UNIX II: Files, data, and running programs on the cluster

In UNIX I you logged in to the cluster, moved around the file system, looked at files, and wrote a first shell script. This lecture covers the everyday skills you need to *work* on the cluster: organizing files, getting data onto the cluster, finding software, running a program on a compute node, and turning in homework with Git.

What you'll learn

- Copy, move, rename and delete files and folders safely (`cp`, `mv`, `rm`)
- Refer to many files at once with wildcards (`*`, `?`, `[ ]`, `{a,b}`)
- Make shortcuts to files and folders with symbolic links (`ln -s`)
- Read `ls -l` output, and change permissions to share files with your lab (`chmod`)
- Check how much space you are using (`du`, `df`, `check_quota`)
- Send output to files and to other programs (`>`, `>>`, `2>`, `|`, `tee`)
- Search inside files with `grep`
- Compress and uncompress data (`gzip`, `zcat`, `tar`)
- Download data (`curl`, `wget`) and check that it downloaded correctly (`md5sum`)
- Copy files between your laptop and the cluster (`scp`, `rsync`)
- Watch and stop running programs (`ps`, `top`, `kill`, `jobs`, `bg`, `fg`)
- Find and load software with `module`, and run it on a compute node with `srun`
- Clone, commit and push your homework with Git

Each topic links to the lecture where it continues. A quick reference table and practice exercises are at the end.

Before you start: get the class data

Log in to the cluster (see UNIX I) and make a folder for the class in your bigdata space. The class data files live in two GitHub repositories. Clone both (you only need to do this once):

```
mkdir -p ~/bigdata/gen220
cd ~/bigdata/gen220
git clone https://github.com/biodataprog/GEN220.git       # small files in GEN220/data
git clone https://github.com/biodataprog/GEN220_data.git  # genome/ and tabular/ data
mkdir -p unix2          # a folder to practice in for this lecture
cd unix2
```

You can also browse them on the web: GEN220/data and GEN220_data. If you already cloned them, get any updates with `git pull` inside each folder.

Your layout should now look like this. All the examples below are run from inside `unix2`, so the data files are one folder up (`../`):

```
~/bigdata/gen220/
    GEN220/data/          codon_table.txt, rice_random_exons.bed, numbers.txt, ...
    GEN220_data/genome/   S_cerevisiae.fasta.gz, S_cerevisiae.gff3.gz, *.pep.gz, ...
    GEN220_data/tabular/  threatened-species.csv.gz, airport-codes.csv.gz
    unix2/                <- you are here
```

```
ls ../GEN220/data
ls ../GEN220_data/genome
```

```
codon_table.txt      numbers.txt                  rice_random_exons.bed
Ecoli_K-12.fasta.gz  rice_chr6_3kSNPs_filt.bed.gz  yeast_orfs-to-chr1.FASTA.tab

E_coli_K12.pep.gz                Orthogroups.csv             S_cerevisiae.ORFs.fasta.gz
E_coli_O157_H7.pep.gz            S_cerevisiae.fasta.gz       S_enterica_CT18.fasta.gz
Ecoli-vs-Senterica.BLASTP.tab.gz S_cerevisiae.gff3.gz        S_enterica.pep.gz
Ecoli-vs-Yersinia.BLASTP.tab.gz
```

Working with files

Copying, moving, renaming and deleting

cp - copy

`cp SOURCE DESTINATION` makes a copy. If the destination is an existing folder, the copy goes inside it with the same name.

```
cp ../GEN220/data/codon_table.txt .                  # copy into the current folder (.)
cp ../GEN220/data/codon_table.txt codons_backup.txt  # copy to a new name
mkdir tables
cp codon_table.txt tables/                           # copy into a folder
cp codon_table.txt codons_backup.txt tables/  # several files: last one is the folder
ls tables
```

```
codon_table.txt  codons_backup.txt
```

Copying a folder needs `-r` (recursive: the folder and everything inside it):

```
cp tables backup_tables
```

```
cp: -r not specified; omitting directory 'tables'
```

```
cp -r tables backup_tables
ls backup_tables
```

```
codon_table.txt  codons_backup.txt
```

Warning: `cp` silently **overwrites** a file that already exists at the destination. Useful options:

Option	Meaning
`-r`	copy folders and their contents (recursive)
`-i`	ask before overwriting (interactive)
`-n`	never overwrite an existing file
`-v`	print what is being copied (verbose)
`-p`	keep the original modification time and permissions

```
cp -i codon_table.txt tables/
```

```
cp: overwrite 'tables/codon_table.txt'? n
```

mv - move or rename

Moving a file and renaming a file are the same operation. `mv` is instant, even for huge files, as long as you stay on the same file system (e.g. within your bigdata folder).

```
mv codons_backup.txt codons_old.txt     # rename
mv codons_old.txt tables/               # move into a folder
mv -v tables/codons_old.txt .           # move it back here; -v says what happened
mv tables codon_tables                  # folders can be renamed too (no -r needed)
```

```
renamed 'tables/codons_old.txt' -> './codons_old.txt'
```

`mv` also overwrites without asking; `mv -i` asks first, `mv -n` never overwrites.

rm and rmdir - delete

```
rm codons_old.txt              # delete a file
rm -i codon_tables/*.txt       # ask about each file first
rmdir codon_tables             # delete an EMPTY folder
rm -r backup_tables            # delete a folder and everything in it
```

```
rm: remove regular file 'codon_tables/codon_table.txt'? y
rm: remove regular file 'codon_tables/codons_backup.txt'? y
```

There is no trash can and no undo on the command line. Before running `rm -r` or `rm -rf` (`-f` = force, never ask), run `ls` with exactly the same arguments to see what would be deleted. Be especially careful with wildcards (next section): `rm * .txt` (note the space) deletes *everything* in the folder and then complains that `.txt` does not exist.

