Eric's sed one-liners are divided into several sections:

* 1. [File spacing](http://www.catonmat.net/blog/sed-one-liners-explained-part-one/) (explained in part one).
* 2. [Numbering](http://www.catonmat.net/blog/sed-one-liners-explained-part-one/) (explained in part one).
* 3. [Text conversion and substitution](http://www.catonmat.net/blog/sed-one-liners-explained-part-one/) (explained in part one).
* 4. [Selective printing of certain lines](http://www.catonmat.net/blog/sed-one-liners-explained-part-one/) (explained in part two).
* **5. Selective deletion of certain lines (explained in this part)**.
* **6. Special applications (explained in this part)**.
* 7. [Release of Sed One-Liners Explained e-book.](http://www.catonmat.net/blog/sed-book/)

**1. File spacing.**

**1. Double-space a file.**

sed G

This sed one-liner uses the 'G' command. If you grabbed the cheat-sheet you'd see that the command 'G' appends to the pattern space. It appends a newline followed by the contents of hold buffer. In this example the hold buffer is empty all the time (only three commands 'h', 'H' and 'x' modify hold buffer), so we end up simply appending a newline to the pattern space. Once all the commands have been processed (in this case just the 'G' command), sed puts the contents of pattern space to output stream followed by a newline. There we have it, two newlines -- one added by the 'G' command and the other by output stream. File has been double spaced.

**2. Double-space a file which already has blank lines in it. Do it so that the output contains no more than one blank line between two lines of text.**

sed '/^$/d;G'

Sed allows to restrict commands only to certain lines. This one-liner operates only on lines that match a regular expression /^$/. Which are those? Those are the empty lines. Note that before doing the regular expression match sed pushes the input line to pattern space. When doing it, sed strips the trailing newline character. The empty lines contain just the newline character, so after they have been put into pattern space, this only character has been removed and pattern space stays empty. Regular expression /^$/ matches an empty pattern space and sed applies 'd' command on it, which deletes the current pattern space, reads in the next line, puts it into the pattern space and aborts the current command, and starts the execution from the beginning. The lines which do not match emptiness get a newline character appended by the 'G' command, just like in one-liner #1.

In general sed allows to restrict operations to certain lines (5th, 27th, etc.), to a range of lines (lines 10-20), to lines matching a pattern (lines containing the word "catonmat"), and to lines between two patterns (lines between "catonmat" and "coders").

**3. Triple-space a file.**

sed 'G;G'

Several sed commands can be combined by separating them by ';'. Such commands get executed one after another. This one-liner does twice what the one-liner #1 does -- appends two newlines (via two 'G' commands) to output.

**4. Undo double-spacing.**

sed 'n;d'

This one-liner assumes that even-numbered lines are always blank. It uses two new commands - 'n' and 'd'. The 'n' command prints out the current pattern space (unless the '-n' flag has been specified), empties the current pattern space and reads in the next line of input. We assumed that even-numbered lines are always blank. This means that 'n' prints the first, third, fifth, ..., etc. line and reads in the following line. The line following the printed line is always an empty line. Now the 'd' command gets executed. The 'd' command deletes the current pattern space, reads in the next line, puts the new line into the pattern space and aborts the current command, and starts the execution from the first sed command. Now the the 'n' commands gets executed again, then 'd', then 'n', etc.

To make it shorter - 'n' prints out the current line, and 'd' deletes the empty line, thus undoing the double-spacing.

**5. Insert a blank line above every line that matches "regex".**

sed '/regex/{x;p;x;}'

This one liner uses the restriction operation together with two new commands - 'x' and 'p'. The 'x' command exchanges the hold buffer with the pattern buffer. The 'p' command duplicates input -- prints out the entire pattern space. This one-liner works the following way: a line is read in pattern space, then the 'x' command exchanges it with the empty hold buffer. Next the 'p' command prints out emptiness followed by a newline, so we get an empty line printed before the actual line. Then 'x' exchanges the hold buffer (which now contains the line) with pattern space again. There are no more commands so sed prints out the pattern space. We have printed a newline followed by the line, or saying it in different words, inserted a blank line above every line.

Also notice the { ... }. This is command grouping. It says, execute all the commands in "..." on the line that matches the restriction operation.

**6. Insert a blank line below every line that matches "regex".**

sed '/regex/G'

This one liner combines restriction operation with the 'G' command, described in one-liner #1. For every line that matches /regex/, sed appends a newline to pattern space. All the other lines that do not match /regex/ just get printed out without modification.

**7. Insert a blank line above and below every line that matches "regex".**

sed '/regex/{x;p;x;G;}'

This one-liner combines one-liners #5, #6 and #1. Lines matching /regex/ get a newline appended before them and printed (x;p;x from #5). Then they are followed by another newline from the 'G' command (one-liner #6 or #1).

**2. Numbering.**

**8. Number each line of a file (named *filename*). Left align the number.**

sed = filename | sed 'N;s/\n/\t/'

One-liners get trickier and trickier. This one-liner is actually two separate one-liners. The first sed one-liner uses a new command called '='. This command operates directly on the output stream and prints the current line number. There is no way to capture the current line number to pattern space. That's why the second one-liner gets called. The output of first one-liner gets piped to the input of second. The second one-liner uses another new command 'N'. The 'N' command appends a newline and the next line to current pattern space. Then the famous 's///' command gets executed which replaces the newline character just appended with a tab. After these operations the line gets printed out.

To make it clear what '=' does, take a look at this example file:

line one

line two

line three

Running the first one-liner 'sed = filename', produces output:

1

line one

2

line two

3

line three

Now, the 'N' command of the second one-liner joins these lines with a newline character:

1\nline one

2\nline two

3\nline three

The 's/\n/\t/' replaces the newline chars with tabs, so we end up with:

1 line one

2 line two

3 line three

The example is a little inaccurate as line joining with a newline char happens line after line, not on all lines at once.

**9. Number each line of a file (named *filename*). Right align the number.**

sed = filename | sed 'N; s/^/ /; s/ \*\(.\{6,\}\)\n/\1 /'

This one-liner is also actually two one-liners. The first one liner numbers the lines, just like #8. The second one-liner uses the 'N' command to join the line containing the line number with the actual line. Then it uses two substitute commands to right align the number. The first 's' command 's/^/ /' appends 5 white-spaces to the beginning of line. The second 's' command 's/ \*\(.\{6,\}\)\n/\1 /' captures at least six symbols up to a newline and replaces the capture and newline with the back-reference '\1' and two more whitespace to separate line number from the contents of line.

I think it's hard to understand the last part of this sed expression by just reading. Let's look at an example. For clearness I replaced the '\n' newline char with a '@' and whitespace with '-'.

$ echo "-----12@contents" | sed 's/-\*\(.\{6,\}\)@/\1--/'

----12--contents

The regular expression '-\*\(.\{6,\}\)@' (or just '-\*(.{6,})@') tells sed to match some '-' characters followed by at least 6 other characters, followed by a '@' symbol. Sed captures them (remembers them) in \1.

In this example sed matches the first '-' (the '-\*' part of regex), then the following six characters "----12" and '@' (the '\*\(.\{6,\}\)@' part of regex). Now it replaces the matched part of the string "-----12@" with the contents of captured group which is "----12" plus two extra whitespace. The final result is that "-----12@" gets replaced with "----12--".

