find, grep, sed, & awk

Have covered basics of filesystems…

These 2 lectures outline the next phase of course:

find, grep, regex, awk, (s)ed

before final phase of bash scripting

These will be covered in more detail soon.

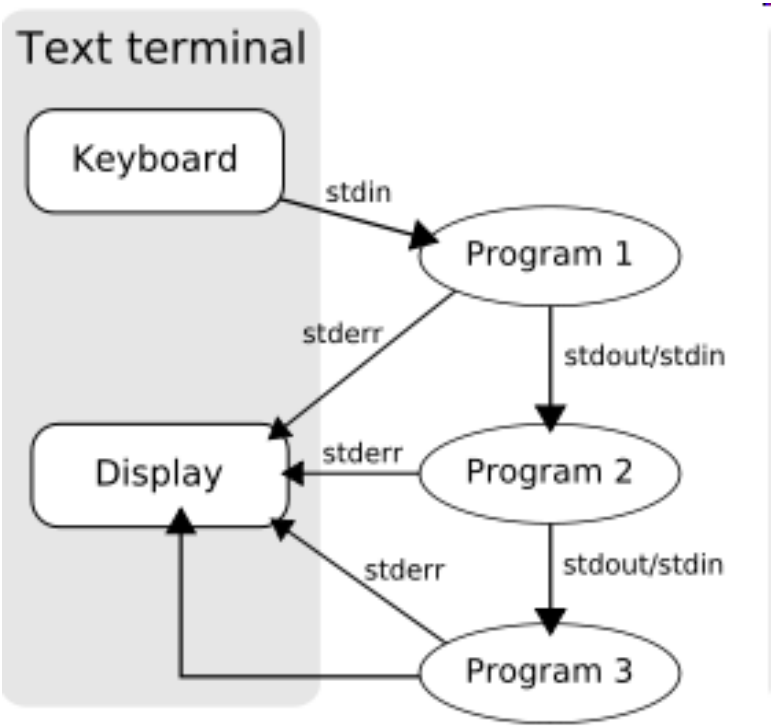
This is just so you can see the end-game
While progressing through details!
...in case you can’t see the wood, for the trees!

Brief review

All can use regex to specify string patterns

- find – (& locate, which) to find files by names
  - or other attributes such as size, age etc.
  - Remember locate does same but needs to index first, delayed
- grep – to find strings within files
  - But can pipe output ls –R and use grep in place of find
  - How do you think find is done anyway…?
- sed – to edit strings within files
  - Cryptic + powerful = danger! Always test first.
- awk – to split and process fields within files.
  - flexible typeless parsing language
  - can act as a basic accounting/monitoring language
  - can even edit like sed.

Unix philosophy



"This is the Unix philosophy:

Write programs that do one thing and do it well.

Write programs to work together.

Write programs to handle text streams, because that is a universal interface."

