming_quotes.txt
12
naming things, and off-by-1 errors by Leon Bambrick

$ sed -n '/off/{p; =}' programming_quotes.txt
naming things, and off-by-1 errors by Leon Bambrick
12
```

Address range

So far, filtering has been based on specific line number or lines matching the given `/REGEXP/FLAGS` pattern. Address range gives the ability to define a starting address and an ending address, separated by a comma.

```
$ # note that all the matching ranges are printed
$ sed -n '/are/,/by/p' programming_quotes.txt
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan
There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick

$ # same as: sed -n '3,8!p'
$ seq 15 24 | sed '3,8d'
15
16
23
24
```

Line numbers and string matching can be mixed.

```
$ sed -n '5,/use/p' programming_quotes.txt
Some people, when confronted with a problem, think - I know, I will
use regular expressions. Now they have two problems by Jamie Zawinski

$ # same as: sed '/smart/Q'
$ # inefficient, but this will work for multiple file inputs
$ sed '/smart/,$d' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
```

If the second filter condition doesn't match, lines starting from first condition to last line of the input will be matched.

```
$ # there's a line containing 'affect' but doesn't have matching pair
$ sed -n '/affect/,/XYZ/p' programming_quotes.txt
A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

The second address will always be used as a filtering condition only from the line that comes after the line that satisfied the first address. For example, if the same search pattern is used for both the addresses, there'll be at least two lines in output (provided there are lines in the input after the first matching line).

```
$ # there's no line containing 'worth' after the 9th line
$ # so, rest of the file gets matched
$ sed -n '9,/worth/p' programming_quotes.txt
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

As a special case, the first address can be `0` if the second one is a search pattern. This allows the search pattern to be matched against first line of the file.

```
$ # same as: sed '/in/q'
$ # inefficient, but this will work for multiple file inputs
$ sed -n '0,/in/p' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.

$ # same as: sed '/not/q'
$ sed -n '0,/not/p' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan
```

Relative addressing

Prefixing `+` to line number as the second address gives relative filtering. This is similar to using `grep -A<num> --no-group-separator` but `grep` will start a new group if a line matches within context lines.

```
$ # line matching 'not' and 2 lines after
$ # won't be same as: grep -A2 --no-group-separator 'not'
$ sed -n '/not/,+2p' programming_quotes.txt
by definition, not smart enough to debug it by Brian W. Kernighan

Some people, when confronted with a problem, think - I know, I will
A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

$ # the first address can be a line number too
$ # helpful when it is programmatically constructed in a script
$ sed -n '5,+1p' programming_quotes.txt
Some people, when confronted with a problem, think - I know, I will
use regular expressions. Now they have two problems by Jamie Zawinski
```

You can construct an arithmetic progression with start and step values separated by the `~` symbol. `i~j` will filter lines numbered `i+0j` , `i+1j` , `i+2j` , `i+3j` , etc. So, `1~2` means all odd numbered lines and `5~3` means 5th, 8th, 11th, etc.

```
$ # print even numbered lines
$ seq 10 | sed -n '2~2p'
2
4
6
8
10

$ # delete lines numbered 2+0*4, 2+1*4, 2+2*4, etc
$ seq 7 | sed '2~4d'
1
3
4
5
7
```

If `i,~j` is used (note the `,`) then the meaning changes completely. After the start address, the closest line number which is a multiple of `j` will mark the end address. The start address can be specified using search pattern as well.

