

11 - UTILITY COMMANDS

- Unit Objectives:
 - Overview 11-2
 - Finding patterns: grep 11-3
 - Extracting fields: cut 11-7
 - Sorting records: sort 11-10
 - Translating characters: tr 11-15
 - Putting it together again: paste 11-19
 - Standard input: the - 11-22
 - Automating changes: sed 11-23
 - Comparing text files: diff 11-32
 - Comparing binary files: cmp 11-33
 - Searching for files: find 11-37
 - Counting lines, words & characters: wc 11-39
 - Combining most of it... 11-41
 - Exercises 11-43
 - Exercise solutions 11-46

OVERVIEW

- Every Unix system has at least a basic set of powerful utility commands as part of its distribution.
- Most of these utilities are designed in such a way that they can be connected to one another using pipes. This is due to the fact that many of them will read their input from the **standard input** if no filename arguments are given, and they will write their output to the standard output (by default, your screen).
- NOTE: Any command which reads standard input by default will assume that the keyboard is the standard input source, in which case executing the command without redirection of standard input from some other source will result in your cursor positioned on a blank line on your screen. This indicates that the command is waiting for input. You may type records, one per line, to be processed and then end the standard input by typing a `<CONTROL-D>` as the first character of a line.
- Because of this behavior, most of the utility commands do not in any way attempt to modify the source of input, as it may be a dynamically generated stream of data. Instead, they send their output to the standard output.

FINDING PATTERNS: grep

- The `grep` command searches one or more named input files (or the standard input, by default) looking for records that contain a specified pattern, given in the form of a regular expression, and writes those selected records to the standard output, as indicated by this generic command line synopsis:

```
grep [-options] regular-expression [file ...]
```

- The name `grep` is derived from the documentation for the early Unix line editor, `ed`, which indicated that while editing a file, you could print out all lines that matched a regular expression using this generic editor command:

```
g/RE/p
↑   ↑   ↑
|   |   |
|   |   | print records where match occurred
|   |   | Regular Expression to search for
|   |   | global search (all records)
```

- Here are some of the more popular options available to `grep`:
 - `-v` print lines that do NOT contain the regular expression (inverted match)
 - `-n` include record **number**
 - `-c` print only the **count** of records
 - `-i` case insensitive matching

FINDING PATTERNS: grep (continued)

➤ -l list filenames only

- Here is a sequence of commands that indicate how `grep` works:

➤ Show the input file to be used in the following examples:

```
$ cat sample
```

```
This student guide is designed to introduce  
people to the fundamentals of the Unix  
operating system.
```

```
The material it contains is applicable to  
many versions of Unix and Unix-like  
operating systems, including Solaris,  
HP-UX, AIX, SCO-Unix, and the many  
different versions of Linux.
```

➤ Print the records containing a `U`:

```
$ grep U sample
```

```
people to the fundamentals of the Unix  
many versions of Unix and Unix-like  
HP-UX, AIX, SCO-Unix, and the many
```

➤ Print the records containing an `x` or a `y` or a `z` (if the regular expression contains any metacharacters recognized by the shell, the expression should be quoted):

```
$ grep "[xyz]" sample
```

```
people to the fundamentals of the Unix  
operating system.
```

```
many versions of Unix and Unix-like  
operating systems, including Solaris,  
HP-UX, AIX, SCO-Unix, and the many  
different versions of Linux.
```

FINDING PATTERNS: grep (continued)

- Print the records that do NOT contain a U:

```
$ grep -v U sample
```

```
This student guide is designed to introduce  
operating system.
```

The material it contains is applicable to operating systems, including Solaris, different versions of Linux.

- Print records that do NOT contain a U or u (using the inverted and "case insensitive" options):

```
$ grep -vi U sample
```

```
operating system.
```

The material it contains is applicable to

- Number the records that contain a . as the last character. Note that since the . matches any character in a regular expression, it must be "escaped" using \ :

```
$ grep -n "\.$" sample
```

```
3:operating system.  
9:different versions of Linux.
```

- Print a count of the records that contain a . as the last character:

```
$ grep -c "\.$" sample
```

```
2
```

- Print names of the files that contain an x:

```
$ grep -l x sample sample2 sample3
```

```
sample  
sample3
```

FINDING PATTERNS: grep (continued)

- Print the records that begin with `usr` in the file `/etc/passwd`:

```
$ grep "^usr" /etc/passwd
usr01:x:501:501:Student 01:/home/usr01:/usr/local/bin/ksh
usr02:x:502:502:Student 02:/home/usr02:/usr/local/bin/ksh
usr03:x:503:503:Student 03:/home/usr03:/usr/local/bin/ksh
usr04:x:504:504:Student 04:/home/usr04:/usr/local/bin/ksh
usr05:x:505:505:Student 05:/home/usr05:/usr/local/bin/ksh
usr06:x:506:506:Student 06:/home/usr06:/usr/local/bin/ksh
usr07:x:507:507:Student 07:/home/usr07:/usr/local/bin/ksh
usr08:x:508:508:Student 08:/home/usr08:/usr/local/bin/ksh
usr09:x:509:509:Student 09:/home/usr09:/usr/local/bin/ksh
usr10:x:510:510:Student 10:/home/usr10:/usr/local/bin/ksh
usr11:x:511:511:Student 11:/home/usr11:/usr/local/bin/ksh
usr12:x:512:512:Student 12:/home/usr12:/usr/local/bin/ksh
usr13:x:513:513:Student 13:/home/usr13:/usr/local/bin/ksh
usr14:x:514:514:Student 14:/home/usr14:/usr/local/bin/ksh
usr15:x:515:515:Student 15:/home/usr15:/usr/local/bin/ksh
usr16:x:516:516:Student 16:/home/usr16:/usr/local/bin/ksh
usr17:x:517:517:Student 17:/home/usr17:/usr/local/bin/ksh
```

