Data Analysis in Geophysics
ESCI 7205

Class 5

Bob Smalley

Basics of UNIX commands

Connecting remotely

Basics of the UNIX/Linux Environment

On a Mac running OS-X to connect to the SUNs, from a terminal window enter

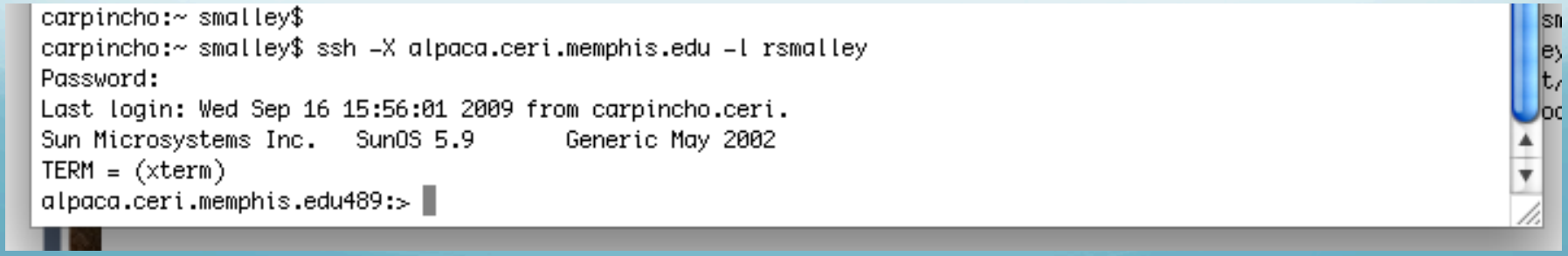
```
ssh –X alpaca.ceri.memphis.edu –l rsmalley
```

The –X flag gives us X-windows graphics capability.

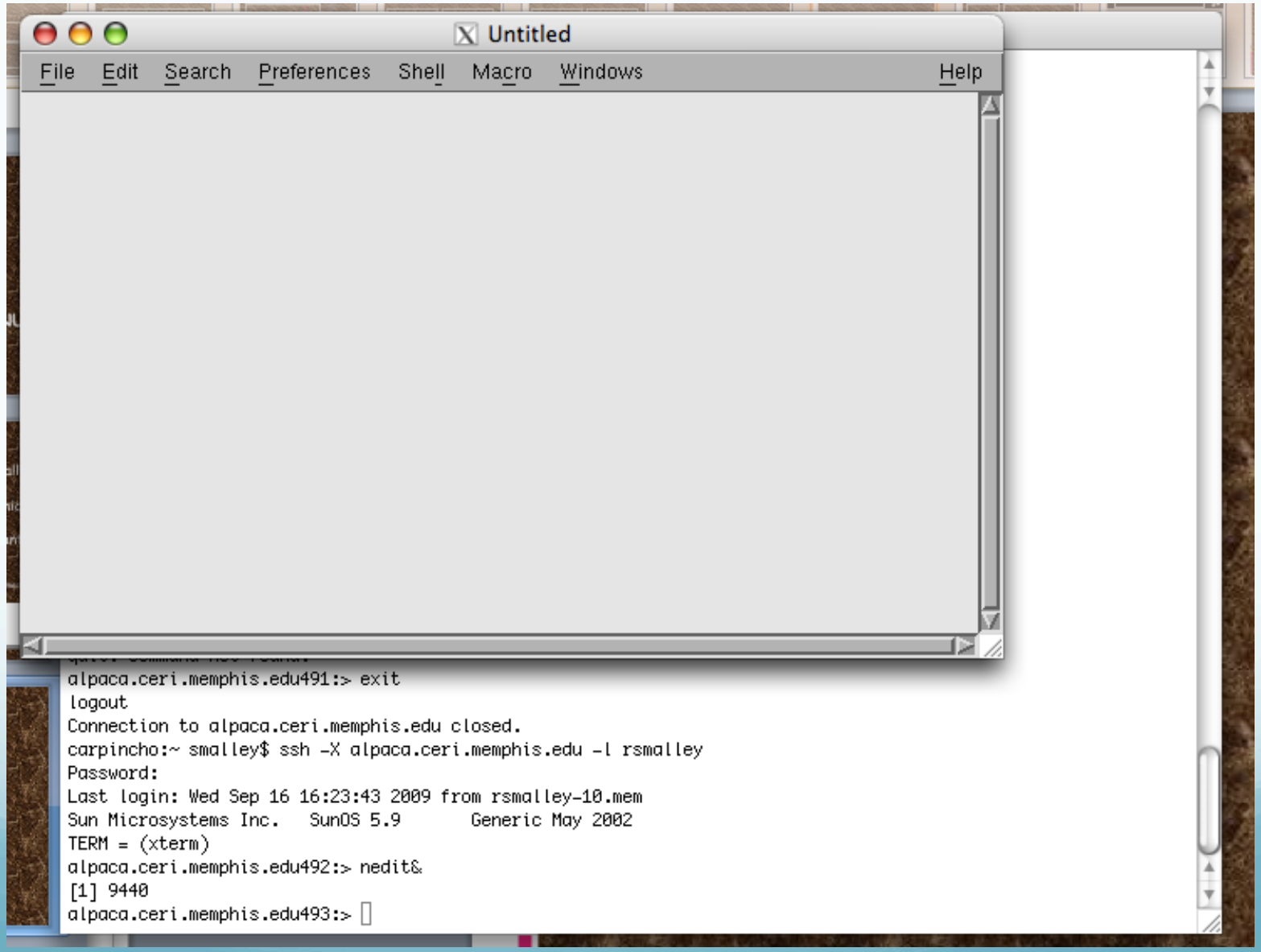
Next is the name of the machine we want to connect to (alpaca.ceri.memphis.edu).

The –l flag passes the username.

(Without this flag, it will pass whatever your username is on the mac.)



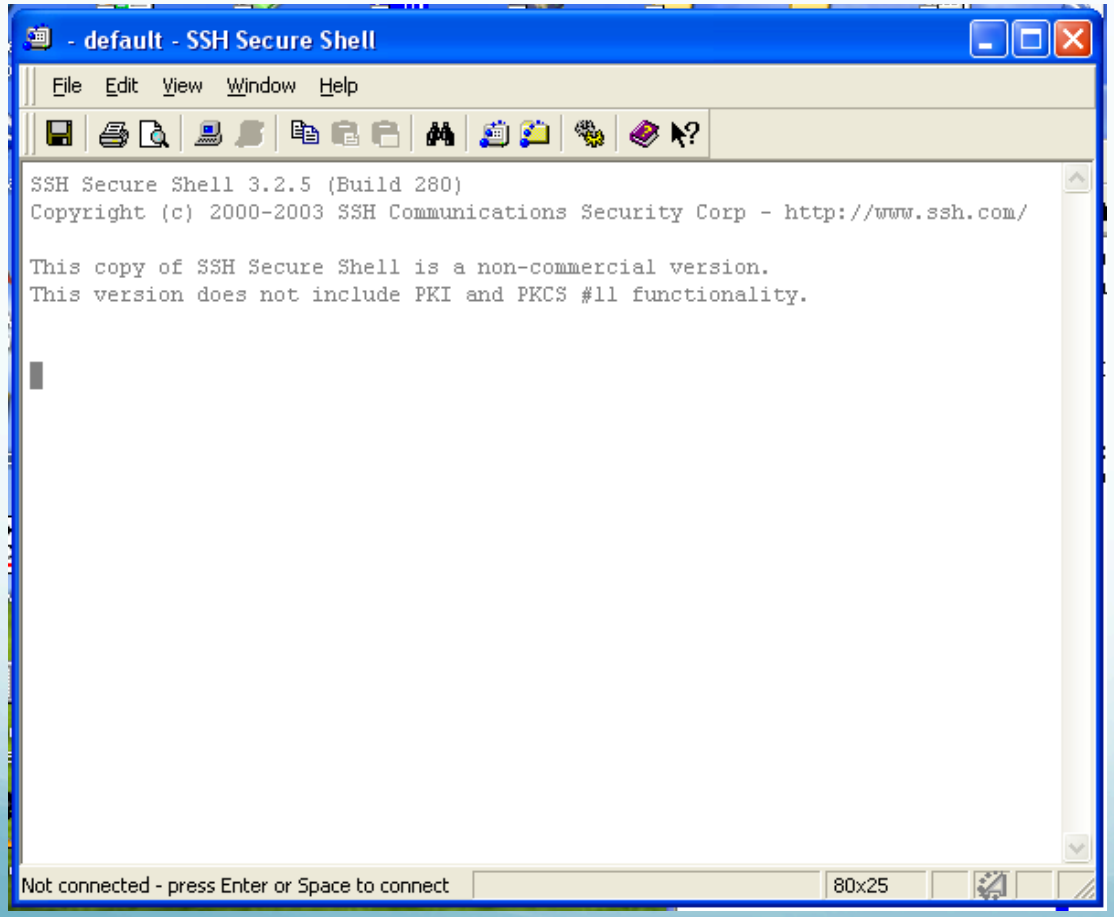
Try running nedit on the SUN.
On the mac – we get X graphics automatically

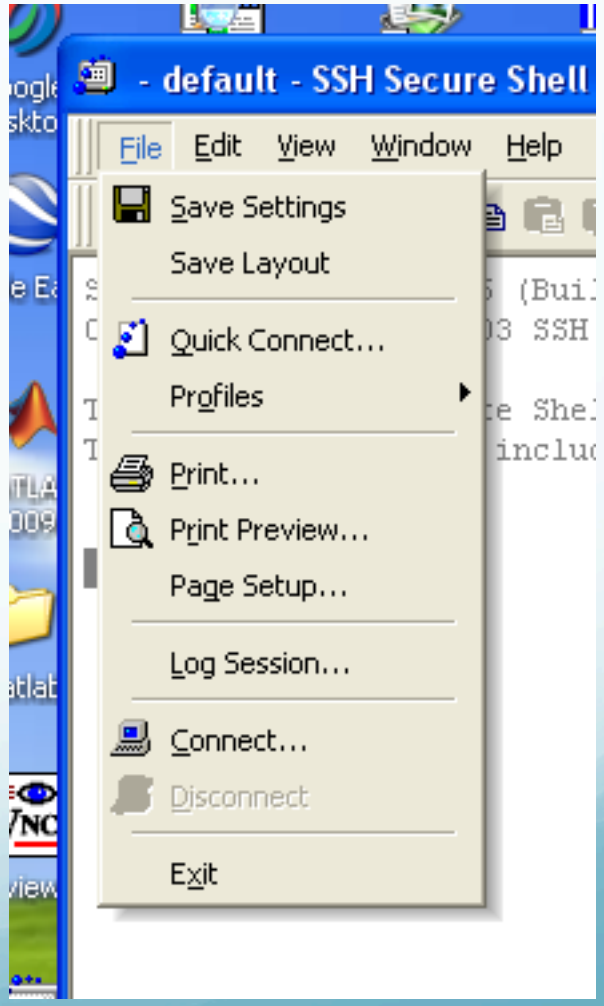


On the PC it is a few more clicks, but first we need (to install) two programs
SSH Secure Shell Client and Exceed (part of the Hummingbird package).

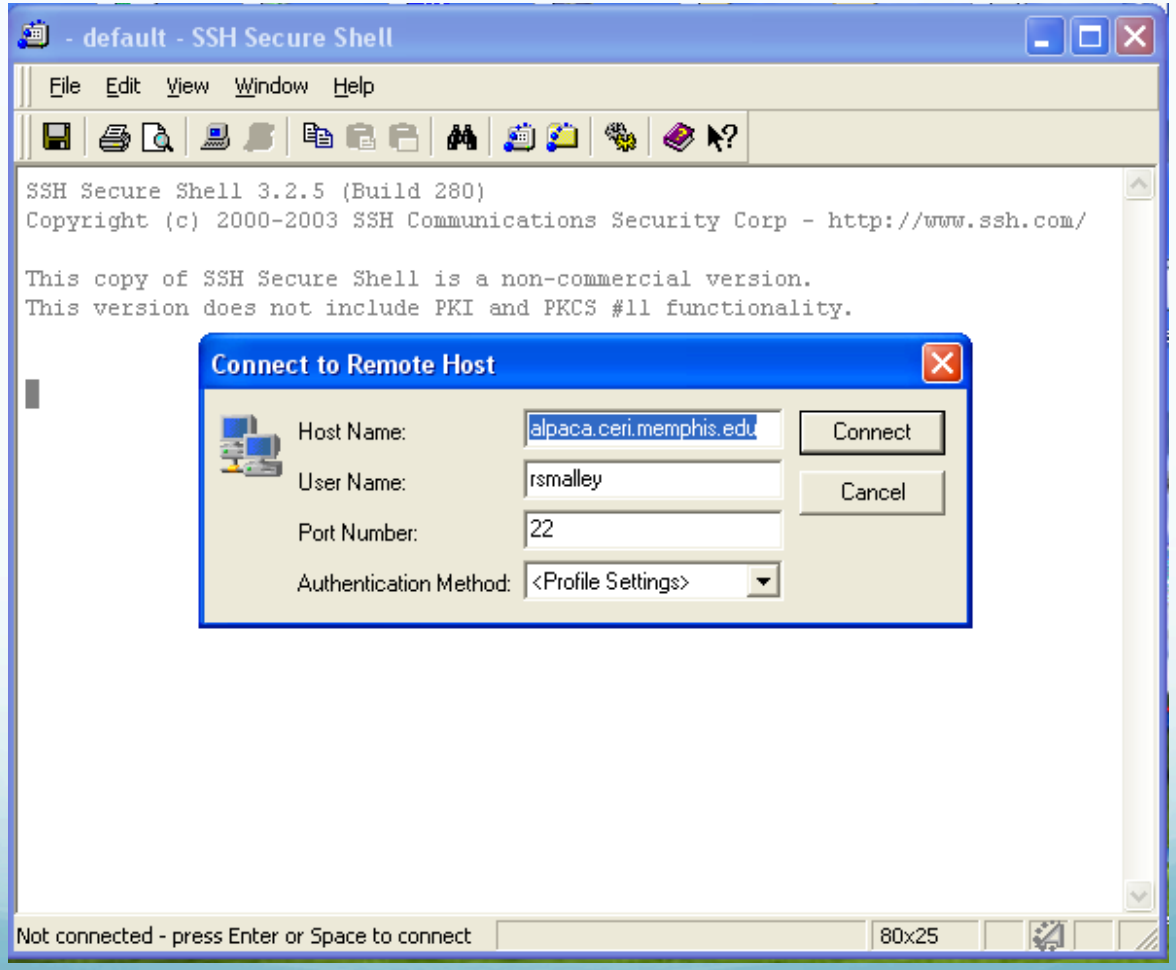
Double click on exceed (it will start up and put an icon in the tray, it does not have a window).
Double click on SSH Secure Shell Client

You will get this window (left). Now we have to connect to a machine. Click on File and then connect.

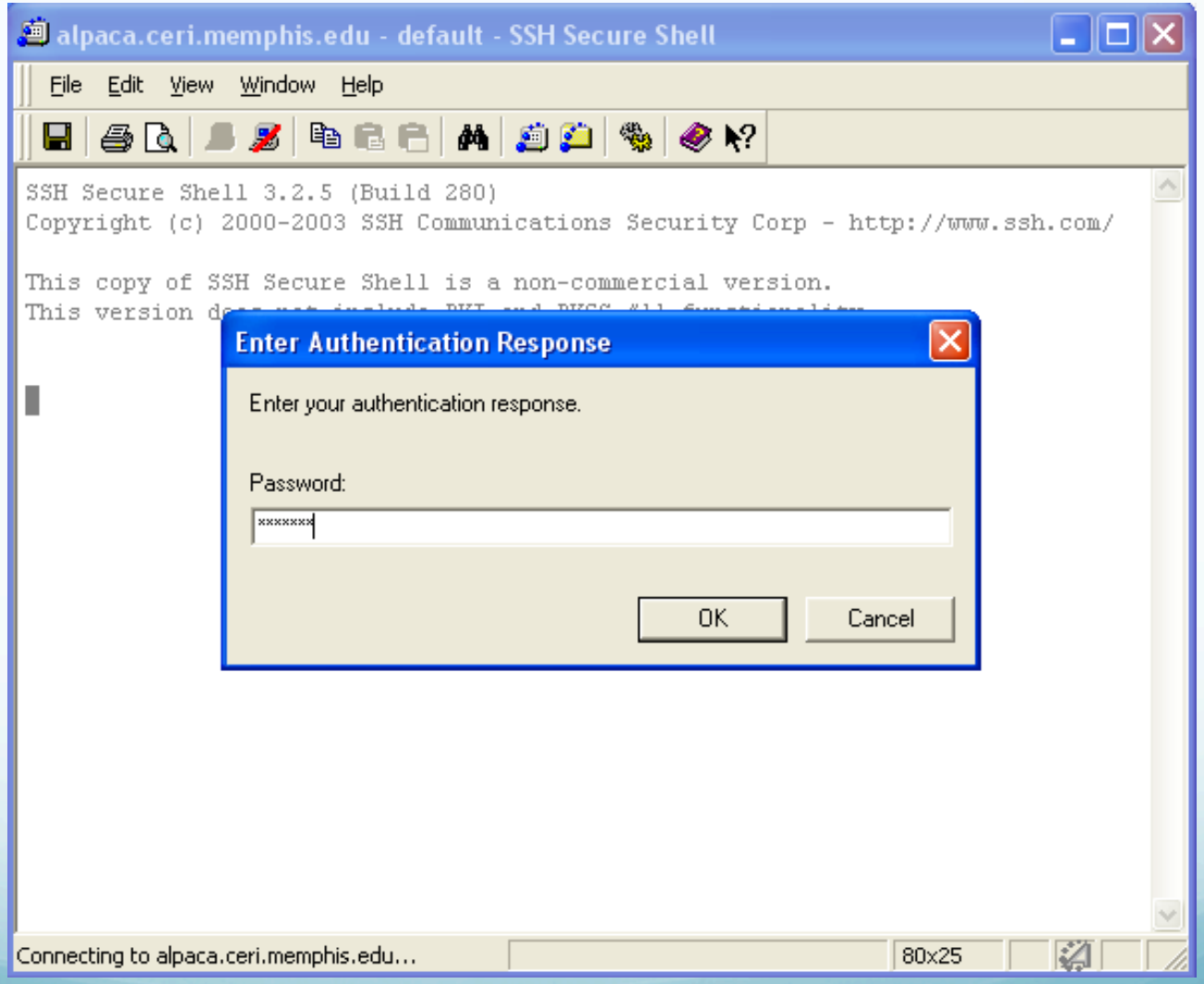




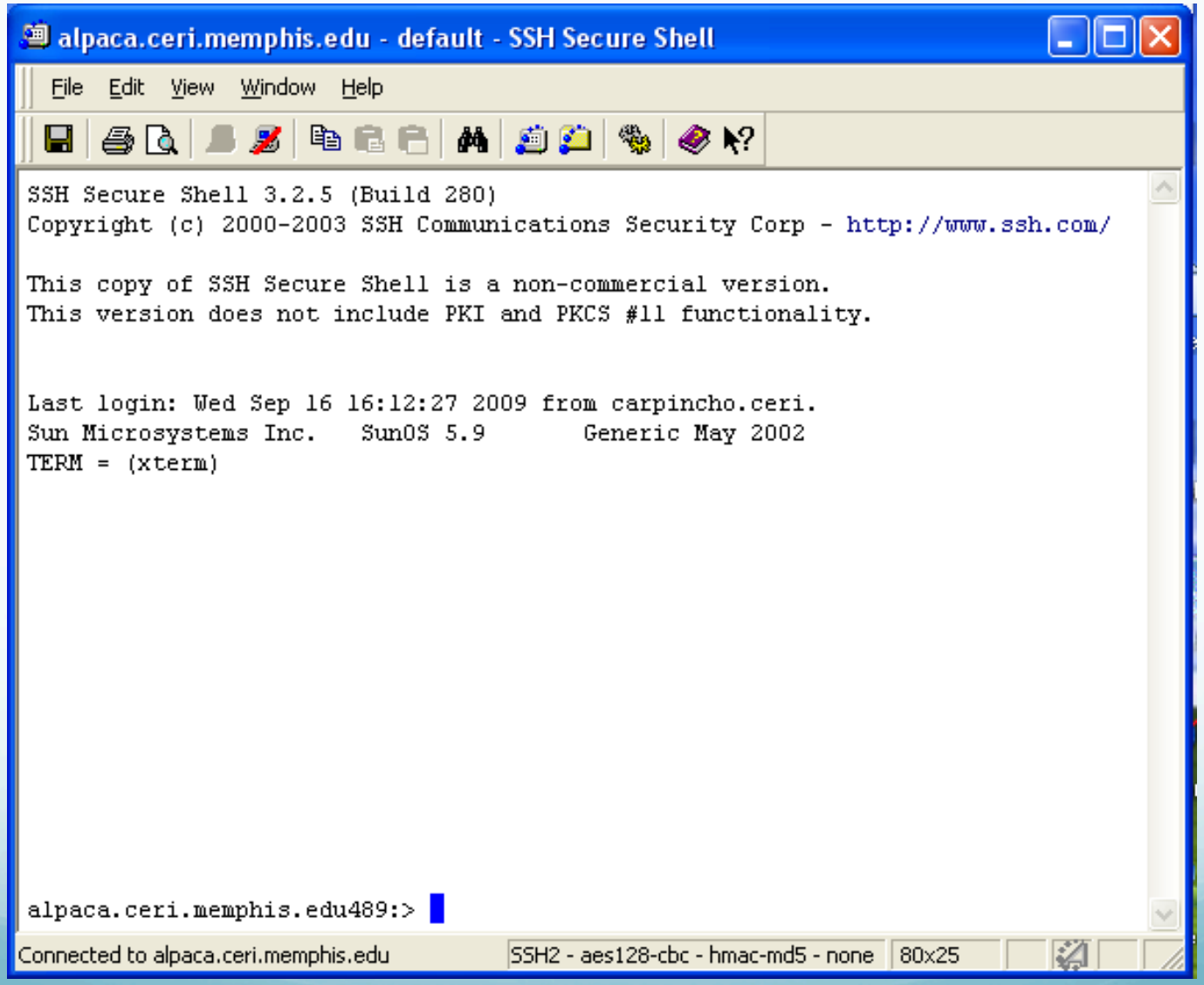
This brings up the connect dialog. Put in the host name you want to connect to and your username. Leave the other stuff alone (default). Click connect.