If you do delete something important, the HPCC keeps snapshots: home directories daily for one week (`/rhome/.snapshots/`) and bigdata weekly for one month (`/bigdata/.snapshots/`). See the HPCC storage manual. Anything newer than the last snapshot is gone, so this is a last resort, not a plan.

Wildcards (globbing)

Wildcards let you name many files with one pattern. The **shell** expands the pattern into a list of matching file names *before* the command runs, so wildcards work with every command (`ls`, `cp`, `mv`, `rm`, `wc`, `zcat`, ...).

Pattern	Matches	Example	Matches e.g.
`*`	any characters (zero or more)	`*.txt`	`codon_table.txt`, `numbers.txt`
`?`	exactly one character	`chr?.bed`	`chr1.bed`, `chrX.bed` (not `chr10.bed`)
`[abc]`	one character from the set	`[cn]*`	files starting with `c` or `n`
`[0-9]`	one character in a range	`*[0-9].gz`	names ending in a digit then `.gz`
`[!abc]`	one character NOT in the set	`[!S]*`	files not starting with `S`
`{a,b}`	each of the listed words	`*.{txt,bed}`	all `.txt` and `.bed` files

```
cd ../GEN220/data
ls *.txt
ls r*
ls *.bed*
ls [cn]*
ls *.{txt,bed}
wc -l *.txt
```

```
codon_table.txt  numbers.txt
rice_chr6_3kSNPs_filt.bed.gz  rice_random_exons.bed
rice_chr6_3kSNPs_filt.bed.gz  rice_random_exons.bed
codon_table.txt  numbers.txt
codon_table.txt  numbers.txt  rice_random_exons.bed
  64 codon_table.txt
 100 numbers.txt
 164 total
```

```
cd ../../GEN220_data/genome
ls S_*                          # all the Saccharomyces and Salmonella files
ls *.pep.gz                     # all the protein files
zcat *.pep.gz | grep -c '>'     # total proteins in the three files: 14293
cd ../../unix2                  # back to our practice folder
```

To see what a pattern expands to, `echo` it. This is a good habit before `rm` or `mv`:

```
echo ../GEN220/data/*.gz
```

```
../GEN220/data/Ecoli_K-12.fasta.gz ../GEN220/data/rice_chr6_3kSNPs_filt.bed.gz
```

Things to know:

- If nothing matches, bash passes the pattern through unchanged, so you get an error like `ls: cannot access '*.fastq': No such file or directory`.
- `*` does not match names starting with `.` (hidden files like `.bashrc`); use `ls -a`.
- Quotes turn wildcards off: `echo "*.txt"` prints `*.txt`. This matters in `grep` and `find` patterns (see UNIX III and UNIX IV).
- Braces `{}` are not really wildcards: they generate words whether or not the files exist. That makes them handy for making names:

```
echo sample{A,B,C}.fastq
echo chr{1..5}.bed
mkdir -p project/{data,scripts,results}      # make three folders at once
cp codon_table.txt{,.bak}          # same as: cp codon_table.txt codon_table.txt.bak
```

```
sampleA.fastq sampleB.fastq sampleC.fastq
chr1.bed chr2.bed chr3.bed chr4.bed chr5.bed
```

(The last `cp` only works if `codon_table.txt` is in your folder, as it is after the `cp` examples above.)

Symbolic links: shortcuts to files and folders

A **symbolic link** (symlink) is a small file that just points to another file or folder, like a shortcut on your desktop. Programs that open the link actually read the target. We use links all the time in bioinformatics to:

- use a big shared file (a genome, a database) without making a copy of it
- give a long file name a short, simple name
- keep one copy of the data and "see" it from several project folders

You already have one: on the HPCC, `~/bigdata` in your home folder is a link to your real bigdata folder (`/bigdata/LABNAME/USERNAME`).

Make a link with `ln -s TARGET LINKNAME` (the target comes first, like `cp`):

```
ln -s ../GEN220_data/genome/S_cerevisiae.fasta.gz yeast_genome.fasta.gz
ls -l yeast_genome.fasta.gz
```

```
lrwxrwxrwx 1 yourname gen220 43 Sep 29 10:05 yeast_genome.fasta.gz ->
    ../GEN220_data/genome/S_cerevisiae.fasta.gz
```

(On your screen this is one line.) The `l` at the start of the line and the `->` show it's a link. It is only 43 bytes (the length of the path it stores), but it acts like the 3.6 MB genome file:

```
zcat yeast_genome.fasta.gz | head -n 2
```

```
>chrI
CCACACCACACCCACACACCCACACACCACACCACACACCACACCACACCCACACACACACATCCTAACACTACCCTAAC
```

If you leave off the link name, the link gets the same name as the target, in the current folder:

```
ln -s ../GEN220_data/genome/S_cerevisiae.gff3.gz     # makes ./S_cerevisiae.gff3.gz
ln -s ../GEN220/data data                            # links work for folders too
ls data/
```

Find out where a link really points:

```
readlink yeast_genome.fasta.gz        # what is stored in the link
realpath yeast_genome.fasta.gz        # the full, final path of the real file
```

```
../GEN220_data/genome/S_cerevisiae.fasta.gz
/bigdata/gen220/yourname/gen220/GEN220_data/genome/S_cerevisiae.fasta.gz
```

`realpath` follows *every* link, including `~/bigdata` itself, so it shows the real location under `/bigdata/LABNAME/USERNAME` (here the `gen220` lab).