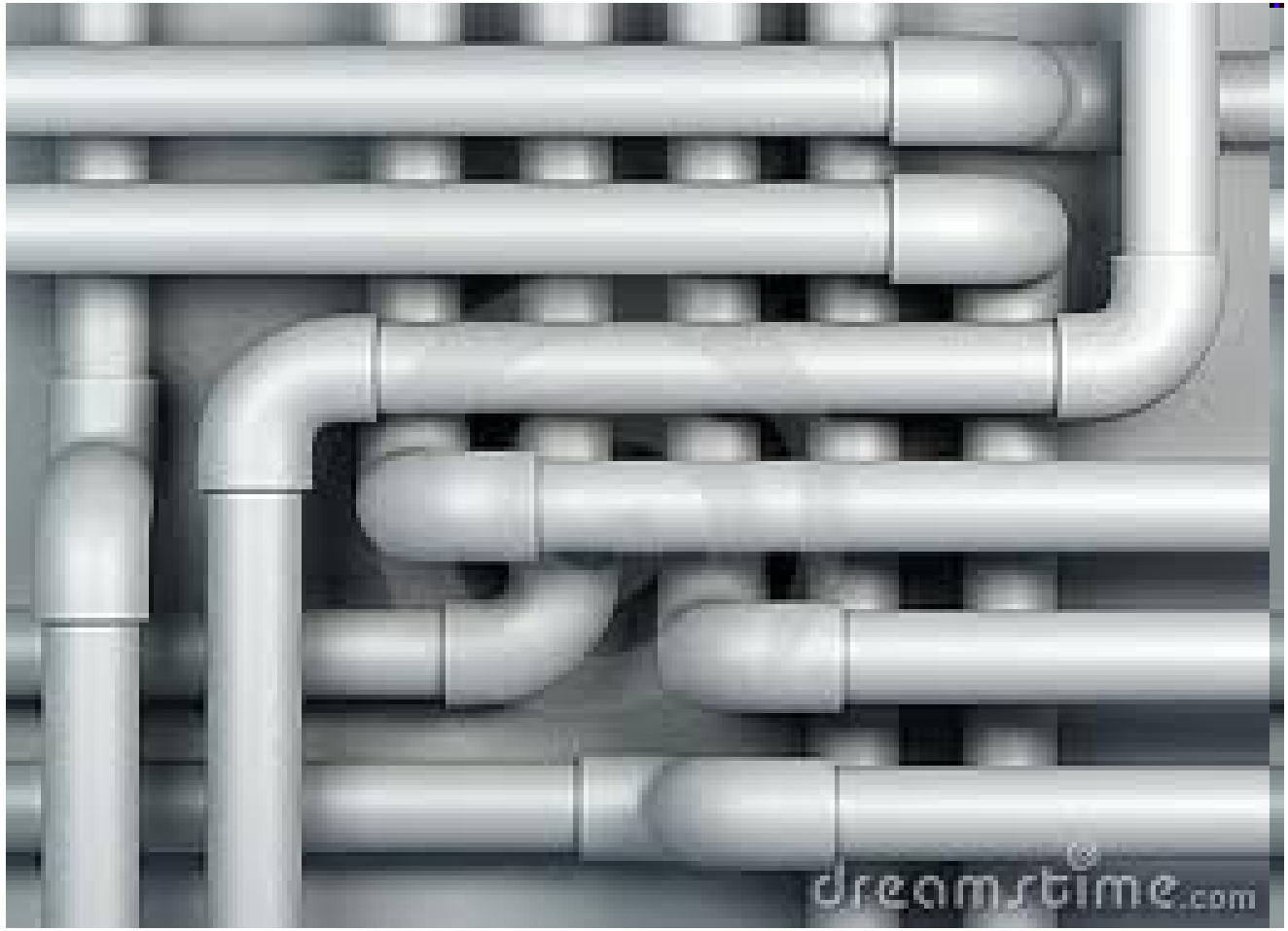
--Doug McIlroy,

Inventor of Unix pipes

Why learn command-line utils?

- Simple – “do one thing”
- Flexible – built for re-use
- Fast – no graphics, no overhead
- Ubiquitous – available on every machine
- Durable ~ 50 years so far …
  - But of course updates under the hood.
- First – recap ways to combine simple tools

Part 0 – pipes and `xargs`



Some simple programs

List files in current working directory:

```
$ ls
foo bar bazoo
```

Count lines in file `foo`:

```
$ wc –l foo
42 foo
```

Putting programs together

```
$ ls | wc –l
3
```

Specs say -l counts newlines,
(but seems to count tabs at times)
w counts words, seems consistent

```
$ ls | xargs wc –l
42 foo
31 bar
12 bazoo
85 total
```

- l for **l**ine count,
not 1 for one

xargs – for commands that don't
- extracts arguments from stdin
  (above, from piped 'ls')
- Runs the following command on these arguments one by one.

-exec or | xargs?

- -exec has crazy syntax… { } \; for assumed args

- | xargs
  - fits Unix philosophy.
  - may fail for filenames having whitespace, quotes or slashes.

