Linux sort command

Updated: 11/06/2021 by Computer Hope

The **sort** command sorts the contents of a text file, line by line.

Overview

sort is a simple and very useful command which will rearrange the lines in a text file so that they are sorted, numerically and alphabetically. By default, the rules for sorting are:

- Lines starting with a number will appear before lines starting with a letter.
- Lines starting with a letter that appears earlier in the alphabet will appear before lines starting with a letter that appears later in the alphabet.
- Lines starting with a lowercase letter will appear before lines starting with the same letter in uppercase.

The rules for sorting can be changed according to the options you provide to the **sort** command; these are listed below.

Syntax

```
sort [OPTION]... [FILE]...
```

```
sort [OPTION]... --files0-from=F
```

Options

-b, **--ignore-leading-blanks**	Ignore leading blanks.
-d, **--dictionary-order**	Consider only blanks and alphanumeric characters.
-f, **--ignore-case**	Fold lower case to upper case characters.
-g, **--general-numeric-sort**	Compare according to general numerical value.
-i, **--ignore-nonprinting**	Consider only printable characters.
-M, **--month-sort**	Compare (unknown) < '**JAN**' < ... < '**DEC**'.
-h, **--human-numeric-sort**	Compare human readable numbers (e.g., "**2K**", "**1G**").
-n, **--numeric-sort**	Compare according to string numerical value.

-R, **--random-sort**	Sort by random hash of keys.
--random-source=*FILE*	Get random bytes from *FILE*.
-r, **--reverse**	Reverse the result of comparisons.
--sort=*WORD*	Sort according to *WORD*: general-numeric **-g**, human-numeric **-h**, month **-M**, numeric **-n**, random **-R**, version **-V**.
-V, **--version-sort**	Natural sort of (version) numbers within text.

Other options

--batch-size=*NMERGE*	Merge at most *NMERGE* inputs at once; for more use temp files.
-c, **--check**, **--check=***diagnose-first*	Check for sorted input; do not sort.
-C, **--check=***quiet*, **--check=***silent*	Like **-c**, but do not report first bad line.
--compress-program=*PROG*	Compress temporaries with *PROG*; decompress them with *PROG* **-d**.
--debug	Annotate the part of the line used to sort, and warn about questionable usage to stderr.
--files0-from=*F*	Read input from the files specified by NUL-terminated names in file *F*; If *F* is '-' then read names from standard input.
-k, **--key=***POS1*[,*POS2*]	Start a key at *POS1* (origin 1), end it at *POS2* (default end of line). See *POS* syntax below.
-m, **--merge**	Merge already sorted files; do not sort.
-o, **--output=***FILE*	Write result to FILE instead of standard output.
-s, **--stable**	Stabilize sort by disabling last-resort comparison.
-t, **--field-separator=***SEP*	Use *SEP* instead of non-blank to blank transition.
-T, **--temporary-directory=***DIR*	Use *DIR* for temporaries, not **$TMPDIR** or **/tmp**; multiple options specify multiple directories.
--parallel=*N*	Change the number of sorts run concurrently to *N*.
-u, **--unique**	With **-c**, check for strict ordering; without **-c**, output only the first of an equal run.
-z, **--zero-terminated**	End lines with 0 byte, not newline.
--help	Display a help message, and exit.

--version	Display version information, and exit.

POS takes the form *F*[**.***C*][*OPTS*], where *F* is the field number and *C* the character position in the field; both are origin **1**. If neither **-t** nor **-b** is in effect, characters in a field are counted from the beginning of the preceding whitespace. *OPTS* is one or more single-letter ordering options, which override global ordering options for that key. If no key is given, use the entire line as the key.

SIZE may be followed by the following multiplicative suffixes:

%	1% of memory
b	1
K	1024 (default)

...and so on for **M**, **G**, **T**, **P**, **E**, **Z**, **Y**.

With no *FILE*, or when *FILE* is a dash ("**-**"), **sort** reads from the standard input.

Also, note that the locale specified by the environment affects sort order; set **LC_ALL=C** to get the traditional sort order that uses native byte values.

Examples

Let's say you have a file, **data.txt**, which contains the following ASCII text:

```
apples
oranges
pears
kiwis
bananas
```

To sort the lines in this file alphabetically, use the following command:

```
sort data.txt
```

...which will produce the following output:

```
apples
bananas
kiwis
oranges
pears
```

Note that this command does not actually change the input file, **data.txt**. To write the output to a new file, **output.txt**, redirect the output like this:

```
sort data.txt > output.txt
```

...which will not display any output, but will create the file **output.txt** with the same sorted data from the previous command. To check the output, use the **cat** command:

```
cat output.txt
```

...which displays the sorted data:

```
apples
bananas
kiwis
oranges
pears
```

You can also use the built-in **sort** option **-o**, which allows you to specify an output file:

```
sort -o output.txt data.txt
```

Using the **-o** option is functionally the same as redirecting the output to a file; neither one has an advantage over the other.