**10. Number each non-empty line of a file (called filename).**

sed '/./=' filename | sed '/./N; s/\n/ /'

This one-liner is again two one-liners. The output of the first one-liner gets piped to the input of second. The first one-liner filters out lines with at least one character in them. The regular expression '/./' says: match lines with at least one char in them. When the empty lines (containing just a newline) get sent to the pattern space, the newline character gets removed, so the empty lines do not get matched. The second one-liner does the same one-liner #8 did, except that only numbered lines get joined and printed out. Command '/./N' makes sure that empty lines are left as-is.

**11. Count the number of lines in a file (emulates "wc -l").**

sed -n '$='

This one-liner uses a command line switch "-n" to modify sed's behavior. The "-n" switch tells sed not to send the line to output after it has been processed in the pattern space. The only way to make sed output anything with the "-n" switch being on is to use a command that modifies the output stream directly (these commands are '=', 'a', 'c', 'i', 'I', 'p', 'P', 'r' and 'w'). In this one-liner what seems to be the command "$=" is actually a restriction pattern "$" together with the "=" command. The restriction pattern "$" applies the "=" command to the last line only. The "=" command outputs the current line number to standard output. As it is applied to the last line only, this one-liner outputs the number of lines in the file.

**3. Text Conversion and Substitution.**

**12. Convert DOS/Windows newlines (CRLF) to Unix newlines (LF).**

sed 's/.$//'

This one-one liner assumes that all lines end with CR+LF (carriage return + line feed) and we are in a Unix environment. Once the line gets read into pattern space, the newline gets thrown away, so we are left with lines ending in CR. The 's/.$//' command erases the last character by matching the last character of the line (regex '.$') and substituting it with nothing. Now when the pattern space gets output, it gets appended the newline and we are left with lines ending with LF.

The assumption about being in a Unix environment is necessary because the newline that gets appended when the pattern space gets copied to output stream is the newline of that environment.

**13. Another way to convert DOS/Windows newlines (CRLF) to Unix newlines (LF).**

sed 's/^M$//'

This one-liner again assumes that we are in a Unix environment. It erases the carriage return control character ^M. You can usually enter the ^M control char literally by first pressing Ctrl-V (it's control key + v key) and then Ctrl-M.

**14. Yet another way to convert DOS/Windows newlines to Unix newlines.**

sed 's/\x0D$//'

This one-liner assumes that we are on a Unix machine. It also assumes that we use a version of sed that supports hex escape codes, such as GNU sed. The hex value for CR is 0x0D (13 decimal). This one-liner erases this character.

**15-17. Convert Unix newlines (LF) to DOS/Windows newlines (CRLF).**

sed "s/$/`echo -e \\\r`/"

This one-liner also assumes that we are in a Unix environment. It calls shell for help. The 'echo -e \\\r' command inserts a literal carriage return character in the sed expression. The sed "s/$/char/" command appends a character to the end of current pattern space.

**18. Another way to convert Unix newlines (LF) to DOS/Windows newlines (CRLF).**

sed 's/$/\r/'

This one-liner assumes that we use GNU sed. GNU sed is smarter than other seds and can take escape characters in the replace part of s/// command.

**19. Convert Unix newlines (LF) to DOS/Windows newlines (CRLF) from DOS/Windows.**

sed "s/$//"

This one-liner works from DOS/Windows. It's basically a no-op one-liner. It replaces nothing with nothing and then sends out the line to output stream where it gets CRLF appended.

**20. Another way to convert Unix newlines (LF) to DOS/Windows newlines (CRLF) from DOS/Windows.**

sed -n p

This is also a no-op one-liner, just like #19. The shortest one-liner which does the same is:

sed ''

**21. Convert DOS/Windows newlines (LF) to Unix format (CRLF) from DOS/Windows.**

sed "s/\r//"

Eric says that this one-liner works only with UnxUtils sed v4.0.7 or higher. I don't know anything about this version of sed, so let's just trust him. This one-liner strips carriage return (CR) chars from lines. Then when they get output, CRLF gets appended by magic.

Eric mentions that the only way to convert LF to CRLF on a DOS machine is to use tr:

tr -d \r <infile >outfile

**22. Delete leading whitespace (tabs and spaces) from each line.**

sed 's/^[ \t]\*//'

Pretty simple, it matches zero-or-more spaces and tabs at the beginning of the line and replaces them with nothing, i.e. erases them.

**23. Delete trailing whitespace (tabs and spaces) from each line.**

sed 's/[ \t]\*$//'

This one-liner is very similar to #22. It does the same substitution, just matching zero-or-more spaces and tabs at the end of the line, and then erases them.

**24. Delete both leading and trailing whitespace from each line.**

sed 's/^[ \t]\*//;s/[ \t]\*$//'

This one liner combines #22 and #23. First it does what #22 does, erase the leading whitespace, and then it does the same as #23, erase trailing whitespace.

**25. Insert five blank spaces at the beginning of each line.**

sed 's/^/ /'

It does it by matching the null-string at the beginning of line (^) and replaces it with five spaces "     ".

**26. Align lines right on a 79-column width.**

sed -e :a -e 's/^.\{1,78\}$/ &/;ta'

This one-liner uses a new command line option and two new commands. The new command line option is '-e'. It allows to write a sed program in several parts. For example, a sed program with two substitution rules could be written as "sed -e 's/one/two/' -e 's/three/four'" instead of "sed 's/one/two/;s/three/four'". It makes it more readable. In this one-liner the first "-e" creates a label called "a". The ':' command followed by a name crates a named label. The second "-e" uses a new command "t". The "t" command branches to a named label if the last substitute command modified pattern space. This branching technique can be used to create loops in sed. In this one-liner the substitute command left-pads the string (right aligns it) a single whitespace at a time, until the total length of the string exceeds 78 chars. The "&" in substitution command means the matched string.

Translating it in modern language, it would look like this:

while (str.length() <= 78) {

str = " " + str

}

**27. Center all text in the middle of 79-column width.**

sed -e :a -e 's/^.\{1,77\}$/ & /;ta'

This one-liner is very similar to #26, but instead of left padding the line one whitespace character at a time it pads it on both sides until it has reached length of at least 77 chars. Then another two whitespaces get added at the last iteration and it has grown to 79 chars.

Another way to do the same is

sed -e :a -e 's/^.\{1,77\}$/ &/;ta' -e 's/\( \*\)\1/\1/'

This one-liner left pads the string one whitespace char at a time until it has reached length of 78 characters. Then the additional "s/\( \*\)\1/\1/" command gets executed which divides the leading whitespace "in half". This effectively centers the string. Unlike the previous one-liner this one-liner does not add trailing whitespace. It just adds enough leading whitespace to center the string.

**28. Substitute (find and replace) the first occurrence of "foo" with "bar" on each line.**

sed 's/foo/bar/'

This is the simplest sed one-liner possible. It uses the substitute command and applies it once on each line. It substitutes string "foo" with "bar".

**29. Substitute (find and replace) the fourth occurrence of "foo" with "bar" on each line.**

sed 's/foo/bar/4'

This one-liner uses a flag for the substitute command. With no flags the first occurrence of pattern is changed. With a numeric flag like "/1", "/2", etc. only that occurrence is substituted. This one-liner uses numeric flag "/4" which makes it change fourth occurrence on each line.

**30. Substitute (find and replace) all occurrence of "foo" with "bar" on each line.**

sed 's/foo/bar/g'

This one-liner uses another flag. The "/g" flag which stands for global. With global flag set, substitute command does as many substitutions as possible, i.e., all.