- exec { } \; is
  - slow, runs new instance of command once for each arg.
  - not sensible, sorts 'alphabetically,
    - rather than order original input was presented,
    - would give wrong results if sequence was critical

GNU version of xargs

xargs [-0prtx] [-E eof-str] [-e[eof-str]] [--eof[=eof-str]] [--null] [-d delimiter] [--delimiter delimiter] [-I replace-str] [-i[replace-str]] [--replace[=replace-str]] [-l[max-lines]] [-L max-lines] [--max-lines[=max-lines]] [-n max-args] [--max-args=max-args] [-s max-chars] [--max-chars=max-chars] [-P max-procs] [--max-procs=max-procs] [--interactive] [--verbose] [--exit] [--no-run-if-empty] [--arg-file=file] [--show-limits] [--version] [--help] [**command** [initial-arguments]]

xargs reads items from the standard input,

- delimited by
  - blanks (which can be protected with double or single quotes or a backslash)
  - or newlines,
- and executes the **command** (default is /bin/echo) one or more times with any initial-arguments followed by items read from standard input. Blank lines on the standard input are ignored.

Unix filenames containing blanks and newlines, are incorrectly processed by xargs.

Use the '-0' option, to overcome this and ensure the program which produces input for xargs also uses a null character as a separator, such as find with '-print0' option.

BSD Unix version of xargs

xargs [-0opt] [-E eofstr] [-I replstr [-R replacements]] [-J replstr] [-L number]
[-n number [-x]] [-P maxprocs] [-s size] [utility [argument ...]]

The xargs utility

•reads space, tab, newline and end-of-file delimited strings from the standard input

•and executes utility with the strings as arguments.

Arguments specified on the command line are given to utility upon each invocation, followed by some number of the arguments read from the standard input of xargs.

The utility is repeatedly executed until standard input is exhausted.

Spaces, tabs and newlines may be embedded in arguments using
single (“ ‘ “) or double (‘ " ‘) quotes or backslashes (“ \ “).

Single quotes escape all non-single quote characters, excluding newlines.

Double quotes escape all non-double quote characters, excluding newlines.

Any single character, including newlines, may be escaped by a backslash.

The options are as follows:

-0 expect NUL (“\0") characters as separators, instead of spaces and newlines.
This is expected to be used in concert with the -print0 function in find(1).

Alternatives: find, locate, which

find

- search for filenames in a directory hierarchy according to criteria

locate

- searches indexes updated by system periodically,
- so locate may not be up to date!

which cmd – finding path to utilities : cmd can be a blank separated list

- gives pathname of the program/executable which performs the command
- search follows the normal sequence of the users $PATH;
- reminder $PATH is an environment variable with dirs of cmds
- so the first match in the normal PATH is returned, and run!

find : search for a file in a directory tree to a specified level

Linux - the GNU version

find [-H] [-L] [-P] [path...] [expression]

This searches the directory tree rooted at each given file name by evaluating the given expression from left to right, according to the rules of precedence (see section OPERATORS), until the outcome is known (the left hand side is false for and operations, true for or), at which point find moves on to the next file name. If no starting-point is specified, the current directory '.' is assumed.

Mac Unix BSD

find [-H | -L | -P] [-EXdsx] [-f path] path ... [expression]
find [-H | -L | -P] [-EXdsx] -f path [path ...] [expression]

The find utility recursively descends the directory tree, for each path listed, evaluating an expression (composed of the ``primaries'' and ``operands'' listed below) in terms of each file in the tree.

GNU version

All of these are subject to the depth search restrictions from –maxdepth, -mindepth.

-P

Never follow symbolic links; just take the information from the properties of the symbolic link itself. This is the default behaviour in the absence of any options.

-L

Follow symbolic links.

-H

Do not follow symbolic links, except when the file is a link specified on the command line. If the link is broken, information about the link, itself is used as a fallback.

If more than one of -H, -L and -P is specified, each overrides the others; the last one takes effect… but check, don't bet your life on it!

