

Some utility programs.

diff: Compare Two Text Files

You have old version C file and new version, find the changes:

```
diff old.c new.c
```

Outputs lines to add, lines to delete, lines to replace.

diff: Compare Two Text Files

You have old version C file and new version, find the changes:

```
diff old.c new.c
```

Outputs lines to add, lines to delete, lines to replace.

Exit code: 0 = no difference; 1 = has difference.

⇒ Can write shell script if-then-else to act on that.

diff: Compare Two Text Files

You have old version C file and new version, find the changes:

```
diff old.c new.c
```

Outputs lines to add, lines to delete, lines to replace.

Exit code: 0 = no difference; 1 = has difference.

⇒ Can write shell script if-then-else to act on that.

You have old version C project directory and new version, find the changes:

```
diff -r olldir newdir
```

diff: Compare Two Text Files

You have old version C file and new version, find the changes:

```
diff old.c new.c
```

Outputs lines to add, lines to delete, lines to replace.

Exit code: 0 = no difference; 1 = has difference.

⇒ Can write shell script if-then-else to act on that.

You have old version C project directory and new version, find the changes:

```
diff -r olldir newdir
```

Full features and doc: [link](#)

Version control systems use `diff` or internal equivalent.

diff Basic Output Format

Sample files: `smallscript-v1`, `smallscript-v2`

```
diff smallscript-v1 smallscript-v2
```

2,3d1 < dryrun= < verbose=	v1 2–3 not in v2 at 1 (d = delete)
7c5 < dryrun=y --- > dryrun=yes	v1 7 is v2 5, but changed (c = change)
13a12 > ;;	v2 12 not in v1 at 13 (a = add)

diff “Unified” Output Format

```
diff -u smallscript-v1 smallscript-v2
```

<pre>--- smallscript-v1 <time> +++ smallscript-v2 <time></pre>	The files
<pre>@@ -1,16 +1,15 @@</pre>	Next chunk: v1 1–16, v2 1–15
<pre>#!/bin/sh -dryrun= -verbose= while [\$# -gt 0]; do case "\$1" in -n) - dryrun=y + dryrun=yes ;; (etc.)</pre>	- is in v1, + is in v2

Similar to `git diff` and github commit display.

grep: Search in Text File

Why B36 should be a prerequisite

You specify a “pattern”, grep outputs matching lines.

Video clip: [Aho's clip](#). In particular:

- ▶ “pattern”: regular expression
- ▶ “the algorithm” translates regular expression to non-deterministic finite-state automaton, then sees if it accepts your input
- ▶ “the program”: grep

Example: Pick out HTML start tags: `grep '<[a-zA-Z]*>' index.html`

Exit code: 0 = found something; 1 = no match

Full features and doc: [link](#)

grep's Regular Expressions 1/2

Tricky: Similar to but different from shell patterns.

Some base cases:

<code>c</code>	matches the letter c
<code>ace</code>	matches the string ace (concatenation, next slide)
<code>[fin]</code>	matches f or i or n
<code>[a-g]</code>	matches any character in that range
<code>[^fin]</code>	matches any character except f, i, n
<code>[^a-g]</code>	matches any character except that range
<code>.</code>	matches any character
<code>^</code>	matches beginning of line
<code>\$</code>	matches end of line
<code>\b</code>	matches empty string at edge of word

`\b` Example: look for word “int”, so not “printf”:

```
grep '\bint\b' mycfile.c
```

grep's Regular Expressions 2/2

Tricky: Similar to but different from shell patterns.

Some inductive cases. Let r, s be grep regular expressions. From high to low precedence:

without -E	with -E	
$\backslash(r\backslash)$	(r)	parenthesizing
$r\backslash?$	$r?$	0 or 1 time of matching r
r^*	r^*	0 or more times of matching r
$r\backslash+$	r^+	1 or more times of matching r
rs	rs	concatenation
$r\backslash s$	$r s$	r or s

sort

Sort, or check-if-sorted, or merge sorted files. But by what key? Default: whole line.
Customizable by...

Sample input ([fruits.txt](#)), 3 fields per line:

Frank	orange	104
Albert	strawberry	79
Tim	orange	52

Sort by 3rd field (the numbers):

```
sort -b -k 3,3n
```

'n' means treat as number not string. (Exercise: What if omitted?)

Sort by 2nd field (the fruits); when tie, by 3rd field:

```
sort -b -k 2,2 -k 3,3n
```

--debug shows what is actually used as key(s).

Full features and doc: [link](#)

find: Look for Files

Automatic recursive traversal of a directory tree and operate on selected files.

Full feature and doc: [link](#).

Typical form:

```
find dir ... expression
```

For each dir given, start there and recurse down. The expression determines which files to pick out, and what to do with them.

find Expressions: Tests

Filename matching: `-name '*.pdf'`

Regular file vs directory: `-type f`, `-type d`

Owning user or group: `-user trebla`, `-group cmsusers`

Permissions: `-readable`, `-writable`, `-executable`

Times:

`-mtime +3 -mtime -6`

(last modified 3–6 days ago)

`-mmin +3 -mmin -6`

(last modified 3–6 minutes ago)

find Expressions: Logical Connectives

Multiple tests already ANDed together. But can also use explicit -a, -and

OR: -o, -or

NOT: prefix !

Also parentheses. Example:

```
find mydir '!' '(' -mmin +3 -mmin -6 ')'
```

find Expressions: Actions

If no action, implicitly `-print`

`-print`: Print pathname.

`-delete`: Delete.

Example: Find Python files and put in zip file:

```
find . -name '*.py' | zip a08-homework.zip -@
```

Example: Find Python files and print their paths and delete:

```
find . -name '*.py' -print -delete
```