**31. Substitute (find and replace) the first occurrence of a repeated occurrence of "foo" with "bar".**

sed 's/\(.\*\)foo\(.\*foo\)/\1bar\2/'

Let's understand this one-liner with an example:

$ echo "this is foo and another foo quux" | sed 's/\(.\*\)foo\(.\*foo\)/\1bar\2/'

this is bar and another foo quux

As you can see, this one liner replaced the first "foo" with "bar".

It did it by using two capturing groups. The first capturing group caught everything before the first "foo". In this example it was text "this is ". The second group caught everything after the first "foo", including the second "foo". In this example " and another foo". The matched text was then replaced with contents of first group "this is " followed by "bar" and contents of second group " and another foo". Since " quux" was not part of the match it was left unchanged. Joining these parts the resulting string is "this is bar and another foo quux", which is exactly what we got from running the one-liner.

**32. Substitute (find and replace) only the last occurrence of "foo" with "bar".**

sed 's/\(.\*\)foo/\1bar/'

This one-liner uses a capturing group that captures everything up to "foo". It replaces the captured group and "foo" with captured group itself (the \1 back-reference) and "bar". It results in the last occurrence of "foo" getting replaced with "bar".

**33. Substitute all occurrences of "foo" with "bar" on all lines that contain "baz".**

sed '/baz/s/foo/bar/g'

This one-liner uses a regular expression to restrict the substitution to lines matching "baz". The lines that do not match "baz" get simply printed out, but those that do match "baz" get the substitution applied.

**34. Substitute all occurrences of "foo" with "bar" on all lines that DO NOT contain "baz".**

sed '/baz/!s/foo/bar/g'

Sed commands can be inverted and applied on lines that DO NOT match a certain pattern. The exclamation "!" before a sed commands does it. In this one-liner the substitution command is applied to the lines that DO NOT match "baz".

**35. Change text "scarlet", "ruby" or "puce" to "red".**

sed 's/scarlet/red/g;s/ruby/red/g;s/puce/red/g'

This one-liner just uses three consecutive substitution commands. The first replaces "scarlet" with "red", the second replaced "ruby" with "red" and the last one replaces "puce" with "red".

If you are using GNU sed, then you can do it simpler:

gsed 's/scarlet\|ruby\|puce/red/g'

GNU sed provides more advanced regular expressions which support alternation. This one-liner uses alternation and the substitute command reads "replace 'scarlet' OR 'ruby' OR 'puce' with 'red'".

**36. Reverse order of lines (emulate "tac" Unix command).**

sed '1!G;h;$!d'

This one-liner acts as the "tac" Unix utility. It's tricky to explain. The easiest way to explain it is by using an example.

Let's use a file with just 3 lines:

$ cat file

foo

bar

baz

Running this one-liner on this file produces the file in reverse order:

$ sed '1!G;h;$!d' file

baz

bar

foo

The first one-liner's command "1!G" gets applied to all the lines which are not the first line. The second command "h" gets applied to all lines. The third command "$!d" gets applied to all lines except the last one.

Let's go through the execution line by line.

Line 1: Only the "h" command gets applied for the first line "foo". It copies this line to hold buffer. Hold buffer now contains "foo". Nothing gets output as the "d" command gets applied.  
Line 2: The "G" command gets applied. It appends the contents of hold buffer to pattern space. The pattern space now contains. "bar\nfoo". The "h" command gets applied, it copies "bar\nfoo" to hold buffer. It now contains "bar\nfoo". Nothing gets output.  
Line 3: The "G" command gets applied. It appends hold buffer to the third line. The pattern space now contains "baz\nbar\nfoo". As this was the last line, "d" does not get applied and the contents of pattern space gets printed. It's "baz\nbar\nfoo". File got reversed.

If we had had more lines, they would have simply get appended to hold buffer in reverse order.

Here is another way to do the same:

sed -n '1!G;h;$p'

It silences the output with "-n" switch and forces the output with "p" command only at the last line.

These two one-liners actually use a lot of memory because they keep the whole file in hold buffer in reverse order before printing it out. Avoid these one-liners for large files.

**37. Reverse a line (emulates "rev" Unix command).**

sed '/\n/!G;s/\(.\)\(.\*\n\)/&\2\1/;//D;s/.//'

This is a very complicated one-liner. I had trouble understanding it the first time I saw it and ended up asking on [comp.unix.shell](http://groups.google.com/group/comp.unix.shell/browse_frm/thread/b3885f5692585f64?tvc=1) for help.

Let's re-format this sed one-liner:

sed '

/\n/ !G

s/\(.\)\(.\*\n\)/&\2\1/

//D

s/.//

'

The first line "/\n/ !G" appends a newline to the end of the pattern space if there was none.

The second line "s/\(.\)\(.\*\n\)/&\2\1/" is a simple s/// expression which groups the first character as \1 and all the others as \2. Then it replaces the whole matched string with "&\2\1", where "&" is the whole matched text ("\1\2"). For example, if the input string is "1234" then after the s/// expression, it becomes "1234\n234\n1".

The third line is "//D". This statement is the key in this one-liner. An empty pattern // matches the last existing regex, so it's exactly the same as: /\(.\)\(.\*\n\)/D. The "D" command deletes from the start of the input till the first newline and then resumes editing with first command in script. It creates a loop. As long as /\(.\)\(.\*\n\)/ is satisfied, sed will resume all previous operations. After several loops, the text in the pattern space becomes "\n4321". Then /\(.\)\(.\*\n\)/ fails and sed goes to the next command.

The fourth line "s/.//" removes the first character in the pattern space which is the newline char. The contents in pattern space becomes "4321" -- reverse of "1234".

There you have it, a line has been reversed.

**38. Join pairs of lines side-by-side (emulates "paste" Unix command).**

sed '$!N;s/\n/ /'

This one-liner joins two consecutive lines with the "N" command. They get joined with a "\n" character between them. The substitute command replaces this newline with a space, thus joining every pair of lines with a whitespace.

**39. Append a line to the next if it ends with a backslash "\".**

sed -e :a -e '/\\$/N; s/\\\n//; ta'

The first expression ':a' creates a named label "a". The second expression looks to see if the current line ends with a backslash "\". If it does, it joins it with the line following it using the "N" command. Then the slash and the newline between joined lines get erased with "s/\\\n//" command. If the substitution was successful we branch to the beginning of expression and do the same again, in hope that we might have another backslash. If the substitution was not successful, the line did not end with a backslash and we print it out.

Here is an example of running this one-liner:

$ cat filename

line one \

line two

line three

$ sed -e :a -e '/\\$/N; s/\\\n//; ta' filename

line one line two

line three

Lines one and two got joined because the first line ended with backslash.

**40. Append a line to the previous if it starts with an equal sign "=".**

sed -e :a -e '$!N;s/\n=/ /;ta' -e 'P;D'

This one-liner also starts with creating a named label "a". Then it tests to see if it is not the last line and appends the next line to the current one with "N" command. If the just appended line starts with a "=", one-liner branches the label "a" to see if there are more lines starting with "=". During this process a substitution gets executed which throws away the newline character which came from joining with "N" and the "=". If the substitution fails, one-liner prints out the pattern space up to the newline character with the "P" command, and deletes the contents of pattern space up to the newline character with "D" command, and repeats the process.

Here is an example of running it:

$ cat filename

line one

=line two

=line three

line four

$ sed -e :a -e '$!N;s/\n=/ /;ta' -e 'P;D' filename

line one line two line three

line four

Lines one, two and three got joined, because lines two and three started with '='. Line four got printed as-is.