Basic find examples

```
.  after find refers to the current directory,
Other paths may be specified.
$ find . –name Account.java
$ find /etc –name '*.conf'    # etc dir has config
$ find . –name '*.xml'
$ find . -not -name '*.java' -maxdepth 4
$ find . \(-name '*jsp' –o –name '*xml'\)
```

- -**i**name : case-**i**nsensitive
- Bash (not regex) logicals:- ! == -not -o == OR -a == AND
- Quotes – rules for quotes in shells are ‘twisted’
  - Will do later, Best ignored and just used
- \ backslash forces following char to be takenliterally, rather than a control or metacharacter

Find and do stuff

```
$ find . –name '*.java' | xargs wc –l | sort
```

```
Other options using exec with {} for arg list:
$ find . –name '*.java' -exec wc –l {} \; | sort
$ find . –name '*.java' -exec wc –l {} + | sort
```

More details on next slide…

For general post-processing, of files found,
use pipes & xargs etc. as above

Limited by your imagination. `mv, rm, cp, chmod.`

Find by type

Files:

```
$ find . –type f
```

Directories:

```
$ find . –type d
```

Links: (more in relation to broken links later!)

```
$ find . –type l
```

By modification time

Numeric arguments can be specified as

+n for greater than n,

-n for less than n,

n for exactly n.

Times

– time – days
– min – mins
– newer – than a specified file
– c – change
– a – access

Changed within day:

```
$ find . –mtime -1
```

Changed within minute:

```
$ find . –mmin -15
```

Variants aren't especially useful.

`-time`, `-cmin`,

`-atime`, `-amin`

By modification time, II

Compare to file

- newer than foo.txt

```
$ find . –newer foo.txt
```

-NOT (!) newer

```
$ find . ! –newer foo.txt
```

By modification time, III

Compare to date

$ find . -type f -newermt '2010-01-01'

Between dates

$ find . -type f -newermt '2010-01-01' > ! -newermt '2010-06-01'

Also can be done with bash logical ops

find . -type f \(-newermt '2010-01-01' -a ! -newermt '2010-06-01' \)

-a for logical AND
\(& \) to backslash (escape) brackets so they delineate logical expression

-newermt – m=modification time; t=interpret following timestring directly
(See manual for more,
other options exist.. a,B,c = access, Birth(created),c-inode status change time refn.)

Find by permissions

```
$ find . –perm 644

$ find . –perm –u=w
$ find . –perm –ug=w
$ find . –perm –o=x
```

Find by size

Less than 1 kB:

```
$ find . –size -1k
```

More than 100MB:

```
$ find . –size +100M
```

`find` summary:

- Can search by name, path, depth, permissions, type, size, modification time, and more.
- Once you find what you want, pipe it to `xargs or exec` for further processing

Part 2: `grep`

- **g**lobal / **r**egular **e**xpression / **p**rint
- From ed command `g/re/p`
- For finding text inside files.

Basic usage:

```
$ grep <string> <file or directory>
$ grep 'new FooDao' Bar.java
$ grep Account *.xml
$ grep –r 'Dao[Impl|Mock]' src
```

- Recursive flag is typical
- Quote string if spaces or regex.
- Don't quote shell filename with * wildcards!

  (* will be interpreted as regex multiplier not shell wildcard

  e.g. ls b* will list all files beginning with b, followed by anything

  – But ls ‘b*’ will only list files whose names are a string of b’s!)

Common `grep` options

Case-insensitive search:

```
$ grep –i foo bar.txt
```

Only find word matches:

```
$ grep –rw foo src
```

Display line number:

```
$ grep –nr 'new Foo()' src
```

Filtering results

Inverted search:

```
$ grep –v foo bar.txt
```

Prints lines not containing `foo`.

Typical use:

```
$ grep –r User src | grep –v svn
```

Using `find … | xargs grep …` is faster.

More `grep` options - Context

Search for multiple terms:

$ grep -e foo –e bar baz.txt

Find surrounding lines (c- for centred):

$ grep –r –C 2 foo src

$ man ls | grep -c 10 link

Similarly –A or –B will print lines before and after the line containing match.