EXTRACTING FIELDS: cut

- The `cut` command searches one or more named input files (or the standard input, by default) attempting to cut out the specified characters or fields from each record of input and write those characters to the standard output, as indicated by these two generic command line synopses:

```
cut -ccharacter-list [-options] [file ...]
cut -ffield-list [-options] [file ...]
```

- Fields are, by default, consecutive strings of non-tab characters separated from each other by a single tab character. Fields are numbered starting with 1 (one). The `-d` option may be used to specify a different field separator using the generic syntax:

```
-ddelimiter
```

- The character or field list consists of integers separated by commas and possibly including inclusive ranges using a hyphen, as in these examples:

- Cut out characters 1 through 10 and 30 through the end of each record from file `sample`:

```
$ cut -c1-10,30- sample
This studed to introduce
people to the Unix
operating
```

```
The materipplicable to
many versix-like
operating Solaris,
HP-UX, AIX many
different
```

EXTRACTING FIELDS: cut (continued)

- Cut out characters 1,3,5 and 7 from file `sample`:

```
$ cut -c1,3,5,7 sample
Ti t
pol
oeai

Temt
mn e
oeai
H-X
dfee
```

- Cut out fields 1 and 7 from file `/etc/passwd` using a colon (`:`) as the field delimiter. Each record in the `/etc/passwd` file has seven fields delimited by colons. The first is the user id and the seventh is the startup program (usually a shell; if empty, it defaults to a Bourne shell). Only a portion of the output is shown:

```
$ cut -f1,7 -d: /etc/passwd
root:/bin/bash
bin:
daemon:
adm:
lp:
sync:/bin/sync
shutdown:/sbin/shutdown
halt:/sbin/halt
mail:
news:
uucp:
ftp:
nobody:
xfs:/bin/false
named:/bin/false
walt:/usr/local/bin/ksh
usr01:/usr/local/bin/ksh
usr17:/usr/local/bin/ksh
js:/bin/bash
```

EXTRACTING FIELDS: cut (continued)

- Here is an example of using a pipe to combine `grep` and `cut` that uses `grep` to filter through only the records from the `/etc/passwd` file beginning with `usr` and then using `cut` to print the first and third colon-delimited fields of each record (the user id and UID number):

```
$ grep "^usr" /etc/passwd | cut -f1,3 -d:  
usr01:501  
usr02:502  
usr03:503  
usr04:504  
usr05:505  
usr06:506  
usr07:507  
usr08:508  
usr09:509  
usr10:510  
usr11:511  
usr12:512  
usr13:513  
usr14:514  
usr15:515  
usr16:516  
usr17:517
```

SORTING RECORDS: `sort`

- The `sort` command sorts the records of one or more named input files (or the standard input, by default) and writes them to the standard output. By default, they are sorted low to high ASCII order. Here is a simplified generic syntax:

```
sort [-options] [file ...]
```

- Here are some of the more popular options available to `sort`:
 - `-r` reverse order (typically high to low ASCII)
 - `-n` numeric sort
 - `-f` fold lower case to uppercase (case insensitive sort)
 - `-o file` send **output** to `file` instead of standard output
 - `-t char` fields are separated by `char` delimiter (the default field separator is any sequence of spaces and tabs); fields are numbered starting with 0 (zero)
 - `+n` sort key begins at beginning of field `n`
 - `-m` sort key ends at beginning of field `m`
 - `-b` ignore leading **blanks** in fields

SORTING RECORDS: sort (continued)

- Here is a sequence of commands that indicate how `sort` works:
 - Show the input file to be used in the following examples. These are the first ten states in order by statehood date, and showing the year of statehood and the resident population as of April 1, 2000. The three fields are separated by single tab characters:

```
$ cat states
Delaware      1787      783600
Pennsylvania  1787      12281054
New Jersey    1787      8414350
Georgia       1788      8186453
Connecticut   1788      3405565
Massachusetts 1788      6349097
Maryland      1788      5296486
South Carolina 1788      4012012
New Hampshire 1788      1235786
Virginia      1788      7078515
```

- Default sort (low to high ASCII):

```
$ sort states
Connecticut   1788      3405565
Delaware      1787      783600
Georgia       1788      8186453
Maryland      1788      5296486
Massachusetts 1788      6349097
New Hampshire 1788      1235786
New Jersey    1787      8414350
Pennsylvania  1787      12281054
South Carolina 1788      4012012
Virginia      1788      7078515
```

SORTING RECORDS: sort (continued)

- Reverse sort (high to low ASCII):

```
$ sort -r states
Virginia      1788      7078515
South Carolina 1788      4012012
Pennsylvania  1787      12281054
New Jersey    1787      8414350
New Hampshire 1788      1235786
Massachusetts 1788      6349097
Maryland      1788      5296486
Georgia       1788      8186453
Delaware      1787      783600
Connecticut   1788      3405565
```

- Sort on field number 2 (the third field!), there is a single TAB character between the quotes:

<TAB> character here



```
$ sort -t"      " +2 states
Pennsylvania  1787      12281054
New Hampshire 1788      1235786
Connecticut   1788      3405565
South Carolina 1788      4012012
Maryland      1788      5296486
Massachusetts 1788      6349097
Virginia      1788      7078515
Delaware      1787      783600
Georgia       1788      8186453
New Jersey    1787      8414350
```