**41. Digit group (commify) a numeric string.**

sed -e :a -e 's/\(.\*[0-9]\)\([0-9]\{3\}\)/\1,\2/;ta'

This one-liner turns a string of digits, such as "1234567" to "1,234,567". This is called commifying or digit grouping.

First the one-liner creates a named label "a". Then it captures two groups of digits. The first group is all the digits up to last three digits. The last three digits gets captures in the 2nd group. Then the two matching groups get separated by a comma. Then the same rules get applied to the line again and again until all the numbers have been grouped in groups of three.

Substitution command "\1,\2" separates contents of group one with a comma from the contents of group two.

Here is an example to understand the grouping happening here better. Suppose you have a numeric string "1234567". The first group captures all the numbers until the last three "1234". The second group captures last three numbers "567". They get joined by a comma. Now the string is "1234,567". The same stuff is applied to the string again. Number "1" gets captured in the first group and the numbers "234" in the second. The number string is "1,234,567". Trying to apply the same rules again fail because there is just one digit at the beginning of string, so the string gets printed out and sed moves on to the next line.

If you have GNU sed, you can use a simpler one-liner:

gsed ':a;s/\B[0-9]\{3\}\>/,&/;ta'

This one-liner starts with creating a named label "a" and then loops over the string the same way as the previous one-liner did. The only difference is how groups of three digits get matched. GNU sed has some additional patterns. There are two patterns that make this one-liner work. The first is "\B", which matches anywhere except at a word boundary. It's needed so we did not go beyond word boundary. Look at this example:

$ echo "12345 1234 123" | sed 's/[0-9]\{3\}\>/,&/g'

12,345 1,234 ,123

It's clearly wrong. The last 123 got a comma added. Adding the "\B" makes sure we match the numbers only at word boundary:

$ echo "12345 1234 123" | sed 's/\B[0-9]\{3\}\>/,&/g'

12,345 1,234 123

The second is "\>". It matches the null string at the end of a word. It's necessary because we need to to match the right-most three digits. If we did not have it, the expression would match after the first digit.

**42. Add commas to numbers with decimal points and minus signs.**

gsed -r ':a;s/(^|[^0-9.])([0-9]+)([0-9]{3})/\1\2,\3/g;ta'

This one-liner works in GNU sed only. It turns on extended regular expression support with the "-r" switch. Then it loops over a line matching three groups and separates the first two from the third with a comma.

The first group makes sure we ignore a leading non-digit character, such as + or -. If there is no leading non-digit character, then it just anchors at the beginning of the string which always matches.

The second group matches a bunch of numbers. The third group makes sure the second group does not match too many. It matches 3 consecutive numbers at the end of the string.

Once the groups have been captured, the "\1\2,\3" substitution is done and the expression is looped again, until the whole string has been commified.

**43. Add a blank line after every five lines.**

sed 'n;n;n;n;G;'

The "n" command is called four times in this one-liner. Each time it's called it prints out the current pattern space, empties it and reads in the next line of input. After calling it four times, the fifth line is read into the pattern space and then the "G" command gets called. The "G" command appends a newline to the fifth line. Then the next round of four "n" commands is done. Next time the first "n" command is called it prints out the newlined fifth line, thus inserting a blank line after every 5 lines.

The same can be achieved with GNU sed's step extension:

gsed '0~5G'

GNU sed's step extensions can be generalized as "first~step". It matches every "step"'th line starting with line "first". In this one-liner it matches every 5th line starting with line 0.

## 4. Selective Printing of Certain Lines.

**44. Print the first 10 lines of a file (emulates "head -10").**

sed 10q

This one-liner restricts the "q" (quit) command to line "10". It means that this command gets executed only when sed reads the 10th line. For all the other lines there is no command specified. When there is no command specified, the default action is to print the line as-is. This one-liner prints lines 1-9 unmodified and at 10th line quits. Notice something strange? It was supposed to print first 10 lines of a file, but it seems that it just printed only the first 9... Worry not! The quit command is sneaky in its nature. Upon quitting with "q" command, sed actually prints the contents of pattern space and only then quits. As a result lines 1-10 get printed!

Please see the [first part of the article](http://www.catonmat.net/blog/sed-one-liners-explained-part-one/) for explanation of "pattern space".

**45. Print the first line of a file (emulates "head -1").**

sed q

The explanation of this one-liner is almost the same as of the previous. Sed quits and prints the first line.

A more detailed explanation - after the first line has been placed in the pattern space, sed executes the "q" command. This command forces sed to quit; but due to strange nature of the "q" command, sed also prints the contents of pattern space. As a result, only the first line gets printed.

**46. Print the last 10 lines of a file (emulates "tail -10").**

sed -e :a -e '$q;N;11,$D;ba'

This one-liner is tricky to explain. It always keeps the last 10 lines in pattern space and at the very last line of input it quits and prints them.

I'll try to explain it. The first "-e :a" creates a label called "a". The second "-e" does the following: "$q" - if it is the last line, quit and print the pattern space. If it is not the last line, execute three commands "N", "11,$D" and "ba". The "N" command reads the next line of input and appends it to the pattern space. The line gets separated from the rest of the pattern space by a new line character. The "11,$D" command executes the "D" command if the current line number is greater than or equal to 11 ("11,$" means from 11th line to end of file). The "D" command deletes the portion of pattern space up to the first new line character. The last command "ba" branches to a label named "a" (beginning of script). This guarantees that the pattern space never contains more than 10 lines, because as line 11 gets appended to pattern space, line 1 gets deleted, as line 12 gets appended line 2 gets deleted, etc.

**47. Print the last 2 lines of a file (emulates "tail -2").**

sed '$!N;$!D'

This one-liner is also tricky. First of all, the "$!" address restricts commands "N" and "D" to all the lines except the last line.

Notice how the addresses can be negated. If "$<command>" restricts a command to the last line, then "$!<command>" restricts the command to all **but** the last line. This can be applied to all restriction operations.

In this one-liner the "N" command reads the next line from input and appends it to pattern space. The "D" command deletes everything in pattern space up to the first "\n" symbol. These two commands always keep only the most recently read line in pattern space. When processing the second-to-last line, "N" gets executed and appends the last line to the pattern space. The "D" does not get executed as "N" consumed the last line. At this moment sed quits and prints out the last two lines of the file.

**48. Print the last line of a file (emulates "tail -1").**

sed '$!d'

This one-liner discards all the lines except the last one. The "d" command deletes the current pattern space, reads in the next line, and restarts the execution of commands from the first. In this case it just loops over itself like "dddd...ddd" until it hits the last line. At the last line no command is executed ("$!d" restricted execution of "d" to all the lines but last) and the pattern space gets printed.

Another way to do the same:

sed -n '$p'

The "-n" parameter suppresses automatic printing of pattern space. It means that without an explicit "p" command (or other commands that act directly on the output stream), sed is dead silent. The "p" command stands for "print" and it prints the pattern space. This one-liner calls the "p" command at the very last line of input. All the other lines are silently discarded.

**49. Print next-to-the-last line of a file.**

Eric gives three different one-liners to do this. The first one prints a blank line if the file contains just 1 line:

sed -e '$!{h;d;}' -e x

This one-liner executes the "h;d" commands for all the lines except the last one ("$!" restricts "h;d" commands to all lines except last). The "h" command puts the current line in hold buffer and "d" deletes the current line, and starts execution at the first sed command ("h;d" gets executed again, and again, ...). At every single line, that line gets copied to hold buffer. At the very last line "h;d" does not get executed. At this moment "x" gets a chance to execute. The "x" command exchanges the contents of hold buffer with pattern space. Remember that the previous line is still in the hold buffer. The "x" command puts it back in pattern space, and sed prints it! There you go, the next-to-last line was printed!