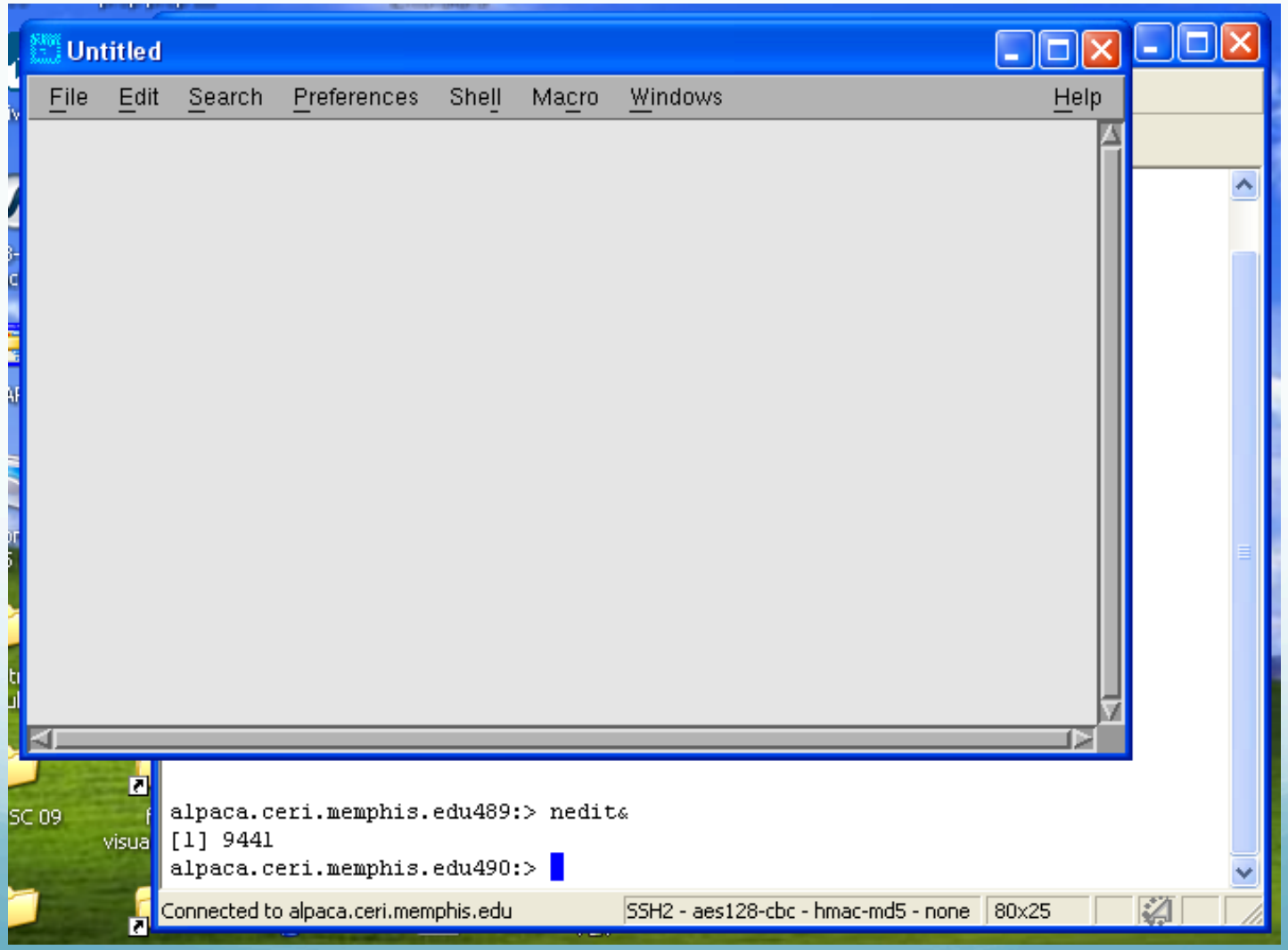
It will now ask for your password.



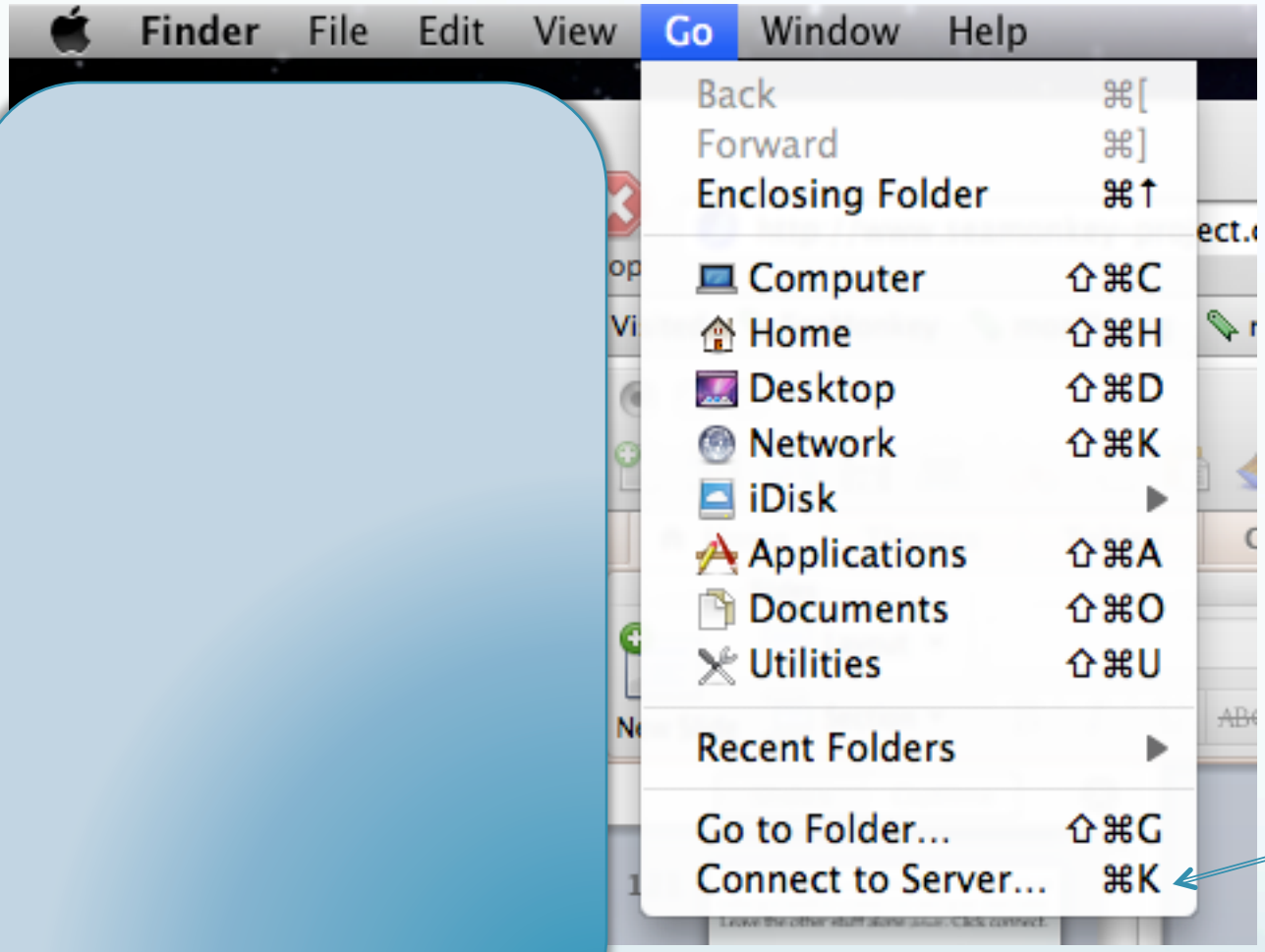
And we are finally connected.



Start nedit in the background (the trailing &). This permits the terminal to continue accepting commands.



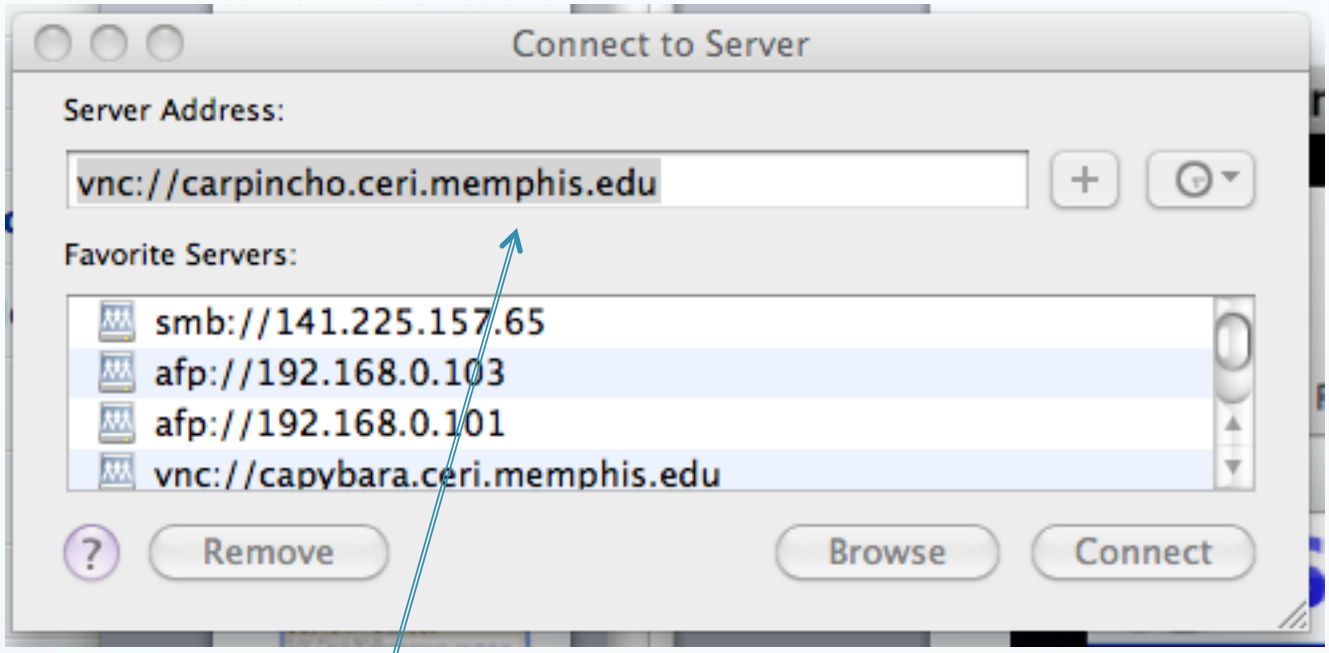
Using Screen Sharing/VNC



Select this under Finder

(will be highlighted – does not come through on screen capture)

Using Screen Sharing/VNC

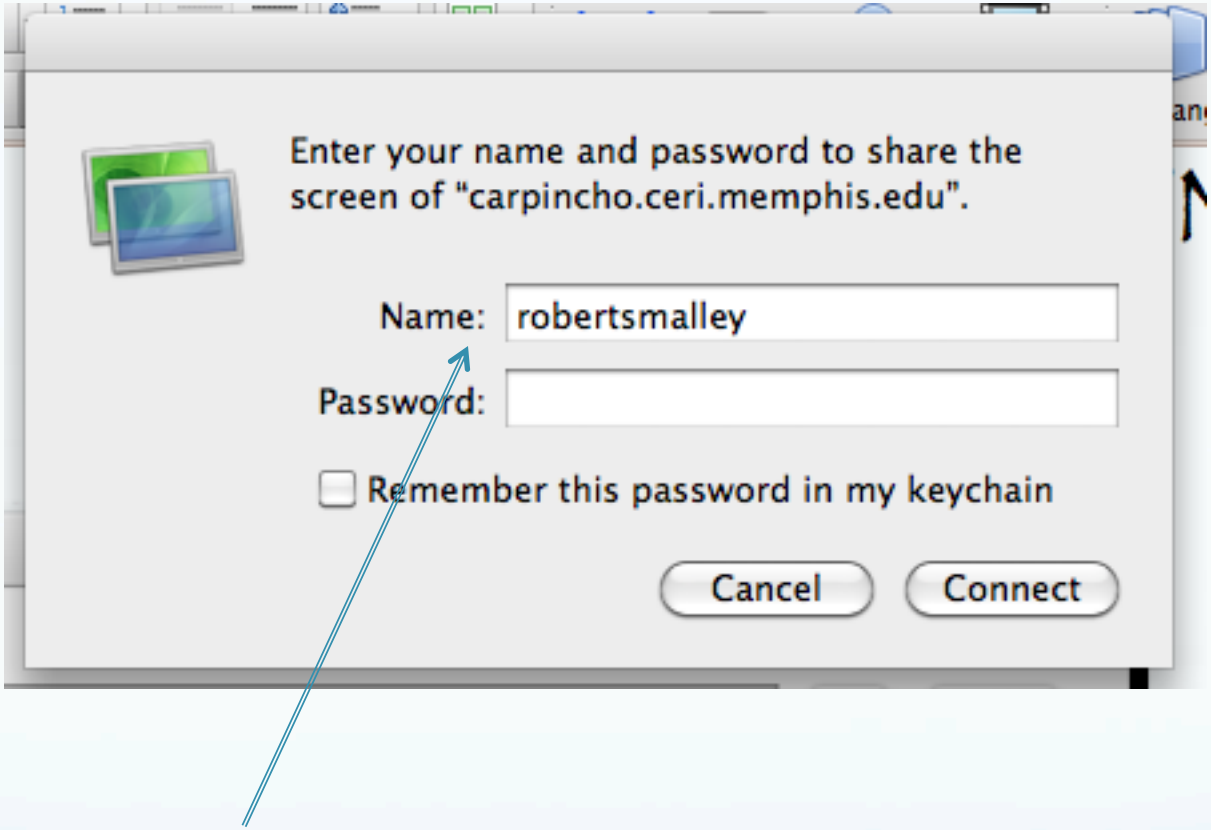


Select address from list, type it in, or browse among machines offering connections.

(will be highlighted – does not come through on screen capture)

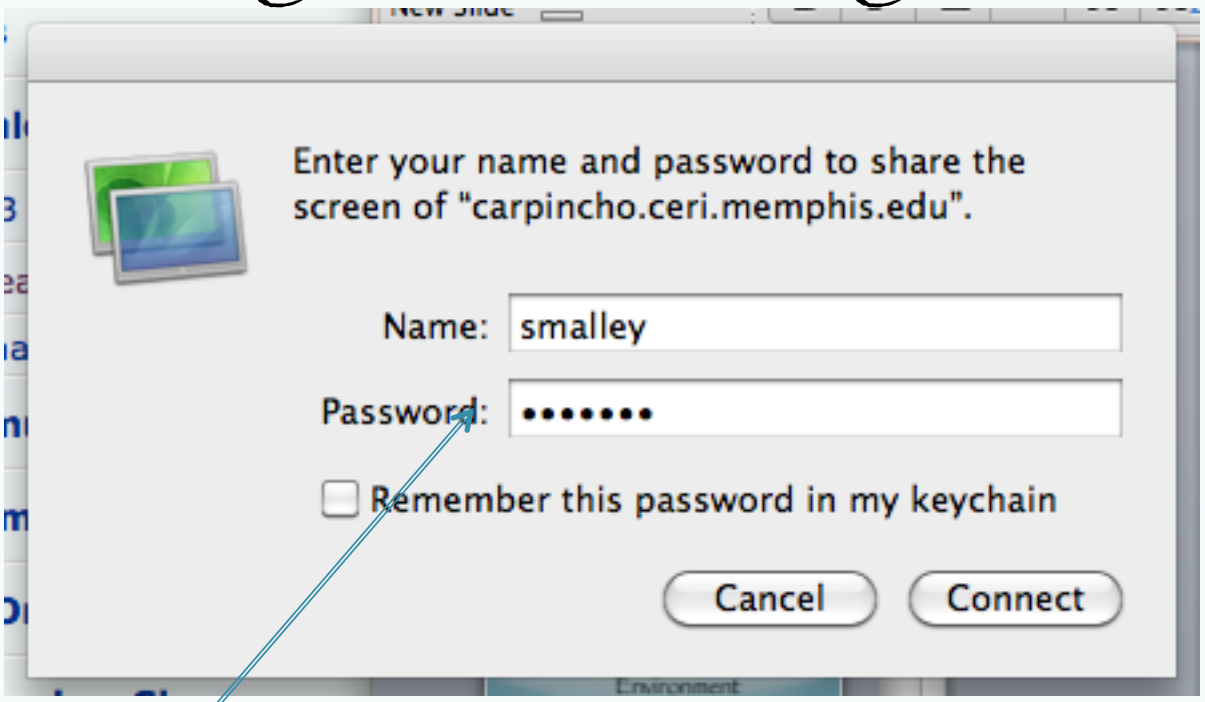
Then click connect

Using Screen Sharing/VNC



Now you get a login screen
It will have automatically put in your username on the LOCAL machine.

Using Screen Sharing/VNC



You may have to change the username to a different one on the REMOTE machine. Plus put in your password.

Not a good idea to have the computer remember your password.

Using Screen Sharing/VNC

Now you will have to log into the REMOTE machine.

The login process is 2 levels – one to connect to the machine (as an authorized user) and one to login (possibly as another authorized user).

Using Screen Sharing/VNC

Grab File Edit Capture Window Help (95%) Tue Sep 13 12:03

Terminal — bash — 153×61

```
ls java.log.4274
java.log.4274
-bash 504 ~ # echo $PATH
```

home | tcd-4-1635-20 | pif_proc_tst.zip | Invitacion Sr. | cant_put_in_pap | Macintosh HD

Robert Smalley's Mac Pro

To access this computer, you need to enter your password.
Name: Robert Smalley
Password: ••••••
Cancel OK

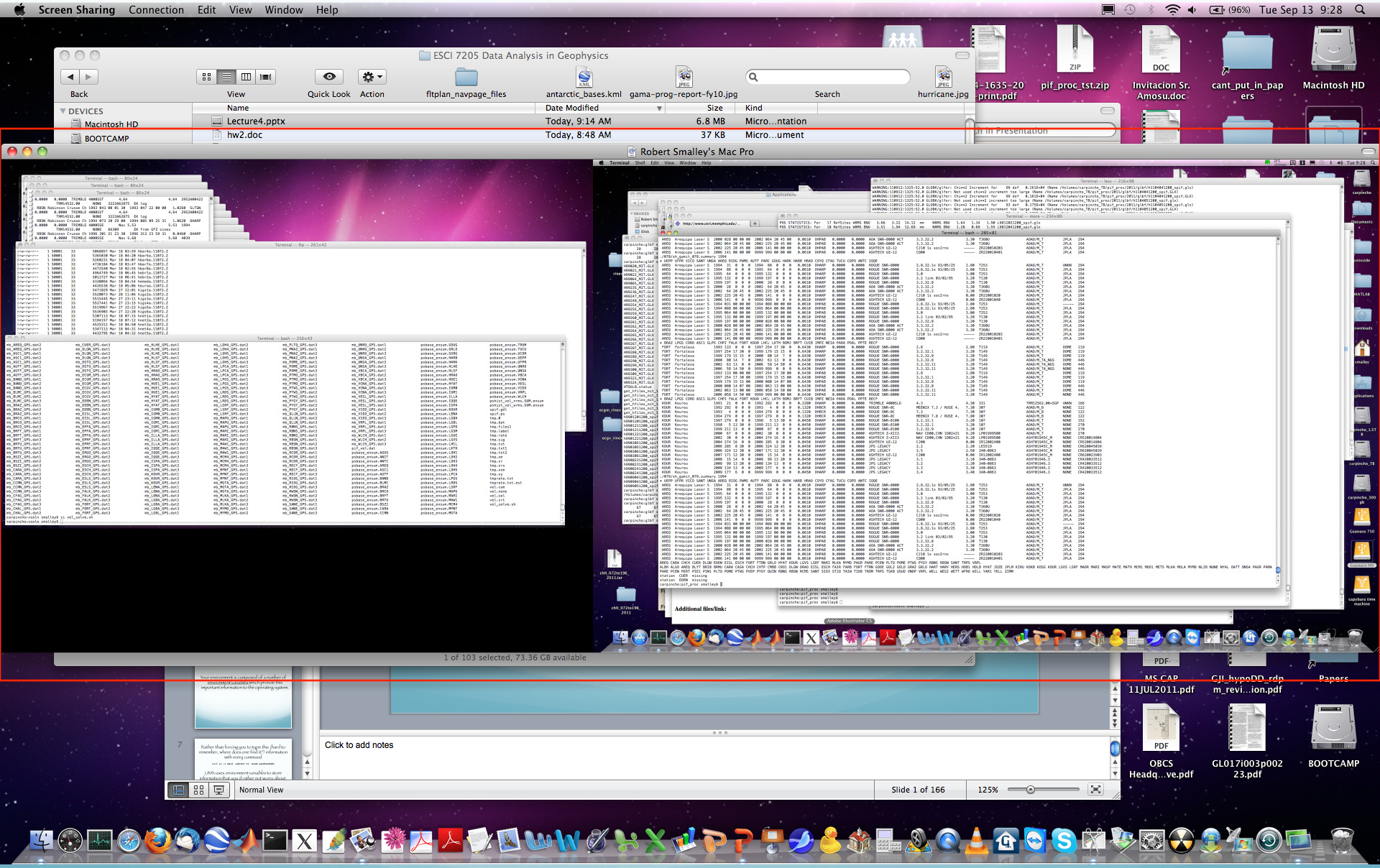
In this example someone is logged in and the screen is locked and you need to enter the password

Now I have the screens – both of them – from the Mac in my office displayed on my laptop! Can be confusing. Cutting and pasting between screens does not work. You are "on" one computer or the other. VNC also works on PC's but you have to install the program and run it on the 'host' machine. It comes already set up on the mac (called screen sharing, but it is the "program" vnc).