Things to know:

- **A relative target is relative to where the link lives**, not to where you were when you made it. If you move a link that uses `../`, or make it in another folder, it can point at nothing. That is a **broken link**: `ls` still lists it (often in red), but using it fails:

```
mkdir sub
# wrong: from inside sub/, ../GEN220_data does not exist
ln -s ../GEN220_data/genome/S_cerevisiae.gff3.gz sub/genes.gff3.gz
zcat sub/genes.gff3.gz | head -n 1
```

```
gzip: sub/genes.gff3.gz: No such file or directory
```

- For files in a fixed place, such as the shared class folder, an **absolute path** target is safest: `ln -s /bigdata/gen220/shared/simple/yeast_gene_names.txt .`
- `rm LINKNAME` deletes only the link, never the target. For a link to a folder, use `rm data`, **not** `rm -r data/` (with the slash, `rm -r` goes inside and deletes the real files).
- Editing through a link edits the original. If you want your own copy to change, use `cp`.
- The link needs permission to read the target, so a link to a file in someone's private folder won't work for you.

You'll use links in later lectures to point at genomes and BLAST databases without copying them.

Reading ls -l and file permissions

`ls -l` (long listing) shows details for each file. `-h` gives human-readable sizes, and `-a` also shows hidden files. Here is part of the listing of our practice folder (yours will have a few more files; `run.sh` stands for a script like the ones you wrote in UNIX I):

```
ls -lh
```

```
total 32K
-rw-r--r-- 1 yourname gen220  938 Sep 29 10:02 codon_table.txt
lrwxrwxrwx 1 yourname gen220   14 Sep 29 10:06 data -> ../GEN220/data
drwxr-xr-x 5 yourname gen220 4.0K Sep 29 10:04 project
-rwxr-xr-x 1 yourname gen220  112 Sep 29 10:10 run.sh
```

```
 -rwxr-xr-x  1  yourname  gen220  112  Sep 29 10:10  run.sh
 |\_/\_/\_/  |     |        |      |       |           |
 | |  |  |   |     owner    group  size    last        name
 | |  |  |   links                        modified
 | |  |  other (everyone else): r-x
 | |  group members: r-x
 | owner (user): rwx
 type: - file, d directory, l link
```

The three permission letters are:

Letter	On a file	On a folder
`r` read	look at the contents	list the files in it (`ls`)
`w` write	change the contents	create, rename or delete files in it
`x` execute	run it as a program or script	go into it (`cd`) and reach files inside
`-`	permission not given	permission not given

Each set applies to one kind of person: **u**ser (the owner), **g**roup, and **o**thers. Note that deleting a file depends on the *folder's* `w` permission, not the file's.

chmod - change permissions

The symbolic form says who (`u`, `g`, `o`, `a` = all), `+` or `-` or `=`, and which permissions:

```
chmod +x run.sh              # let it be run as a program (you saw this in UNIX I)
chmod go-r notes.txt         # remove read for group and others
chmod g+w shared_table.tsv   # let your group edit this file
chmod u=rwx,g=rx,o= results  # owner everything, group read/enter, others nothing
chmod -R g+rX results        # -R: recursively; capital X adds x only to folders
chmod a-w codon_table.txt    # read-only for everyone (protect raw data from accidents)
```

The number form uses one digit per set: `r`=4, `w`=2, `x`=1, added up.

Number	Letters	Typical use
755	`rwxr-xr-x`	scripts and folders others may read
644	`rw-r--r--`	ordinary data files
700	`rwx------`	private folders (e.g. `~/.ssh`)
600	`rw-------`	private files (e.g. SSH keys)

Making your raw data read-only is a good habit: then `>` can't overwrite it by mistake.

```
chmod a-w codon_table.txt
echo oops > codon_table.txt
```

```
-bash: codon_table.txt: Permission denied
```

Groups and shared lab folders

Every file belongs to one owner and one **group**. On the HPCC your group is your lab; during this class everyone is in the `gen220` group. See your groups with:

```
groups
id
```

Bigdata space is organized by lab:

- `/bigdata/LABNAME/USERNAME` - your own folder (`~/bigdata` points here)
- `/bigdata/LABNAME/shared` - a folder shared by everyone in the lab

For this class that is `/bigdata/gen220/USERNAME` and `/bigdata/gen220/shared`. Anyone in the group can read a file if it has group `r` permission (and every folder above it has group `x`). To let lab members *change* files, they also need group `w`. HPCC's permission advice suggests:

- keep your own bigdata folder at `u=rwx,g=rx,o=` (the group can read, but not change it)
- don't give group-write to your whole folder; add `g+w` only to the sub-folders you share
- for a session of work in a shared folder, run `umask u=rwx,g=rwx,o=` so new files you create are group-writable (the HPCC recommends *not* putting this in your `.bashrc`)

A few other commands you may meet: `chgrp LABNAME file` changes a file's group, and `chmod g+s folder` makes new files created inside a folder belong to the folder's group. More in the HPCC sharing and permissions manuals.

How much space am I using?

du - disk usage of files and folders

```
du -h ../GEN220_data/genome/S_cerevisiae.fasta.gz    # one file, human readable
du -sh ../GEN220_data                                # -s: one total for the folder
du -sh ../GEN220_data/*                              # a total for each item inside
du -h -d 1 ../GEN220_data | sort -h                  # one level deep, sorted by size
```

```
3.6M    ../GEN220_data/genome/S_cerevisiae.fasta.gz
41M ../GEN220_data
8.0K    ../GEN220_data/LICENSE
4.0K    ../GEN220_data/README.md
14M ../GEN220_data/genome
8.0K    ../GEN220_data/scripts
6.1M    ../GEN220_data/tabular
...
```

(Your numbers may be a little different: `du` measures space used on the disk, which depends on the file system. The `.git` folder, which holds the history, is often the biggest item.) `ls -lh` shows the size of the file contents instead, and `du -ch *.gz` adds up several files with a `total` line.