SORTING RECORDS: sort (continued)

- Sort on field number 2 numerically:

```
$ sort -t"      " -n +2 states
Delaware      1787      783600
New Hampshire 1788      1235786
Connecticut   1788      3405565
South Carolina 1788      4012012
Maryland      1788      5296486
Massachusetts 1788      6349097
Virginia      1788      7078515
Georgia       1788      8186453
New Jersey    1787      8414350
Pennsylvania  1787      12281054
```

- Sort on field number 1 (admission year) and then alphabetically by state name (2 keys):

```
$ sort -t"      " +1 -2 +0 -1 states
Delaware      1787      783600
New Jersey    1787      8414350
Pennsylvania  1787      12281054
Connecticut   1788      3405565
Georgia       1788      8186453
Maryland      1788      5296486
Massachusetts 1788      6349097
New Hampshire 1788      1235786
South Carolina 1788      4012012
Virginia      1788      7078515
```

SORTING RECORDS: sort (continued)

- Sort on field number 2 (population) in reverse order and store sorted records in the same file:

```
$ sort -r -t" " -n +2 -o states states
Pennsylvania 1787 12281054
New Jersey 1787 8414350
Georgia 1788 8186453
Virginia 1788 7078515
Massachusetts 1788 6349097
Maryland 1788 5296486
South Carolina 1788 4012012
Connecticut 1788 3405565
New Hampshire 1788 1235786
Delaware 1787 783600
```

- NOTE: The above example uses the `-o` option.
- If you try to do it this way, you'll wind up with an EMPTY `states` file, because the shell will "lobber" the `states` file when it sees `> states`, after which `sort` doesn't have much work to do:

```
$ sort -r -t" " -n +2 states > states
$ cat states
$
      ↑
  wrong...
```

TRANSLATING CHARACTERS: tr

- The `tr` (**t**ranslate or **t**ransliterate) command reads records from the standard input, performs translations and writes the translated records to the standard output. Note that `tr` can NOT be given an input filename on the command line; it MUST read from standard input. Here are two simplified generic synopses, using **bold** to indicate where the standard input is coming from:

```
tr [-options] set1 [set2] < file
```

```
command | tr [-options] set1 [set2]
```

- Here are two of the most popular options available to `tr`:
 - `-d` **delete** characters in *set1*; perform no translation, *set2* not required
 - `-s` **squeeze** out multiple consecutive occurrences of translated characters

TRANSLATING CHARACTERS: tr (continued)

- Here is a sequence of commands that indicate how `tr` works:
 - Show the input file to be used in the following examples. This file contains the names of some U.S. cities and the average July rainfall in inches. The city and state are separated from the rainfall by a varying number of space characters. Each example starts with the original file (they are not a succession):

```
$ cat rain
New York, NY      4.4
Atlanta, GA      5.3
Chicago, IL      3.6
Dallas, TX       2.2
Honolulu, HI     0.6
Miami, FL        6.0
```

- Translate decimal points from periods to commas:

```
$ tr . , < rain
New York, NY      4,4
Atlanta, GA      5,3
Chicago, IL      3,6
Dallas, TX       2,2
Honolulu, HI     0,6
Miami, FL        6,0
```

- Delete the commas:

```
$ tr -d , < rain
New York NY      4.4
Atlanta GA      5.3
Chicago IL      3.6
Dallas TX       2.2
Honolulu HI     0.6
Miami FL        6.0
```

TRANSLATING CHARACTERS: tr (continued)

- Translate lower case to upper case, using brackets and a hyphen ([-]) as the inclusive range operator (use quotes so the shell doesn't "see" the [] which are filename generation metacharacters):

```
$ tr "[a-z]" "[A-Z]" < rain
NEW YORK, NY      4.4
ATLANTA, GA       5.3
CHICAGO, IL       3.6
DALLAS, TX        2.2
HONOLULU, HI      0.6
MIAMI, FL         6.0
```

- Reduce multiple spaces to single spaces. Each pair of quotes contains a single space character (it's a "dummy" translation; no translation really occurs, but `tr` "thinks" it has translated something into spaces, so the "squeeze out" option can be used):

```
$ tr -s " " " " < rain
New York, NY 4.4
Atlanta, GA 5.3
Chicago, IL 3.6
Dallas, TX 2.2
Honolulu, HI 0.6
Miami, FL 6.0
```

TRANSLATING CHARACTERS: tr (continued)

- Translate a's to e's, i's to o's and u's to y's (just to demonstrate the point!):

```
$ tr aiu eoy < rain
New York, NY      4.4
Atlente, GA      5.3
Chocego, IL      3.6
Delles, TX       2.2
Honolyly, HI     0.6
Moemo, FL        6.0
```

- There are many more unusual forms of translation. Here is one that translates all lower case vowels into x's:

```
$ tr aeiouy "[x*]" < rain
Nxw Yxrk, NY     4.4
Atlxntx, GA     5.3
Chxcxgx, IL     3.6
Dxllx, TX       2.2
Hxnxlxlx, HI    0.6
Mxxmx, FL       6.0
```

- Control characters and other "unusual" characters may be represented using their octal ASCII code preceded by a backslash (that must be quoted) and a 0 (zero). Here's an example that deletes form-feed characters from a file called `ff_file` and writes its output to a new file called `clean_file`:

```
$ tr -d "\014" < ff_file > clean_file
```