Sorting in reverse order

You can perform a reverse-order sort using the **-r** flag. For example, the following command:

```
sort -r data.txt
```

...will produce the following output:

```
pears
oranges
kiwis
bananas
apples
```

Handling mixed-case data

But what about situations where you have a mixture of upper- and lower-case letters at the beginning of your lines? In cases like this, the behavior of **sort** can seem confusing, but really it just needs some

more information from you to sort the data the way you want. Let's take a closer look.

Let's say our input file **data.txt** contains the following data:

```
a
b
A
B
b
c
D
d
C
```

sorting this data without any options, like this:

```
sort data.txt
```

...will produce the following output:

```
a
A
b
b
B
c
C
d
D
```

As you can see, it's sorted alphabetically, with lowercase letters always appearing before uppercase letters. This sort is "case-insensitive", and this is the default for GNU **sort**, which is the version of **sort** used in GNU/Linux.

At this point you might be asking yourself, well, if case-insensitive sorting is the default, then what is the "**-f**/**--ignore-case**" option for? The answer has to do with localization settings and bytewise sorting.

In brief, "localization" refers to what language the operating system uses, which at the most basic level defines what characters it uses. Each letter in the system is represented in a certain order. Changing the locale settings will affect what characters the operating system is using, and – most relevant to sorting – what order they are encoded in. For an example, refer to the United States English ASCII encoding table. As you can see from the table, a capital A ("**A**") is character number 65, and lowercase a ("**a**") is character number 97. So you might expect **sort** to arrange its output so that capital letters come before lowercase letters.

Defining operating system locale is a subject which goes beyond the scope of this page, but for now, it will suffice to say that to achieve bytewise sorting, we need to set the environment variable **LC_ALL** to **C**.

Under the default Linux shell, **bash**, we can accomplish this with the following command:

```
export LC_ALL=C
```

This sets the environment variable **LC_ALL** to the value **C**, which will enforce bytewise sorting. Now if we run the command:

```
sort data.txt
```

...we will see the following output:

```
A
B
C
D
a
b
b
c
d
```

...and *now*, the **-f**/**--ignore-case** option has the following effect:

```
A
a
B
b
b
C
c
D
d
```

...performing a "case-insensitive bytewise" sort.

Tip: If you are using the **join** command in conjunction with **sort**, be aware that there is a known incompatibility between the two programs – unless you define the locale. If you are using **join** and **sort** to process the same input, it is highly recommended that you set **LC_ALL** to **C**, which will standardize the localization used by all programs.

Checking for sorted order

If you just want to check to see if your input file is already sorted, use the **-c** option:

```
sort -c data.txt
```

If your data is unsorted, you will receive an informational message reporting the line number of the first unsorted data, and what the unsorted data is:

```
sort: data.txt:3: disorder: A
```

Sorting multiple files using the output of find

One useful way to sort data is to sort the input of multiple files, using the output of the **find** command. The most reliable (and responsible) way to accomplish this is to specify that **find** produces a NUL-terminated file list as its output, and to pipe that output into **sort** using the **--files0-from** option.

Normally, **find** outputs one file on each line; in other words, it inserts a line break after each file name it outputs. For instance, let's say we have three files named **data1.txt**, **data2.txt**, and **data3.txt**. **find** can generate a list of these files using the following command:

```
find -name "data?.txt"
```

This command uses the question mark wildcard to match any file that has a single character after the word "data" in its name, ending in the extension "**.txt**". It produces the following output:

```
./data1.txt
./data3.txt
./data2.txt
```

It would be nice if we could use this output to tell the **sort** command, "sort the data in any files found by **find** as if they were all one big file." The problem with the standard **find** output is, even though it's easy for humans to read, it can cause problems for other programs that need to read it in. Because file names can include non-standard characters, so in some cases, this format will be read incorrectly by another program.

The correct way to format **find**'s output to be used as a file list for another program is to use the **-print0** option when running **find**. This terminates each file name with the NUL character (ASCII character number zero), which is universally illegal to use in file names. This makes things easier for the program reading the file list, since it knows that any time it sees the NUL character, it can be sure it's at the end of a file name.