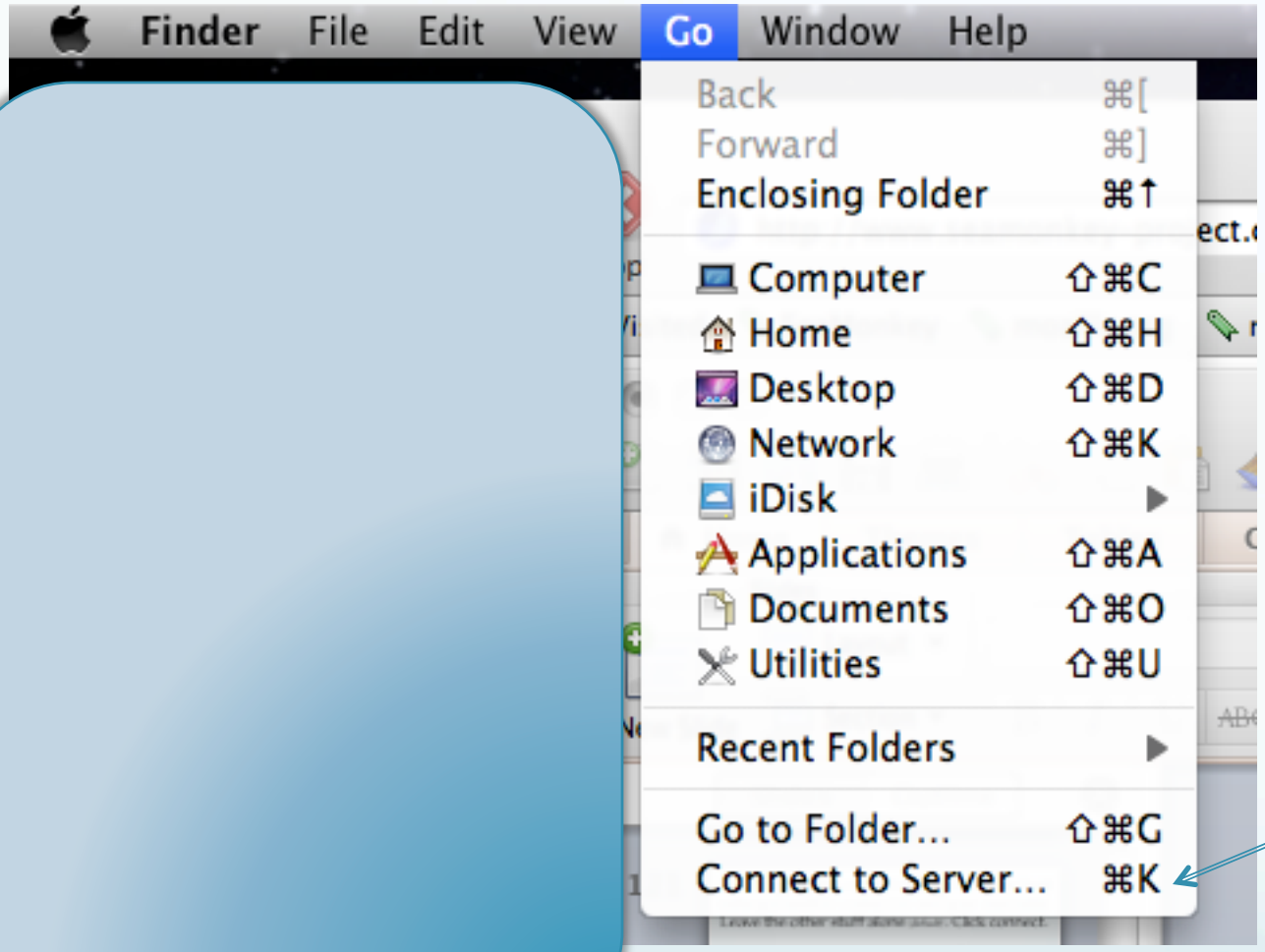
Slide 130 of 168 125%

MS CAP 11JUL2011.pdf | GJI_hypoDD_rdp m_revi...ion.pdf | Papers | OBCS Headq...ve.pdf | GL017i003p002 23.pdf | BOOTCAMP

Using Screen Sharing/VNC



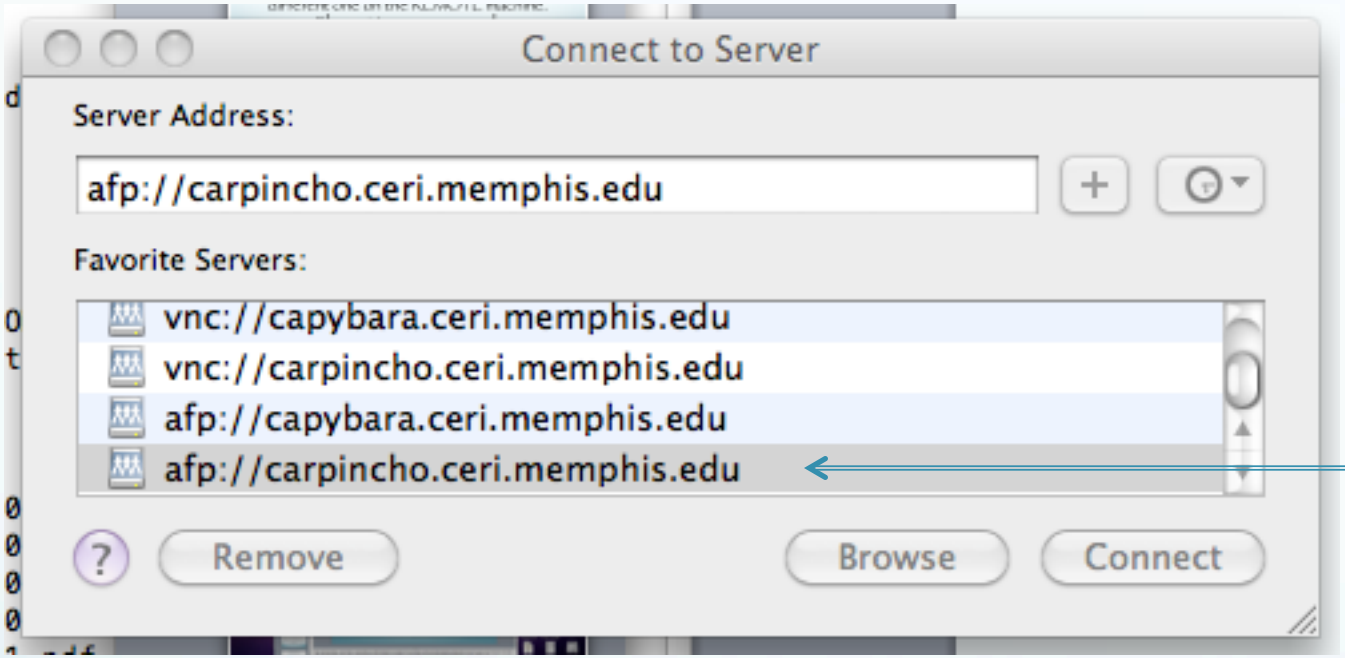
Sharing disks



Select this again

(will be highlighted – does not come through on screen capture)

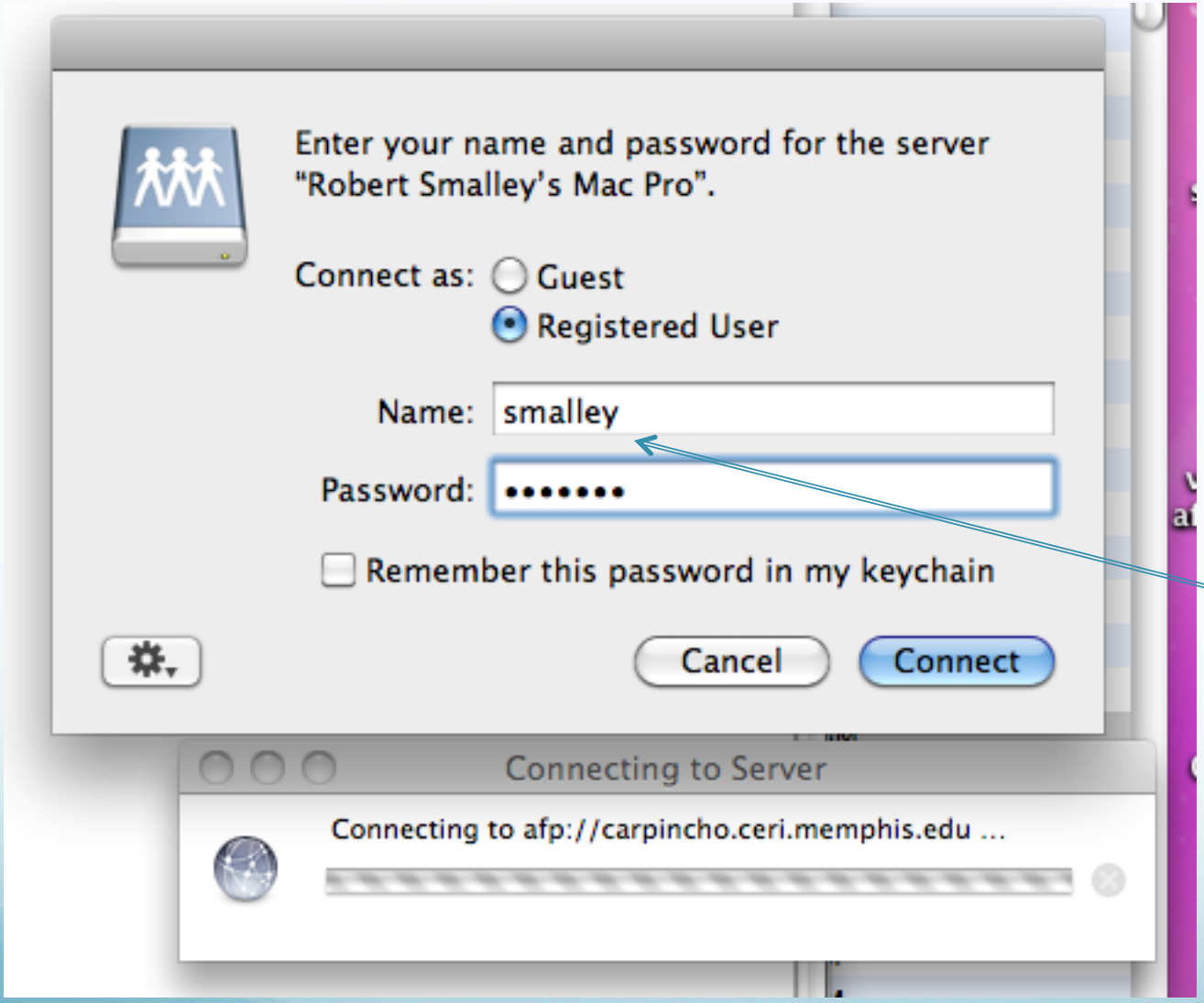
Sharing disks



Now select this instead
(will be highlighted – does not come through on screen capture. When you double click it goes up top. Or type it in up top.)

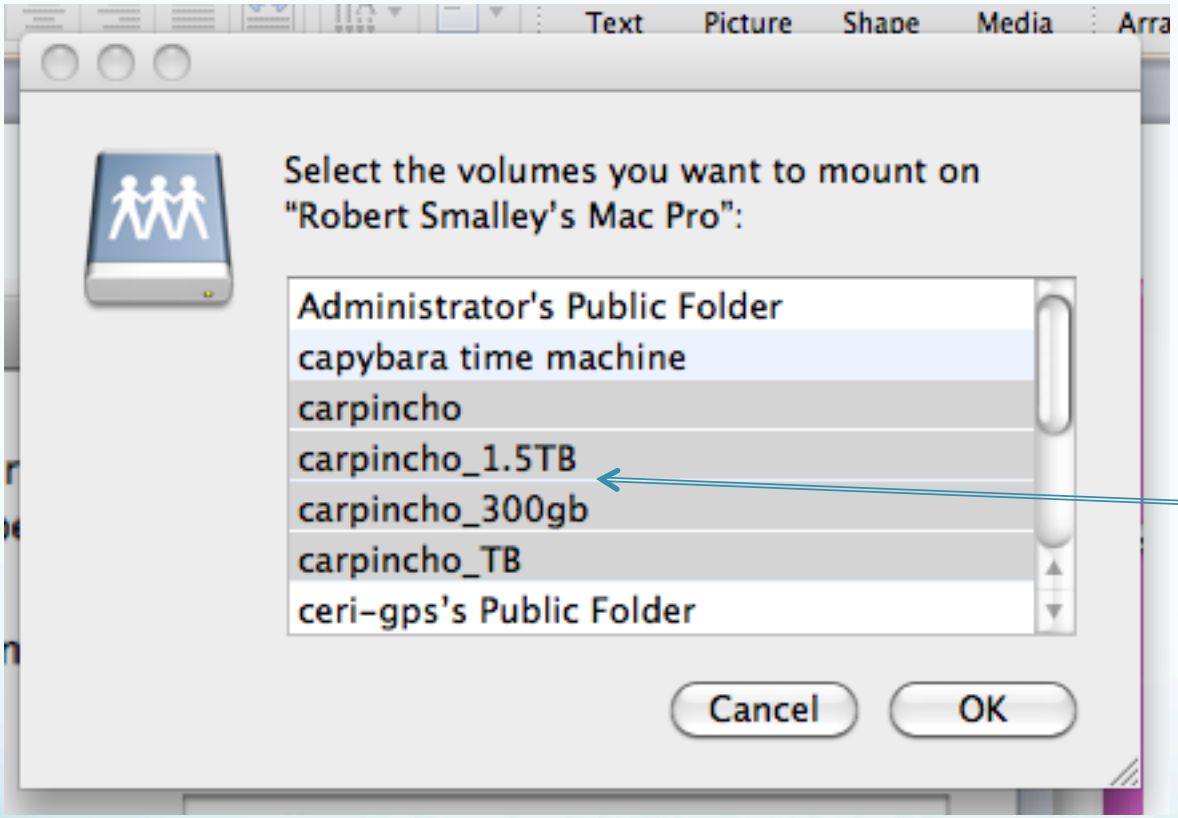
"afp" is apple file protocol – it is how the Mac shares disks with other Macs.

Sharing disks



You have to log in.

Sharing disks



Select the disks you want to mount. (it will be blue)

Sharing disks

They show up on the desktop and can be accessed under /Volumes in the UNIX file structure.

```
-bash 520 ~ # cd /Volumes
-bash 521 Volumes # ls
BOOTCAMP        Macintosh HD    carpincho       carpincho_1.5TB carpincho_300gb carpincho_TB
-bash 522 Volumes #
```

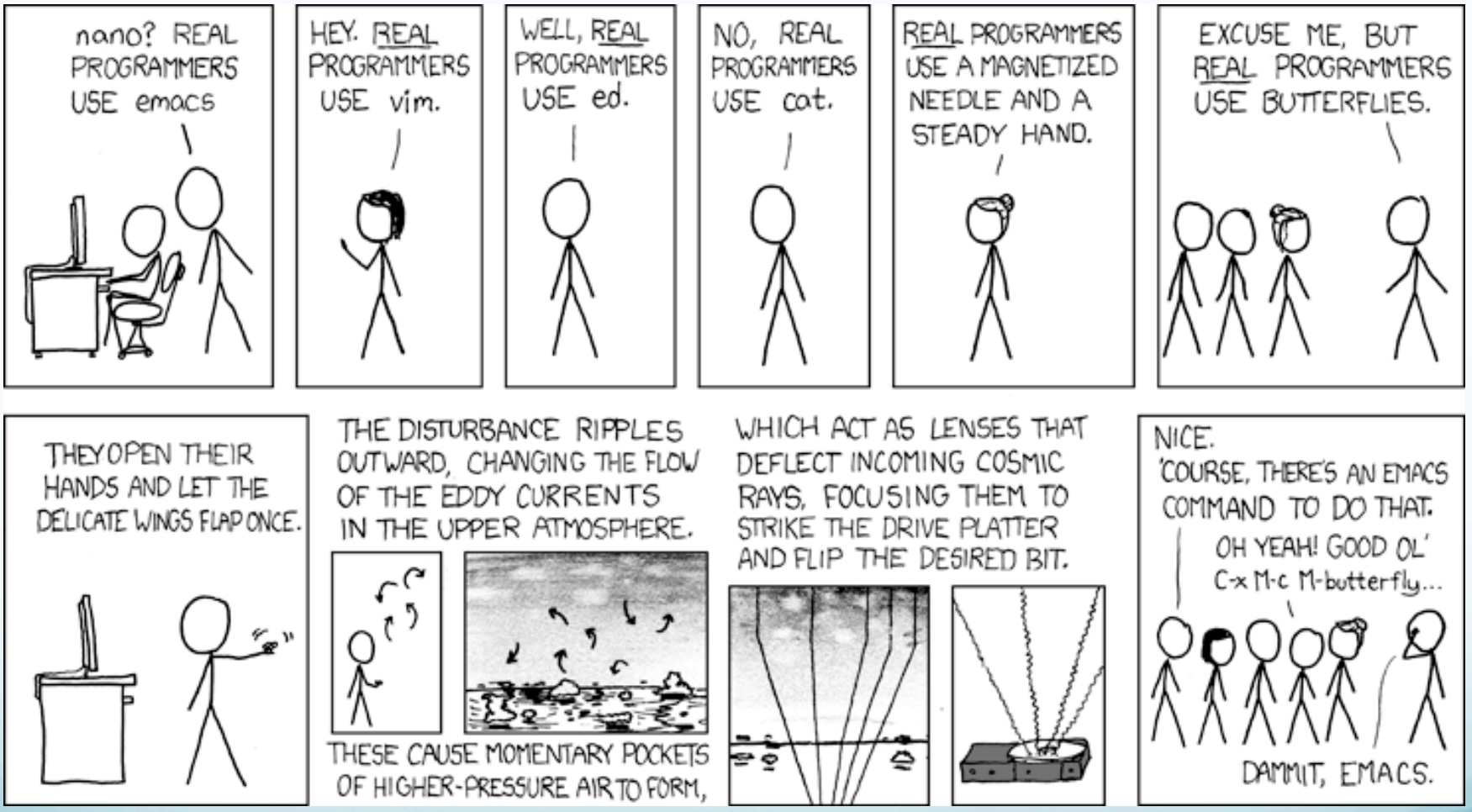
Sharing disks

You can also “mount” disks from the UNIX and PC systems using “samba”

smb://TCP/IP address or name

Text Editing

Basics of the UNIX/Linux Environment


nano? REAL PROGRAMMERS USE emacs
HEY. REAL PROGRAMMERS USE vim.
WELL, REAL PROGRAMMERS USE ed.
NO, REAL PROGRAMMERS USE cat.
REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.
EXCUSE ME, BUT REAL PROGRAMMERS USE BUTTERFLIES.
THEY OPEN THEIR HANDS AND LET THE DELICATE WINGS FLAP ONCE.
THE DISTURBANCE RIPPLES OUTWARD, CHANGING THE FLOW OF THE EDDY CURRENTS IN THE UPPER ATMOSPHERE.
THESE CAUSE MOMENTARY POCKETS OF HIGHER-PRESSURE AIR TO FORM,
WHICH ACT AS LENSES THAT DEFLECT INCOMING COSMIC RAYS, FOCUSING THEM TO STRIKE THE DRIVE PLATTER AND FLIP THE DESIRED BIT.
NICE.
'COURSE, THERE'S AN EMACS COMMAND TO DO THAT.
OH YEAH! GOOD OL' C-x M-c M-butterfly...
DAMMIT, EMACS.

Text Editing Options

Mouse-driven options

nedit: this X-window GUI text editor (that you start from the command line, not by clicking on an icon) allows interactive mouse or keyboard driven text manipulation; colored text and auto-recognition of various standard scripting and programming languages is helpful for debugging scripts and code; appears to be a student favorite at CERI and is available on the Mac and SUN UNIX systems.

Text Editing Options

Mouse-driven options

emacs: a less sleek looking GUI text editor (available at CERI) that allows interactive mouse or keyboard driven text manipulation; it is very powerful and is an old favorite of computer programmers.

(see Steve Brewer if you want to use emacs.)

Text Editing Options

Keyboard-driven options

vi or vim:

vi/vim (vi improved) is a non-GUI text editor that relies primarily on keyboard driven text manipulation; steep learning curve but very powerful;

vim - adds colored text and auto-recognition of various standard scripting and programming languages to vi, helpful for debugging.

vi (& probably vim) found on ALL UNIX systems.

Text Editing Options

Keyboard-driven options

pico: a pared down non-GUI text editor very similar to the email program pine. If you don't know what pine is, use nedit instead.

nedit or vi/vim.

nedit is available on the CERI Mac and SUN UNIX machines because Deshone/Bob/Mitch have installed it.

nedit has a shallow learning curve (execute it and start using! If you need the manual, there is a bug in the program!).

(Also the Mac OS program TextEdit)

nedit or vi/vim.

vi (and typically vim) is available as standard on all UNIX and UNIX-like systems.

vi and vim are hard to learn.

vi and vim are much more powerful (i.e. harder) than nedit.

(vi is aliased to vim on the Mac)

*note to OSX users not on the CERI Student Lab machines, nedit can be downloaded and installed on OSX but you need to be sys admin and know what you are doing....it is not a simple dmg unpack. Xcode is a similar but more powerful editor for code development.

to start nedit

```
%nedit &
```

Which shows another UNIX feature we have mentioned before – the optional "&".

When the "&" is placed at the end of a command line it opens the program in the background so that you can continue to use the terminal window.

&

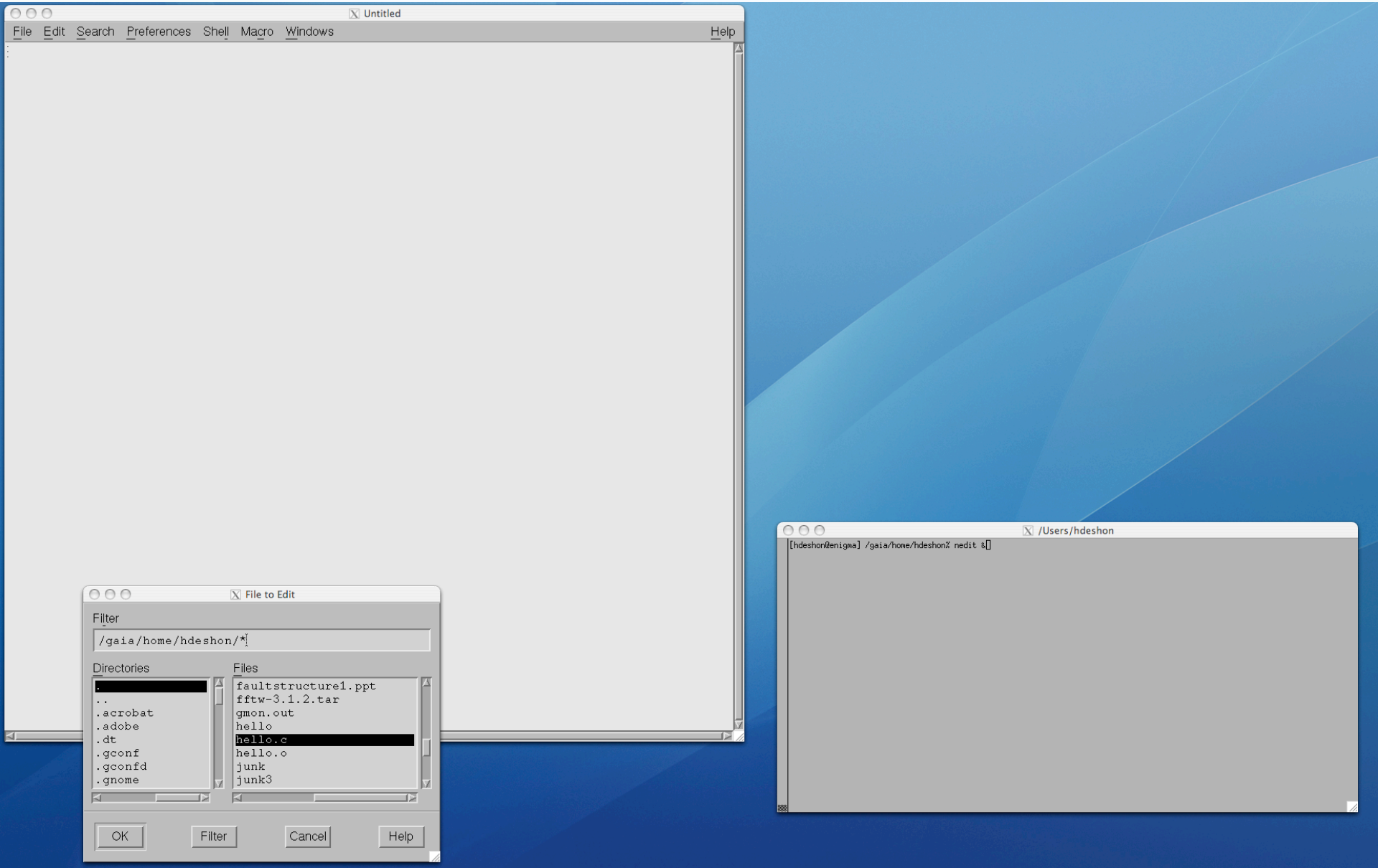
This is a general feature.