```
$ # here, closest multiple of 4 is 4th line
$ seq 10 | sed -n '2,~4p'
2
3
4
$ # here, closest multiple of 4 is 8th line
$ seq 10 | sed -n '5,~4p'
5
6
7
8

$ # line matching on 'regular' is 6th line, so ending is 9th line
$ sed -n '/regular/,~3p' programming_quotes.txt
use regular expressions. Now they have two problems by Jamie Zawinski

A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis
```

n and N commands

So far, the commands used have all been processing only one line at a time. The address range option provides the ability to act upon a group of lines, but the commands still operate one line at a time for that group. There are cases when you want a command to handle a string that contains multiple lines. As mentioned in the preface, this book will not cover advanced commands related to multiline processing and I highly recommend using `awk` or `perl` for such scenarios. However, this section will introduce two commands `n` and `N` which are relatively easier to use and will be seen in coming chapters as well.

This is also a good place to give more details about how `sed` works. Quoting from sed manual: How sed Works:

> sed maintains two data buffers: the active pattern space, and the auxiliary hold space. Both are initially empty.
> sed operates by performing the following cycle on each line of input: first, sed reads one line from the input stream, removes any trailing newline, and places it in the pattern space. Then commands are executed; each command can have an address associated to it: addresses are a kind of condition code, and a command is only executed if the condition is verified before the command is to be executed.
> When the end of the script is reached, unless the -n option is in use, the contents of pattern space are printed out to the output stream, adding back the trailing newline if it was removed. Then the next cycle starts for the next input line.

The **pattern space** buffer has only contained single line of input in all the examples seen so far. By using `n` and `N` commands, you can change the contents of pattern space and use commands to act upon entire contents of this data buffer. For example, you can perform substitution on two or more lines at once.

First up, the `n` command. Quoting from sed manual: Often-Used Commands:

> If auto-print is not disabled, print the pattern space, then, regardless, replace the pattern space with the next line of input. If there is no more input then sed exits without processing any more commands.

```
$ # same as: sed -n '2~2p'
$ # n will replace pattern space with next line of input
$ # as -n option is used, the replaced line won't be printed
$ # then the new line is printed as p command is used
$ seq 10 | sed -n 'n; p'
2
4
6
8
10

$ # if line contains 't', replace pattern space with next line
$ # substitute all 't' with 'TTT' for the new line thus fetched
$ # note that 't' wasn't substituted in the line that got replaced
$ # replaced pattern space gets printed as -n option is NOT used here
$ printf 'gates\nnot\nused\n' | sed '/t/{n; s/t/TTT/g}'
gates
noTTT
used
```

Next, the `N` command. Quoting from sed manual: Less Frequently-Used Commands:

> Add a newline to the pattern space, then append the next line of input to the pattern space. If there is no more input then sed exits without processing any more commands.
> When -z is used, a zero byte (the ascii ‘NUL’ character) is added between the lines (instead of a new line).

```
$ # append next line to pattern space
$ # and then replace newline character with colon character
$ seq 7 | sed 'N; s/\n/:/'
1:2
3:4
5:6
7