A common task: "what is filling up my space?"

```
cd ~/bigdata
du -sh * | sort -h | tail -n 5      # the five biggest things in your bigdata folder
```

df and quotas

`df -h FOLDER` shows the size and free space of the whole file system (disk) that a folder is on. On the HPCC these are huge shared file systems, so `df` does not tell you *your* limit. Your limits (quotas) are:

Location	What it's for	Limit
`/rhome/USERNAME` (`~`)	scripts, config files, small things	50 GB per user
`/bigdata/LABNAME/USERNAME`	your data and results	shared lab quota (many TB)
`/bigdata/LABNAME/shared`	data shared by the lab	same lab quota
`/scratch` (`$SCRATCH` in a job)	fast temporary space during a job	deleted after the job ends

Check your usage with the HPCC's `check_quota` command, or on the web at https://dashboard.hpcc.ucr.edu:

```
check_quota home
check_quota bigdata
```

The output is a short table with the space used and the limit (it will look similar to this but with your own numbers). Put big data files in bigdata, not in your home folder. Scratch and other temporary space are covered in UNIX IV. Source: HPCC storage manual.

Redirection and pipes

Standard input, output and error

Every program has three data streams:

Stream	Number	Default	Used for
standard input (STDIN)	0	keyboard	data the program reads
standard output (STDOUT)	1	screen	the program's results
standard error (STDERR)	2	screen	error and progress messages

Because STDOUT and STDERR both go to the screen, they look the same, but you can send them to different places. Here `ls` prints one result (STDOUT) and one error (STDERR):

```
ls ../GEN220/data/codon_table.txt nosuchfile.txt
```

```
ls: cannot access 'nosuchfile.txt': No such file or directory
../GEN220/data/codon_table.txt
```

Syntax	Meaning
`cmd > file`	STDOUT to a file (create it, or **overwrite** it if it exists)
`cmd >> file`	STDOUT appended to the end of a file
`cmd < file`	STDIN read from a file
`cmd 2> file`	STDERR to a file
`cmd > out.txt 2> err.txt`	results and errors in separate files
`cmd > file 2>&1`	STDERR to the same place as STDOUT (both in one file)
`cmd &> file`	bash shortcut for the same thing
`cmd 2> /dev/null`	throw error messages away (`/dev/null` discards everything)

And `cmd1 | cmd2`, the **pipe**, sends STDOUT of `cmd1` to STDIN of `cmd2` (below).

```
ls ../GEN220/data/codon_table.txt nosuchfile.txt > out.txt
cat out.txt                      # only the result; the error still went to the screen
ls ../GEN220/data/codon_table.txt nosuchfile.txt > out.txt 2> err.txt
cat err.txt                      # the error message is here
ls ../GEN220/data/codon_table.txt nosuchfile.txt > all.txt 2>&1
cat all.txt                      # both
```

```
../GEN220/data/codon_table.txt
ls: cannot access 'nosuchfile.txt': No such file or directory
ls: cannot access 'nosuchfile.txt': No such file or directory
../GEN220/data/codon_table.txt
```

Notes:

- In `> file 2>&1` the order matters: `2>&1 > file` sends errors to the screen, not the file.
- `>` empties the file *before* the command runs, so `sort data.txt > data.txt` destroys `data.txt`. Always write to a new file name.
- Many bioinformatics programs write their results to STDOUT and log messages to STDERR, so `program input.fa > results.txt 2> program.log` keeps the results clean.
- `cat` plus `>` combines files: `cat part1.fa part2.fa > all.fa` writes one file after the other into `all.fa`. For example, one file with all the Valine and Leucine codons: `grep Valine ../GEN220/data/codon_table.txt > val.txt`, then `grep Leucine ../GEN220/data/codon_table.txt > leu.txt`, then `cat val.txt leu.txt > VL.txt` (10 lines). `cat` works on `.gz` files too: `cat a.fq.gz b.fq.gz > both.fq.gz` is a valid gzip file.
- `<` makes a program read a file as its STDIN. For example `wc -l < file` prints only the number (no file name), which is handy in scripts:

```
wc -l ../GEN220/data/codon_table.txt
wc -l < ../GEN220/data/codon_table.txt
```

```
64 ../GEN220/data/codon_table.txt
64
```

Pipes

A pipe `|` sends the output of one program straight into the next, without a temporary file. Small tools joined by pipes are the heart of UNIX data processing:

```
ls ../GEN220_data/genome | wc -l              # how many files in genome/?
zcat ../GEN220_data/genome/S_cerevisiae.fasta.gz | grep '>' | head -n 3  # 3 chromosomes
zcat ../GEN220_data/genome/S_cerevisiae.ORFs.fasta.gz | grep -c '>'     # ORF count
history | grep git                            # which git commands did I run?
grep --help | less                            # page through long output (q quits)
```

```
10
>chrI
>chrII
>chrIII
6713
```

Only STDOUT goes through the pipe; errors still go to the screen. To send both, use `cmd 2>&1 | less` (or `cmd |& less` in bash). You'll build much longer pipelines with `sort`, `cut`, `uniq` and `awk` in the UNIX III data processing lab.

tee - save a copy and keep going

`tee FILE` copies its STDIN to a file **and** to STDOUT, like a T-joint in a pipe. Use it to save an intermediate result, or to watch a program's output while also keeping a log.

```
grep Leucine ../GEN220/data/codon_table.txt | tee leucine_codons.txt | wc -l
cat leucine_codons.txt
```

```
6
CTT L   Leucine
CTC L   Leucine
CTA L   Leucine
CTG L   Leucine
TTA L   Leucine
TTG L   Leucine
```

```
./run_analysis.sh 2>&1 | tee analysis.log      # see messages now AND keep them
command | tee -a analysis.log                  # -a appends instead of overwriting
```

Searching inside files: grep

`grep PATTERN FILE` prints every line of the file that contains the pattern.

```
grep Valine ../GEN220/data/codon_table.txt
```

```
GTT V   Valine
GTC V   Valine
GTA V   Valine
GTG V   Valine
```

The most useful options to start with:

Option	Meaning
`-c`	count the matching lines instead of printing them
`-i`	ignore upper/lower case
`-n`	show the line number of each match
`-v`	invert: lines that do **not** match
`-w`	match whole words only

```
grep -c Valine ../GEN220/data/codon_table.txt        # 4
grep -c valine ../GEN220/data/codon_table.txt        # 0 - grep is case sensitive
grep -c -i valine ../GEN220/data/codon_table.txt     # 4
grep -n Stop ../GEN220/data/codon_table.txt          # which lines are stop codons?
```

```
4
0
4
62:TAA  *   Stop
```

```
63:TAG  *   Stop
64:TGA  *   Stop
```

Be careful what a pattern matches. `rice_random_exons.bed` has 1000 exons on the 12 rice chromosomes. Searching for `Chr1` also matches `Chr10`, `Chr11` and `Chr12`; `-w` fixes that:

```
grep -c Chr1 ../GEN220/data/rice_random_exons.bed
grep -c -w Chr1 ../GEN220/data/rice_random_exons.bed
grep -c -v -w Chr1 ../GEN220/data/rice_random_exons.bed
```

```
328
146
854
```

Put the pattern in single quotes when it has spaces or special characters, such as `'>'` (without quotes the shell would treat `>` as redirection and overwrite a file!). For compressed files, use `zgrep` or `zcat | grep`. Plain `grep` on a `.gz` file searches the compressed bytes and gives a meaningless answer:

```
zgrep -c '>' ../GEN220_data/genome/S_cerevisiae.ORFs.fasta.gz
zgrep -w RAD51 ../GEN220_data/genome/S_cerevisiae.ORFs.fasta.gz
```

```
6713
>YER095W RAD51 SGDID:S000000897, Chr V from 349980-351182, Genome Release 64-2-1
```

Patterns can also be *regular expressions*: `grep '^>'` matches `>` only at the start of a line, and `grep -E` allows more complex patterns. More `grep` options and regular expressions are in the UNIX III lab; regular expressions in Python come in Python V.

Compressing, downloading and transferring data

Compression

Most biological data files are plain text (FASTA, FASTQ, GFF, VCF, BLAST tables) and text compresses very well. Compressing saves disk space (your lab pays for it on the cluster) and makes copying files between computers faster. **Keep large data files compressed** and read them compressed - most tools and our own scripts can do this.

gzip: the standard

`gzip` is by far the most common format; files end in `.gz`.

```
gzip file.fa              # compress: replaces file.fa with file.fa.gz
gunzip file.fa.gz         # uncompress: replaces file.fa.gz with file.fa
gzip -k file.fa           # -k keeps the original file too
gzip -c file.fa > copy.fa.gz    # -c writes to STDOUT, so you choose the output name
gzip -t file.fa.gz        # test that a compressed file is complete and not corrupted
```

Note that `gzip` and `gunzip` **replace** the input file. That surprises people the first time.

Reading compressed files without uncompressing them

`zcat` prints an uncompressed copy to STDOUT (the screen or a pipe) and leaves the `.gz` file alone. There are `z` versions of other tools too:

```
zcat file.fa.gz | head -n 4         # the first 4 lines
zless file.fa.gz                    # page through it (like less)
zgrep -c '^>' file.fa.gz            # count the FASTA headers (like grep)
zcat *.gz | wc -l                   # total lines in several compressed files
```

Try it on the class data:

```
zcat ../GEN220_data/genome/S_cerevisiae.gff3.gz | head -n 2
zgrep -c '^>' ../GEN220_data/genome/S_cerevisiae.fasta.gz   # 17 (16 + chrmt)
```

```
##gff-version 3
#date Tue Jan 13 13:06:13 2015
17
```

(On a Mac, `zcat` only works on `.Z` files; use `gzcat` or `zcat < file.gz` instead.)

Programs can also *write* compressed output by piping to `gzip`:

```
blastn -query query.fa -db db.fa -outfmt 6 | gzip -c > blastresult.tsv.gz
```

Many bioinformatics tools (BWA, minimap2, samtools, most aligners and assemblers) read `.gz` input directly, and so can Python (`gzip.open(filename, "rt")`), pandas, R (`readr::read_csv`), and DuckDB. There is usually no need to uncompress.

Other formats

Here is how the yeast ORF file (11.5 MB FASTA) compresses with each tool:

Tool	Extension	Compressed size	Speed	Notes
`gzip`	`.gz`	3.8 MB (33%)	medium	works everywhere; the default choice
`pigz -p 4`	`.gz`	3.8 MB (33%)	fast	gzip using several CPUs, same file format
`bgzip`	`.gz`	3.8 MB (33%)	fast	gzip in blocks, can be indexed (see below)
`bzip2`	`.bz2`	3.1 MB (27%)	slow to uncompress	`bzcat`, `pbzip2` for multiple CPUs
`xz`	`.xz`	2.8 MB (25%)	very slow to compress	smallest files; `xzcat`
`zstd`	`.zst`	3.6 MB (31%)	very fast	modern; `zstdcat`, `zstd -d`

- On the cluster use `pigz` instead of `gzip` for big files: `pigz -p 4 file.fastq` (and ask for 4 CPUs in your job).
- Already compressed files don't get smaller: re-compressing a `.gz` file, a BAM file, or PNG/JPEG images gains nothing (our `.gz` file actually got a little bigger).
- `.zip` files (common from Windows and websites) are different: list the contents with `unzip -l file.zip` and extract with `unzip file.zip`.