So if you have a program that will take 10 minutes to run and is putting its output into a file (not the screen) and it does not need interactive input, you can run it with the & at the end and it will go off and do its thing in the "background" and you can continue working in the window.

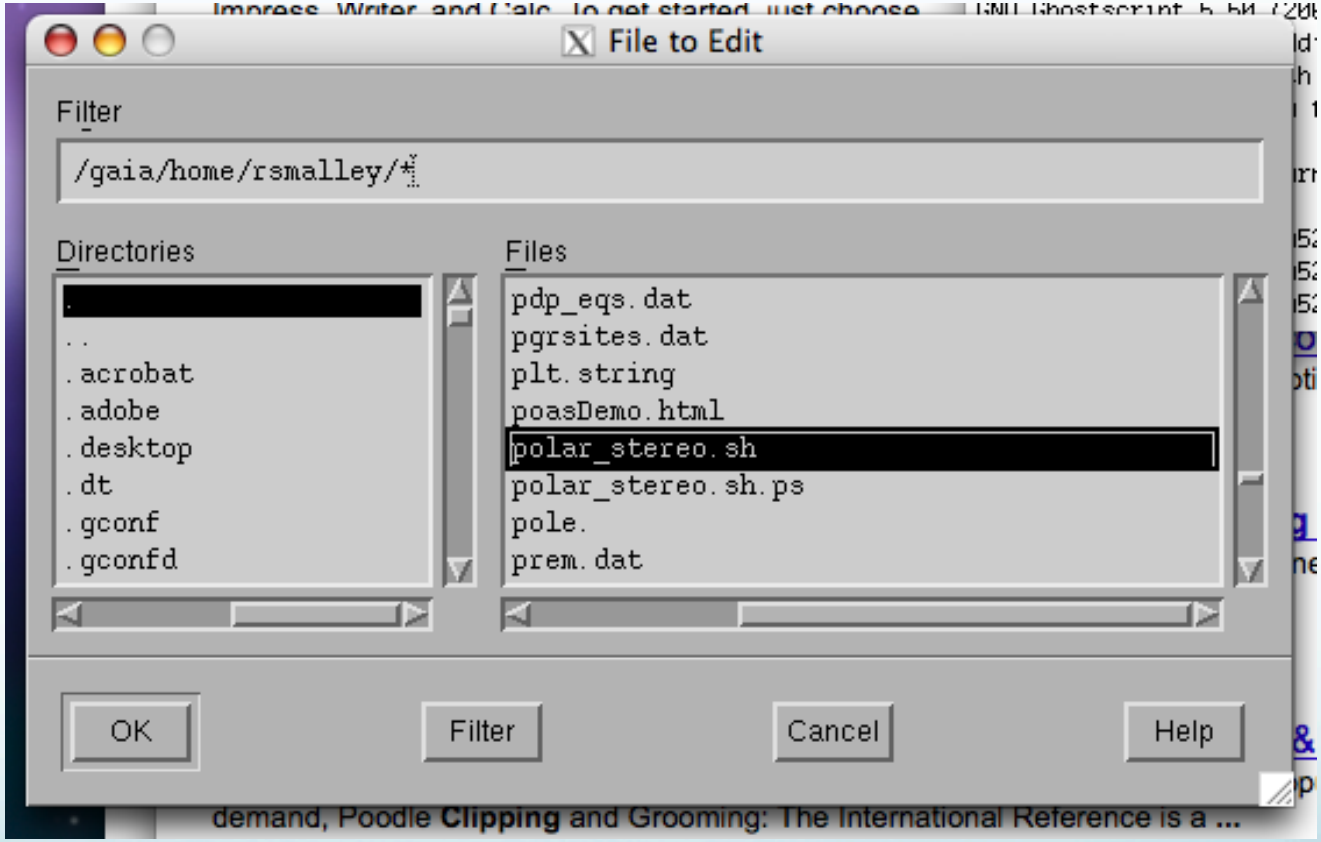
This was a much more important before the days of window based GUIs.
Now just open more windows and move between them.

This is what it looks like

(using a mac that is ssh'd into the suns)



Works similar to WORD. File/open – get dialog box. Select file to open.



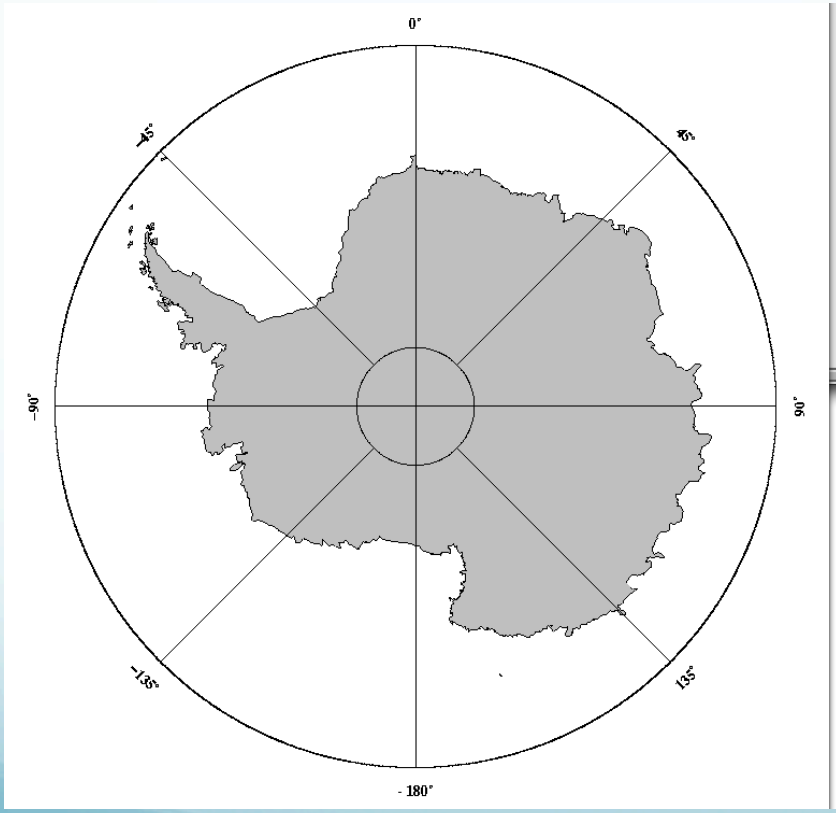
This is the file. It is a shell script (bourne shell – sh). It makes a map using the GMT package.

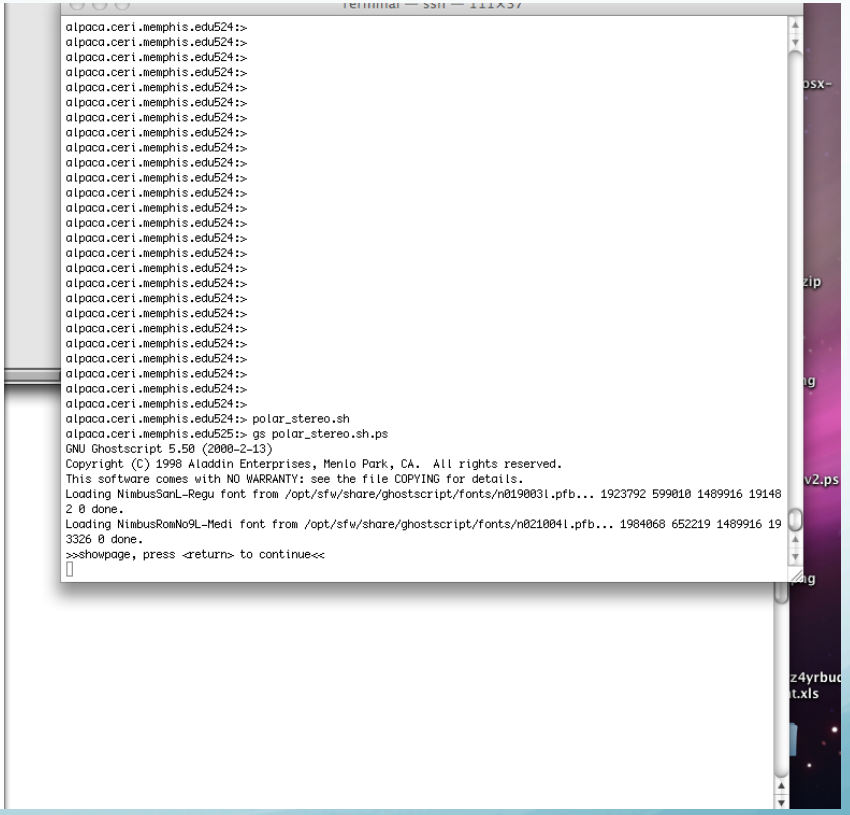
polar_stereo.sh – /gaia/home/rsmalley/

File Edit Search Preferences Shell Macro Windows Help

```
#!/bin/sh
#Js gets long and lat of center (0/-90) then scale as radius/lat (3.5 inches and oblique lat)
pscoast -R-180/180/-90/-60 -Js0/-90/3.5i/-60 -Ba45g45/45g45 -Dl -A250 -G192 -W1/0 -P > $0.ps
```

Here's what you get when you 1st run it, and 2nd display it (two steps).





But first – a little more about files on UNIX.

Files on UNIX are “flat”

Just strings of bytes with the information contained in the file.

What do we mean by this?

The files do not have headers or tailers with metadata about the file, icons, etc.

UNIX does not provide

<u>Indexed</u>
or <u>relational database</u>

files.
(but you can write a program to provide them! Oh the power of UNIX.).

To UNIX

EVERYTHING is a file, which is a string of bytes.

All equal.

Command line editors

ed

Don't even think of using ed.
(it is a "line" editor, edits one line with cryptic commands.)
(except you will use it without realizing when you use vi!)

If you accidently type it, enter ^D to get out.

Command line editors

<u>edit</u>

Don't even think of using <u>edit</u>.

If you accidently type it, enter "exit" to get out.
(SUN only, does not exist on Mac)

Command line editors

sed

sed:

powerful command line text editor.
("the ultimate" stream editor, non-interactive).

It takes standard in, edits each line, and spits it to standard out.

It uses regular expressions for pattern matches.

Very powerful (i.e. hard to use).

sed:

sed has several commands, but most people only learn the substitute command: **s**.

The substitute command changes all occurrences of the regular expression into a new value.

A simple example is changing "day" in the "old" file to "night" in the "new" file:

```
%sed s/day/night/ <old > new
```

You don't see anything

sed:

```
%echo day | sed s/day/night/
night
```

It does what you tell it.
(here you send edited file to screen, so you see it but don't actually have it saved.)

```
%echo Sunday | sed 's/day/night/'
Sunnight
```

sed:

4 parts to substitute command

`s` Substitute command

`/../../` Delimiter

`day` - Regular Expression Pattern Search Pattern

`night` - Replacement string

sed:

Most examples of sed are incomprehensible (heavy use of regular expressions [will do regular expressions soon] plus sed only expressions)

```
sed 's/[^ ]*/(&)/' < old > new
sed 's/[^ ][^ ]*/(&)/g' < old > new
sed 's/^\([^:]*\):[^:]:/\1::/' </etc/passwd >/etc/password.new
```

count the number of lines in the three files `f1 f2 f3` that don't begin with a "#"

```
sed 's/^#.*//' f1 f2 f3 | grep -v '^$' | wc -l
```

sed:

Is very useful when you really need it.

vi and vim

(uses the same command set as ed/edit/sed! This is Unix, reuse the same tools.)

to start it up
(on Student Mac Lab machines vi is aliased to vim)

```
%vim [name-of-file]
```

vi and vim are have what is called a “modal” interface.

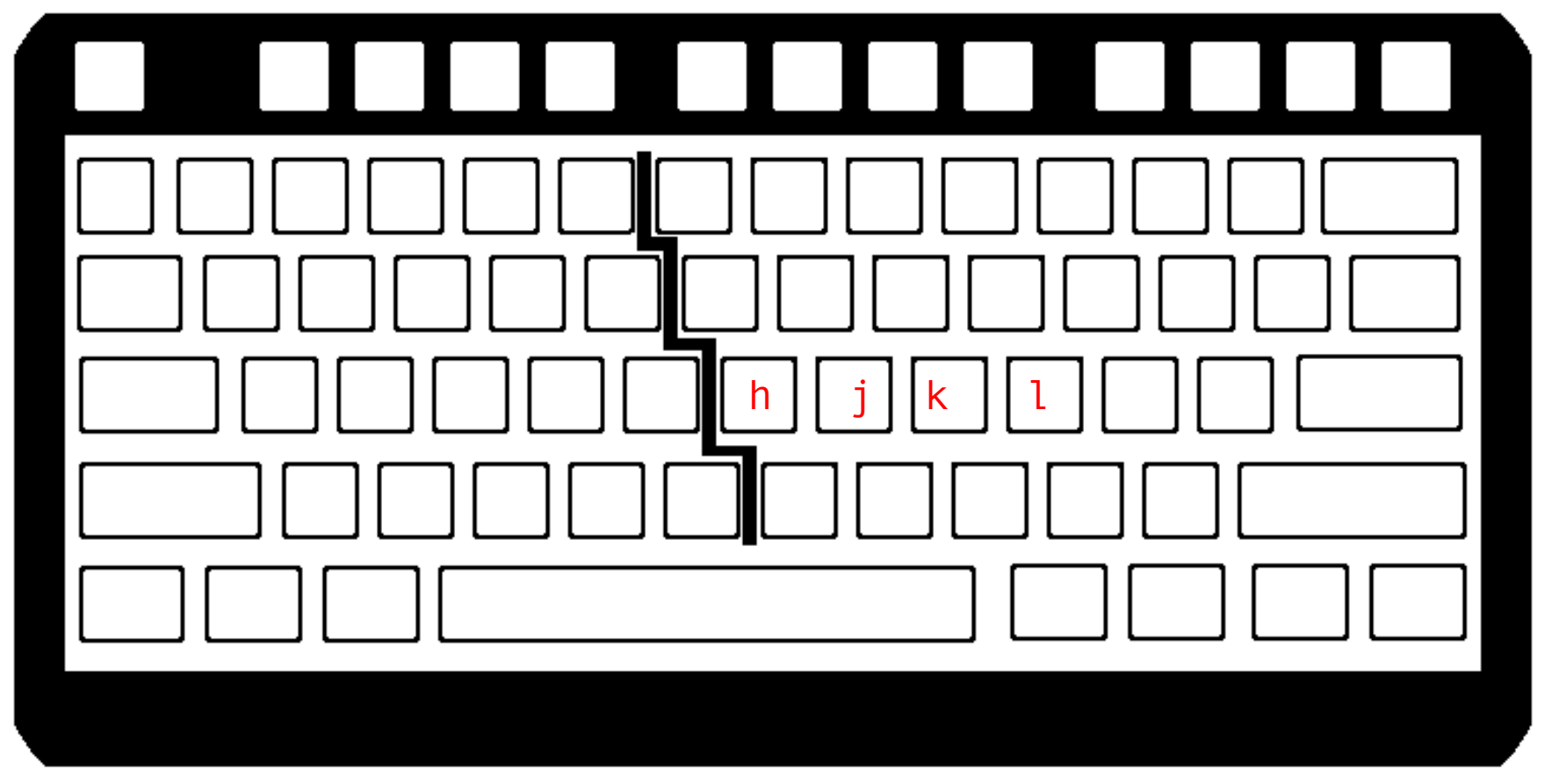
They have two modes

“normal” = command mode
insert = input mode

Entering text takes place in insert mode and the editing power comes to the fore in command mode.

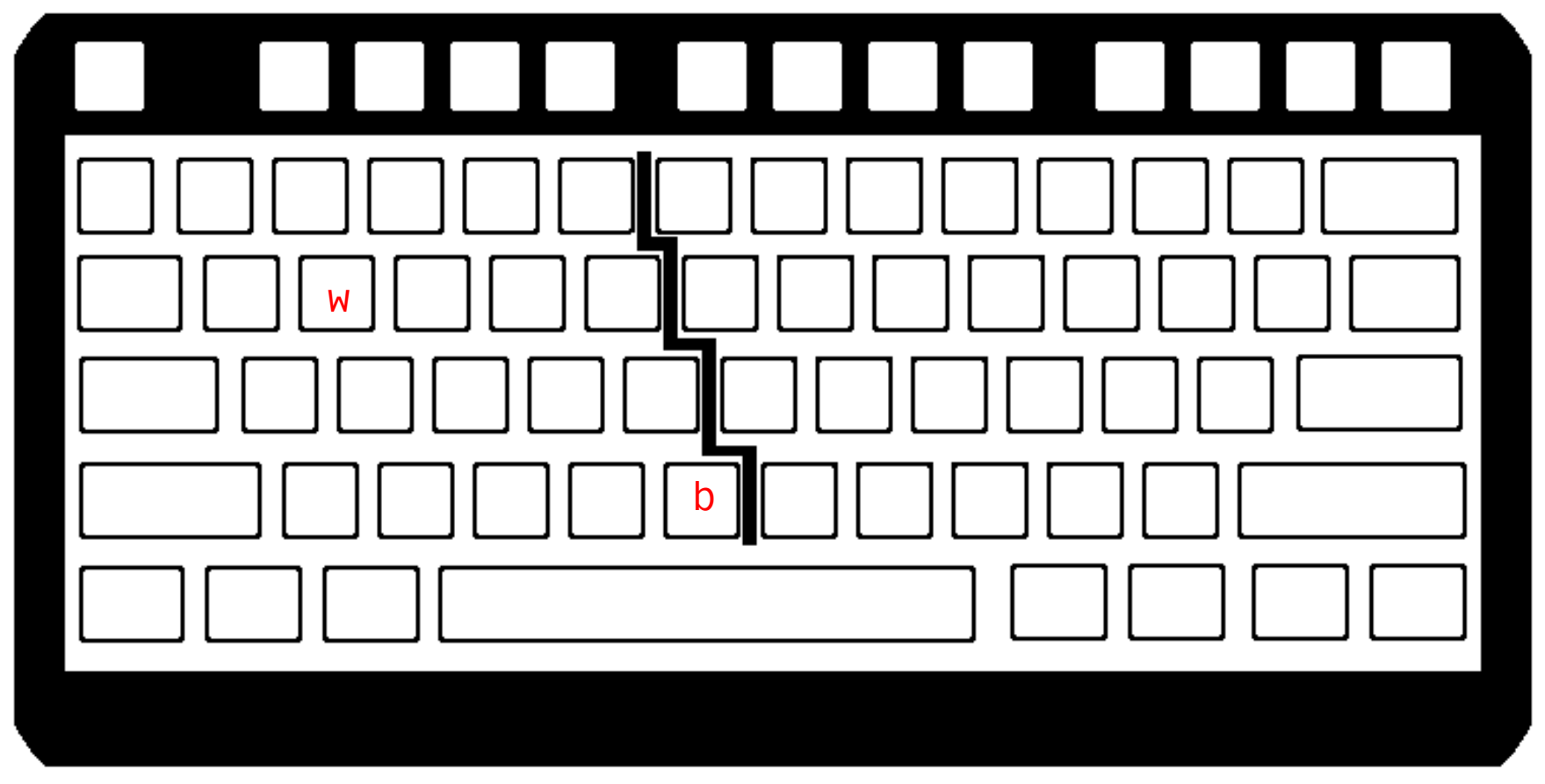
Use “esc” (escape) to return to command mode from insert mode.