PUTTING IT TOGETHER AGAIN: paste

- The `paste` command can be thought of as the opposite of the `cut` command.
- `paste` reads records from the named input files and writes to the standard output the corresponding records from each input file joined by single tab characters. A hyphen (-) may be used in place of a filename to indicate standard input:

```
paste [-options] [file ...]
```

- `paste` may use the `-d` option followed by a list of characters which it will use instead of tabs to paste the records together (`d` for **d**elimiter). These characters are sometimes called "the glue":

➤ `-dchars` **d**elimit joined records with *chars*,
one at a time and recycled if necessary

PUTTING IT TOGETHER AGAIN: paste (continued)

- Here is a sequence of commands that indicate how `paste` works:
 - Show the input files to be used in the following examples. One file contains the names of the first five alphabetical states of the United States, along with the names of their respective capital cities, separated from the state names by a single tab character. The other has the same state names arranged alphabetically, along with the name of the state bird, also tab delimited:

```
$ cat state_caps
Alabama      Montgomery
Alaska      Juneau
Arizona      Phoenix
Arkansas     Little Rock
California   Sacramento
```

```
$ cat state_birds
Alabama      Yellowhammer
Alaska      Willow Ptarmigan
Arizona      Cactus Wren
Arkansas     Mockingbird
California   California Valley Quail
```

- Cut out the birds from `state_birds` and save those records in a file named `birds`:

```
$ cut -f2 state_birds > birds
$ cat birds
Yellowhammer
Willow Ptarmigan
Cactus Wren
Mockingbird
California Valley Quail
```

PUTTING IT TOGETHER AGAIN: paste (continued)

- `paste state_caps` and `birds` together and save the output in `state_caps_birds`:

```
$ paste state_caps birds > state_caps_birds
$ cat state_caps_birds
Alabama      Montgomery   Yellowhammer
Alaska      Juneau       Willow Ptarmigan
Arizona     Phoenix     Cactus Wren
Arkansas    Little Rock  Mockingbird
California  Sacramento  California Valley Quail
```

STANDARD INPUT: THE -

- Many Unix utilities that allow filenames as command line arguments also allow a single dash (-) to be used on the command line in place of a file pathname. This becomes the insertion point for the standard input that is often coming from a pipe. This example shows accomplishing the same action as the previous example in a single command line (note the - as the 2nd argument to paste):

↓

```
$ cut -f2 state_birds | paste state_caps - > state_caps_birds
$ cat state_caps_birds
Alabama      Montgomery   Yellowhammer
Alaska       Juneau       Willow Ptarmigan
Arizona      Phoenix      Cactus Wren
Arkansas     Little Rock  Mockingbird
California   Sacramento   California Valley Quail
```

AUTOMATING CHANGES: sed

- Perhaps one of the most powerful utilities provided with every Unix system is `sed` (**s**tream **e**ditor).
- The `sed` command incorporates the same fundamental editor commands as those used by the original Unix line editor, `ed`. These are the same as the "ex" style (or "colon-mode") commands supported by the `vi/ex` editor.
- `sed` reads and processes one or more input files specified as command line arguments, or the standard input if no filename arguments are specified.
- `sed` writes its output to the standard output.
- `sed` supports the use of `-` to indicate standard input as one of several command line filename arguments.
- `sed` allows an editor command to appear (typically quoted) as a command line argument. When specifying more than one editor command in this fashion, each must be preceded by the `-e` option:

```
sed 'edit-command' [file ...]
```

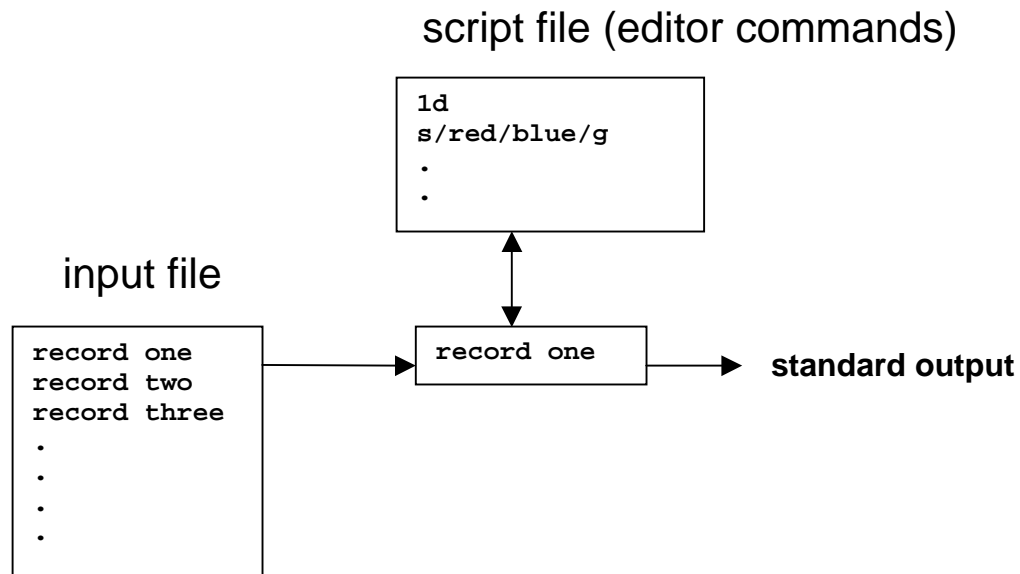
```
sed -e 'edit-command' -e 'edit-command' [file ...]
```

- Editor commands may be placed in a "script file", the pathname of which appears as the argument of the `-f` option:

```
sed -f script-file [file ...]
```

AUTOMATING CHANGES: sed (continued)

- Here's how `sed` works:
 - Each input record is read into an area in memory known as the "pattern space".
 - Each editor command is examined to see whether its address matches the address currently in the pattern space. Editor commands with matching addresses are applied to the text in the pattern space.
 - The pattern space is "flushed" to the standard output.
 - The cycle is repeated until all input records have been processed.