`grep` summary:

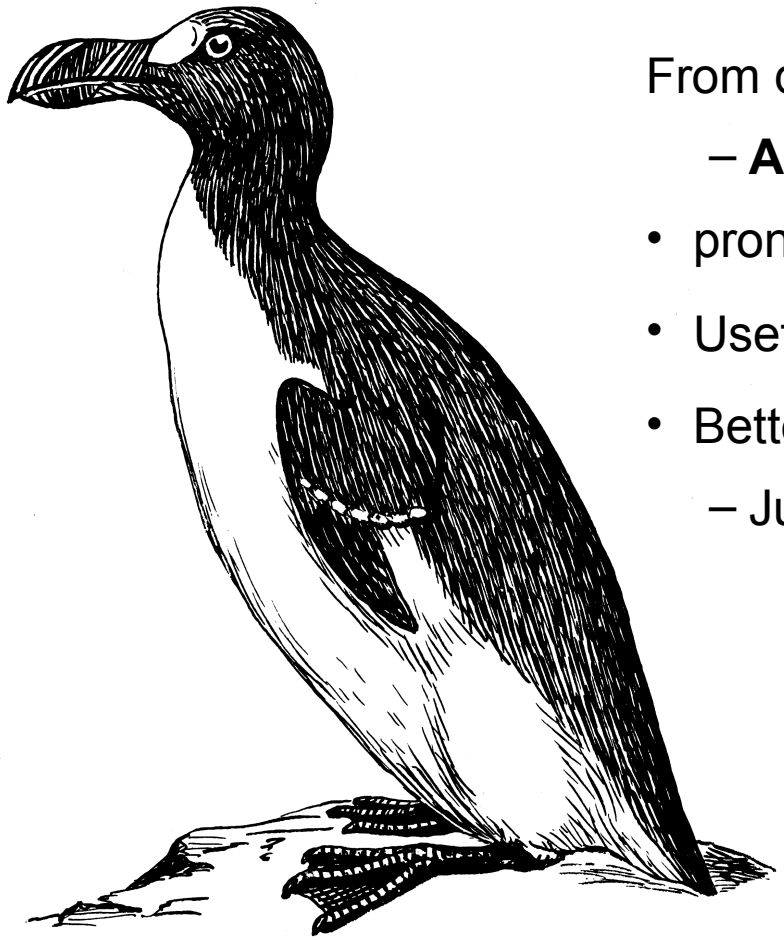
- `-r` **r**ecursive search
- `-i` case **i**nsensitive
- `-w` whole **w**ord
- `-n` prefix match output with line **n**umber
- `-e` multipl**e** searches of regex patterns
- `-f` can specify patterns in a file
- `-A` **A**fter
- `-B` **B**efore
- `-C` **C**entered

A,B, C followed by NUM output
NUM lines After, Before or Centred

links summary: from lab chat

- Links
  - save space & admin & confusion
  - Only one copy, changes to all
- Analogy of sorts
  - Hard-like cash in hand
  - Soft-in bank account-bank broke
- Better analogy :- acct. access
  - hard – like full joint acct. all access
  - Soft – like parent with no will!?
  - Analogies always fail in the end.

Part 4: awk, gawk, mawk GNU & Modern versions



From developers initials

- **A**ho, **W**einberger, **K**ernighan

- pronounced *auk.*
- Useful for text-munging.
- Better explanation to follow
  - Just a sampler

Simple awk programs – parse fields !!

```
Print $n: n=0,1,2 => fields all, 1, 2

In lines below:-
- first $ is cmd prompt
- others field number reference

$ echo 'Jones 123' | awk '{print $0}'
Jones 123

$ echo 'Jones 123' | awk '{print $1}'
Jones

$ echo 'Jones 123' | awk '{print $2}'
123
```

Apache Webserver Common Log File Format

111.222.333.123 HOME - [01/Feb/1998:01:08:39 -0800] "GET /bannerad/ad.htm HTTP/1.0" 200 198

111.222.333.123 HOME - [01/Feb/1998:01:08:46 -0800] "GET /bannerad/ad.htm HTTP/1.0" 200 28083

111.222.333.123 AWAY - [01/Feb/1998:01:08:53 -0800] "GET /bannerad/ad7.gif HTTP/1.0" 200 9332

111.222.333.123 AWAY - [01/Feb/1998:01:09:14 - 0800] "GET /bannerad/click.htm HTTP/1.0" 200 207

Fields

1: User Address IP or domain name of the user accessing the site.
2: RFC931 This field is used to log the domain for multi-homed web servers.
3: User Authentication
4: Date/Time Date and time the user accessed the site.
5: GMT Offset Number of hours from GMT (if this is +0000 it is logged in GMT time).
6: Action The particular operation of the hit (this must be in quotes).
7: Return Code The return code indicates whether or not the action was successful etc.
8: Size The size of the file sent.