vi/vim - moving the cursor in command mode



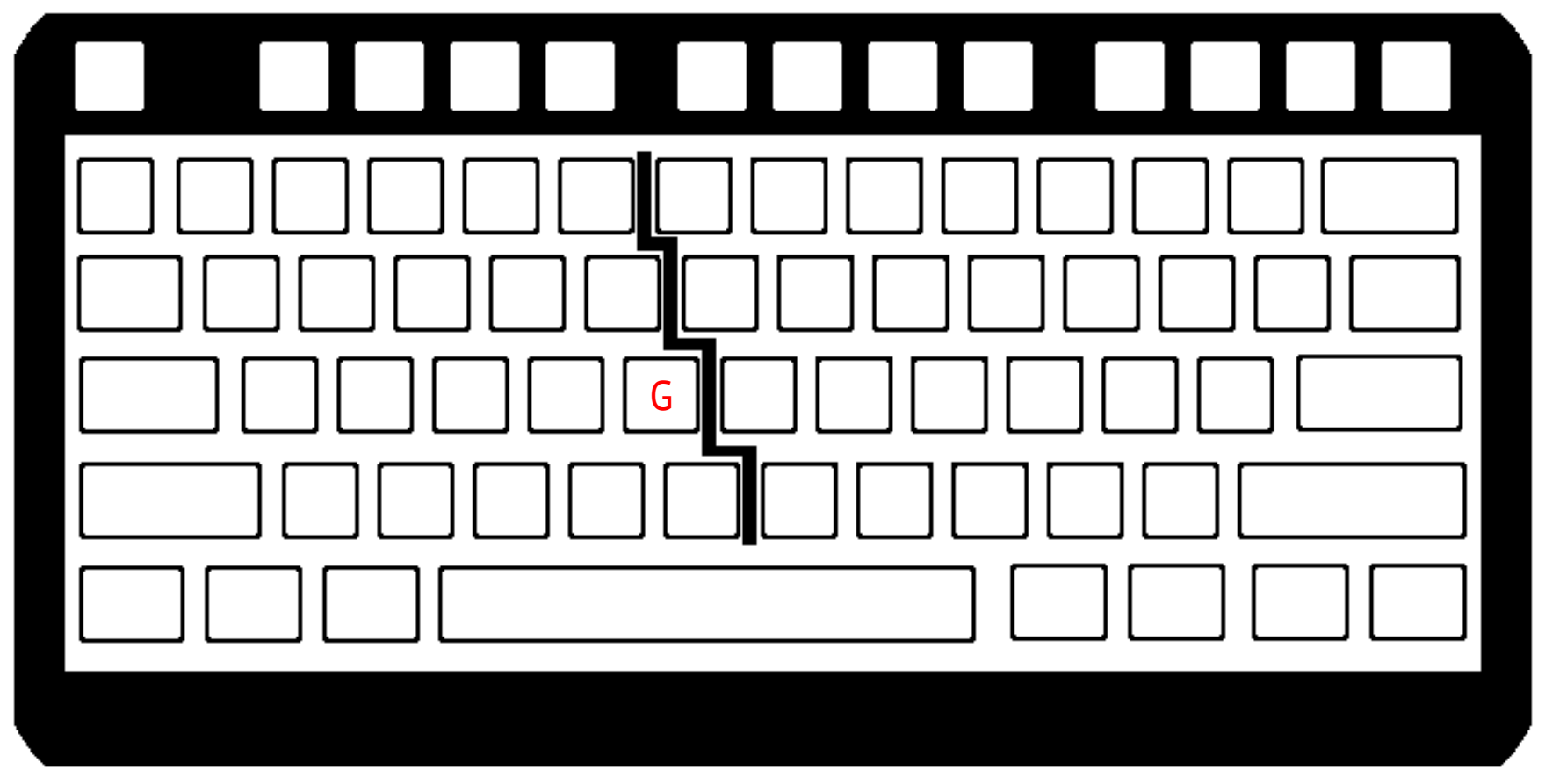
h (or backspace or left arrow) – cursor left
j (or return or down arrow) – cursor down
k (or up arrow) – cursor up
l (or space or right arrow) – cursor right

vi/vim - moving the cursor in command mode



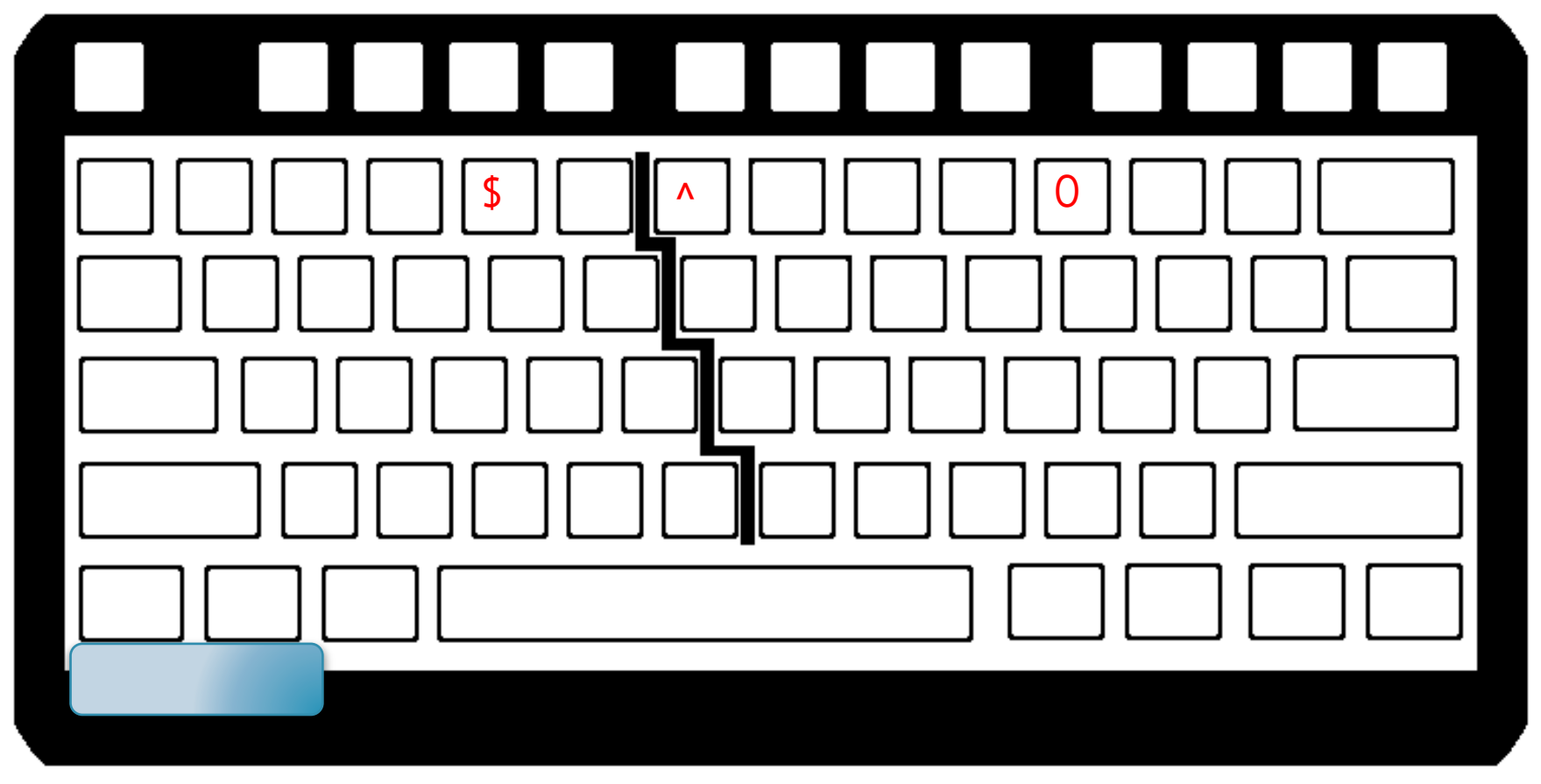
w – beginning next word
b – beginning preceeding word

vi/vim - moving the cursor in command mode



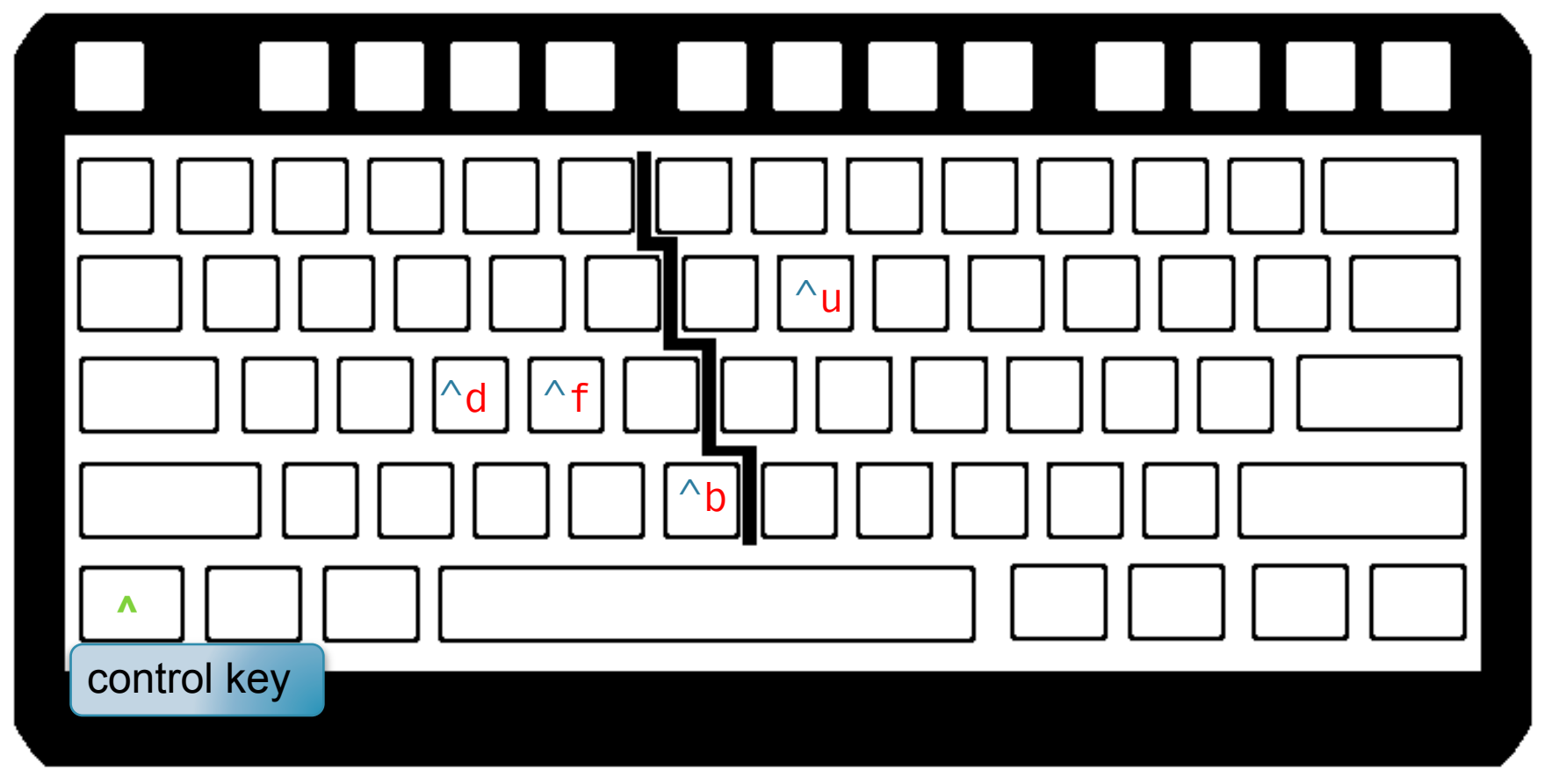
G – move cursor to beginning of last line
nG – move cursor to beginning of line n

vi/vim - moving the cursor in command mode



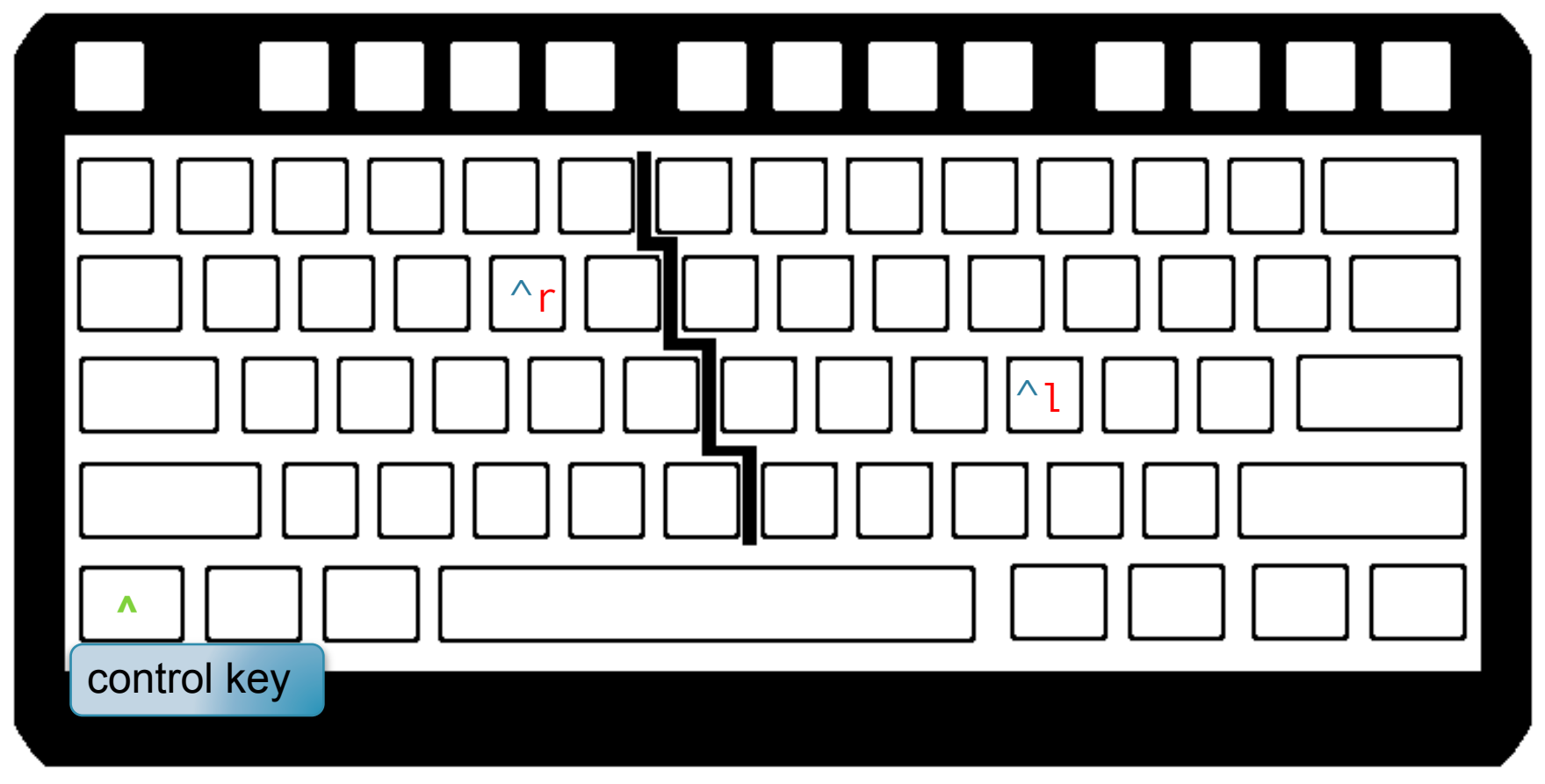

```
$ -- go to end of line (eol)
0 -- go to beginning of line (bol)
^ -- go to first character at bol
```

vi/vim - moving the cursor in command mode

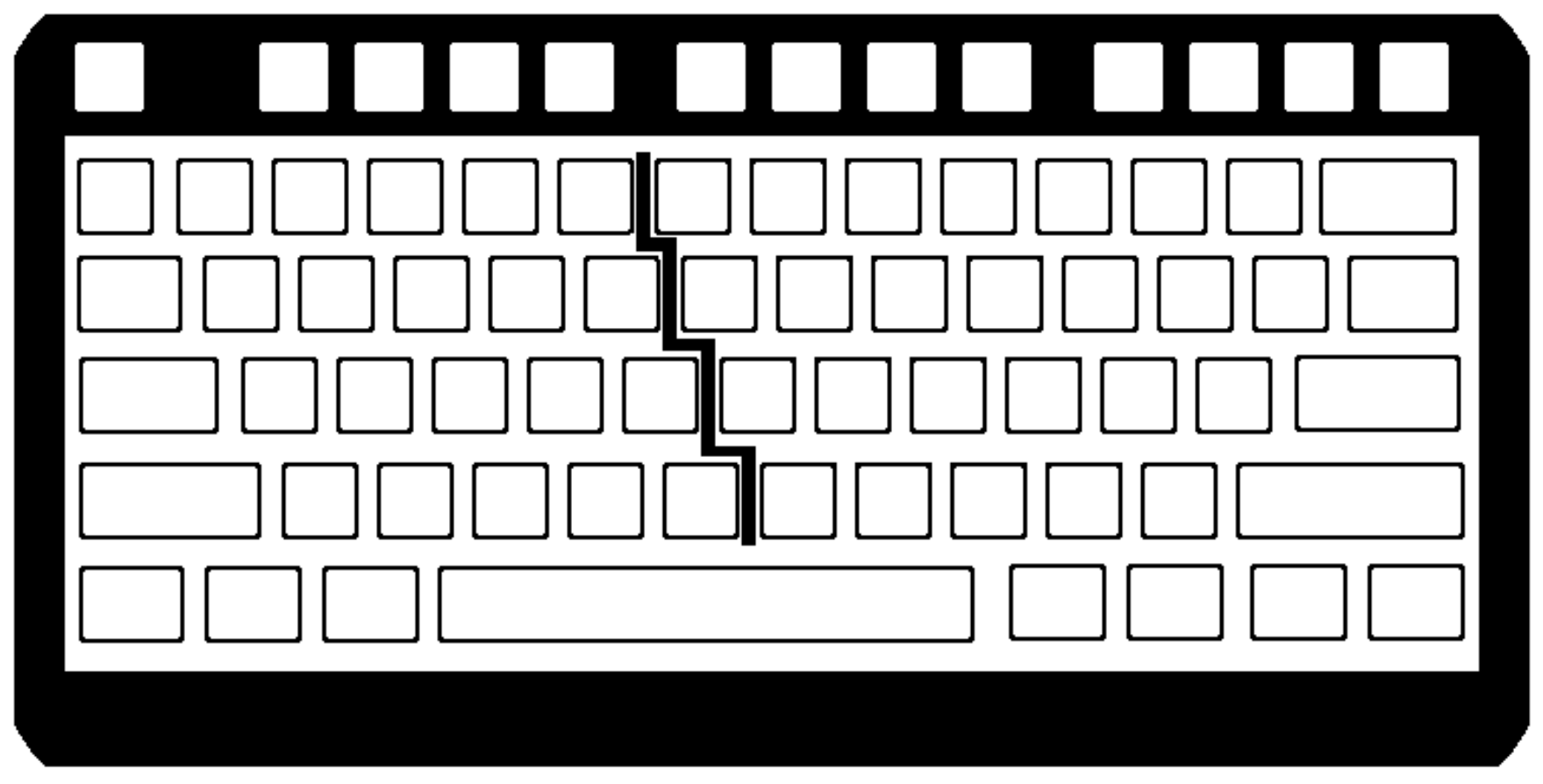



```
^f -- scroll screen forward one screen
^b -- scroll screen backwards one screen
^d -- scroll screen forward one half screen
^u -- scroll screen backwards one half screen
```

vi/vim - moving the cursor in command mode




```
^l -- redraw screen
^r -- redraw screen removing deleted lines
```

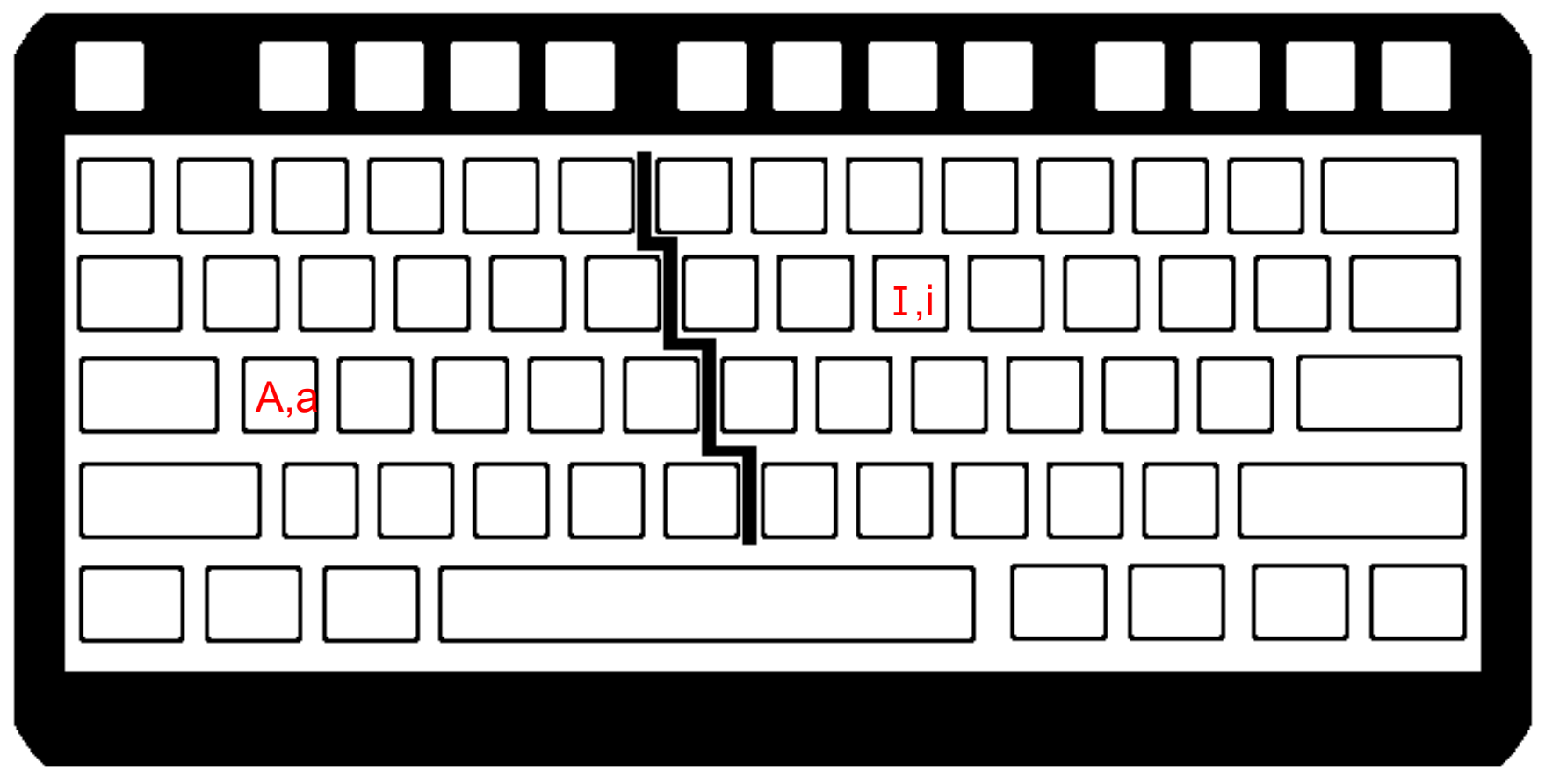


Don't use the arrow keys

(even though you can – unless you are not on a teletype that does not have them – no arrow keys on keyboard above) .