AUTOMATING CHANGES: sed (continued)

- The editor commands used are those used by the `ed` (and `ex`) editors, and include:
 - `a` **append**
 - `c` **change**
 - `d` **delete**
 - `i` **insert**
 - `p` **print**
 - `s` **substitute**
- The editor commands use the same addressing scheme as the `ed` and `ex` editors with one major difference:
 - In `ed` or `ex`, the absence of an address usually indicates that the action is to be applied to the current line by default. The same is true in `sed`, except EVERY line is the "current line" at some point in time (when it's in the pattern space) so the absence of an address is equivalent to the address range `1, $` in `ed` or `ex` (first through and including last line).

AUTOMATING CHANGES: sed (continued)

- Here are a few examples of editor commands that can be used with `sed`:

- `1d` Delete line 1.
- `3,5p` Print lines 3 through 5.
- `s/[Bb]lue/purple/g` Attempt to substitute purple in place of blue or Blue globally on every line.
- `/adj/a\
As an intransitive verb:\`
I blue when I hold my breath. Add two lines of text after any line containing "adj".

- Now we'll apply these commands:

- Show the input file to be processed with `sed`:

```
$ cat poem
Here is a poem that's simple but true:
Roses are red, violets are blue.
As a noun: Blue is a color.
As an adjective: Blue skies are beautiful.
As a transitive verb: What will blue litmus?
```

AUTOMATING CHANGES: sed (continued)

- Show the script file containing the editor commands:

```
$ cat sed_script
1d
3,5p
s/[Bb]lue/purple/g
/adj/a\
As an intransitive verb:\
I blue when I hold my breath.
```

- Put them together:

```
$ sed -f sed_script poem
Roses are red, violets are purple.
As a noun: Blue is a color.
As a noun: purple is a color.
As an adjective: Blue skies are beautiful.
As an adjective: purple skies are beautiful.
As an intransitive verb:
I blue when I hold my breath.
As a transitive verb: What will blue litmus?
As a transitive verb: What will purple litmus?
```

AUTOMATING CHANGES: sed (continued)

- The order of the editor commands is important, as they are examined in the order that they're presented:
 - 1st record read:
Here is a poem that's simple but true:
 - 1st editor command examined matches address (line 1):
1d
 - The line is deleted, the pattern space is now empty and no other editor address match except the non-addressed line, but there's nothing to examine anyway.
 - The empty pattern space is flushed (no output).

 - 2nd record read:
Roses are red, violets are blue.
 - 1st editor command examined does not match:
1d
 - 2nd editor command examined does not match:
3,5p
 - 3rd editor command matches address (current line) and substitution is attempted. blue becomes purple:
s/[Bb]lue/purple/g
 - 4th editor command examined does not match:
/adj/a\
As an intransitive verb:\nI blue when I hold my breath.
 - The modified pattern space is flushed:
Roses are red, violets are purple.

AUTOMATING CHANGES: sed (continued)

- 3rd record read:
As a noun: Blue is a color.
- 1st editor command examined does not match:
1d
- 2nd editor command examined matches and the pattern space is printed:
3,5p
As a noun: Blue is a color.
- 3rd editor command matches address (current line) and substitution is attempted. Blue becomes purple:
s/[Bb]lue/purple/g
- 4th editor command examined does not match:
/adj/a\
As an intransitive verb:\nI blue when I hold my breath.
- The modified pattern space is flushed.
As a noun: purple is a color.

- 4th record read:
As an adjective: Blue skies are beautiful.
- 1st editor command examined does not match:
1d
- 2nd editor command examined matches and the pattern space is printed:
3,5p
As an adjective: Blue skies are beautiful.
- 3rd editor command matches address (current line) and substitution is attempted. Blue becomes purple:
s/[Bb]lue/purple/g
- 4th editor command examined matches and two lines of text are appended to the pattern space:
/adj/a\
As an intransitive verb:\nI blue when I hold my breath.

AUTOMATING CHANGES: sed (continued)

- The modified pattern space is flushed.

As an adjective: purple skies are beautiful.

As an intransitive verb:

I blue when I hold my breath.

- 5th record read:

As a transitive verb: What will blue litmus?

- 1st editor command examined does not match:

1d

- 2nd editor command examined matches and the pattern space is printed:

3,5p

As a transitive verb: What will blue litmus?

- 3rd editor command matches address (current line) and substitution is attempted. blue becomes purple:

s/[Bb]lue/purple/g

- 4th editor command examined does not match:

/adj/a\

As an intransitive verb:\

I blue when I hold my breath.

- The modified pattern space is flushed.

As a transitive verb: What will purple litmus?

- No more input records remain.