In case there is just 1 line in the file, only the "x" command gets executed. As the hold buffer initially is empty, "x" puts emptiness in pattern space (I use word "put" here but it actually exchanges the pattern space with hold space). Now sed prints the contents of pattern space, but it's empty, so sed prints out just a blank line.

The second prints the first line if the file contains just 1 line:

sed -e '1{$q;}' -e '$!{h;d;}' -e x

This sed-one liner is divided in two parts. The first part "1{$q;}" handles the case when the file contains just a single line. The second part "$!{h;d;} x" is exactly the same as in the previous one-liner! Thus, I need to explain just the first part.

The first part says - if it is the first line "1", then execute "$q". The "$q" command means - if it is the last line, then quit. What it effectively does is it quits if the first line is the last line (i.e. file contains just one line). Remember from one-liner #44 that before quitting sed prints the contents of pattern space. As a result, if the file contains just one line, sed prints it.

The third prints nothing for 1 line files:

sed -e '1{$d;}' -e '$!{h;d;}' -e x

This one-liner is again divided in two parts. The first part is "1{$d;}" and the second is exactly the same as in the previous two one-liners. I will explain just the first part.

The first part says - if it is the first line "1", then execute "$d". The "$d" command means - if it is the last line, then delete the pattern space and start all over again. In case the first line is the last (only one line in file), there is nothing more to be done and sed quits, printing nothing.

**50. Print only the lines that match a regular expression (emulates "grep").**

sed -n '/regexp/p'

This one-liner suppresses automatic printing of pattern space with the "-n" switch and makes use of "p" command to print only the lines that match "/regexp/". The lines that do not match this regex get silently discarded. The ones that match get printed. That's it.

Another one-liner that does the same:

sed '/regexp/!d'

This one-liner deletes all the lines that do not match "/regexp/". The other lines get printed by default. The "!" before "d" command inverts the line matching.

**51. Print only the lines that do not match a regular expression (emulates "grep -v").**

sed -n '/regexp/!p'

This one-liner is the inverse of the previous.

The "-n" prevents automatic printing of pattern space. The "/regexp/" restricts the "!p" command only to lines that match "/regexp/", but the "!" switch prevents "p" from acting on these lines. What happens is "p" acts on all lines that do not match "/regexp/", and they get "p"rinted.

sed '/regexp/d'

This one-liner is the inverse of the previous (#50).

This one-liner executed the "d" (delete) command on all lines that match "/regexp/", thus leaving only the lines that do not match. They get printed automatically.

**52. Print the line immediately before regexp, but not the line containing the regexp.**

sed -n '/regexp/{g;1!p;};h'

This one-liner saves each line in hold buffer with "h" command. If a line matches the regexp, the hold buffer (containing the previous line) gets copied to pattern space with "g" command and the pattern space gets printed out with "p" command. The "1!" restricts "p" not to print on the first line (as there are no lines before the first).

**53. Print the line immediately after regexp, but not the line containing the regexp.**

sed -n '/regexp/{n;p;}'

First of all, this one-liner disables automatic printing of pattern space with "-n" command line argument. Then, for all the lines that match "/regexp/", this one-liner executes "n" and "p" commands. The "n" command is the only command that depends on "-n" flag explicitly. If "-n" is specified it will empty the current pattern space and read in the next line of input. If "-n" is not specified, it will print out the current pattern space before emptying it. As in this one-liner "-n" is specified, the "n" command empties the pattern space, reads in the next line and then the "p" command prints that line out.

**54. Print one line before and after regexp. Also print the line matching regexp and its line number. (emulates "grep -A1 -B1").**

sed -n -e '/regexp/{=;x;1!p;g;$!N;p;D;}' -e h

First let's look at "h" command at the end of script. It gets executed on every line and stores the current line in pattern space in hold buffer. The idea of storing the current line in hold buffer is that if the next line matches "/regexp/" then the previous line is available in hold buffer.

Now let's look at the complicated "/regexp/{=;x;1!p;g;$!N;p;D;}" command. It gets executed only if the line matches "/regexp/". The first thing it does is it prints the current line number with "=" command. Then, it exchanges the hold buffer with pattern space by using the "x" command. As I explained, the "h" command at the end of the script makes sure that the hold buffer always contains the previous line. Now we have put it in the pattern space with "x" command. Next, if it's not the first line, "1!p" prints the pattern space, effectively printing the previous line. Now the "g" command gets executed. It copies the original line that was just exchanged with hold buffer back to pattern space. Now the "$!N" executes. If it is not the last line, "N" appends the next line to the current pattern space (and separates them with "\n" char). Pattern space now contains the line that matched "/regexp/" and the next line. The "p" command prints that. "D" deletes the current line (line that matched "/regexp/") from pattern space and finally "h" gets executed again, that puts the contents of pattern space into hold buffer. As "D" deleted the current line, the next line was put in hold buffer.

**55. Grep for "AAA" and "BBB" and "CCC" in any order.**

sed '/AAA/!d; /BBB/!d; /CCC/!d'

This one-liner inverts the "d" command to be executed on lines that do not contain either "AAA", "BBB" or "CCC". If a line does not contain one of them, it gets deleted and sed proceeds to the next line. Only if all three of the patterns are present, does the sed print the line.

**56. Grep for "AAA" and "BBB" and "CCC" in that order.**

sed '/AAA.\*BBB.\*CCC/!d'

This one-liner deletes lines that do not match regexp "/AAA.\*BBB.\*CCC/". For example, a line "AAAfooBBBbarCCC" will get printed but "AAAfooCCCbarBBB" baz will not.

It can also be written as:

sed -n '/AAA.\*BBB.\*CCC/p'

This one-liner prints lines that contain AAA...BBB...CCC in that order.

**57. Grep for "AAA" or "BBB", or "CCC".**

sed -e '/AAA/b' -e '/BBB/b' -e '/CCC/b' -e d

This one-liner uses the "b" command to branch to the end of the script if the line matches "AAA" or "BBB" or "CCC". At the end of the script the line gets implicitly printed. If the line does not match "AAA" or "BBB" or "CCC", the script reaches the "d" command that deletes the line.

gsed '/AAA\|BBB\|CCC/!d'

This one-liner works with GNU sed. GNU sed allows alternation operator | to be used to match separate things. It's a more compact way of saying match "AAA" or "BBB", or "CCC".

If you are using GNU sed, then there is actually no need to escape the pipes |. You may specify the "-r" command line option to use extended regular expressions. This way this one liner becomes:

gsed -r '/AAA|BBB|CCC/!d'

or

gsed -rn '/AAA|BBB|CCC/p'

**58. Print a paragraph that contains "AAA". (Paragraphs are separated by blank lines).**

sed -e '/./{H;$!d;}' -e 'x;/AAA/!d;'

First notice that this one-liner is divided in two parts for clearness. The first part is "/./{H;$!d;}" and the second part is "x;/AAA/!d".

The first part has an interesting pattern match "/./". What do you think it does? Well, a line separating paragraphs would be a blank line, meaning it would not have any characters in it. This pattern matches only the lines that are not separating paragraphs. These lines get appended to hold buffer with "H" command. They also get prevented from printing with "d" command (except for the last line, when "d" does not get executed ("$!" restricts "d" to all but the last line)). Once sed sees a blank line, the "/./" pattern no longer matches and the second part of one-liner gets executed.

