13

Search and Sort Tools

Unix philosophy is to provide a rich set of generic tools, each with a variety of options. These primitive Unix tools can be combined in imaginative ways (by using pipes) to enhance user productivity. The tool suite also facilitates to build either a user customised application or a more sophisticated and specialised tool.

We shall discuss many primitive tools that are useful in the context of text files. These tools are often called "filters" because these tools help in searching the presence or absence of some specified pattern(s) of text in text files. Tools that fall in this category include *ls*, *grep*, and *find*. For viewing the output from these tools one uses tools like *more*, *less*, *head*, *tail*. The *sort* tool helps to sort and tools like *wc* help to obtain statistics about files. In this chapter we shall dwell upon each of these tools briefly. We also illustrate some typical contexts of usage of these tools.

13.1 *grep, egrep,* AND *fgrep*

grep stands for general regular expression parser. *egrep* is an enhanced version of *grep*. It allows a greater range of regular expressions to be used in pattern matching. *fgrep* is for fast but fixed string matching. As a tool, *grep* usage is basic and it follows the syntax

```
grep options pattern files
```

with the semantics that search the file(s) for lines with the pattern and options as command modifiers. The following example[1] shows how we can list the lines with *int* declarations in a program called *add.c*. Note that we could use this trick to collate all the declarations from a set of files to make a common include file of definitions.

```
bhatt@SE-0 [~/UPE] >>grep int ./C/add.c
extern int a1; /* First operand */
extern int a2; /* Second operand */
extern int add();
printf("The addition gives %d \n", add());
```

Note: print has *int* in it!!

In other words, *grep* matches string literals. A little later we
options to make partial patterns for intelligent searches. We co
the lines in all the *c* programs in that directory. In such a usag
shown in the example below:

```
grep int ./C/*.c | more
```

This shows the use of a pipe with another tool *more* which is a g
more offers a one screen at a time view of a long file. As stated i
is a program called *less* which additionally permits scrolling.

Regular expression conventions: Table 13.1 shows m
expression conventions. In Table 13.1, RE, RE1, and RE2 denot
practice we may combine Regular Expressions in arbitrary ways
egrep is an enhanced *grep* that allows additionally the al
capabilities. Note that an RE may be enclosed in parentheses. T
make a file called testfile with entries as shown. Next, we shall try
various options. Below we show a session using our text file ca

Table 13.1 Regular expression options

Regular expression symbol	*The effect of choi*
c	matches character c
^c	matches a line that starts w
c$	matches c with the end of l
[]	matches any one of the char
.	matches any character
*	matches zero or more charac
RE*	matches zero, or more, repe
RE1RE2	matches two concatenated e

Table 13.2 Regular expression combinati

RE = A regular expression	*The interpre*
RE+	matches one, or more,
RE?	matches none, or one,
RE1 | RE2	matches occurrence of

earch and Sort Tools

ride a rich set of generic tools, each with a variety of options
s can be combined in imaginative ways (by using pipes) to
The tool suite also facilitates to build either a user customised
isticated and specialised tool.
primitive tools that are useful in the context of text files. These
ers" because these tools help in searching the presence or
pattern(s) of text in text files. Tools that fall in this category
For viewing the output from these tools one uses tools like
rt tool helps to sort and tools like *wc* help to obtain statistics
we shall dwell upon each of these tools briefly. We also
exts of usage of these tools.

7

lar expression parser. *egrep* is an enhanced version of *grep*
regular expressions to be used in pattern matching. *fgrep* is
hing. As a tool, *grep* usage is basic and it follows the syntax

rch the file(s) for lines with the pattern and options as
owing example[1] shows how we can list the lines with *int*
ed *add.c*. Note that we could use this trick to collate all the
s to make a common include file of definitions.

c

e sessions on my machine.

In other words, *grep* matches string literals. A little later we will see how we may use options to make partial patterns for intelligent searches. We could have used **.c* to list the lines in all the *c* programs in that directory. In such a usage it is better to use it as shown in the example below:

```
grep int ./C/*.c | more
```

This shows the use of a pipe with another tool *more* which is a good screen viewing tool. *more* offers a one screen at a time view of a long file. As stated in the last chapter, there is a program called *less* which additionally permits scrolling.

Regular expression conventions: Table 13.1 shows many of the *grep* regular expression conventions. In Table 13.1, RE, RE1, and RE2 denote regular expressions. In practice we may combine Regular Expressions in arbitrary ways as shown in Table 13.2. *egrep* is an enhanced *grep* that allows additionally the above pattern matching capabilities. Note that an RE may be enclosed in parentheses. To practise the above, we make a file called testfile with entries as shown. Next, we shall try matching patterns using various options. Below we show a session using our text file called testfile.