(they are much slower as you have to take you right hand off the keyboard.)

to enter insert mode from command mode



i -- insert

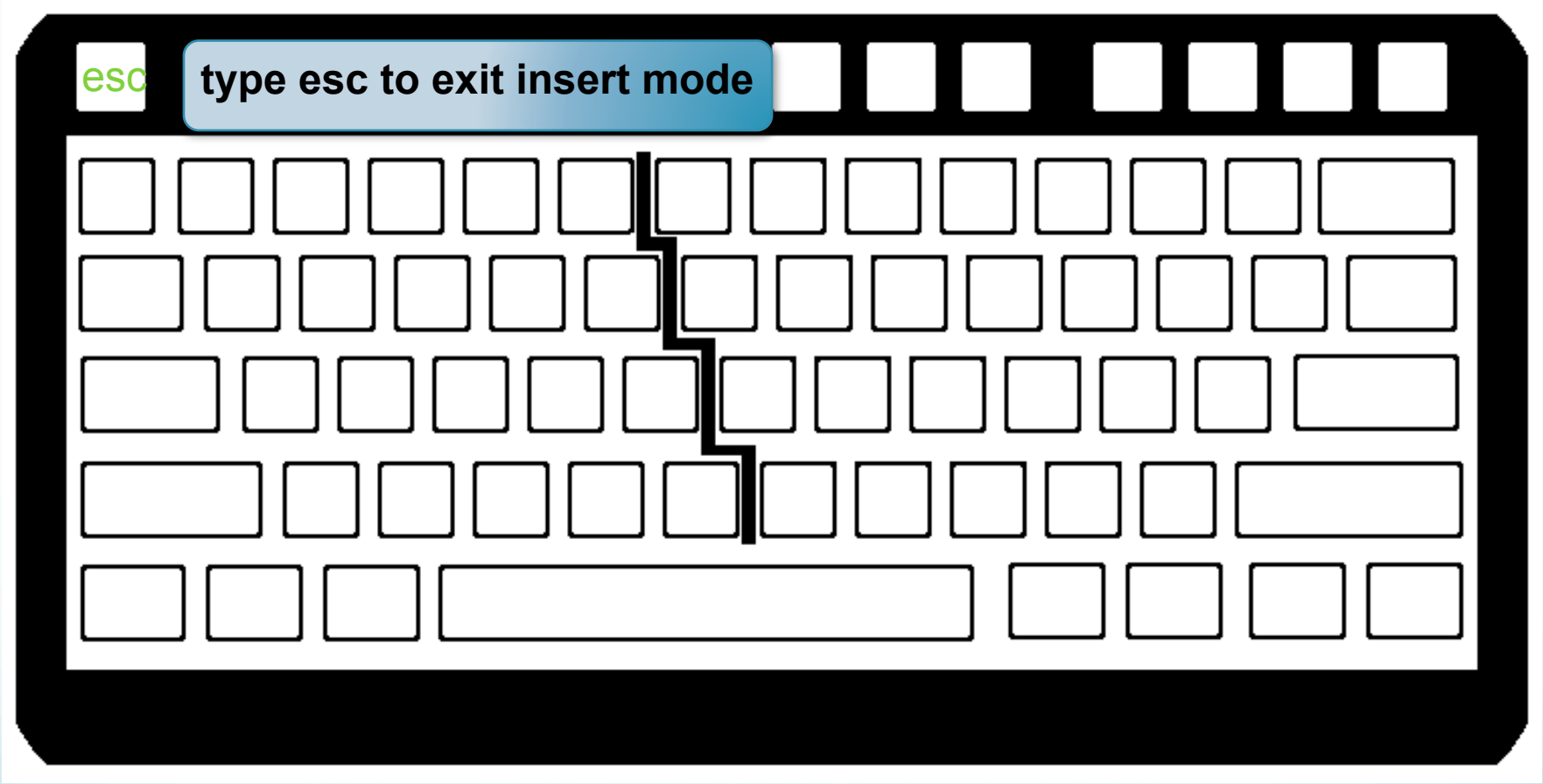
a – append

A -- append at eol

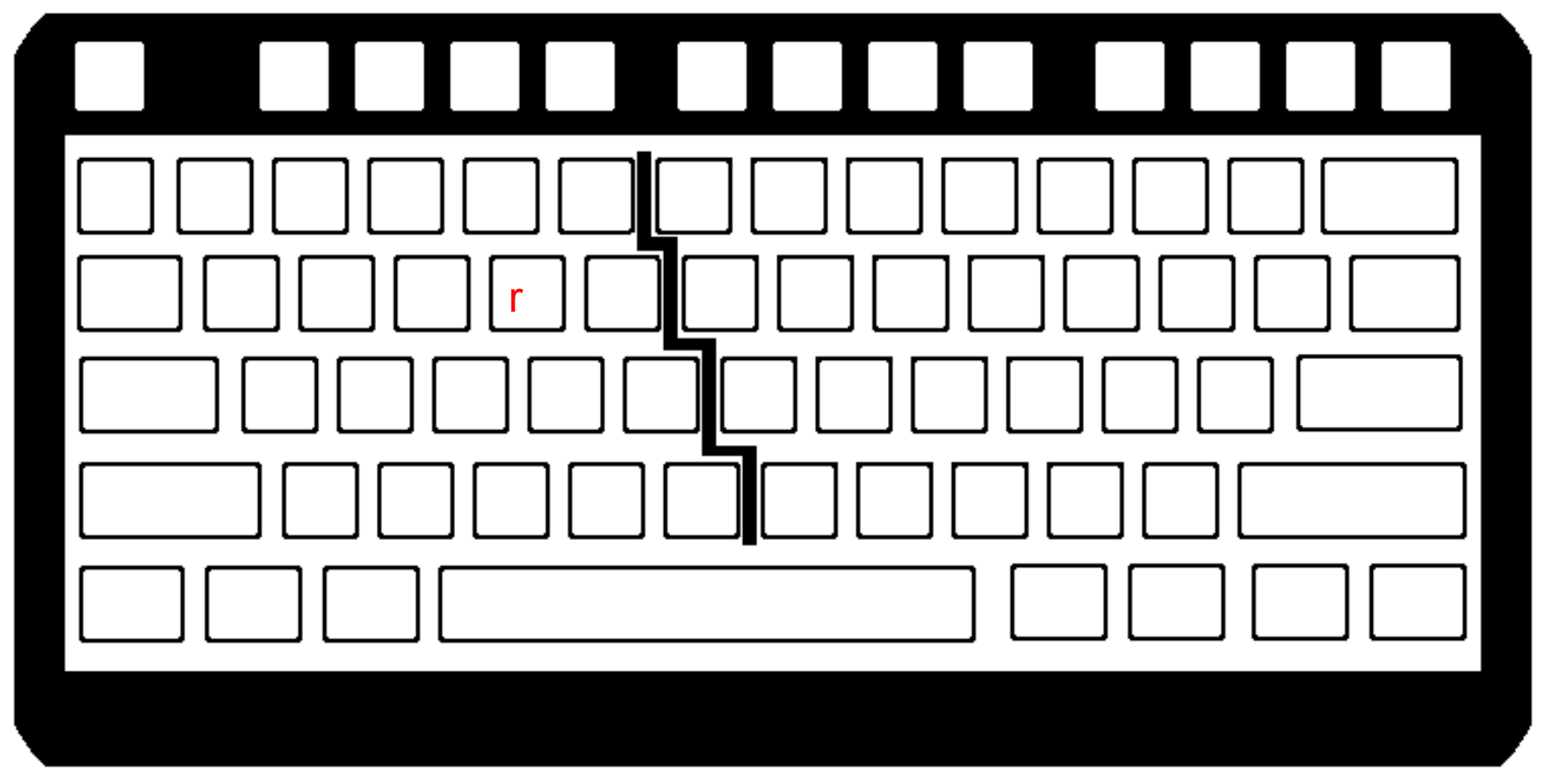
I -- insert at bol

o – start new line and enter insert mode

esc (escape) to exit insert mode and return to command mode.

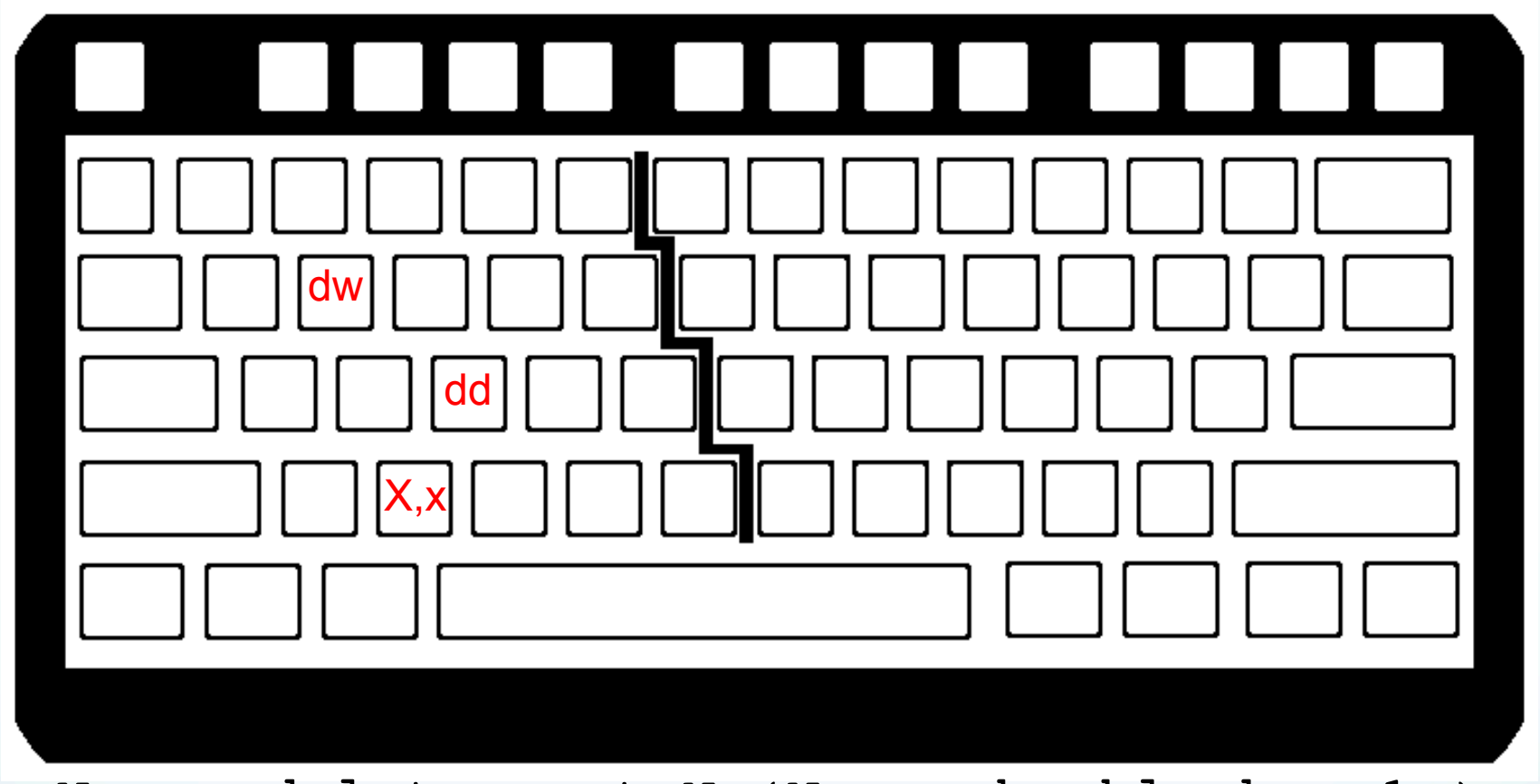


to substitute a single letter from command mode



r – replace (overwrite) character under cursor
R – replaces (overwrites) characters until you exit (esc)

deleting text from command mode



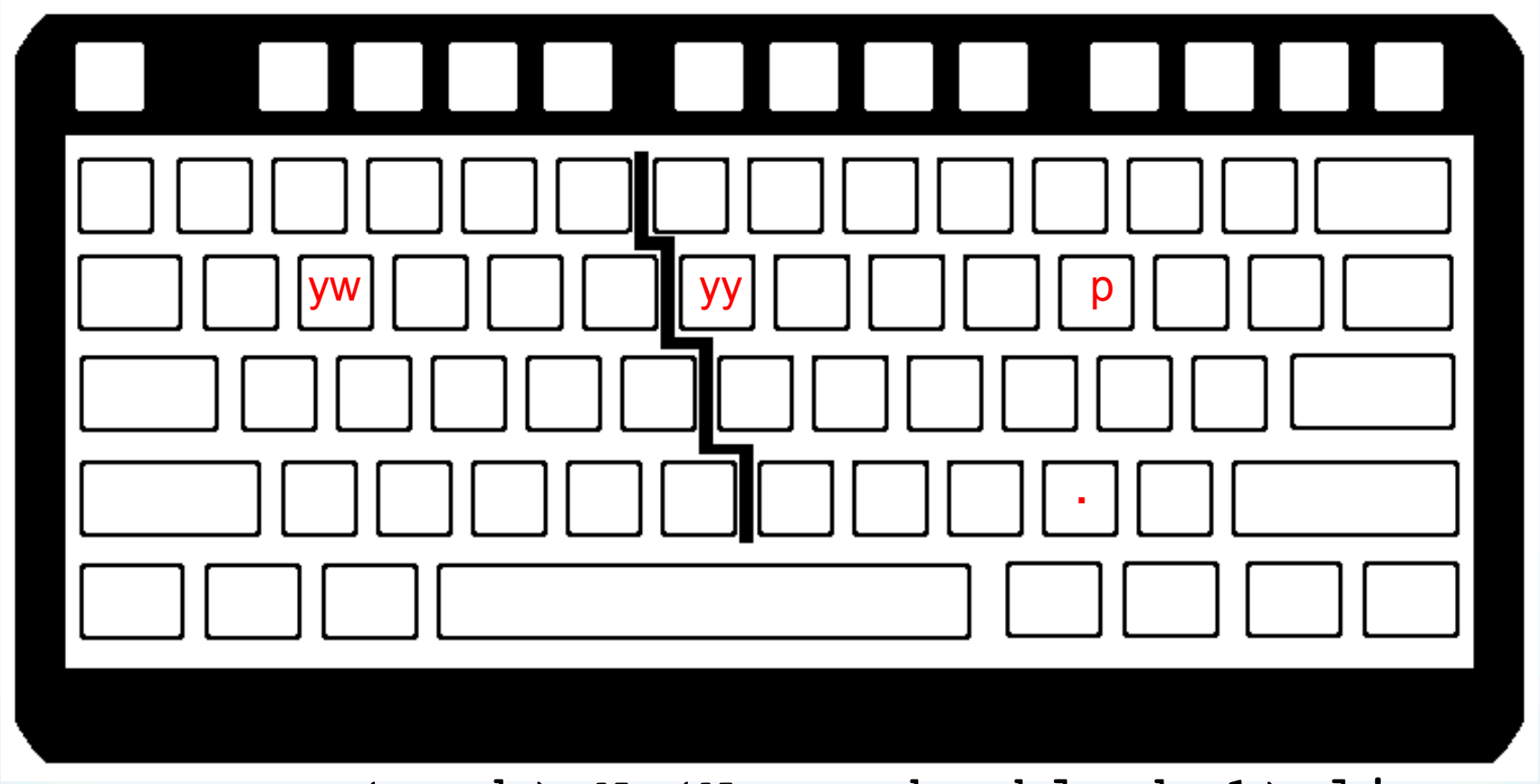
Nx -- delete next N (N can be blank = 1) characters behind cursor

NX -- delete next N (N can be blank = 1) characters in front of cursor

Ndw -- delete next N (N can be blank = 1)words

Ndd -- delete next N (N can be blank = 1)lines

copy, paste, repeat from command mode



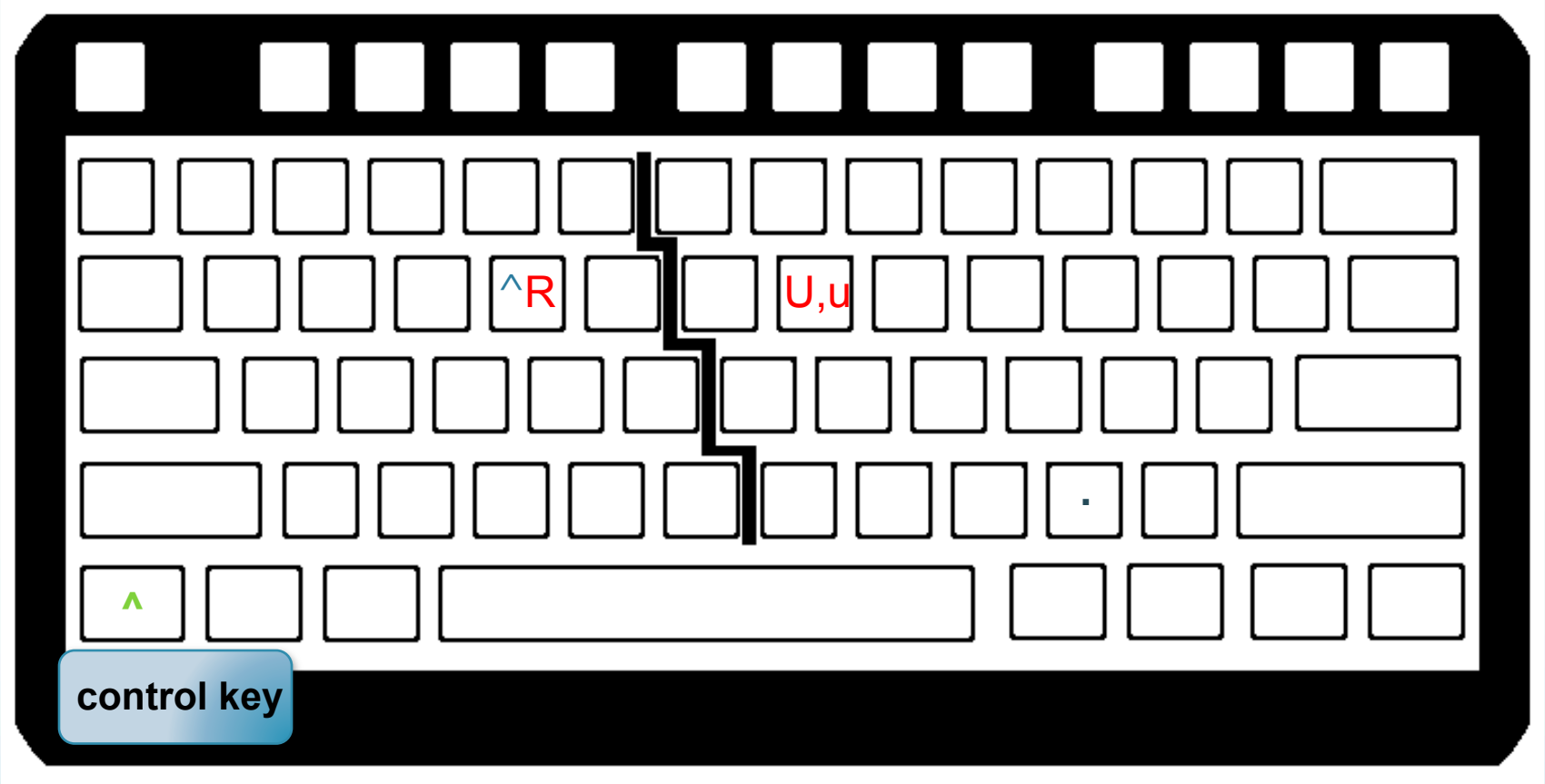
Nyy -- copy (yank) N (N can be blank=1) lines to the “clipboard” (does not remove/erase them)

Nyw – copy (yank) N (N can be blank=1) words

p – paste from the clipboard after the cursor

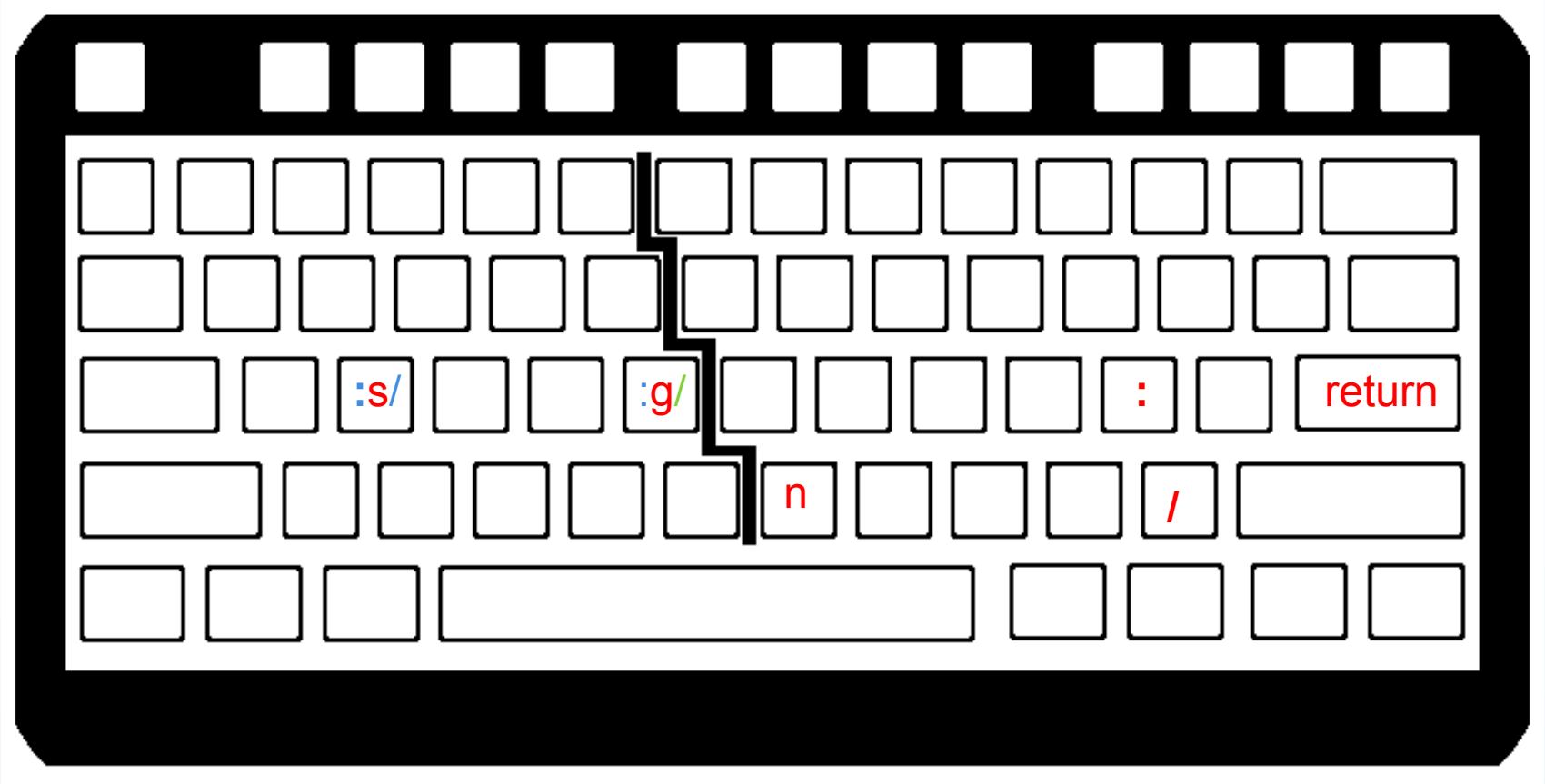
. -- repeat last command

undo and redo from command mode




```
u -- undo last change
U -- undo all changes to the line
^R -- redo change
```