The second part exchanges the hold buffer with pattern space by using the "x" command. The pattern space now contains the whole paragraph of text. Next sed tests if the paragraph contains "AAA". If it does, sed does nothing which results in printing the paragraph. If the paragraph does not contain "AAA", sed executes the "d" command that deletes it without printing and restarts execution at first command.

**59. Print a paragraph if it contains "AAA" and "BBB" and "CCC" in any order.**

sed -e '/./{H;$!d;}' -e 'x;/AAA/!d;/BBB/!d;/CCC/!d'

This one-liner is also split in two parts for clarity. The first part is exactly the same as the first part of previous one-liner. The second part is very similar to one-liner #55 and also the previous.

The "x" command in the 2nd part does exactly the same as in previous one-liner, it exchanges the hold buffer, that contains the paragraph with pattern space. Next sed does three tests - it tests if the paragraph contains "AAA", "BBB" and "CCC". If the paragraph does not contain even one of them, the "d" command gets executed that purges the paragraph. If it contains all three patterns, sed happily prints the paragraph.

**60. Print a paragraph if it contains "AAA" or "BBB" or "CCC".**

sed -e '/./{H;$!d;}' -e 'x;/AAA/b' -e '/BBB/b' -e '/CCC/b' -e d

The first part is exactly the same as in previous two one-liners and does not require explanation. The second part that happens to be "-e 'x;/AAA/b' -e '/BBB/b' -e '/CCC/b' -e d" is almost exactly the same as in one-liner #57.

The "x" command exchanges the paragraph stored in hold buffer with the pattern space. Then it tests if the pattern space (paragraph) contains "AAA", if it does, sed branches to end of script with "b" command, that happily makes sed print the paragraph. If "AAA" did not match, sed does exactly the same testing for pattern "BBB". If it again did not match, it tests for "CCC". If none of these patterns were found, sed executes the "d" command that deletes everything and restarts this one-liner.

Here is another way to do the same with GNU sed:

gsed '/./{H;$!d;};x;/AAA\|BBB\|CCC/b;d'

This one-liner is exactly the same as previous one. It just compresses the three tests for "AAA", "BBB" or "CCC" into one "/AAA\|BBB\|CCC/" as explained in one-liner #57.

**61. Print only the lines that are 65 characters in length or more.**

sed -n '/^.\{65\}/p'

This one-liner prints lines that are 65 characters in length or more. It does it by using a regular expression "^.{65}" that matches any 65 characters at the beginning of line. If there are less than 65 characters, the regex does not match and the line does not get printed (as automatic printing was disabled with "-n" command line option).

**62. Print only the lines that are less than 65 chars.**

sed -n '/^.\{65\}/!p'

This one-liner inverts the previous one. If the line matches 65 characters, then it is not printed "!p". If it does not match, it gets printed.

Another way to do the same:

sed '/^.\{65\}/d'

This one-liner deletes all lines that match 65 characters. All others implicitly get printed.

**63. Print section of a file from a regex to end of file.**

sed -n '/regexp/,$p'

This one-liner uses a tricky range match "/regex/,$". It matches lines starting from the first line that matches "/regex/" to the end of file "$". The "p" command prints these lines. All other lines get silently discarded.

**64. Print lines 8-12 (inclusive) of a file.**

sed -n '8,12p'

This is another type of range match. This range matches a section of lines between two lines numbers (inclusive). In this case it's lines [8 to 12].

sed '8,12!d'

This is the same one-liner, just written differently. It deletes lines that are outside of range [8, 12] and prints those in this range.

**65. Print line number 52.**

sed -n '52p'

This one-liner restricts the "p" command to line "52". Only this line gets "p"rinted.

sed '52!d'

This one-liner deletes all lines except line 52. Line 52 gets printed.

sed '52q;d'

This one is the smartest. It quits at line 52 with "q" command. The previous two one-liners would loop over all the remaining lines and do nothing. Remember from one-liner #44 that quit command prints the pattern space with it. The "d" command makes sure that no other line gets printed while sed gets to line 52.

**66. Beginning at line 3, print every 7th line.**

gsed -n '3~7p'

This one-liner uses a line range match extension of GNU sed. A line range in format "first~step" matches every step'th line starting from first. In this one-liner it's "3~7", meaning match every 7th line starting from 3rd. The "-n" flag prevents printing any other lines, and "p" in "3~7p" prints the matched line.

For everyone else, this one-liner works:

sed -n '3,${p;n;n;n;n;n;n;}'

This one-liner executes commands "p;n;n;n;n;n;n" for lines starting the 3rd line. The "3,$" is a line range match that restricts commands by line numbers. The "$" means end of file and "3" means 3rd line.

The "p;n;n;n;n;n;n" command prints the line, then skips 6, prints the 7th, skips 6, prints the 14th, etc. As it starts executing at line 3, the effect is - print line 3, skip 6, print line 10, skip 6, print line 17, ...**.** That is, print every 7th line beginning at 3rd.

**67. Print section of lines between two regular expressions (inclusive).**

sed -n '/Iowa/,/Montana/p'

This one-liner prints all the lines between the first line that matches a regular expression "Iowa" and the first line that matches a regular expression "Montana".

It uses a range match "/start/,/finish/" that matches all lines starting from a line that matches "start" and ending with the first line that matches "finish".

## 5. Selective Deletion of Certain Lines

**68. Print all lines in the file except a section between two regular expressions.**

sed '/Iowa/,/Montana/d'

This one-liner continues where the previous left off. [One-liner #67](http://www.catonmat.net/blog/sed-one-liners-explained-part-two/) used the range match "/start/,/finish/" to print lines between two regular expressions (inclusive). This one-liner, on the other hand, deletes lines between two regular expressions and prints all the lines outside this range. Just to remind you, a range "/start/,/finish/" matches all lines starting from the first line that matches a regular expression "/start/" to the first line that matches a regular expression "/finish/". In this particular one-liner the "d", delete, command is applied to these lines. The delete command prevents the matching lines from ever seeing the light.

For example, suppose your input to this one-liner was:

Florida

<strong>Iowa

New York

San Jose

Montana</strong>

Texas

Fairbanks

Then after the sed program has finished running, the output is:

Florida

Texas

Fairbanks

We see this output because the lines from Iowa to Montana matched the "/Iowa/,/Montana/" range match (i put the matched lines in bold) and were deleted.

**69. Delete duplicate, consecutive lines from a file (emulates "uniq").**

sed '$!N; /^\(.\*\)\n\1$/!P; D'

This one-liner acts as the "uniq" Unix utility. So how does it work? First of all, for every line that is not the very last line of input, sed appends the next line to the pattern space by the "N" command. The "N" command is restricted to all but the last line by "$!" restriction pattern. The newly appended line is separated from the previous line by the "\n" character. Next, the pattern space is matched against "/^\(.\*\)\n\1$/" regular expression. This regular expression captures the previous line up to "\n" character and saves it in the match group "\1". Then it tests if the newly appended line is the same as the previous one. If it is not, the "P" gets executed. If it is, the "P" command does not get executed. The "P" command prints everything in the pattern space up to the first "\n" character. Next the "D" command executes and deletes everything up to the first "\n" char, leaving only the newly read line in pattern space. It also forces the sed script to begin from the first command.