Table 13.1 Regular expression options

Regular expression symbol	*The effect of choice*
c	matches character c
^c	matches a line that starts with character c
c$	matches c with the end of line
[]	matches any one of the characters in [ ]
.	matches any character
*	matches zero or more characters
RE*	matches zero, or more, repetitions of RE
RE1RE2	matches two concatenated expressions RE1RE2

Table 13.2 Regular expression combinations

RE = A regular expression	*The interpretation*
RE+	matches one, or more, repetitions of RE
RE?	matches none, or one, occurrence of RE
RE1 | RE2	matches occurrence of RE1 or RE2

```
aaa
a1a1a1
456
10000001
This is a test file.

bhatt@SE–0 [F] >>grep '[0–9]' testfile
a1a1a1
456
10000001

bhatt@SE–0 [F] >>grep '^4' testfile
456

bhatt@SE–0 [F] >>grep '1$' testfile
a1a1a1
10000001

bhatt@SE–0 [F] >>grep '[A–Z]' testfile
This is a test file.

bhatt@SE–0 [F] >>grep '[0–4]' testfile
a1a1a1
456
10000001

bhatt@SE–0 [F] >>fgrep '000' testfile
10000001

bhatt@SE–0 [F] >>egrep '0..' testfile
10000001
```

\The back slash is used to consider a special character literally. This is required when the character used is also a command option as in case of –, * etc.

See the example below where we are matching a period symbol.

```
bhatt@SE–0 [F] >>grep '\.' testfile
This is a test file.
```

We may use a character's characteristics as options in *grep*. The options available are shown in Table 13.3.

Table 13.3 Choosing match options

The option	*The effect of choice*
–i	to ignore case like in so SO So etc.
–l	asks grep to not list lines, just lists filenames only
–v	select lines that do not match

```
bhatt@SE–0 [F] >>grep –v 'a1' testfile
aaa
456
10000001
This is a test file.

bhatt@SE–0 [F] >>grep 'aa' testfile
aaa
bhatt@SE–0 [F] >>grep –w 'aa' testfile
bhatt@SE–0 [F] >>grep –w 'aaa' testfile
aaa
bhatt@SE–0 [F] >>grep –l 'aa' testfile
testfile
```

Context of use: Suppose we wish to list all subdirectori

```
ls –l | grep ^d

bhatt@SE–0 [M] >>ls –l | grep ^d
drwxr–xr–x 2 bhatt bhatt 512 Oct 15 13:15 M1
drwxr–xr–x 2 bhatt bhatt 512 Oct 15 12:37 M2
drwxr–xr–x 2 bhatt bhatt 512 Oct 15 12:37 M3
drwxr–xr–x 2 bhatt bhatt 512 Oct 16 09:53 RAND
```

Suppose we wish to select a certain font and also wish to as a bold font with size 18. We may list these with the instru

```
xlsfonts | grep bold\–18 | more

bhatt@SE–0 [M] >>xlsfonts | grep bold\–18 | more
lucidasans–bold–18
lucidasans–bold–18
lucidasanstypewriter–bold–18
lucidasanstypewriter–bold–18
```

Suppose we wish to find out at how many terminals a cer moment. The following command will give us the required in

```
who | grep username | wc –l > count
```

The *wc* with *–l* options gives the count of lines. Also, who | g every line matched with the given pattern (username) in it.

13.2 USING *find*

find is used when a user, or a system administrator, needs a certain file under a specified subtree in a file hierarchy. T

```
find path expression action
```

stfile

le

e

tfile

tfile

le

e

a special character literally. This is required when the character used is also a command

here we are matching a period symbol.

r's characteristics as options in *grep*. The options available are

Table 13.3 Choosing match options

The effect of choice
to ignore case like in so SO So etc.
asks grep to not list lines, just lists filenames only
select lines that do not match
select lines that contain whole words only

```
bhatt@SE-0 [F] >>grep 'aa' testfile
aaa
bhatt@SE-0 [F] >>grep -w 'aa' testfile
bhatt@SE-0 [F] >>grep -w 'aaa' testfile
aa
bhatt@SE-0 [F] >>grep -l 'aa' testfile
testfile
```

Context of use: Suppose we wish to list all subdirectories in a certain directory.

```
ls -l | grep ^d
```

```
bhatt@SE-0 [M] >>ls -l | grep ^d
drwxr-xr-x 2 bhatt bhatt 512 Oct 15 13:15 M1
drwxr-xr-x 2 bhatt bhatt 512 Oct 15 12:37 M2
drwxr-xr-x 2 bhatt bhatt 512 Oct 15 12:37 M3
drwxr-xr-x 2 bhatt bhatt 512 Oct 16 09:53 RAND
```