Old example – same process:

fcrawler.looksmart.com [26/Apr/2000:00:00:12] "GET /contacts.html HTTP/1.0" 200 4595 "-"
fcrawler.looksmart.com [26/Apr/2000:00:17:19] "GET /news/news.html HTTP/1.0" 200 16716 "-"
ppp931.on.bellglobal.com [26/Apr/2000:00:16:12] "GET /download/windows/asctab31.zip HTTP/1.0" 200 1540096 "http://www.htmlgoodies.com/downloads/freeware/webdevelopment/15.html"
123.123.123.123 [26/Apr/2000:00:23:48] "GET /pics/wpaper.gif HTTP/1.0“ 200 6248 "http://www.jafsoft.com/asctortf/"
123.123.123.123 [26/Apr/2000:00:23:47] "GET /asctortf/ HTTP/1.0" 200 8130 "http://search.netscape.com/Computers/Data_Formats/Document/Text/RTF"
123.123.123.123 [26/Apr/2000:00:23:48] "GET /pics/5star2000.gif HTTP/1.0" 200 4005 "http://www.jafsoft.com/asctortf/"
123.123.123.123 [26/Apr/2000:00:23:50] "GET /pics/5star.gif HTTP/1.0" 200 1031 "http://www.jafsoft.com/asctortf/"
123.123.123.123 [26/Apr/2000:00:23:51] "GET /pics/a2hlogo.jpg HTTP/1.0" 200 4282 "http://www.jafsoft.com/asctortf/"
<snip>

Built-in variables: NF, NR

- NR – Number of Record
- NF – Number of Fields
  - If the 2nd last fied, $(NF-2) = 200 then it indicates web request was successful
- With $, gives field, otherwise number

```
$ awk '{print NR, $(NF-2)}' server.log
1 200
2 200
```

Hits from unique IP address – field 1

```
$ awk '{print $1}' | sort | uniq -c
| sort -rn server.log
```

- Print $1 : extracts IP address
- 1st Sort : sorts them (bash cmd)
- Uniq -c : replaces many identical lines by one with count of how many
- 2nd Sort -rn : sorts by frequency numeric (n) and reverse (r) order

Word frequency in text

```
$ gawk '
{for (i=1;i<=NF;i++) count[$i]++}
END {for (word in count)
       printf word, count[word]}' |
sort -k2 -nr
```

- Loops over all lines from first line
- Increments array count with word as index
- In the END, prints all words & count
- Sort -k2 -rn :
  - sorts key 2 (count), by frequency
    i.e. numeric (n) and reverse (r) order

Structure of an awk program

condition { actions }

```
$ awk 'END { print NR }' server.log
9

$ awk '$1 ~ /^[0-9]+.*/ { print $1,$7}' \
> server.log
123.123.123.123 6248
123.123.123.123 8130
```

Changing delimiter

```
$ awk 'BEGIN {FS = ":"} ; {print $2}'
```

- FS – Field Seperator string
- BEGIN and END are special patterns
  - BEGIN runs it's statement once only at start
  - END runs it's statement once only at end
  - most others are run for all applicable data lines

Or from the command line:

- With the -F flag for Field Separator

```
$ awk –F: '{ print $2 }'
```

Get date out of `server.log`

```
$ awk '{ print $2 }' server.log
[26/Apr/2000:00:00:12]

$ awk '{ print $2 }' server.log \
> | awk –F: '{print $1}
[26/Apr/2000

$ awk '{ print $2 }' server.log \
> | awk –F: '{print $1} | sed 's/\[//'
26/Apr/2000
```

Maintaining state in awk

As noted in the lecture,

The original slide is erroneous, with `{ b += $(NF-1) }`

- probably from a rushed cut and paste and forgetting to edit,
- instead of counting chars, it finds the total sum of the 2nd last field in every line…

To count chars, just accumulate the length of the entire record, but will see again.

i.e. replace `{ b += $(NF-1) } with` { b += length($0) } in each below.

