Sort and grep continued.
Continuation from previous lecture notes on sort and grep

sort [options] file

key options
-n numeric sort
-u unique
-f ignore case
-k key (start a key at POS1, end it at POS2 (origin 1)). The -k1,1 notation sets the start and stop limits of the first sort key to the first field.
-t field-separator=SEP; use SEP instead of non-blank to blank transition

The sort command sorts a file alphanumerically by line. In effect, it groups according to the first element

Examples:
#temp file contains:

again over there
redo hello there
stop saying over
echo Over there

```
sort -k3 temp
```
stop saying over
again over there
echo Over there
redo hello there

#Here there are ties. If equal, sort will sort according to entire line. This could change the order of lines that are equal at the sort position.

```
sort -k3 -k2 temp
```
stop saying over
echo Over there
redo hello there
again over there

Finally, what is the difference between:
```
sort -k2 temp
sort -fk2 temp
```

temp2 contains:
10 2 5
1 3 5
20 3 5

```
4    2    6
10   1    6
10   0    7
10   0    5
sort temp2
1    3    5
10   0    5
10   0    7
10   1    6
10   2    5
20   3    5
4    2    6

sort -k1 -k3 temp2
1    3    5
10   0    5
10   0    7
10   1    6
10   2    5
20   3    5
4    2    6
#This seems odd because item 3 is before item 4. Column 2 is contributing to the result.

sort -k1,1 -k3,3  temp2
1    3    5
10   0    5
10   2    5
10   1    6
10   0    7
20   3    5
4    2    6
#This is more what one would expect. We stop the sort on column 1.

sort -n -k1,1 -k3,3  temp2
1    3    5
4    2    6
10   0    5
10   2    5
10   1    6
10   0    7
20   3    5
#Note what the "n" does.

Here is another new file temp3
a       10
a       10
b       15
```

b 13
b 15

```
sort –uk2n temp3
```
a 10
b 13
b 15

uniq

uniq outputs all lines with *consecutive* duplicates removed.

```
cut –f1 temp3 | uniq –c #Gives us counts of occurrences.
```

Related (we practiced on a .gff3 file)

```
cut –f1 temp3 | sort -u <filename> #will give you unique members of a column
cut –f1 temp3 | sort -u <filename> | wc #will count them
```

spaces in files

#separate the elements of temp1 by spaces

```
cut –f1 –d ' ' temp1 | uniq
```

#Now do the same command with tab delimited temp1 file.
#What is the difference?

We can see what the spaces really are by using:

```
cat –vet <filename>
```

^I is a tab
$ marks the end of the line

grep

Many text-based data files use a common syntax of one record per line. Grep searches lines for matches. We will discuss in more detail later.

```
grep [options] <pattern> file
```

key options

-c prints a count of matching lines
-l lists the names of files with matching lines, but not the individual matched lines
- I ignore case
-A prints lines after the match
-B prints lines before the match
-E extended regular expression (later)
-r Recursively search subdirectories listed
-v reverse match- prints those lines that don't have pattern
-n prints line numbers
-f takes a list of patterns from file; one per line

-e allows multiple patterns to be matched
-E parse pattern as extended regular expression (later)
- w is the word boundary-- i.e. 259 would match 259 not 2597.

What grep does and does not match of course makes sense but can get complex.

Here is a file temp

again over there
echo Over there
stop saying over
redo hello there

Some questions grep can answer:
What lines contain Over?
What lines don't contain Over?
Does any line match “do”?
What lines have ll, lh, tl, or th?
What is the overlap between names in one file and records(rows) in a second?
… and more!

```
grep Over temp
```
echo Over there

#note case sensitivity
#the -i flag means "ignore case"
```
grep –i Over temp
```
again over there
stop saying over
echo Over there

#what lines don't match?

#The -v option turns the logic of the grep command upside down; it shows lines that don't match the pattern.

```
grep –iv Over temp
```
redo hello there

We can search for do
```
grep –i do temp
```
redo hello there
##note how grep matched do even though it was in a word.

```
grep –iw do temp
```
<NIL>
why?

how many lines have a match to a pattern?
```
grep –ic Over temp
```

#what line numbers have a matching pattern?
```
grep –ni over temp
```

#What lines have ll, lh, tl, or th?
```
grep [lt][lh] temp
```

How about the beginning of a line?
^
```
grep ^again temp
```

#You can put the pattern in quotes. It is needed if you want spaces.
`grep 'hello  there' temp` #works, though of course, spaces won't match tabs.
`grep hello  there temp` #generates an error

Still more…
You may have a list of entities in one file that you want to search for in another file.