Suppose we wish to select a certain font and also wish to find out if it is available as a bold font with size 18. We may list these with the instruction shown below.

```
xlsfonts | grep bold\-18 | more
```

```
bhatt@SE-0 [M] >>xlsfonts | grep bold\-18 | more
-b&h-lucidasans-bold-r-normal--18-180-75-75-p-111-iso8859-1
-b&h-lucidasans-bold-r-normal--18-180-75-75-p-111-iso8859-15
-b&h-lucidasanstypewriter-bold-r-normal--18-180-75-75-m-110-iso8859-1
-b&h-lucidasanstypewriter-bold-r-normal--18-180-75-75-m-110-iso8859-15
```

Suppose we wish to find out at how many terminals a certain user is logged in at the moment. The following command will give us the required information:

```
who | grep username | wc -l > count
```

The *wc* with *-l* options gives the count of lines. Also, who | grep will output one line for every line matched with the given pattern (username) in it.

13.2 USING *find*

find is used when a user, or a system administrator, needs to determine the location of a certain file under a specified subtree in a file hierarchy. The syntax and use of *find* is

```
find path expression action
```

where path identifies the subtree, expression helps to identify the file and the action specifies the action one wishes to take. Let us now see a few typical usages.

- List all the files in the current directory.

```
bhatt@SE–0 [F] >>find . –print /* only path and action specified */

.
./ReadMe
./testfile
```

- Finding files which have been created after a certain other file was created.

```
bhatt@SE–0 [F] >>find . –newer testfile

.
./ReadMe
```

  There is an option *mtime* to find modified files in a certain period over a number of days.

- List all the files which match the partial pattern test. One should use only shell metacharacters for partial matches.

```
bhatt@SE–0 [F] >>find . –name test*
./testfile

bhatt@SE–0 [F] >>find ../../ –name test*
../../COURSES/OS/PROCESS/testfile
../../UPE/F/testfile
```

- I have a file called linkedfile with a link to testfile. The *find* command can be used to find the links.

```
bhatt@SE–0 [F] >>find ./ –links 2
./
./testfile
./linkedfile
```

- Finding out the subdirectories under a directory.

```
bhatt@SE–0 [F] >>find ../M –type d
../M
../M/M1
../M/M2
../M/M3
../M/RAND
```

- Finding files owned by a certain user.

```
bhatt@SE–0 [F] >>find /home/georg/ASM–WB –user georg –type d
/home/georg/ASM–WB
/home/georg/ASM–WB/examples
/home/georg/ASM–WB/examples/Library
/home/georg/ASM–WB/examples/SimpleLang
/home/georg/ASM–WB/examples/InstructionSet
/home/georg/ASM–WB/Projects
```

The file type options are: *f* for text file, *c* for character s
and *p* for pipes.

Strings: Sometimes one may need to examine if there
object or a binary file. This can be done using a string co

```
string binaryfileName | more
```

As an example, let us see its use below:[2]

```
bhatt@SE–0 [F] >>strings ../M/RAND/main | more
The value of seed is %d
at A
The value is %f
at B
```

ctags and etags: These commands are useful in the
look up patterns like *c* function calls. You may look up ma
you are a power user of *c*.

13.2.1 Sort Tool

For *sort* tool Unix treats each line in a text file as data. In
lines of text in text file. It is possible to give an option
descending order. We shall demonstrate its usage through

```
bhatt@SE–0 [F] >>sort
aaa
bbb
aba
^d terminates the input ....see the output below
aaa
aba
bbb

bhatt@SE–0 [F] >>sort testfile
10000001
456
This is a test file.
a1a1a1
aaa
```

(Use *–r* option for descending order.)

```
bhatt@SE–0 [F] >>sort testfile –o outfile
bhatt@SE–0 [F] >>
```

One can see the outfile for sorted output. *sort* repeats
use a filter *uniq* to get sorted output with unique lines. Le
have repetition of a few lines and then use *uniq* as shown

```
bhatt@SE–0 [F] >>sort testfile | uniq | more
10000001
```

/* only path and action specified */

ve been created after a certain other file was created.

r testfile

ne to find modified files in a certain period over a numb

match the partial pattern test. One should use only sh
tial matches.

test*

me test*
estfile

lfile with a link to testfile. The *find* command can be used t

2

ectories under a directory.