Counting bytes must include the line endings chars, but will avoid for now,

- the point is to show how to maintain an accumulated total using a var… just a demo.

Find total chars transferred from `server.log`

```
$ awk '{ b += length($0) } END { print b }' server.log
1585139
```

Find total chars transferred to `fcrawler`

```
$ awk '$1 ~ /^fcraw.*/ { b += length($0) } END { print b }'\
> server.log
21311
```

One more example

Want to eliminate commented out code in large code base.

Make a one-liner to identify classes that > 50% comments,

A C comment line has “//” as first non-whitespace chars, conventionally followed by a space… hence $1.

```
$ awk '$1 == "//" { a+=1 } END { if (a*2 >
  NR) {print FILENAME, NR,  a}}'
```

Example, continued.

To execute on all Java classes:

```
$ find src -name '*.java' -exec awk '$1 == "//"
  { a+=1 } END { if (a * 2 > NR) {print FILENAME, NR,
  a}}' {} \;
```

- Directory src is conventionally where a project's source code is
- Here –`exec` with `\;` is the right choice, as the awk program is executed for each file individually.
- It should be possible to use `xargs` and FNR, but KISS.
  - NR total record count for all files read
  - FNR record count within current file

awk summary

- `NF` – Number of Field
- `NR` – Number of Records
- `FILENAME` – filename
- `BEGIN, END` – special events
- `FS` – Field Seperator (or –F).
- `awk 'condition { actions }'`

awk extras

not covered here, only basic use in scripts – good to know exist, if ever needed.

- Command line debugging (-D flag)
  - Breakpoints : on reaching code
  - Watchpoints : on changing data
    - Allows user to view program state:
      - values at levels (variables - registers)
      - Watchpoints easier to locate badass values
  - Stackframe : function call history
- Profiling
- Interprocess communication (IPC) & Network Programming

Part 3: sed -

stream **ed**itor

- programmable
- for modifying files & streams
- can do entire filesystem

sed command #1: s

As a filter, it redirects files
(echo just prints on screen, piped to sed here below)
(sed uses ed like cmds : s for substitute/switch/swap/

```
$ echo 'foo' | sed 's/foo/bar/'
bar

$ echo 'foo foo' | sed 's/foo/bar/'
bar foo
```

's/foo/bar/g' – global (only global for all occurrences within line)
Since sed works on all lines within files, line by line,
So there is no global file replace option – not needed.

Typical uses

To test & preview changes onscreen before writing to files.

NB sed does not save changes unless written out to a file.

```
$ sed 's/foo/bar/g' old

$ sed 's/foo/bar/g' old > new


$ sed –i 's/foo/bar/g' file
```

-i extension

edits file **i**n place,

saving backup with extension, if given.

```
$ <stuff> | xargs sed –i 's/foo/bar/g'
```

Real life example I

Sample scenario: - (far out example perhaps)

Each time a batch job is run,

a flag file gets it's only line set to YES,

and the job can't be tested again

until it is reverted to NO.

$ sed –i 's/YES/NO/' flagfile

- Run again to change file again
  - Re-run last command with up-arrow.
  - Or append as script to batch job script
- avoid context switches for increased efficiency.

Real life example II

Modern IDE's with global edits within project files, perform this basic powerful sed functionality, which was invaluable B4 IDE's

Probably not as used now, unless doing remote updates on an inaccessible minimalist system, with minimal bandwidth, otherwise development would be on a powerful local system followed by upload to minimalist remote system.

Assume a bunch of test cases say:

Assert.assertStuff which could be assertStuff, since using JUnit 3.

```
$ find src/test/ -name '*Test.java' \
> | xargs sed –i 's/Assert.assert/assert/'
```

Backslash at end of line '\'
merely signifies line continuation in scripts.

Real life example III

Windows CR-LF is mucking things up.