Here is a file: temp_search

hello
stop

```
grep –f temp_search temp3
```
redo hello there
stop saying over
#what happened

```
grep –l 10 *
```
this searches all files for -l within the current directory

#my working directory is Temp. What does this command do?
```
grep –Rl 10 /Users/lewis/Temp/
```

Finally grep -A (n) and -B (n) will return both the matching line and (n) lines after(A) or before (B)

#####
Redirecting input and output
We sometimes want to keep the results from a unix program to do further work on them.

<u>Standard output</u>

Unix treats the output of programs likes a stream of data that can be redirected to different places. The official term for the output of any command is **standard output (stdout)**. Your computer screen is the default destination for stdout.

Rather than have the output go to the screen, we can use the **redirect operator** to send it to a new file. (e.g grep as an example: grep x filename > outfile).

If you accidentally redirect a command's output to a file that exists, it will clobber it. Redirection will re-write files, even if there is no output.

Example
`nano important_file`
type in some text

```
grep string someanalysis > important_file
"grep: targetfile: No such file or directory" (Error message)
```

#will replace the file with an empty file even when nothing is there.

set -o noclobber" #prevents this
```
grep string targetfile > important_file
"-bash: important_file: cannot overwrite existing file" (Error
message)
```

Appending to an existing file:
`>>`
is the "append redirect operator"
`echo "Thanks" >> letter.txt`
#text is appended immediately after last letter, no carriage return.

Using redirects with the cat command is common. The cat command prints the entire contents of a file.
```
cat a.txt > all.txt
cat b.txt >> all.txt
cat c. txt >> all. txt
cat a.txt b.txt c.txt > all.txt
cat *.txt > all.txt
```
#they will be combined depending on the ASCII values of their filenames.

Standard input (also known as stdin)

Data stream, usually text, that can be provided as input for UNIX program.
There is a default source of information for a program: the keyboard.

Standard input only refers to the input used by commands. As of now, most take their input from a file we specify as arguments. This is separate from standard input. We are not asked for more information

e.g. less <filename>

#filename is an argument to the command. It is not the standard input.

tr, transliteration, requires some typed input after the command has started-- this comes from standard input.
tr: specify a range of characters that are to be changed into others. tr 'A' a

The user can now provide standard input.
control d to escape

Redirecting standard input
We can redirect standard input to come from a file with the "<" redirect operator. It is used less than redirecting standard output.
tr 'A' a < yeast_chr1.fa
Very few commands (like tr) need to read from standard input. Still...

grep 'pattern' datafile #here, datafile is an argument.
grep 'pattern' < datafile #read contents as standard input

Redirecting both
tr 'A' 'a' < yeast > processed #redirecting both input and output of command

Remember the cat command prints the entire contents of a file.
#Also useful for doing something line by line...
cat hello | tr '\n' '\t'
tr= translate characters.

Standard Error
< redirect standard input from a file
> redirect standard output to a file

There is another unix data stream- standard error. This data stream may be produced when a command is used incorrectly. It is distinct from stdout.

Example
```
echo "hello" > greeting
echo "goodbye" > farewell
cat greeting farewell

cat greeting farrewell #get standard out and standard error
```
“hello
cat: farrewell: No such file or directory”

```
cat greeting farrewell  > output_file #now get standard out
cat output_file #note that hello is in the new file and farewell is not.
```

This code shows that messages are controlled by a separate data stream (standard error) from the normal output. When we redirect standard output to a file, the file does not contain error messages.

You may want to capture standard error to a file or turn it off entirely.
standard output has a label 1
standard error has a label 2

```
cat greeting farrewell  2> errors
```
#now we see STDOUT on the screen.
#`head errors` to get errors
2>> to append. No space.

You can combine standard out and standard error into one stream with &>

You may want to turn off standard error so you can see stdout. Redirect standard error to the null device /dev/null #the black hole.
cat hero villlian 2> /dev/null
cat hero villian 2> errors > outfile

cat errors