This way it loops over all lines, comparing two consecutive lines. If they are equal, the first line gets deleted, and a new line gets appended to what's left. If they are not equal, the first one gets deleted, and deleted.

I think it's hard to understand what is going on from this description. I'll illustrate it with an example. Suppose this is the input:

foo

foo

foo

bar

bar

baz

The first thing sed does is it reads the first line of input in pattern space. The pattern space now contains "foo". Now the "N" command executed. The pattern space now contains "foo\nfoo". Next the pattern space is tested against "/^\(.\*\)\n\1$/" regular expression. This regular expression matches because "\(.\*\)" is "foo" and "/^\(.\*\)\n\1$/" is "foo\nfoo", exactly what we have in the pattern space. As it matched, the "P" command does not get executed. Now the "D" command executes, deleting the everything up to first "\n" from pattern space. The pattern space now contains just "foo". The "D" command forces sed to start from the first command. Now the "N" is executed again, the pattern space now contains "foo\nfoo" again and the same thing happens, "P" does not get executed and "D" deletes the first "foo", leaving the pattern space with just "foo" in it. Now the "N" gets executed once again, this time "bar" gets appended to pattern space. It contains "foo\nbar" now. The regular expression "/^\(.\*\)\n\1$/" does not match and "P" gets executed, printing "foo". After that "D" gets executed wiping "foo" from pattern space. The pattern space now contains "bar". The commands restart and "N" gets executed, it appends the next "bar" to current pattern space. Now it contains "bar\nbar". Just like with "foo\nfoo", nothing gets printed, and "D" deletes the first "bar", leaving pattern space with "bar". The one-liner restarts its execution. Now "N" reads in the final line "baz". The pattern space contains "bar\nbaz" which does not match the regular expression. The "P" prints out the "bar" and "D" deletes "bar". Now "N" does not get executed because we are at the last line of input. The "$!N" restricts "N" to all lines but last. At this moment pattern space contains only the last "baz", the regular expression does not match, so "baz" gets printed. The "D" command executes, emptying the pattern space. There is no more input and sed quits.

The output for this example is:

foo

bar

baz

I think this is one of the most detailed explanations I have written about a single one liner. :)

**70. Delete duplicate, nonconsecutive lines from a file.**

sed -n 'G; s/\n/&&/; /^\([ -~]\*\n\).\*\n\1/d; s/\n//; h; P'

This is a very tricky one-liner. It stores the unique lines in hold buffer and at each newly read line, tests if the new line already is in the hold buffer. If it is, then the new line is purged. If it's not, then it's saved in hold buffer for future tests and printed.

A more detailed description - at each line this one-liner appends the contents of hold buffer to pattern space with "G" command. The appended string gets separated from the existing contents of pattern space by "\n" character. Next, a substitution is made to that substitutes the "\n" character with two "\n\n". The substitute command "s/\n/&&/" does that. The "&" means the matched string. As the matched string was "\n", then "&&" is two copies of it "\n\n". Next, a test "/^\([ -~]\*\n\).\*\n\1/" is done to see if the contents of group capture group 1 is repeated. The capture group 1 is all the characters from space " " to "~" (which include all printable chars). The "[ -~]\*" matches that. Replacing one "\n" with two was the key idea here. As "\([ -~]\*\n\)" is greedy (matches as much as possible), the double newline makes sure that it matches as little text as possible. If the test is successful, the current input line was already seen and "d" purges the whole pattern space and starts script execution from the beginning. If the test was not successful, the doubled "\n\n" gets replaced with a single "\n" by "s/\n//" command. Then "h" copies the whole string to hold buffer, and "P" prints the new line.

**71. Delete all lines except duplicate consecutive lines (emulates "uniq -d").**

sed '$!N; s/^\(.\*\)\n\1$/\1/; t; D'

This sed one-liner prints only the duplicate lines. This sed one-liner starts with reading in the next line from input with the "N" command. As I already mentioned, the current line and the next get separated by "\n" character after "N" executes. This one-liner also restrics "N" to all lines but last with "$!" restriction. Now a substitution "s/^\(.\*\)\n\1$/\1/" is tried. Similarly to one-liner #69, this substitution replaces two repeating strings with one. For example, a string "foo\nfoo" gets replaced with just "foo". Now, if this substitution was successful (there was a repeated string), the "t" command takes the script to the end where the current pattern space gets printed automatically. If the substitution was not successful, "D" executes, deleting the non-repeated string. The cycle continues and this way only the duplicate lines get printed once.

Let's take a look at an example. Suppose the input is:

foo

foo

bar

baz

This one-liner reads the first line and immediately executes the "N" command. The pattern space now is "foo\nfoo". The substitution "s/^\(.\*\)\n\1$/\1/" is tried and it's successful, because "foo" is repeated twice. The pattern space now contains just a single "foo". As the substitution was successful, "t" command branches to the end of the script. At this moment "foo" gets printed. Now the cycle repeats. Sed reads in "bar", the "N" command appends "baz" to "bar". The pattern space now is "bar\nbaz". The substitution is tried, but it's not successful, as "bar" is not repeated. As the substitution failed, "t" does nothing and "D" executes, deleting "bar" from pattern space. The pattern space is left with single "baz". Command "N" no longer executes as we reached end of file, substitution fails, "t" fails, and "D" deletes the "baz".

The end result is:

foo

Just as we expected - only the duplicate line got printed.

**72. Delete the first 10 lines of a file.**

sed '1,10d'

This one-liner restricts the "d" command to a range of lines by number. The "1,10" means a range matching lines 1 to 10 inclusive. On each of the lines the "d" command gets executed. It deletes the current pattern space, and restarts the commands from beginning. The default action for lines > 10 is to print the line.

**73. Delete the last line of a file.**

sed '$d'

This one-liner restricts the "d" command to the last line of file. It's done by specifying the special char "$" as the line to match. It matches only the last line. The last line gets deleted, but the others get printed implicitly.

**74. Delete the last 2 lines of a file.**

sed 'N;$!P;$!D;$d'

This one-liner always keeps two lines in the pattern space. At the very last line, it just does not output these last two. All the others before last two get output implicitly. Let's see how it does it. As soon as sed reads the first line of input in pattern space, it executes the first command "N". It places the 2nd line of input in pattern space. The next two commands "$!P" and "$!D" print the first part of pattern space up to newline character, and delete this part from pattern space. They keep doing it until the very last line gets appended to pattern space by "N" command. At this moment the last two lines are in pattern space and "$d" executes, deleting them both. That's it. Last two lines got deleted.

If there is just one line of data, then it outputs it.

**75. Delete the last 10 lines of a file.**

sed -e :a -e '$d;N;2,10ba' -e 'P;D'

This is really straight forward one-liner. It always keeps 10 lines in pattern-space, by appending each new input line with "N", and deleting the 11th excessive line with "D". Once the end of file is reached, it "d" the whole pattern space, deleting the last 10 lines.

sed -n -e :a -e '1,10!{P;N;D;};N;ba'

This is also a straight forward one-liner. For the lines that are not 1-10, it appends them to pattern space with "N". For lines > 10, it prints the first line in pattern space with "P", appends another line with "N" and deletes the printed line with "D". The "D" command causes sed to branch to the beginning of script! The "N;ba" at the end never, ever gets executed again for lines > 10. It keeps looping this way "P", "N", "D", always keeping 10 lines in pattern space and printing line-10 on each cycle. The "N" command causes script to quit if it tries to read past end of file.