AUTOMATING CHANGES: sed (continued)

- The `-n` option (no autoflush) disables the automatic flushing of the pattern space; only records explicitly printed will be written to the standard output:

➤ Without `-n`:

```
$ sed -e '3,5p' poem
Here is a poem that's simple but true:
Roses are red, violets are blue.
As a noun: Blue is a color.
As a noun: Blue is a color.
As an adjective: Blue skies are beautiful.
As an adjective: Blue skies are beautiful.
As a transitive verb: What will blue litmus?
As a transitive verb: What will blue litmus?
```

➤ With `-n`:

```
$ sed -n -e '3,5p' poem

As a noun: Blue is a color.
As an adjective: Blue skies are beautiful.
As a transitive verb: What will blue litmus?
```

COMPARING FILES: `diff`

- The `diff` command compares two text files and reports any **differences** between them.
- The two files to be compared are specified as command line arguments.
- `diff` supports the use of `-` to indicate standard input as either one of the two filename arguments.
- The reporting syntax typically attempts to explain what additions, changes or deletions must be made to the first file so that it will be identical to the second file.

COMPARING TEXT FILES: diff (continued)

- Here is a sequence of commands that demonstrate how `diff` works. It compares `file1` against `file2`:

➤ Show the input files to be compared with `diff`:

```
$ cat file1
After I drive on the parkway, I park in my driveway.
Paris is beautiful in
    the spring.
Jumbo shrimp are smaller than microbreweries!
On a bright sunny day, in the middle of the night,
I heard a lie I believe is true.
```

```
$ cat file2
After I park on the parkway, I drive in my driveway.
Paris is beautiful in the
    the spring.
On a bright sunny day, in the middle of the night,
I heard a lie I believe is true.
If you don't believe me,
Ask the deaf man, he heard it too!
```

➤ Examine the differences using `diff`:

```
$ diff file1 file2
1,2c1,2
< After I drive on the parkway, I park in my driveway.
< Paris is beautiful in
---
> After I park on the parkway, I drive in my driveway.
> Paris is beautiful in the
4d3
< Jumbo shrimp are smaller than microbreweries!
6a6,7
> If you don't believe me,
> Ask the deaf man, he heard it too!
```

COMPARING TEXT FILES: diff (continued)

- If the two files compare as identical, `diff` produces no output at all.
- If there are any differences, `diff` will generate output to indicate what must be done to the first file so as to make it the same as the second file.
- The output may include these three letters preceded and/or followed by line (record) numbers:
 - a add these lines
 - c change these lines
 - d delete these lines
- The interpretation of the numbers appearing before or after the `a`, `c` or `d` are explained using these examples:
 - 6a6,7 After line 6 in the first file add the lines which are lines 6 through 7 in the second file.
 - 1,2c1,2 Change lines 1 through 2 in the first file so that they look like lines 1 through 2 in the second file.
 - 4d3 Delete line 4 in the first file that appears to be after what is line 3 in the second file.

COMPARING TEXT FILES: diff (continued)

- Following the number and letter combination, the lines themselves are shown with left hand "gutter" characters indicating which file they're excerpted from:
 - < Excerpted from the first file.
 - > Excerpted from the second file.
- In the case of changed lines, the lines excerpted from the first file are separated from the lines excerpted from the second file by a row of three dashes:
 - --- Separates excerpted lines from each file.

COMPARING BINARY FILES: `cmp`

- `diff` is designed to work with text files. Text files are files containing printable characters.
- If you need to compare binary files (such as compiled object code, which cannot be "viewed" in a conventional sense), use `cmp`:

```
$ cmp file1 file2
file1 file2 differ: char 9, line 1
```

- `cmp` produces no output if the files are identical, otherwise it simply reports the character offset at which the first difference was found.

SEARCHING FOR FILES: `find`

- `find` is designed to examine one or more portions of the file system in a recursive fashion, searching "downward" from one or more specified starting directories.

- Here is a very general syntax of the `find` command:

```
find directory ... [-criteria ...] [-action]
```

- `find` generates the file pathnames of all the files it finds that match the specified search criteria.
- The search criteria are specified as keywords that begin with a hyphen (-), and usually followed by a required argument.
- An implied "and" condition occurs when two or more search criteria are specified.
- An "or" condition may be specified by placing `-o` between two search criteria.
- Parenthesis (`()`) may be used for clarity or to override precedence of combined "and" and "or" conditions. They must be "escaped" from the shell with backslashes:
`\(... \)`.
- If any file pathnames are generated, `find` then applies the specified action to those pathnames, or prints them to the standard output, by default. Note that earlier versions of `find` do not exhibit this default behavior; an action must be specified.

SEARCHING FOR FILES: find (continued)

- The pathnames generated will either be absolute or relative, governed by the format of the starting directories.
- Here are some examples of `find` (output is NOT shown):

- Start in the current directory and print the relative pathnames of all files named `xfile`:

```
$ find . -name xfile -print
```

- Start in the `/etc` directory and print the absolute pathnames of all files modified less than seven days ago:

```
$ find /etc -mtime -7 -print
```

- Start in both the `/tmp` and `/var/tmp` directories and print the absolute pathnames of all files owned by `usr17`, modified more than 30 days ago and larger than 1000 KB (1 MB):

```
$ find /tmp /var/tmp -user usr17 -mtime +30 -size +1000k -print
```

- Start in the `/tmp` directory and remove (`rm`) all files modified more than 180 days ago. When using the `-exec` action, a command line follows, and must be terminated by an escaped semicolon (`\;`) and should include an empty pair of curly braces (`{ }`) to indicate where each file pathname should be inserted:

```
$ find /tmp -mtime +180 -exec rm {} \;
```

COUNTING LINES, WORDS & CHARACTERS: WC

- `wc` reads files named on the command line (or the standard input if no filenames are given) and reports to the standard output the number of:
 - lines (logical records separated by `<NEWLINE>` characters)
 - words (separated by one or more `<SPACES>` or `<TABS>`)
 - characters (bytes).