)e d

a certain user.

org/ASM–WB –user georg –type d

Library
SimpleLang
nstructionSet

bject or a binary file. This can be done using a string command with the syntax:

ing binaryfileName | more

As an example, let us see its use below:[2]

hatt@SE–0 [F] >>strings ../M/RAND/main | more
he value of seed is %d
A
he value is %f
B

ctags and etags: These commands are useful in the context when one wishes to look up patterns like *c* function calls. You may look up man pages for its description if you are a power user of *c*.

13.2.1 Sort Tool

The *sort* tool Unix treats each line in a text file as data. In other words, it basically sorts lines of text in text file. It is possible to give an option to list lines in ascending or descending order. We shall demonstrate its usage through examples given below:

hatt@SE–0 [F] >>sort
aaa
bbb
aba
d terminates the input ....see the output below
aaa
aba
bbb

hatt@SE–0 [F] >>sort testfile
0000001
456
This is a test file.
hatatt
aaa

(Use *–r* option for descending order.)

hatt@SE–0 [F] >>sort testfile –o outfile
hatt@SE–0 [F] >>

One can see the outfile for sorted output. *sort* repeats all identical lines. It helps to use a filter *uniq* to get sorted output with unique lines. Let us now modify our testfile to have repetition of a few lines and then use *uniq* as shown below:

hatt@SE–0 [F] >>sort testfile | uniq | more
0000001

[2] I tend to use print statement "at A", followed by "at B" markers in my programs to keep track of the programs as I develop them. These are the ASCII strings that appear within binary.

```
456
This is a test file.
a1a1a1
aaa
```

In Table 13.4 we list the options that are available with *sort*. *sort* can also be used to merge files. Now we will split a file and then show the use of merge. Of course, the usage is in the context of merge–sort.

Table 13.4 Sort options

The sort option	*The effect of choice*
–r	To get a sort in decreasing order of value
–b	To ignore leading blank spaces
–f	To fold uppercase to lowercase for comparison
–i	To ignore characters outside the ASCII range
–n	To perform numeric comparison. A number may have leading blanks.

One often uses filter commands such as *sort* and *grep* in conjunction with *wc, more, head,* and *tail* commands available in Unix. System administrators use *who* in conjunction with *grep, sort, find* to keep track of terminal usage and also for lost or damaged files.

split: split command helps one to split a file into smaller sized segments. For instance, if we *split* ReadMe file with the following command:

```
split – l 20 ReadMe seg
```

Upon execution we get a set of files segaa, segab, etc. each with 20 lines in it. (Check the line count using *wc*.) Now merge using sorted segaa with segab.

```
sort –m segaa segab > check
```

A clever way to merge all the split files is to use *cat* as shown below:

```
cat seg* > check
```

The file check should have 40 lines in it. Clearly, *split* and *merge* would be useful to support merge sort and for assembling a set of smaller files that can be sent over a network using e-mail whenever there are restrictions on the size of attached files.

In the next chapter we shall learn about the AWK tool in Unix. Evolution of AWK is a very good illustration of how more powerful tools can be built. AWK evolves from the (seemingly modest) generic tool *grep*!!

EXERCISES

1. What algorithm would you use if you were implementing *grep*?
2. When should *fgrep* be preferred to *grep* and vice versa?
3. How do we use *ls –alF grep* and *more* as a one-line command to display all the

4. How do we search for a given pattern in the last 10 lines directory?
5. Whenever we run a *grep* from the /dev directory (for exam keyboard does not work. Essentially, while *grep* is runn shell and presses a key on the keyboard, there is no respo other directory does not cause this problem. Find out w
6. What is the difference between *locate* and *find* command
7. How can one use *find* to locate some strings, given two the current directory?
8. Assume that the last file you created yesterday was called *find* to determine the files which have been created toda
9. Can one use *find* to locate two given strings in an unknow file?
10. Here is a poser: Why does *find* fail to work with *–exec* shell (cmd.exe)? Try a few combinations and you shall get message. Why?
11. Give a description of the algorithm you would have use
12. How does one sort the contents of a directory (recursive subdirectories) so that one can view the largest files fir
13. How does *sort* work when it is provided with a file whic the main memory?
14. We all like to create aliases. One may create these durir them in .bashrc file. One may do this to append these alias be something like *alias >>* .bashrc. Unfortunately, one end So we may try using *sort* and *uniq*. Is it right? What r
15. We have a problem with a file we have just received. It h no separators. The records vary in length with the max 163 characters. The records generally appear to be of t

```
L1XXXXXXXXWXAXXXX..... .....
L3XXXXXXXXVXCXXXX... ..... ..... ..
L2XXXXXXXXZXBXXXX..... .....
```

The first two characters represent a record type whi represent a customer number. We need to sort the file fir 17 and then based on characters 1 to 2 to a newfile.