Win has carriage return (CR) & line feed (LF)

Below…

- first $ is cmd prompt,
- Last $ is standard end of line regex

$ sed 's/.$//' winfile > unixfile

Replaces \r\n with \n (always inserted)

$ sed 's/$/\r/' unixfile > winfile

Replaces \n with \r\n.

Sed regex are fairly restrictive...for speed

POSIX.2 Basic Regular Expressions (BREs) should be supported,
but not completely due to performance problems,
e.g. backreference needs memory : storage & speed processing

The backslash options are supported:

\n	newline
\t	tab
\w	word
\b	word boundary....

The -E option uses extended regular expressions (EREs)

- has been supported for years by GNU sed,
- is now included in POSIX.

Check installation and GNU online documentation for updates.

Capturing groups

```
$ echo 'john doe' | sed 's/\b\(\w\)/\U\1/g'
John Doe

$ echo 'Dog Cat Pig' | sed 's/\b\(\w\)/(\1)/g'
(D)og (C)at (P)ig
```

Explanation : (idea only, full details follow in a few lectures)

‘ … ‘ == issue commands within single quote

s/…/(\1)/g == put brackets around first previous substring

\(…\) == formal substring specification,
(but takes \b here!)

\b , \w == beginning of word, & word respectively

\U == switch to Uppercase

Exercise: formatting phone #.

Convert all strings of 10 digits to (###) ###-####.

Conceptually, we want:

's/(\d{3})(\d{3})(\d{4})/(\1) \2-\3/g’

Where **\n** refers to the nth match of terms in (….)

- Must escape parenthesis and braces with \… { , } etc.
- Brackets are not escaped. ( ) are OK
- But \d (for decimal digit) is/was not supported in sed regex, so rewrite as:

's/\([0-9]\{3\}\)\([0-9]\{3\}\)\([0-9]\{4\}\)/(\1) \2-\3/g’

NB d for delete is supported, but not \d for digit!

Check GNU regex syntaxes

Exercise: trim whitespace

Trim leading whitespace:

```
$ sed -i 's/^[ \t]*//' t.txt
```

Trim trailing whitespace:

```
$ sed –i 's/[ \t]*$//' t.txt
```

Trim leading and trailing whitespace:

```
$ sed -i 's/^[ \t]*//;s/[ \t]*$//' t.txt
```

Add comment line to file with s:

'1s/^/\/\/ Copyright FooCorp\n/'

- Prepends // Copyright FooCorp\n
  - // is comment in C syntax like languages
- 1 restricts to first line, similar to vi search.
- ^ matches start of line.

- Example use with find to insert copyright in all .java or .py files etc.

Insert Shebang function!

In a .bashrc: (normal now rather than .bash_profile)

```
function shebang {
  sed –i '1s/^/#!\/usr\/bin\/env python\n\n' $1
  chmod +x $1
}
```

Prepends #!/usr/bin/env python

(assume python interpreter /usr/bin/env python

and make file executable)

Run by $ shebang my.py

$1 refers to first argument of function,

In this case… my.py

sed command #2: d - delete

Delete lines containing foo:

```
$ sed –i '/foo/ d' file
```

Delete lines starting with #:

```
$ sed –i '/^#/ d' file
```

Delete first two lines:

```
$ sed –i '1,2 d' file
```

More delete examples:

Delete blank lines:

```
$ sed '/^$/ d' file
```

Delete up to first blank line (email header):

```
$ sed '1,/^$/ d' file
```

Note that we can combine range with regex;

From line 1 to an empty line specified with regex

Real life example II, ctd

Again in a software project, a bunch of test classes may have the following unnecessary line:

```
import junit.framework.Assert;
```

```
$find src/test/ -name *.java | xargs \
> sed -i '/import junit.framework.Assert;/d'
```

Backslash at end of line '\' merely signifies line continuation in scripts.

`sed` summary

Back in the day, without decent IDE's for sofware dev:

- With only `s` and `d` you should probably find a use for `sed` once a week.
- Combine with `find` for better results.
- `sed` gets better as your regex improves.
- Syntax often matches `vi`.

Back in the day, might comeback in a day when power, comms and chips are & everywhere & small