- Here is a very general syntax of the `wc` command:

```
wc [-options] [filename ... ]
```

- The universal options are:
 - `-l` (el) Report only the line count.
 - `-w` Report only the word count.
 - `-c` Report only the character count.
- By default, the behavior is as if the options were given as `-lwc`.
- If more than one input file is specified, `wc` also reports a final total of the requested tallies.

COUNTING LINES, WORDS & CHARACTERS: wc (continued)

- For example:

```
$ wc file1
   6  42 218  file1
```

```
$ wc file1 file2
   6  42 218  file1
   7  50 236  file2
  13  92 454  total
```

```
$ wc -w file1
   42  file1
```

```
$ who | wc
   1   6  61
```

- In the last example, I was the only one on the system at the time!

COMBINING MOST OF IT...

- Here is a single command line, broken out over many individual lines (using the `\` to "hide" the newlines entered when the `<ENTER>` key is pressed; it is required at the end of the third and fourth lines as part of `sed`'s insert command syntax) that uses many of the commands described in this section of the document. The syntax `<TAB>` indicates where the `<TAB>` key was pressed:

```
$ who | grep '^usr' | tr -s ' ' '<TAB>' | cut -f1,3- | \
sed 's/\(.*<TAB>.*\)<TAB>\(.*\)<TAB>\(.*\)/\1 \2 \3/' | \
sort | sed 'li\
USER      LOGGED IN AT\
-----'
```

USER	LOGGED IN AT
----	-----
usr01	Jan 8 9:34
usr05	Jan 8 12:30
usr17	Jan 8 14:27

← the output

- Here's how it works:
 - The output of `who` is piped into `grep` which filters through only records beginning with "usr".
 - The output of `grep` is piped into `tr` which translates spaces to tabs and squeezes out multiple consecutive occurrences of those converted characters.
 - The output of `tr` is piped into `cut` which cuts out copies of fields 1 and 3 through the end of each record, based on a single tab character as the delimiter.
 - The output of `cut` is piped into `sed` that essentially converts all but the first tab back into a single space.

COMBINING MOST OF IT... (continued)

- The output of `sed` is piped into `sort`, which produces an ascending ASCII sorted output.
- The output of `sort` is piped into another `sed` which inserts two header records before line 1.
- This could have been accomplished many different ways. That's usually the case in the Unix environment, since there are so many powerful utilities and each of them typically has many options available to it.

EXERCISES

1. Change into the `Lab11` directory. Use the `grep` command to print all of the records found in all of the files in the current directory that begin with `A` (a capital A).
2. Given the nature of the `rain` file, in which each record contains a city and state and a decimal number representing the average July rainfall, use `grep` to print the records for states that have 2 or more inches of rain but less than 5 inches (i.e. 2.0 through 4.9).
3. Use `grep` to print the records from the `state_birds` file that contain 2 consecutive identical lower case letters.
4. Use `grep` and one of its options to print the names of the files in the current directory that contain any tab characters.
5. Use the `cut` command to print only the state names and population figures from the `states` file. In the `states` file each record has three tab-delimited fields that are state name, year of statehood and population.
6. Use the `sort` command to print out the `state_birds` file sorted by name of bird, which is the second field of each record and is separated from the first field (state name) by a single tab character.
7. Use the `tr` command to remove all new-line characters from the file called `words`. The octal ASCII code for the new-line character is `012`.

EXERCISES (continued)

8. Use the `paste` command to join together the files `text1`, `text2` and `text3` in that order, using a single space as the "glue" and printing the result to the screen.
9. Repeat the previous exercise, pasting together the three text files but redirecting the output to a file called `text`. Then use the `paste` command again and paste together the **standard input** and the newly created `text` file, redirecting the output to a file called `numbered_text` and furnishing the numbers 1 through 5, one per line, from the keyboard. Use a `<CONTROL-D>` on a line by itself to terminate the keyboard entry.
10. Execute a long listing (`ls -l`) of the current directory and redirect its output into a pipe to the `sed` command. Furnish two edit commands as argument to `sed`: one to delete the first line of input and another to globally substitute a single tab character in place of any sequence of one or more spaces. Redirect the output of `sed` into a pipe to the `cut` command and use it to cut out only fields 1, 5 and 9. This should produce output showing only the file type and permissions, the file size in bytes and the name of each file.
11. Repeat the previous exercise adding a final pipe to the `sort` command so as to sort the output numerically by file size from largest to smallest.

EXERCISES (continued)

12. Use the `find` command to start searching from your home (login) directory and print the full (absolute) pathnames of all regular files that have been modified less than 1 day (24 hours) ago. Discard any errors by redirecting them to `/dev/null`.

EXERCISE SOLUTIONS

In some of these exercise solutions, the output is not shown so as not to "clutter" the pages.