**76. Delete every 8th line.**

gsed '0~8d'

This one-liner only works with GNU Sed only. It uses a special address range match "first~step" that matches every step'th line starting with the first. In this one-liner first is 0 and step is 8. Zero is not a valid physical line number, so the very first line of input does not match. The first line to match is 8th, then 16th, then 24th, etc. Each line that matches is deleted by "d" command.

sed 'n;n;n;n;n;n;n;d;'

This is a portable version. The "n" command prints the current pattern space, empties it, and reads in the next line. It does so for every 7 lines, and 8th line gets deleted with "d". This process continues until all input has been processed.

**77. Delete lines that match regular expression pattern.**

sed '/pattern/d'

This one-liner executes the "d" command on all lines that match "/pattern/". The "d" command deletes the line and skips to the next line.

**78. Delete all blank lines in a file (emulates "grep '.'".**

sed '/^$/d'

The regular expression "/^$/" in this one-liner tests if the beginning of line matches the end of the line. Only the empty lines have this property and sed deletes them.

Another way to do the same is:

sed '/./!d'

This one-liner tests if the line matches at least one character. The dot "." in the regular expression matches any character. An empty line does not have any characters and it does not match this regular expression. Sed deletes all the lines that do not match this regular expression.

**79. Delete all consecutive blank lines from a file (emulates "cat -s").**

sed '/./,/^$/!d'

This one-liner leaves one blank line at the end of the file, if there are multiple blanks at the end. Other than that, all consecutive blanks are stripped.

It uses an inverse range match "/start/,/finish/!" to "d" delete lines from first blank line, to first non-blank, non-inclusive.

sed '/^$/N;/\n$/D'

This one-liner leaves one blank line at the beginning and end of the file, if there are multiple blanks at both sides. Other than that, all consecutive blanks are stripped.

The consecutive empty lines get appended in pattern space by "/^$/N" command. The "/\n$/D" command matches and deletes blanks until only 1 is left. At that moment it no longer matches, and the line is output.

**80. Delete all consecutive blank lines from a file except the first two.**

sed '/^$/N;/\n$/N;//D'

In case of > 2 blank lines, this one-liner trims them down to two. There is a catch to this one-liner. Let me explain it first. See the last command "//D"? It's a shortcut for "/previous-match/D". In this case it's shortcut for "/\n$/D". Alright, now the one-liner itself. On every empty line, it appends the next to current pattern space with "/^$/N" command. Next it tests if the line just read in was actually a blank line with "/\n$/", if it is, it reads another line in with "N". At this moment it repeats the same test "/\n$/". If the line was a blank one again, it deletes the first blank line and restarts sed script from the beginning. Notice that at all times only 2 consecutive blank lines are in pattern space. This way any number of blank lines get deleted and only two are left.

**81. Delete all leading blank lines at the top of a file.**

sed '/./,$!d'

This one-liner inverts a match "match from the first non-blank line to end of file". It becomes "match from the beginning of file to last blank line".

**82. Delete all trailing blank lines at the end of a file.**

sed -e :a -e '/^\n\*$/{$d;N;ba' -e '}'

This one-liner accumulates blank lines in pattern space until it either hits end or hits a non-blank line. If it hits end, "$d" deletes the whole pattern space (which contained just the trailing blank lines) and quits. If however, it hits non-blank line, the whole pattern space gets printed implicitly and script continues as if nothing had happened.

This one is a portable version.

gsed -e :a -e '/^\n\*$/N;/\n$/ba'

This is the same script, except a shorter version, made to work with Gnu Sed.

**83. Delete the last line of each paragraph.**

sed -n '/^$/{p;h;};/./{x;/./p;}'

This one-liner always keeps the previous line in hold buffer. It's accomplished by 2nd block of commands "/./{x;/./p;}". In this block, the pattern space (1 line) gets exchanged with hold buffer (1 line) by "x" command and if the hold buffer was not empty, it gets printed by "p". The next moment to note is what happens on the first empty line. That is the line after the paragraph. At this moment "/^$/{p;h;}" gets executed, that prints the blank line (but does not print the last line of paragraph!), and puts the blank line in hold buffer. Once a new paragraph is reached, the script executed just like it was the very first paragraph of the input.

## 6. Special Sed Applications

**84. Remove nroff overstrikes.**

Nroff overstrikes are chars that are formatted to stand out in bold. They are achieved like in old typewriters, where you would do backspace and hit the same key again. In nroff it's key CHAR, CTRL+H, CHAR. This one-liner deletes the CHAR, CTRL+H, leaving just plain CHAR.

sed 's/.^H//g'

Press Ctrl+V and then Ctrl+H to insert ^H literally in sed one-liner. It then uses the substitute command to delete any char "." followed by CTRL+H "^H".

Another way to do the same is use a hex escape expression that works in most recent seds:

sed 's/.\x08//g'

Yet another way is to use "echo" and enable interpretation of backslashed characters:

sed 's/.'`echo -e "\b"`'//g'

**85. Print Usenet/HTTP/Email message header.**

gsed -r '/^\r?$/q'

Usenet, HTTP and Email headers are similar. They are a bunch of text lines, separated from the body of the message with two new lines "\r\n\r\n". Some implementations might even go with just "\n\n". This one-liner quits on the first line that is either empty or contains "\r". In other words, it prints the message header and quits.

**86. Print Usenet/HTTP/Email message body.**

sed '1,/^$/d'

This one-liner uses a range match "1,/^$/" to delete lines starting from 1st, and ending with the first blank line (inclusive). As I explained in the previous one-liner #78 above, "/^$/" matches empty lines. All the lines before first blank line in a Usenet/Email message or a HTTP header are message headers. They get deleted.

**87. Extract subject from an email message.**

sed '/^Subject: \*/!d; s///; q'

This one-liner deletes all lines that do not match "^Subject: ". Then it re-uses the match in "s///" to delete "Subject: " part from the line, leaving just the real subject. Please notice how "s///" is equivalent to "s/previous-match//", where "previous-match" is "^Subject: \*" in this one-liner.

**88. Extract sender information from an email message.**

sed '/^From: \*/!d; s///; q'

This one liner is equivalent to the previous one, except it prints sender information from email.

**89. Extract email address from a "Name Surname <email@domain.com>" string.**

sed 's/.\*< \*//;s/ \*>.\*//;

This one-liner strips all symbols before < symbol (and any whitespace after it), and stips all symbols after > symbol (including whitespace before it). That's it. What's left is email@domain.com.

**90. Add a leading angle bracket and space to each line (quote an email message).**

sed 's/^/> /'

This one-liner substitutes zero-width anchor "^" that matches beginning of line with "> ". As it's a zero-width anchor, the result is that "> " gets added to beginning of each line.

**91. Delete leading angle bracket from each line (unquote an email message).**

sed 's/^> //'

It does what it says, deletes two characters ">" and a space " " from the beginning of each line.

**92. Strip HTML tags.**

sed -e :a -e 's/<[^>]\*>//g;/</N;//ba'

Sed is not made for parsing HTML. This is a very crude version of HTML tag eraser. It starts by creating a branch label named "a". Then on each line it substitutes "<[^>]\*>" with nothing as many times as possible ("g" flag for s/// command). The "<[^>]\*>" expression means match match symbol "<" followed by any other symbols that are not ">", and that ends with ">". This is a common pattern in regular expressions for non-greediness. Next, the one-liner tests if there are any open tags left on the line, if there are "N" reads the next line of input to make it work across multiple lines. "//ba" finally branches to the beginning of the script (it's short for "/previous-expression/ba" which in this case is "/</ba").