Archives: tar

`gzip` compresses **one file**. To bundle a whole folder into one file (e.g. to download or copy a project), use `tar` to make an *archive*, usually compressed at the same time (`.tar.gz` or `.tgz`):

```
tar -czf project.tar.gz project/     # c = create, z = gzip it, f = file name to write
tar -tzf project.tar.gz              # t = list what is inside, without extracting
```

```
tar -xzf project.tar.gz              # x = extract into the current folder
tar -xzf project.tar.gz -C /tmp      # extract somewhere else
```

Always list (`-t`) an archive you downloaded before extracting it, to see what it will create.

bgzip and indexes: jumping into the middle of a file

A normal `.gz` file has to be read from the beginning. `bgzip` (from htslib, `module load samtools` or `htslib`) writes gzip-compatible files in independent blocks, so an **index** can point to where each part of the file is. This lets tools pull out one region of a large sorted file instantly:

```
bgzip genes.bed                      # makes genes.bed.gz (zcat can still read it)
tabix -p bed genes.bed.gz            # index: makes genes.bed.gz.tbi (must be sorted)
tabix genes.bed.gz chr1:150-600      # print only features overlapping this region
```

The same idea is used by `samtools faidx` on bgzipped FASTA files, VCF files (`.vcf.gz` + `.tbi`), and BAM files, which are bgzip-compressed internally. You'll see this again in the variant calling lectures.

Downloading data

Most data you analyze will come from a web or FTP site (NCBI, Ensembl, UniProt, a sequencing center). Download it **directly to the cluster** instead of to your laptop and then uploading it: it's much faster and saves a step.

curl

`curl URL` prints what it downloads to STDOUT (the screen), so you normally add an option to save it to a file:

```
# -O: save with the name from the URL (P04637.fasta)
curl -L -O https://rest.uniprot.org/uniprotkb/P04637.fasta
# -o: choose the name
curl -L -o TP53_human.fasta https://rest.uniprot.org/uniprotkb/P04637.fasta
# the same, using redirection
curl -L https://rest.uniprot.org/uniprotkb/P04637.fasta > TP53_human.fasta
head -n 2 TP53_human.fasta
```

```
>sp|P04637|P53_HUMAN Cellular tumor antigen p53 OS=Homo sapiens OX=9606 GN=TP53 PE=1 SV=4
MEEPQSDPSVEPPLSQETFSDLWKLLPENNVLSPLPSQAMDDLMLSPDDIEQWFTEDPGP
```

Always use `-L` (follow redirects). Web sites move pages and send a "redirect" to the new address; without `-L` curl saves the empty redirect message instead. For example, UniProt's old addresses like `https://www.uniprot.org/uniprot/P04637.fasta` now redirect to `https://rest.uniprot.org/uniprotkb/P04637.fasta`:

```
curl -s -O https://www.uniprot.org/uniprot/P04637.fasta     # no -L
ls -l P04637.fasta                                           # 0 bytes!
```

Other useful `curl` options:

Option	Meaning
`-L`	follow redirects
`-O`	save with the file name from the URL
`-o NAME`	save as NAME
`-s`	silent: no progress meter
`-f`	fail on server errors (e.g. 404 Not Found) instead of saving the error page
`-C -`	continue a download that was interrupted

wget

`wget URL` saves to a file named from the URL, follows redirects automatically, and can resume (`-c`) or download many files. If the file already exists, `wget` saves the new one as `FILE.1` instead of replacing it.

```
wget https://rest.uniprot.org/uniprotkb/P04637.fasta
wget -O TP53_human.fasta https://rest.uniprot.org/uniprotkb/P04637.fasta  # choose name
wget -c https://example.org/big_file.fastq.gz                  # resume a download
```

Did it download correctly? Checksums

Large downloads can be cut off or corrupted without any obvious error. Data providers publish a **checksum** for each file: a short "fingerprint" computed from every byte of the file. If you compute the same fingerprint for your copy and it matches, the file is identical. The common kinds are MD5 (`md5sum`) and SHA-256 (`sha256sum`).

Let's download the yeast genome from NCBI along with its checksum file:

```
URL=https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/146/045/GCF_000146045.2_R64
curl -L -O $URL/GCF_000146045.2_R64_genomic.fna.gz
curl -L -O $URL/md5checksums.txt
md5sum GCF_000146045.2_R64_genomic.fna.gz
grep R64_genomic.fna.gz md5checksums.txt       # the published checksum
```

```
88c38b957b721dfc50e6c4df03b6242e  GCF_000146045.2_R64_genomic.fna.gz
88c38b957b721dfc50e6c4df03b6242e  ./GCF_000146045.2_R64_genomic.fna.gz
```

The fingerprints match. `md5sum -c` does the comparison for you, for every file in a checksum list; `--ignore-missing` skips the files in the list that you didn't download:

```
md5sum -c --ignore-missing md5checksums.txt
```

```
./GCF_000146045.2_R64_genomic.fna.gz: OK
```

A damaged file reports `FAILED`: download it again. `sha256sum` works the same way. For `.gz` files, `gzip -t file.gz` is another quick check that the file is complete. When you copy important data between computers, you can make your own checksum list the same way: `md5sum *.fastq.gz > md5sums.txt` before, and `md5sum -c md5sums.txt` after.

Transferring files between your computer and the cluster

Run these commands **on your laptop** (in Terminal on a Mac, or MobaXterm or PowerShell on Windows), not on the cluster. Your laptop can reach the cluster, but the cluster can't reach your laptop. Replace `USERNAME` with your HPCC user name. If you set up the `hpcc` host alias from the SSH keys guide, you can write `hpcc:` instead of `USERNAME@cluster.hpcc.ucr.edu:`.