$ # if line contains 'at', the next line gets appended to pattern space
$ # then the substitution is performed on the two lines in the buffer
$ printf 'gates\nnot\nused\n' | sed '/at/{N; s/s\nnot/d/}'
gated
used
```

See also sed manual: N command on the last line. Escape sequences like `\n` will be discussed in detail later.

See grymoire: sed tutorial if you wish to explore about the data buffers in detail and learn about the various multiline commands.

Cheatsheet and summary

Note	Description
`ADDR cmd`	Execute cmd only if input line satisfies the ADDR condition
	`ADDR` can be REGEXP or line number or a combination of them
`/at/d`	delete all lines based on the given REGEXP
`/at/!d`	don't delete lines matching the given REGEXP
`/twice/p`	print all lines based on the given REGEXP
	as print is default action, usually `p` is paired with `-n` option
`/not/ s/in/**/gp`	substitute only if line matches given REGEXP
	and print only if substitution succeeds
`/if/q`	quit immediately after printing current pattern space
	further input files, if any, won't be processed
`/if/Q`	quit immediately without printing current pattern space
`/at/q2`	both `q` and `Q` can additionally use `0-255` as exit code
`-e 'cmd1' -e 'cmd2'`	execute multiple commands one after the other
`cmd1; cmd2`	execute multiple commands one after the other
	note that not all commands can be constructed this way
	commands can also be separated by literal newline character
`ADDR {cmds}`	group one or more commands to be executed for given ADDR
	groups can be nested as well
	ex: `/in/{/not/{/you/p}}` conditional AND of 3 REGEXPs
`2p`	line addressing, print only 2nd line
`$`	special address to indicate last line of input
`2452{p; q}`	quit early to avoid processing unnecessary lines
`/not/=`	print line number instead of matching line
`ADDR1,ADDR2`	start and end addresses to operate upon

Note	Description
	if ADDR2 doesn't match, lines till end of file gets processed
`/are/,/by/p`	print all groups of line matching the REGEXPs
`3,8d`	delete lines numbered 3 to 8
`5,/use/p`	line number and REGEXP can be mixed
`0,/not/p`	inefficient equivalent of `/not/q` but works for multiple files
`ADDR,+N`	all lines matching the ADDR and `N` lines after
`i~j`	arithmetic progression with `i` as start and `j` as step
`ADDR,~j`	closest multiple of `j` wrt line matching the ADDR
pattern space	active data buffer, commands work on this content
`n`	if `-n` option isn't used, pattern space gets printed
	and then pattern space is replaced with the next line of input
	exit without executing other commands if there's no more input
`N`	add newline (or NUL for `-z`) to the pattern space
	and then append next line of input
	exit without executing other commands if there's no more input

This chapter introduced the filtering capabilities of `sed` and how it can be combined with `sed` commands to process only lines of interest instead of entire input file. Filtering can be specified using a REGEXP, line number or a combination of them. You also learnt various ways to compose multiple `sed` commands. In the next chapter, you will learn syntax and features of regular expressions as implemented in `sed` command.

Exercises

a) Remove only the third line of given input.

```
$ seq 34 37 | sed ##### add your solution here
34
35
37
```

b) Display only fourth, fifth, sixth and seventh lines for the given input.

```
$ seq 65 78 | sed ##### add your solution here
68
69
70
71
```

c) For the input file `addr.txt` , replace all occurrences of `are` with `are not` and `is` with `is not` only from line number **4** till end of file. Also, only the lines that were changed should be displayed in the output.

```
$ cat addr.txt
Hello World
How are you
This game is good
Today is sunny
```

```
12345
You are funny

$ sed ##### add your solution here
Today is not sunny
You are not funny
```

d) Use `sed` to get the output shown below for the given input. You'll have to first understand the logic behind input to output transformation and then use commands introduced in this chapter to construct a solution.

```
$ seq 15 | sed ##### add your solution here
2
4
7
9
12
14
```

e) For the input file `addr.txt`, display all lines from start of the file till the first occurrence of `game`.

```
$ sed ##### add your solution here
Hello World
How are you
This game is good
```

f) For the input file `addr.txt`, display all lines that contain `is` but not `good`.

```
$ sed ##### add your solution here
Today is sunny
```

g) See Gotchas and Tricks chapter and correct the command to get the output as shown below.

```
$ # wrong output
$ seq 11 | sed 'N; N; s/\n/-/g'
1-2-3
4-5-6
7-8-9
10
11

$ # expected output
$ seq 11 | sed ##### add your solution here
1-2-3
4-5-6
7-8-9
10-11
```

h) For the input file `addr.txt`, add line numbers in the format as shown below.

```
$ sed ##### add your solution here
1
```

```
Hello World
2
How are you
3
This game is good
4
Today is sunny
5
12345
6
You are funny
```

i) For the input file `addr.txt`, print all lines that contain `are` and the line that comes after such a line, if any.

```
$ sed ##### add your solution here
How are you
This game is good
You are funny
```

Bonus: For the above input file, will `sed -n '/is/,+1 p' addr.txt` produce identical results as `grep -A1 'is' addr.txt`? If not, why?

j) Print all lines if their line numbers follow the sequence `1, 15, 29, 43, etc` but not if the line contains `4` in it.

```
$ seq 32 100 | sed ##### add your solution here
32
60
88
```