Search from command mode

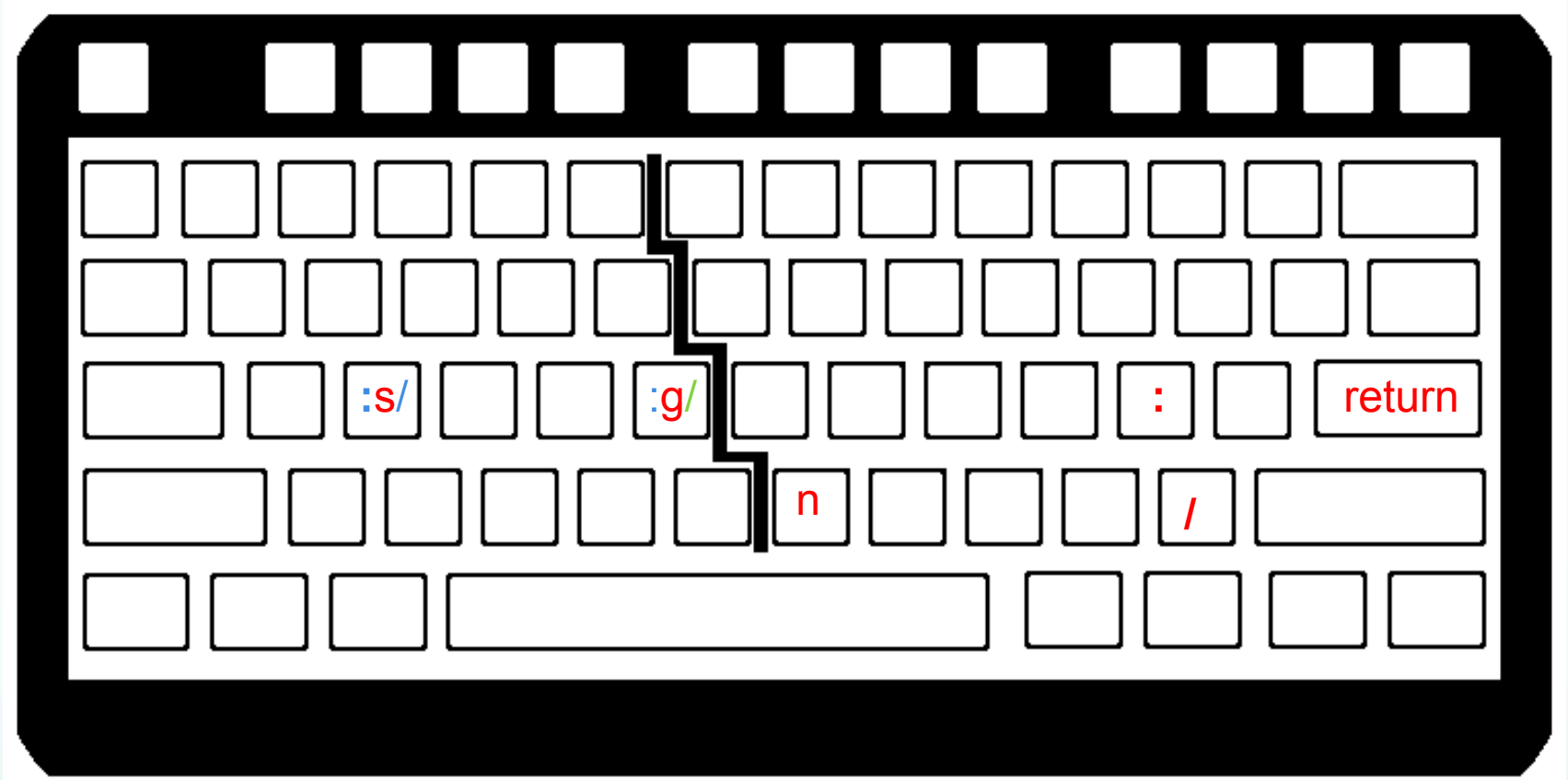


/[word(s)] <CR>-- search for the next instance of the word or words

Uses regular expressions for the pattern matching.

n<CR> -- go to next instance of word or words

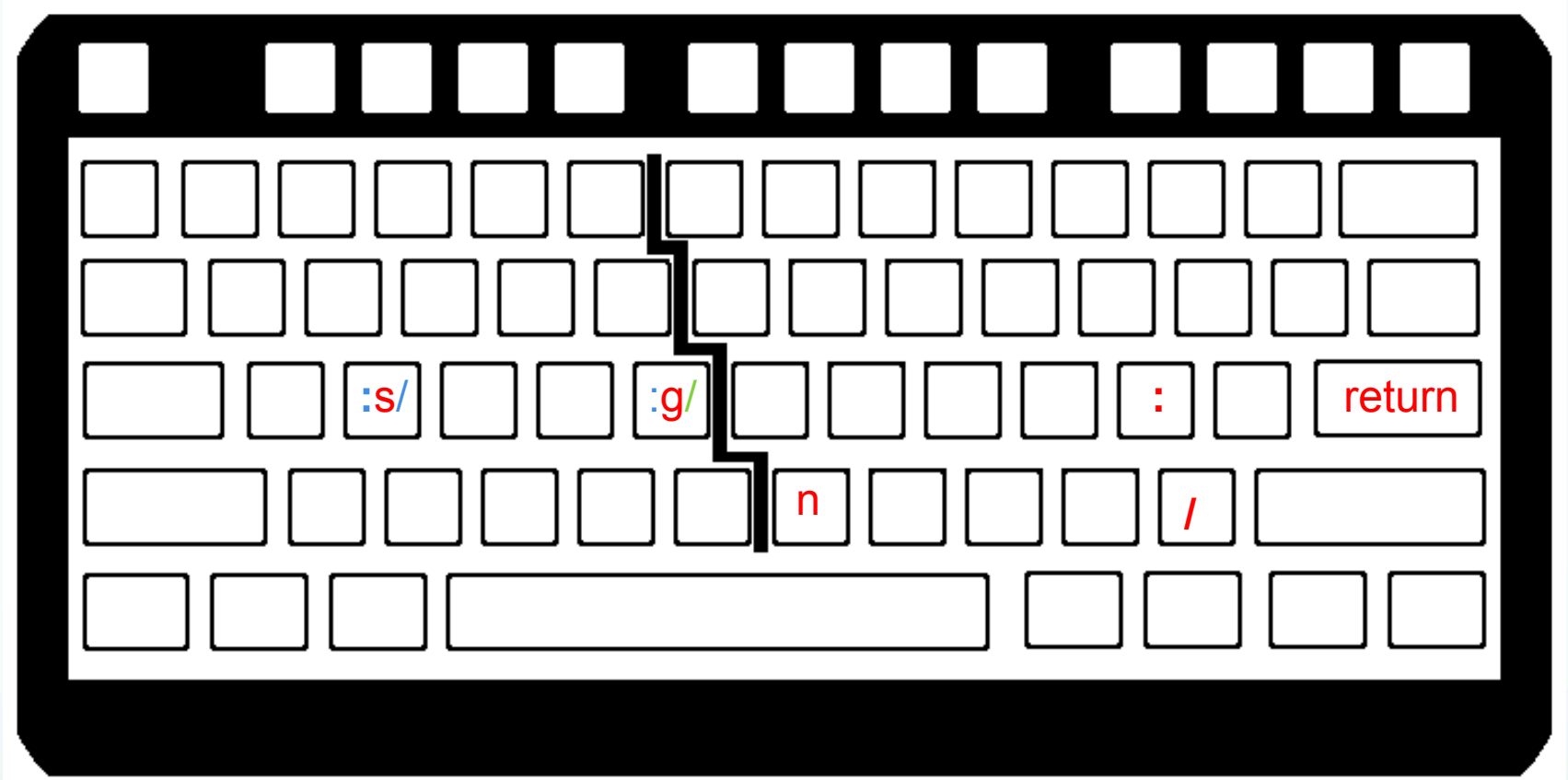
search and replace from command mode



:s/[old]/[new]/[g] <CR> -- substitute old string with new string; does only first instance on line or add optional final "g" for <u>globally on line.</u>

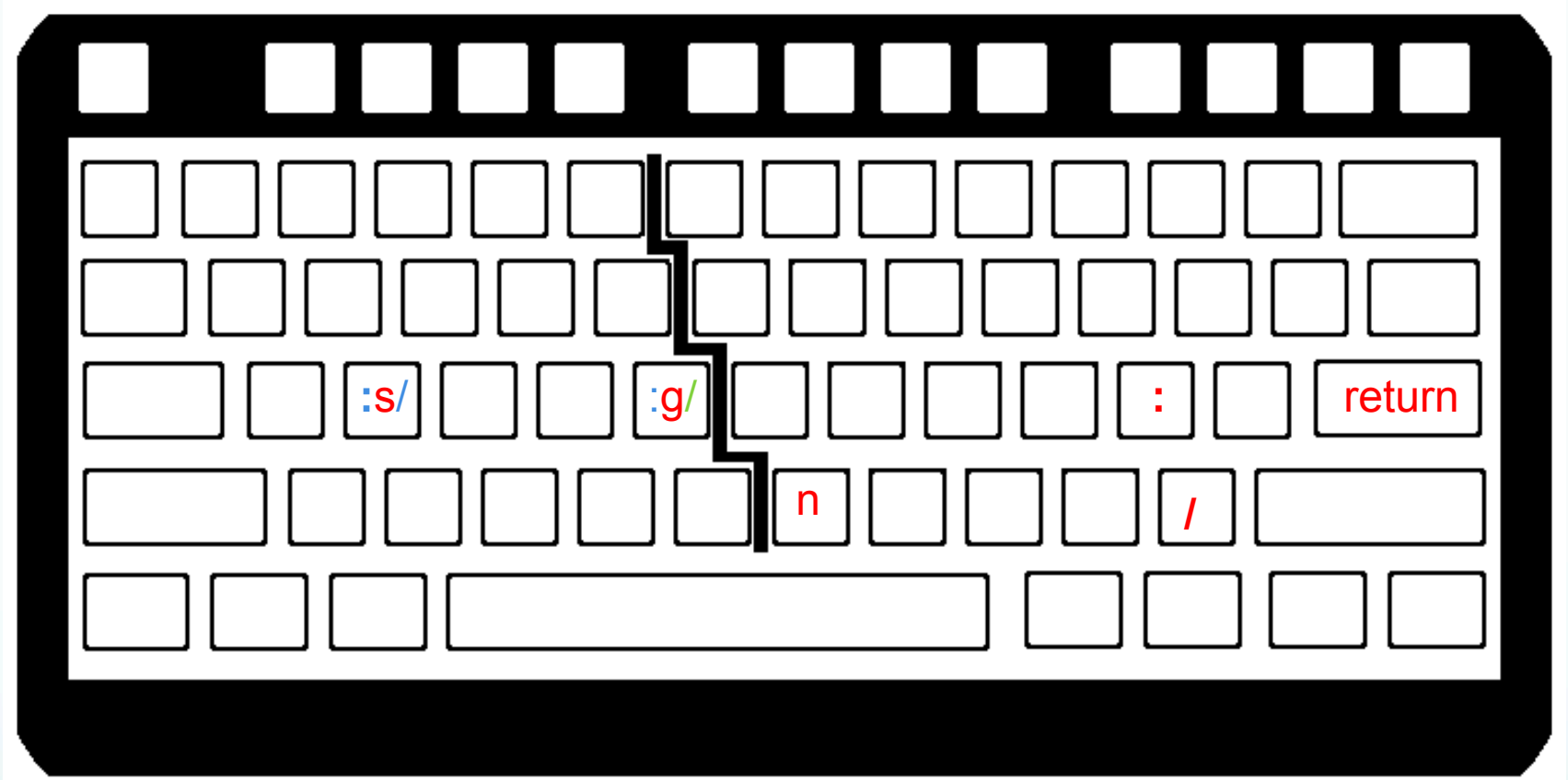
Uses regular expressions for the pattern matching.

search and replace from command mode



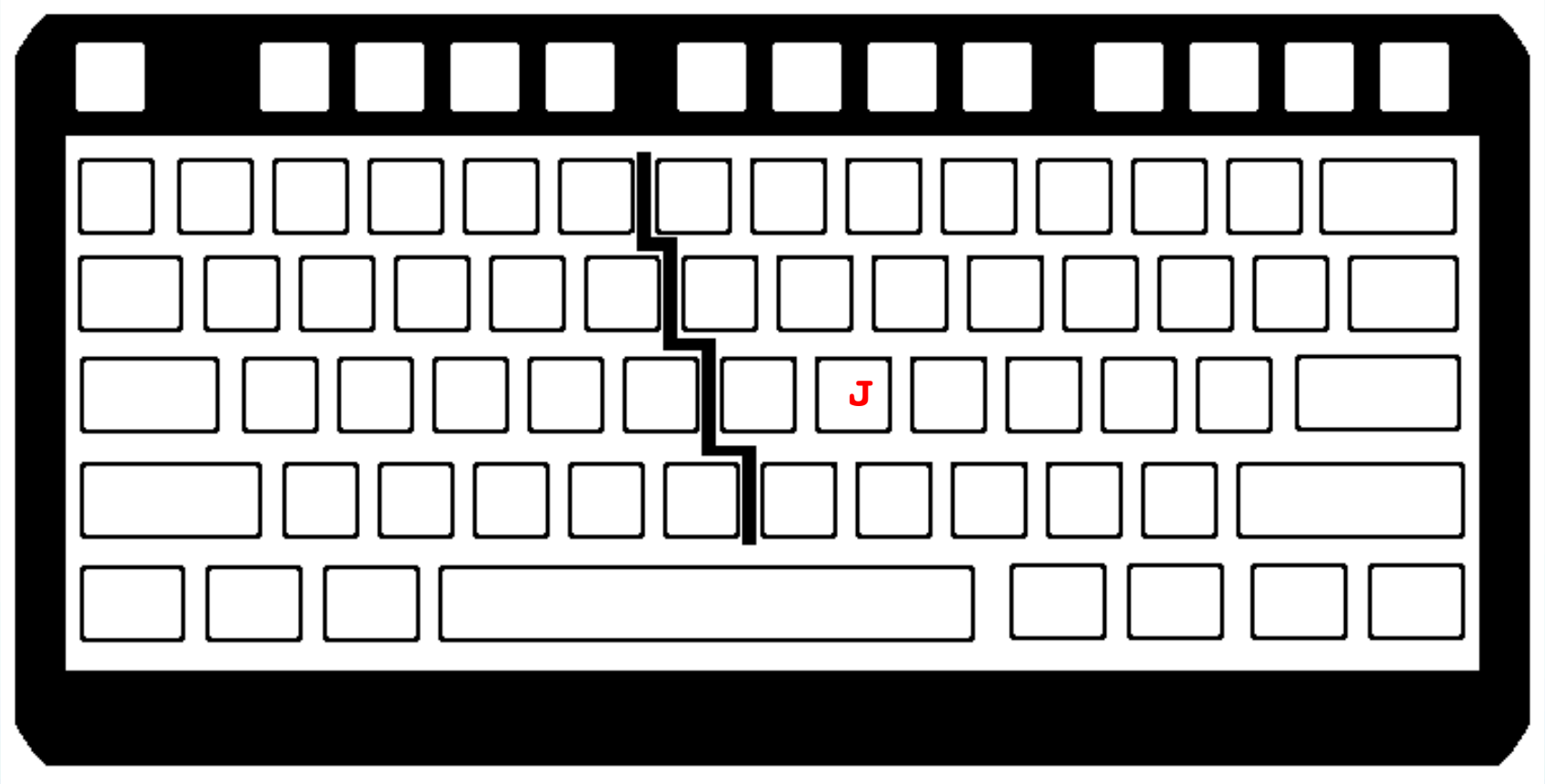
:gs/[old]/[new]/[g]<CR> or :%s/[old]/[new]/[g]<CR> - substitute old string with new string <u>on every line</u>; does only first instance on each line or add optional final "g" for <u>globally on line.</u> Uses regular expressions for the pattern matching.

search and replace from command mode



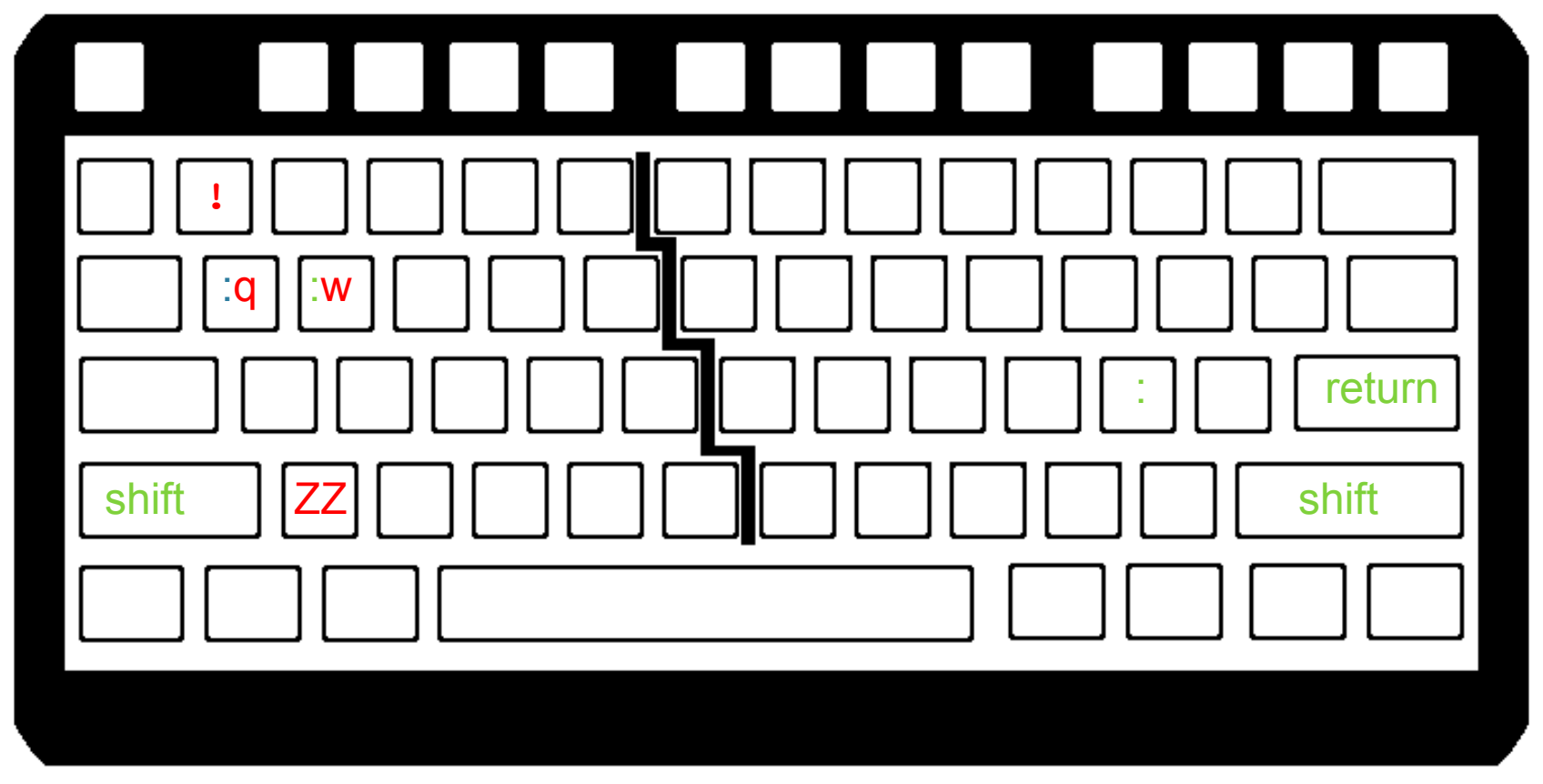
:g/[key]/s/[old]/[new]/[g]<CR> -- globally find string “key”, substitute all old string with new string (first instance per line unless have optional “g” then all instances on line). Uses regular expressions for the pattern matching.

misc



J - takes the line below the current line and appends it to the current line. (end up with one, longer line.)