scp - secure copy

`scp SOURCE DESTINATION` works like `cp`, but either side can be on another computer, written as `USERNAME@HOST:PATH`. A path after the `:` without a leading `/` is relative to your home folder on the cluster.

```
# laptop -> cluster: put a file in your bigdata folder
scp mydata.csv USERNAME@cluster.hpcc.ucr.edu:bigdata/gen220/
# cluster -> laptop: copy a result into the current folder (.) on your laptop
scp USERNAME@cluster.hpcc.ucr.edu:bigdata/gen220/unix2/leucine_codons.txt .
# a whole folder needs -r
scp -r USERNAME@cluster.hpcc.ucr.edu:bigdata/gen220/unix2 .
```

rsync - copy only what changed

`rsync` is better for folders and big files: it can show progress, it only sends files that are new or changed, and if the copy is interrupted you just run the same command again.

```
rsync -av --progress results/ USERNAME@cluster.hpcc.ucr.edu:bigdata/gen220/results/
rsync -av USERNAME@cluster.hpcc.ucr.edu:bigdata/gen220/unix2/ unix2_copy/
rsync -av --dry-run results/ backup/     # -n / --dry-run: show what would be copied
```

- `-a` (archive) copies folders recursively and keeps times and permissions; `-v` lists files.
- **The trailing slash on the source matters.** `rsync -a results/ dest/` copies the *contents* of `results` into `dest`. `rsync -a results dest/` makes `dest/results/`.
- `rsync` also copies between two folders on the same computer, e.g. to make a backup.

Other options

- `sftp USERNAME@cluster.hpcc.ucr.edu` opens an interactive session (`put`, `get`, `ls`, `cd`).
- Graphical programs: FileZilla, Cyberduck (Mac), WinSCP (Windows), or the file browser built into MobaXterm. Connect with protocol SFTP to `cluster.hpcc.ucr.edu`, port 22.
- Very large transfers (hundreds of GB) are best done with Globus.

If you log in with a password, each new connection also asks for Duo two-factor approval. Setting up SSH keys avoids typing your password and makes `scp` and `rsync` much less tedious (see the HPCC login manual).

Running programs

Processes: what is running, and stopping it

Every running program is a **process** with a number (the PID). By default a command runs in the **foreground**: your prompt comes back only when it finishes.

Keys / command	What it does
`Ctrl-C`	stop (kill) the foreground program
`Ctrl-Z`	pause (suspend) the foreground program
`bg`	continue the paused program in the **background**
`fg`	bring a background program back to the foreground
`command &`	start a program in the background right away
`jobs`	list the programs started from this terminal
`ps`	list your processes and their PIDs
`top` / `htop`	live view of processes using CPU and memory (`q` quits)
`kill PID`	ask a process to stop; `kill -9 PID` forces it
`time command`	run a command and report how long it took

Try it with `sleep`, a program that just waits:

```
sleep 60
```

Press `Ctrl-Z`:

```
^Z
[1]+  Stopped                 sleep 60
```

```
bg          # keep running, in the background
jobs
fg          # back in the foreground; now Ctrl-C stops it
```

```
[1]+ sleep 60 &
[1]+  Running                 sleep 60 &
```

```
sleep 60
^C
```

```
sleep 300 &            # start in the background; prints the job number and PID
ps -u $USER            # all your processes
kill 3817              # use the PID that ps showed you
```

```
[1] 3817
    PID TTY          TIME CMD
   3521 pts/12   00:00:00 bash
   3817 pts/12   00:00:00 sleep
   3822 pts/12   00:00:00 ps
```

(Your PIDs will be different.) `top -u $USER` shows only your processes; `htop` is a friendlier version if it is installed.

Background jobs belong to your terminal: if you log out or lose your connection, they are usually killed. For long work, submit a batch job (below and UNIX IV), or use `tmux` to keep a session alive (also in UNIX IV).

The login node is not for computing

When you `ssh` to `cluster.hpcc.ucr.edu` you land on a **login node** (head node) that is shared by everyone who is logged in. It is for editing files, small tests, downloads, and submitting jobs. The real work runs on the cluster's **compute nodes**: more than a hundred machines with over 5,000 CPU cores in total, GPUs, and up to 3 TB of memory on one node (see the HPCC hardware summary). The HPCC rule: don't run computationally intensive tasks on the login nodes; "your process will either run very slow or be killed automatically" (Getting started).

Two ways to get onto a compute node, both through the **SLURM** job scheduler:

- `srun` - an **interactive** session: you get a shell on a compute node and type commands (next section)
- `sbatch` - a **batch** job: you write a script and SLURM runs it when resources are free, even after you log out (short intro below; full details in UNIX IV)

Software modules

The cluster has hundreds of programs installed, often in several versions. To keep them from conflicting, most are not available until you **load** a module, which adds the program to your `PATH` (the list of folders the shell searches for commands; see it with `echo $PATH`). `which PROGRAM` shows where the shell finds a program:

```
which blastn
```

```
/usr/bin/which: no blastn in (/usr/local/bin:/usr/bin:...)
```

```
module avail ncbi-blast       # which versions are installed?
```

```
------------------- /opt/linux/rocky/8.x/x86_64/modules -------------------
ncbi-blast/2.2.22+  ncbi-blast/2.2.30+  ncbi-blast/2.4.0+  ncbi-blast/2.9.0+
ncbi-blast/2.2.25+  ncbi-blast/2.2.31+  ncbi-blast/2.5.0+  ncbi-blast/2.11.0+
...
ncbi-blast/2.14.1+  ncbi-blast/2.16.0+  ncbi-blast/2.17.0+
```

(The exact list and layout will look similar but may differ; the cluster is regularly updated.)

```
module load ncbi-blast            # load the default version
which blastn
blastn -version
module list                       # what is loaded now?
```

```
/opt/linux/rocky/8.x/x86_64/pkgs/ncbi-blast/2.17.0+/bin/blastn
blastn: 2.17.0+
...
```

```
Currently Loaded Modulefiles:
 1) slurm/24.11.1   2) ncbi-blast/2.17.0+   ...
```

(Again, your paths and versions may be different.)