1. Change into the `Lab11` directory. Use the `grep` command to print all of the records found in all of the files in the current directory that begin with `A` (a capital A).

```
$ grep '^A' *
file1:After I drive on the parkway, I park in my driveway.
file2:After I park on the parkway, I drive in my driveway.
file2:Ask the deaf man, he heard it too!
poem:As a noun: Blue is a color.
poem:As an adjective: Blue skies are beautiful
poem:As a transitive verb: What will blue litmus?
rain:Atlanta, GA          5.3
state_birds:Alabama      Yellowhammer
state_birds:Alaska       Willow Ptarmigan
state_birds:Arizona      Cactus Wren
state_birds:Arkansas     Mockingbird
state_caps:Alabama       Montgomery
state_caps:Alaska        Juneau
state_caps:Arizona       Phoenix
state_caps:Arkansas      Little Rock
```

2. Given the nature of the `rain` file, in which each record contains a city and state and a decimal number representing the average July rainfall, use `grep` to print the records for states that have 2 or more inches of rain but less than 5 inches (i.e. 2.0 through 4.9).

```
$ grep '[234]\.' rain
New York, NY          4.4
Chocego, IL           3.6
Delles, TX            2.2
```

EXERCISE SOLUTIONS (continued)

3. Use `grep` to print the records from the `state_birds` file that contain 2 consecutive identical lower case letters.

```
$ grep '\([a-z]\)\1' state_birds
Alabama      Yellowhammer
Alaska       Willow Ptarmigan
California    California Valley Quail
```

4. Use `grep` and one of its options to print the names of the files in the current directory that contain any tab characters.

```
                <TAB> character here
                ↓
$ grep -l ' ' *
file1
file2
state_birds
state_caps
states
```

5. Use the `cut` command to print only the state names and population figures from the `states` file. In the `states` file each record has three tab-delimited fields that are state name, year of statehood and population.

(Only partial output shown)

```
$ cut -f1,3 states
Delaware      783600
Pennsylvania  12281054
New Jersey    8414350
Georgia       8186453
```

EXERCISE SOLUTIONS (continued)

6. Use the `sort` command to print out the `state_birds` file sorted by name of bird, which is the second field of each record and is separated from the first field (state name) by a single tab character.

```
$ sort +1 state_birds
Arizona    Cactus Wren
California California Valley Quail
Arkansas   Mockingbird
Alaska     Willow Ptarmigan
Alabama    Yellowhammer
```

7. Use the `tr` command to remove all new-line characters from the file called `words`. The octal ASCII code for the new-line character is `012`.

The output shown here will wrap around your screen or window, since it is one very long record.

```
$ tr -d '\012' < words
ThisweeksomepeopleareattendingtheUnixFundam
entalsclassusingastudentguidecreatedbyTopSh
elfTechnologies.
```

EXERCISE SOLUTIONS (continued)

8. Use the `paste` command to join together the files `text1`, `text2` and `text3` in that order, using a single space as the "glue" and printing the result to the screen.

<SPACE> character here



```
$ paste -d' ' text1 text2 text3
```

At this point we are doing the lab exercises associated with Chapter 11 of the Unix Fundamentals course. In this lab, you have an opportunity to use some of the very powerful Unix utility commands.

9. Repeat the previous exercise, pasting together the three text files but redirecting the output to a file called `text`. Then use the `paste` command again and paste together the **standard input** and the `text` file, redirecting the output to a file called `numbered_text` and furnishing the numbers 1 through 5, one per line, from the keyboard. Use a <CONTROL-D> on a line by itself to terminate the keyboard entry.

```
$ paste - text > numbered_text
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

```
<CONTROL-D>
```

```
$
```

EXERCISE SOLUTIONS (continued)

10. Execute a long listing (`ls -l`) of the current directory and redirect its output into a pipe to the `sed` command. Furnish two edit commands as argument to `sed`: one to delete the first line of input and another to globally substitute a single tab character in place of any sequence of one or more spaces. Redirect the output of `sed` into a pipe to the `cut` command and use it to cut out only fields 1, 5 and 9. This should produce output showing only the file type and permissions, the file size in bytes and the name of each file.

(Only partial output shown)

```

                2 spaces
                ↓
                1 tab
                ↓
$ ls -l | sed -e '1d' -e 's/ */ /g' | cut -f1,5,9
-rw-rw-r-- 218 file1
-rw-rw-r-- 236 file2
-rw-rw-r-- 187 poem
-rw-rw-r-- 168 rain
-rw-rw-r-- 284 sample
-rw-rw-r-- 85 sed_script
-rw-rw-r-- 121 state_birds
-rw-rw-r-- 92 state_caps
.
.
.
```

EXERCISE SOLUTIONS (continued)

11. Repeat the previous exercise adding a final pipe to the sort command so as to sort the output numerically by file size from largest to smallest.

(Only partial output shown)

```
$ ls -l | sed -e 'ld' -e 's/ */ /g' | cut -f1,5,9 | sort -r -n +1
-rw-rw-r-- 284 sample
-rw-rw-r-- 244 states
-rw-rw-r-- 236 file2
-rw-rw-r-- 218 file1
-rw-rw-r-- 187 poem
-rw-rw-r-- 168 rain
-rw-rw-r-- 121 words
-rw-rw-r-- 121 state_birds
-rw-rw-r-- 108 text1
-rw-rw-r-- 92 state_caps
-rw-rw-r-- 85 sed_script
-rw-rw-r-- 54 text3
-rw-rw-r-- 39 text2
-rw-rw-r-- 26 xfile
```

Here is the same line shown in larger print and broken across two lines using a backslash to "hide" the <ENTER> keystroke (although on most modern systems, simple leaving an "open" pipe at the end of a line implies continuation of the line):

```
$ ls -l | sed -e 'ld' -e 's/ */ /g' | \
cut -f1,5,9 | sort -r -n +1
```

EXERCISE SOLUTIONS (continued)

12. Use the `find` command to start searching from your parent directory and print the relative pathnames of all regular files that have been modified less than 1 day (24 hours) ago. Discard any errors by redirecting them to `/dev/null`.

(No output shown)

```
$ find .. -type f -mtime -1 -print 2> /dev/null
```