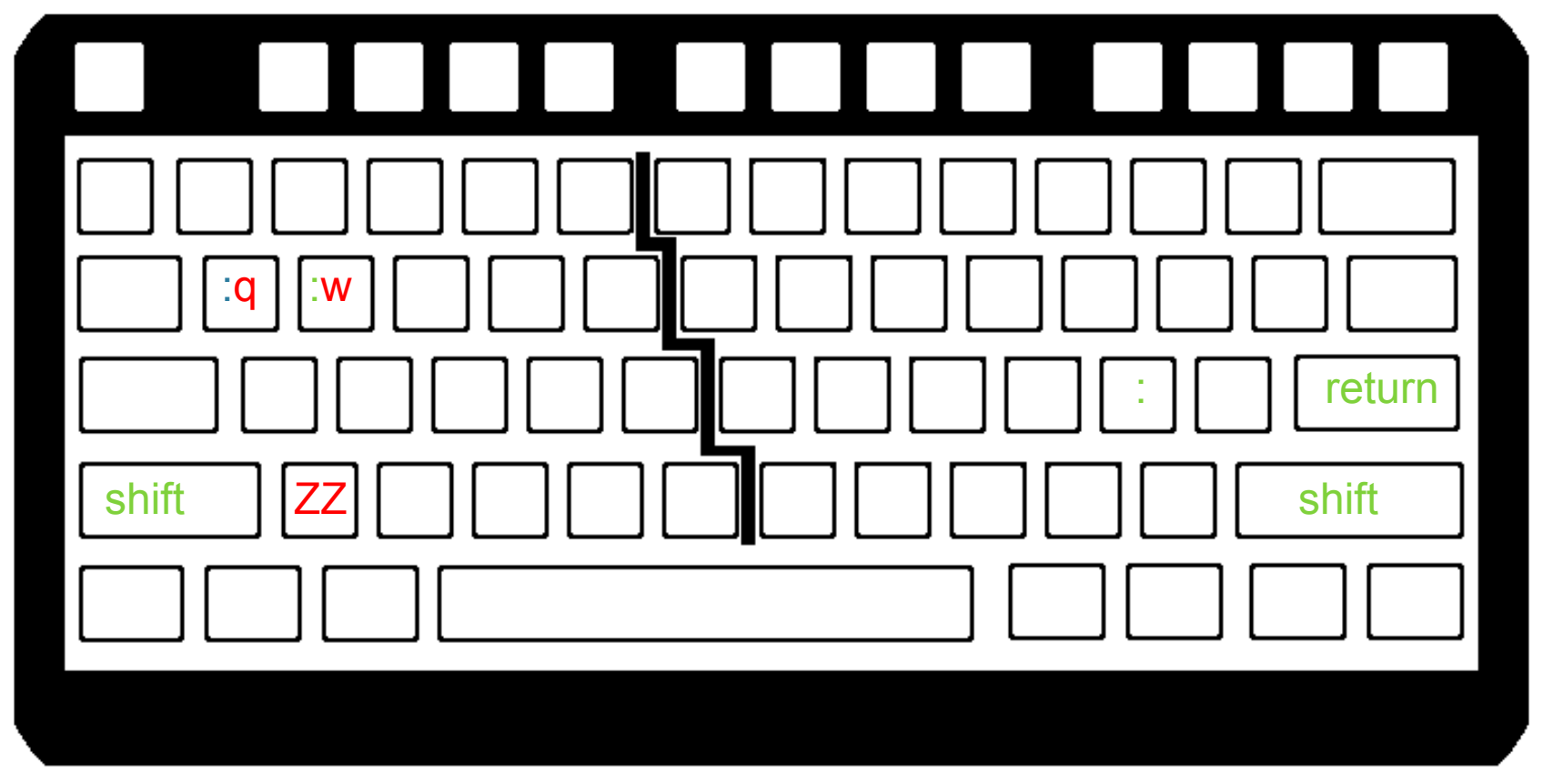
saving and exiting vim



:w[!] [filename] -- [optionally over] write to file filename

:w – overwrites input file given on vi call, remains in vi

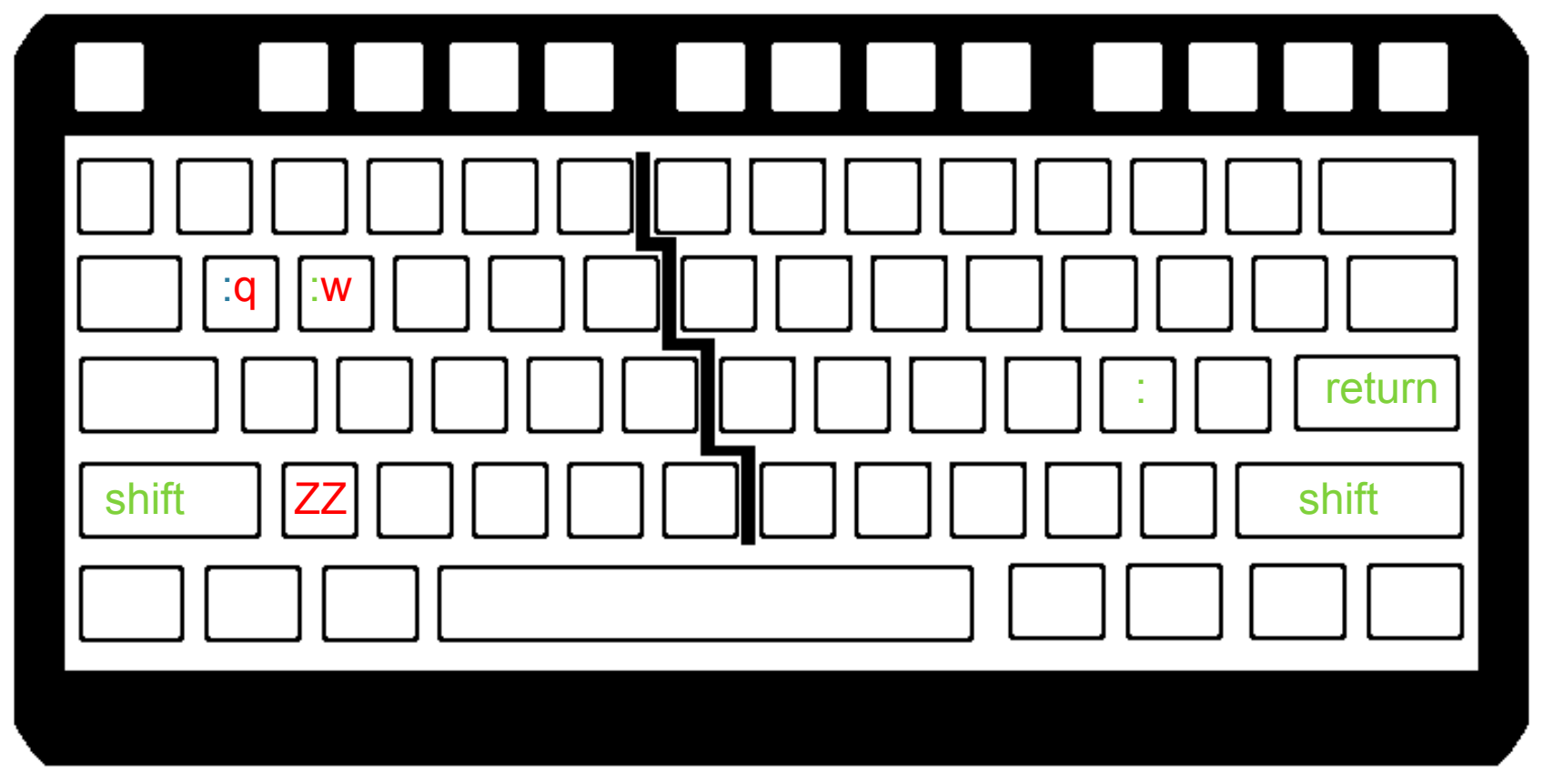
saving and exiting vim



:wq -- overwrites input file given on vi call and quits

ZZ -- overwrite and quit

saving and exiting vim



:q --- quit (does not save). Stops you and you get a message if you have changed anything -

E37: No write since last change (add ! to override)

:q! --- force quit without saving (ignores any changes)

vi and vim

Common options

`-R` read only mode (also `view` in vim – alias for `vim -R`)

`-r {file}` recovery mode using swap file after a crash

vim

Has ability to do column editing

Review

When you want to search for a string of text and replace it with another string of text, you can use the syntax

`:[range]s/search/replace/[g][c][i].`

Range can be -
`n,m` for lines n to m
n,$ for lines n to last (`1,$` for whole file)
or `g,%` for whole file

`g` – global in the line, `c` – confirmation, `i` – ignore case.

Review

`:[range]s/search/replace/[g][c][i].`

The range, global and confirm fields are optional (given in brackets []).

if you just run

`:s/search/replace/`

it will search only the current line and match/replace <u>only the first occurrence</u> of the match.

Review

Ex with range specified, plus "g" at end is for global (on line) replace (all matches <u>on line</u>, not just first)

```
:8,10 s/search/replace/g
```

If you want to search an entire file, and replace all matches, you can use `%` to indicate the whole file as the range, and g for all matches on each line:

```
:%s/search/replace/g
```

other useful features in vi/vim

`:![unix command]` -- allows you to run standard unix commands without exiting vim; very useful with GMT

Example

```
$:!ls *.SAC
```

In command mode the “ **:** ” tells vi that we are doing a command from the ed/edit/sed command list.

If you look in the man pages for vi or vim, it will refer you to the man pages for ed for the command descriptions.

other useful features in vi/vim

`:set hlsearch` -- will highlight all instances of a string when using `/[word]` to search

`>aB` -- indent the block/loop defined by `{}` when cursor is located within the block in question

`:sp` -- split the screen

`^WW` -- use to move from one split screen to the next; useful when writing subroutines within the same file

other useful features in vi/vim

`: set number` or `:set nonumber` -- turn line numbers on/off

`:x` -- jump to line number `x` example `:1`

There are whole books on vi and vim. We are just scratching the surface.

Once you learn one of these, you tend to use them instead of the GUI/"word" like editors.

From the author of "The best of vim tips" web page "15 Years of Vi + 7 years of Vim and still learning 05Aug11 : Last Update "
(I'm not quite sure if this is good or bad!)

Regular Expressions

Basics of the UNIX/Linux Environment

Regular Expressions

If you master regular expressions, searching for text becomes easy.

Regular expressions are accepted input for grep, sed, awk, perl and other unix commands.

Much like learning the shells, it is all about syntax & we'll just scratch the surface here.

Regular Expressions

Unfortunately Regular Expressions use some of the wildcards (very) differently than the shell.

It is quite common for the same character to show up multiple times in an expression and mean different things in each instance!

Basic "regular expressions"

- . : Matches a single character

```
523:> grep P..D samgps.dat
PELD -33.14318 -70.67493 CAP [5] 1993 1997 1998 1999 2002 CHILE OKRT
MOAT -54.9572 -66.79024 SCARP|CAPP|TDF [4] 1998 2000 2007 ARGENTINA
```

But probably not what I was looking for (I was most likely looking for the station PELD and stations whose name starts with P and ends with D, not the other combination – side effects – it does exactly what you tell it).

Basic "regular expressions"

"*": Matches zero or more instances of the preceding character

```
529:> grep AT1*0 samgps.dat
AT01 -31.00523 -68.49972 US|MATE|CAPP [2] 1997 1999 ARGENTINA
AT02 -30.86703 -68.49559 US|MATE|CAPP [4] 1997 1998 1999 2004
AT03 -30.89345 -68.42641 US|MATE|CAPP [5] 1997 1998 1999 2000
AT04 -30.98976 -68.80327 US|MATE|CAPP [5] 1997 1998 1999 2002
AT05 -30.84826 -68.94951 US|MATE|CAPP [5] 1997 1998 1999 2000
AT06 -30.87866 -68.68793 US|MATE|CAPP [5] 1997 1998 1999 2000
AT07 -30.34463 -68.60229 US|MATE|CAPP [4] 1997 1998 1999 2004
AT08 -30.24569 -68.46489 US|MATE|CAPP [4] 1997 1998 1999 2004
AT09 -30.27979 -68.53166 US|MATE|CAPP [3] 1997 1999 2004 ARGE
AT10 -30.28933 -68.54643 US|MATE|CAPP [4] 1997 1999 2000 2004
```

Basic "regular expressions"

"*": Matches zero or more instances of the preceding character

```
529:> grep 'AT1*0' samgps.dat
```

What were we looking for?

AT0…, AT10…,AT110…,AT1110…

Basic "regular expressions"

How do we look for anything and everything (zero or more instances of any character).

The regular expression "*" (the shell wildcard from earlier that does just that – in the shell) does not do it – we just saw that it does zero or more instances of the <u>preceding</u> character.

Basic “regular expressions”

We have enough information.

All we have to do is think UNIX.

Basic "regular expressions"

The "." represents any character.

The "*" is any number of repetitions (including none or zero) of the preceding character.

Basic “regular expressions”

How about

“ . * ”

(dot, splat)

Any character plus zero or more repetitions of any character.

You can think of regular expressions as wildcards on steroids (or LSD).

Basic "regular expressions"

"`.*`": Matches zero or more instances of preceding character

Looking for lines strings with "`YA`" and "`ARG`" with any number characters between

```
-bash 618 # grep YA samgps.dat
YAVI -22.13792 -65.48923 US|CAP|POSGAR07 [1] 2006 ARGENTINA OKRT
HYAT -48.73171 -75.33964 US|CAP|GFZ|SCARP|TRANSFER|BOAT|SENH|PIF
CCYA -21.63037 -65.04788 US|CAP3 [2] 2003 2009 BOLIVIA OKRT
LYAR -18.134395 -70.568644 CALT|CLSD [c] continuous (2005-) CHILE
YANI -37.363806 -73.657833 US|CAP|C2010|RAPID|OPEN [c] continuous
YAPE -29.45242518 -56.91402597 POSGAR07 [1] 2006 ARGENTINA NORT
-bash 619 # grep YA.*ARG samgps.dat
YAVI -22.13792 -65.48923 US|CAP|POSGAR07 [1] 2006 ARGENTINA OKRT
YAPE -29.45242518 -56.91402597 POSGAR07 [1] 2006 ARGENTINA NORT
```

This is how you look for two strings (but have to be in order)

So now we have two kinds of special characters, or metacharacters.

Those that mean something special to the shell (such as the "$" on a shell or environment variable or the "/" in a path, or the *).

And those that are used to specify a pattern in Regular Expressions, such as the *.

And will need a way to "turn off", or escape, the special meaning in both cases.

\ : Escapes the following metacharacter. Tells it to use the following metacharacter as a regular character (i.e. look for a *, don't use it to mean zero or more occurrences).

```
% grep  '\*'  suma.stations | head –n2
*AGD      +11.529000      +042.824000
*       AIS     -37.797000      +077.569000
```

[ ] : Matches members of the sets/ranges within the brackets (set [abclmn] any single match of a,b,c,l,m,n. range [a-c] any single match of letters in range of a to c, i.e. a,b,c.)

```
% grep '[DB]EQ' SUMA.NEW.loc
3478 2005 7 4 16 7 35.23   10.301   93.576   29.9  4.9  0.0 ehb
DEQ Md
3480 2005 7 5 1 52 4.16    1.822   97.068   30.0  6.2  6.8 ehb
BEQ Md
3481 2005 7 5 7 57 27.19    2.244   94.978   15.7  5.1  4.5 ehb
DEQ Md
```

^ : Represents the beginning of a line

```
534:> cat samgps.dat
PELD -33.14318 -70.67493 CAP [5] 1993 1997 1998 1999 2002 CHILE
COGO -31.15343 -70.97526 CAP [3] 1993 1996 2002 CHILE OKRT
MORA -30.20823 -70.78971 CAP [3] 1993 1996 2002 CHILE OKRT
MOR2 -30.20823 -70.78971 CAP [?] CHILE OKRT
TOFO -29.45939 -71.23842 CAP [4] 1993 1996 2001 2002 CHILE
SILA -29.24037 -70.74956 CAP [3] 1993 1996 2002 CHILE OKRT
HUAS -28.47848 -71.22235 CAP [3] 1993 1996 2002 CHILE OKRT
PSTO -28.17157 -69.79377 CAP [3] 1993 1996 2002 CHILE OKRT
GRDA -27.71571 -69.55836 CAP [2] 1993 1996 CHILE OKRT
CALD -27.0827 -70.86208 CAP [5] 1993 1996 1999 2001 2002 CHILE
PNAZ -26.14822 -70.65368 CAP [3] 1993 1996 2001 CHILE OKRT
532:> grep ^P samgps.dat
PELD -33.14318 -70.67493 CAP [5] 1993 1997 1998 1999 2002
PSTO -28.17157 -69.79377 CAP [3] 1993 1996 2002 CHILE OKRT
PNAZ -26.14822 -70.65368 CAP [3] 1993 1996 2001 CHILE OKRT
PPST -20.97508 -68.83487 CAP [3] 1993 1996 2001 CHILE OKRT
PSAG -19.6023 -70.21962 CAP [3] 1993 1996 2001 CHILE OKRT
```

$: Represents the end of the line

```
file example IND.pha
# 1918 9 22 9 54 49.29  -1.698   98.298   15.0  0.0 0 0
COC        274.71 1 P
MAN        346.71 1 P
ZKW        450.71 1 P
# 1926 6 28 3 23 26.82  -0.128  101.514   15.0  0.0 0 0
COC        303.18 1 P

%grep 'P_*$' IND.pha  | head –n2
COC        274.71 1 P
MAN        346.71 1 P

or

%grep –c 'P_*$' IND.pha    the –c flag counts matches
         831857
```

UNIX think practice.

What represents an empty line?

Basic “regular expressions”

We now have all the pieces, we just have to put them together in UNIX think.

Any guesses?

What represents an empty line?

^$

A “beginning of line” (bol), followed by an “end of line”.

(this does not get lines that “look empty” to us, but not UNIX, because they contain only spaces or tabs. This is what makes it such fun!)

Non-printable characters

Here the escape means use the following regular character for a special character (you can't see a tab, but it is a "character" to UNIX).

The following syntax works with a range of commands and programs that recognize regular expressions (sed, awk, perl, printf, etc)

`\t` : for a tab character
`\r` : for carriage return
`\n` : for line feed or new line.
`\s` : for a white space

ASCII table

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

What is actually in a file

```
-bash 628 geolfigs # cat play
line 1


line 4
-bash 629 geolfigs # od -hc play
0000000      696c    656e    3120    0a0a    0a20    696c    656e    3420
          l   i   n   e       1  \n  \n      \n   l   i   n   e       4
0000020      000a
         \n
0000021
-bash 630 geolfigs #
```

The cat output shows us the file as characters. The second output (od = octal dump) shows us the hexadecimal (h, top line) and character (c, bottom line) elements of the file.

What is actually in a file

```
-bash 628 geolfigs # cat play
line 1


line 4
-bash 629 geolfigs # od -hc play
0000000      696c    656e    3120    0a0a    0a20    696c    656e    3420
           l   i   n   e       1  \n  \n      \n   l   i   n   e       4
0000020      000a
          \n
0000021
-bash 630 geolfigs #
```

You can find the ascii values for the letters (`l=6c`, `i=69`, etc.), and the non-printing characters (`\n=new line`) in the ASCII table.

The numbers on the left count the bytes (in base 8 so `0000020=16` in base 10. There are 17 bytes in the file.)

What is actually in a file

```
-bash 628 geolfigs # cat play
line 1


line 4
-bash 629 geolfigs # od -hc play
0000000      696c    656e    3120    0a0a    0a20    696c    656e    3420
           l   i   n   e       1  \n  \n      \n   l   i   n   e       4
0000020      000a
          \n
0000021
-bash 630 geolfigs #
```

Notice that the line separator is just a new line (`\n`).

The `^$` in the Regular expression matches the pair `\n\n` in the file.

What is actually in a file

```
-bash 628 geolfigs # cat play
line 1


line 4
-bash 629 geolfigs # od -hc play
0000000      696c    656e    3120    0a0a    0a20    696c    656e    3420
           l   i   n   e       1  \n  \n      \n   l   i   n   e       4
0000020      000a
          \n
0000021
-bash 630 geolfigs #
```

Notice that while lines 2 and 3 look the same to us (blank lines), they are actually different to the computer. Line 2 is really blank (`0a0a=\n\n`), while line 3 has a space (`0a 0a20=\n \n`).

(the hex display `696c` is "backwards" to the order of the characters `l i`, see me if you want more info.)

To match regular expressions

/ regular expresssion here /

The stuff inside the / is the field you are trying to match or replace.

Don't always need the /. Usually obvious when you need them (eg not for grep, but yes when substituting).

To match a word

`/ word /` is a good attempt at a match a word (words are delimited by leading and following space), but does not get the word when followed by punctuation for example (“ `word.` ”).

`\<word\>` the characters `\<` match the start of a word, while `\>` match the end of a word

(have to escape the < and >, and don’t need the /’s anymore)

now matches the word “word”.

Say you want to find a string and append something to it.

Try this.

```
s/run/&s/
```

Will match <u>run</u> and produce <u>runs</u>.

The & represents the match.

Say you want to find a string and append something to it.

Try this.

`\1` is first match, `\2` is second.

So this will also do it.

```
:%s/\(run\)/\1s/
```

You need the `()`, which needs to be escaped, `\`, (else it will look for `(run)`, not `run`)

The `\( . . . \)` delimiters are used to inform the editor that the text that matches the regular expression inside the parentheses is to be remembered for later use (in the `\1`).

Now we can attempt to understand these

```
sed 's/[^ ]*/(&)/' < old > new
sed 's/[^ ][^ ]*/(&)/g' < old > new
sed 's/^\([^:]*\):[^:]:/\1::/' </etc/passwd >/etc/password.new
```

count the number of lines in the three files `f1 f2 f3` that don't begin with a "#:"

```
sed 's/^#.*//' f1 f2 f3 | grep -v '^$' | wc -l
```

Now we can attempt to understand these

```
sed 's/[^ ]*/(&)/' < old > new
```

We need to see a few more definitions of regular expression elements

[ ] defines a “class” of characters

What are character classes?

A character class matches a single character out of all the possibilities offered by the character class. Inside a character class, different rules apply.

The rules in this section are only valid inside character classes.

The rules outside this section are not valid in character classes, except for a few character escapes that are indicated with "can be used inside character classes".

(see the regular-expressions.info web site link on the class web site)

Now we can attempt to understand these

```
sed 's/[^ ]*/(&)/' < old > new
```

We need to see a few more definitions of regular expression elements

[ ] defines a "class" of characters

Inside a class definition the ^ immediately after the [means negation of the class (outside a class definition it means the beginning of a line)

It is followed by a space [^]

So this matches any single character not equal to a space.

Now we can attempt to understand these

```
sed 's/[^ ]*/(&)/' < old > new
```

Continuing on we are looking for a non-space character, the `[^ ]`, repeated zero or more times, the `*`.

So that is what we are looking for?

The first string of non-spaces.

Now we can attempt to understand these

```
sed 's/[^ ]*/(&)/' < old > new
```

When we find it, we will replace the first occurrence of it with the string represented by the **(&)** which is an open paren, followed by whatever we found (indicated by the ampersand) followed by a closed paren.

Now we can attempt to understand these

```
sed 's/[^ ]*/(&)/' < old > new
```

Notice that the parens here are taken as regular characters (no \)

- why, since parens are metacharacters?

Answer – parens are metacharacters in the match definition, so if I'm looking for parens I have to escape them.

In the output I'm not using search metacharacters for anything so they don't have a special meaning, I'm just specifying what to output.

You can work on the others

```
sed 's/[^ ]*/(&)/' < old > new
sed 's/[^ ][^ ]*/(&)/g' < old > new
sed 's/^\([^:]*\):[^:]:/\1::/' </etc/passwd >/etc/password.new
```

count the number of lines in the three files `f1 f2 f3` that don't begin with a "#"

```
sed 's/^#.*//' f1 f2 f3 | grep -v '^$' | wc -l
```

Regular expressions are like mathematics where each symbol is absolutely essential and means a very specific thing and you better understand all the ramifications and details.

It is not like literature where you can randomly throw out 5% of the letters and still understand it.

Compress multiple occurrences of blank lines into a single blank line

```
:v/./,/./-j
```

Use :helpgrep '\/,\/' *.txt for an explanation.

I'll break down this incredible collapse-multiple-blank-lines command for everyone, now that I finally figured out how it works.
First, however, I'll rewrite it this way to illustrate that some of those slashes have totally different meaning than others:

```
:v_._,/./-1join
```

Note that to delimit expressions like these, just about any symbol can be used in place of the typical slashes... in this case, I used underscores.
What we have is an inverse search (:v, same as :g!) for a dot ('.') which means anything except a newline. So this will match empty lines and proceed to execute [command] on each of them.

```
:v_._[command]
```

The remaining [command] is this, which is a fancy join command, abbreviated earlier as just 'j'.

```
,/./-1join
```

The comma tells it to work with a range of lines:

```
:help :,
```

With nothing before the comma, the range begins at the cursor, which is where that first blank line was. The end of the range is specified by a search, which to my knowledge actually does require slashes. The slash and dot mean to search for anything (again), which matches the nearest non-empty line and offsets by {offset} lines.

```
/./{offset}
```

The {offset} here is -1, meaning one line above. In the original command we just saw a minus sign, to which vim assumes a count of 1 by default, so it did the same thing as how I've rewritten it, but simply with one character fewer to type.

```
/./-1
```

There is a caveat about join that makes this trick possible. If you specify a range of only one line to "join", it will do nothing. For example, this command tells vim to join into one line all lines from 5 to 5, which does nothing:

```
:5,5join
```

In this case, any time you have more than one empty line (the case of interest), the join will see a range greater than one and join them together.
For all single empty lines, join will leave it alone.

There's no good way use a delete command with :v/./ because you have to delete one line for every empty line you find. Join turned out to be the answer.
This command only merges truly "empty" lines... if any lines contain spaces and/or tabs, they will not be collapsed. To make sure you kill those lines, try this:

```
:v/^[^ \t]\+$/,/^[^ \t]\+$/-j
```

Or, to just clean such lines up first,

```
:%s/^[ \t]\+$//g
```

The trick with Regular Expressions is to be able to generate them, not just understand them when provided.

Generating them is usually an iterative process (sort of like passing the law "to see what is in it" [Pelosi], you have to execute the command and see what it does. Then "fix" it, try again, etc. Most normal people can't write these things 100% the first go.)