So, if we run the previous command with the **-print0** option at the end, like this:

```
find -name "data?.txt" -print0
```

...it will produce the following output:

```
./data1.txt./data3.txt./data2.txt
```

You can't see it, but after each file name is a NUL character. This character is non-printable, so it will not appear on your screen, but it's there, and any programs you pipe this output to (**sort**, for example) will see them.

Tip: Be careful how you word the **find** command. It's important to specify **-print0** last; **find** needs this to be specified after the other options.

Okay, but how do we tell **sort** to read this file list and sort the contents of all those files?

One way to do it is to pipe the **find** output to **sort**, specifying the **--files0-from** option in the **sort** command, and specify the file as a dash ("**-**"), which will read from the standard input. Here's what the command will look like:

```
find -name "data?.txt" -print0 | sort --files0-from=-
```

...and it will output the sorted data of any files located by **find** which matches the pattern **data?.txt**, *as if they were all one file*. This example is a very powerful function of **sort** – give it a try.

Comparing only selected fields of data

Normally, **sort** decides how to sort lines based on the entire line: it compares every character from the first character in a line, to the last one.

If, on the other hand, you want **sort** to compare a limited subset of your data, you can specify which fields to compare using the **-k** option.

For instance, if you have an input file **data.txt** With the following data:

```
01 Joe
02 Marie
03 Albert
04 Dave
```

...and you sort it without any options, like this:

```
sort data.txt
```

...you will receive the following output:

```
01 Joe
02 Marie
```

```
03 Albert
04 Dave
```

...as you can see, nothing was changed from the original data ordering, because of the numbers at the beginning of the line — which were already sorted. However, if you want to sort based on the names, you can use the following command:

```
sort -k 2,2 data.txt
```

This command will sort the *second* field, and ignore the first. (The "k" in "**-k**" stands for "key" — we are defining the "sorting key" used in the comparison.)

Fields are defined as anything separated by whitespace; in this case, an actual space character. Our command above will produce the following output:

```
03 Albert
04 Dave
01 Joe
02 Marie
```

...which is sorted by the second field, listing the lines alphabetically by name, and ignoring the numbers in the sorting process.

You can also specify a more complex **-k** option. The complete positional argument looks like this:

-k *POS1*,*POS2*

...where *POS1* is the starting field position, and *POS2* is the ending field position. Each field position, in turn, is defined as:

F.*C*

...where *F* is the field number and *C* is the character within that field to begin the sort comparison.

So, let's say our input file **data.txt** contains the following data:

```
01 Joe Sr.Designer
02 Marie Jr.Developer
03 Albert Jr.Designer
04 Dave Sr.Developer
```

...we can sort by seniority if we specify the third field as the sort key:

```
sort -k 3 data.txt
```

...this produces the following output:

```
03 Albert Jr.Designer
02 Marie Jr.Developer
01 Joe Sr.Designer
04 Dave Sr.Developer
```

Or, we can ignore the first three characters of the third field, and sort solely based on title, ignoring seniority:

```
sort -k 3.3 data.txt
```

```
01 Joe Sr.Designer
03 Albert Jr.Designer
02 Marie Jr.Developer
04 Dave Sr.Developer
```

We can also specify where in the line to *stop* comparing. If we sort based on *only* the third-through-fifth characters of the third field of each line, like this:

```
sort -k 3.3,3.5 data.txt
```

...**sort** will see *only* the same thing on every line: "**.De**" ... *and nothing else*. As a result, **sort** will not see any differences in the lines, and the sorted output will be the same as the original file:

```
01 Joe Sr.Designer
02 Marie Jr.Developer
03 Albert Jr.Designer
04 Dave Sr.Developer
```

Using sort and join together

sort can be especially useful when used in conjunction with the **join** command. Normally **join** will join the lines of any two files whose first field match. Let's say you have two files, **file1.txt** and **file2.txt**. **file1.txt** contains the following text:

```
3       tomato
1       onion
4       beet
2       pepper
```

...and **file2.txt** contains the following:

```
4       orange
3       apple
```

```
1       mango
2       grapefruit
```

If you'd like **sort** these two files *and* **join** them, you can do so all in one command if you're using the **bash** command shell, like this:

```
join <(sort file1.txt) <(sort file2.txt)
```

Here, the **sort** commands in parentheses are each executed, and their output is redirected to **join**, which takes their output as standard input for its first and second arguments; it is joining the sorted contents of both files and gives results similar to the below results.

```
1 onion mango
2 pepper grapefruit
3 tomato apple
4 beet orange
```

Related commands

comm – Compare two sorted files line by line.
join – Join the lines of two files which share a common field of data.
uniq – Identify, and optionally filter out, repeated lines in a file.