Command	What it does
`module avail`	list all modules (long!); `module avail NAME` lists one program's versions
`module load NAME`	load the default version
`module load NAME/VERSION`	load a specific version, e.g. `module load samtools/1.19.2`
`module list`	show what is loaded
`module unload NAME`	remove one module
`module purge`	unload everything
`module show NAME`	show what a module changes (e.g. which folders go on `PATH`)

Tips:

- **Record versions.** Results can change between versions of a program. For your homework and projects, load a specific version (`module load ncbi-blast/2.16.0+`) in your scripts, and write down what you used. `module list` in a job's log is an easy record.
- Modules only last for the current session. Put `module load` lines in your scripts (and job scripts) rather than relying on what you loaded by hand.
- Some modules set other variables too. For example `module load db-ncbi` points the `BLASTDB` variable at pre-downloaded NCBI BLAST databases, so you don't have to download them.
- `module purge` also unloads modules the cluster loads for you when you log in (such as `slurm`, which provides `sbatch` and `squeue`). If those commands disappear, log out and back in.
- If you need software that isn't installed as a module, you can install it yourself with conda (see UNIX IV) or ask support@hpcc.ucr.edu. HPCC's package management page has more.

Running a program on a compute node with srun

`srun` asks SLURM for resources and runs a command on a compute node. With `--pty bash -l` the command is a new login shell, so you get an interactive session:

```
srun -p short -c 2 --mem 4G --time 1:00:00 --pty bash -l
```

Option	Meaning
`-p short`	the **partition** (group of nodes) to use; `short` allows jobs up to 2 hours
`-c 2`	number of CPU cores (long form: `--cpus-per-task 2`)
`--mem 4G`	amount of memory
`--time 1:00:00`	time limit (hours:minutes:seconds); the session ends when it runs out
`--pty bash -l`	run an interactive login shell

It may take a few seconds (or longer if the cluster is busy) before you get a prompt. The prompt then shows the name of a compute node (such as `c05` or `r21`) instead of the login node. Now run your program there. For example, search a human protein against the SwissProt database with BLAST (you'll learn what this means in the BLAST lecture):

```
hostname                                  # which node am I on?
cd ~/bigdata/gen220/unix2
module load ncbi-blast
module load db-ncbi                       # sets BLASTDB to the NCBI databases
curl -L -O https://rest.uniprot.org/uniprotkb/Q5T6X5.fasta
blastp -num_threads 2 -query Q5T6X5.fasta -db swissprot -outfmt 6 -max_target_seqs 5 \
    > Q5T6X5_vs_swissprot.tsv
head Q5T6X5_vs_swissprot.tsv
exit                                      # end the session and give the resources back
```

Notes:

- Ask for what you need. `-c 2` matches `-num_threads 2`: a program only uses the cores you asked for, and bigger requests wait longer in the queue.
- Your files are the same on every node (home and bigdata are shared), so you can edit on the login node and run on a compute node.
- The main partitions are `epyc`, `intel` (the default), `batch`, `short` (2 hour limit), `highmem` and `gpu`. See the HPCC jobs manual and queue policies for their limits.
- `srun` stops when you log out. For anything long, use a batch job.

Batch jobs in one minute

A batch job is a script with your commands, plus `#SBATCH` lines that request resources. Save this as `blast_job.sh`:

```
#!/bin/bash -l
#SBATCH -p short -c 2 --mem 4G --time 1:00:00
#SBATCH --output blast_job.log
module load ncbi-blast
module load db-ncbi
blastp -num_threads 2 -query Q5T6X5.fasta -db swissprot -outfmt 6 \
    > Q5T6X5_vs_swissprot.tsv
```

```
sbatch blast_job.sh           # submit it; prints "Submitted batch job NUMBER"
squeue -u $USER               # is it waiting (PD) or running (R)?
```

That's all you need for now. Writing job scripts, choosing resources, checking finished jobs (`sacct`, `seff`), cancelling (`scancel`), and running hundreds of jobs with job arrays are covered in UNIX IV.

Git: turning in your homework

Homework is turned in through GitHub Classroom: each assignment creates a Git repository for you, you **clone** it to the cluster, do the work, and **commit** and **push** it back. This section is the minimum you need for Homework 1. The Git and GitHub guide covers everything else: one-time setup, SSH keys and tokens, `.gitignore`, undoing mistakes, and working from two computers.

One time only (on the cluster), tell Git who you are:

```
git config --global user.name "Your Name"
git config --global user.email "yourname@ucr.edu"     # the email on your GitHub account
```

Get your homework repository. Accept the assignment link in Canvas, then copy the URL from the green **Code** button on your repository's GitHub page:

```
cd ~/bigdata/gen220
git clone git@github.com:biodataprog/2026-hw1-YOURGITHUBID.git    # with an SSH key
# or: git clone https://github.com/biodataprog/2026-hw1-YOURGITHUBID.git   (token)
cd 2026-hw1-YOURGITHUBID
ls
```

GitHub does not accept your password on the command line. Use an SSH key or a personal access token; see Logging in.

The cycle: edit, add, commit, push.

```
nano filesize.sh              # edit (or use VS Code / Jupyter)
git status                    # what has changed?
git add filesize.sh           # choose the files to save
git commit -m "Homework 1: report file sizes"   # save a snapshot, with a message
git push                      # send it to GitHub
```

```
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    filesize.sh

nothing added to commit but untracked files present (use "git add" to track)
```

```
[main 588de6a] Homework 1: report file sizes
 1 file changed, 2 insertions(+)
 create mode 100644 filesize.sh
```

- `git status` is always safe to run; it tells you what to do next.
- A commit is saved only in your copy on the cluster until you `git push`.
- You can commit and push as many times as you like. **The last push before the deadline is what gets graded**, so check your repository page on GitHub to see that your files are there.
- Commit your scripts, not big data files or results you can regenerate.

Quick reference

Task	Command
copy a file / a folder	`cp file dest` / `cp -r folder dest`
move or rename	`mv old new`
delete a file / a folder	`rm file` / `rm -r folder` (no undo!)
ask before overwriting or deleting	add `-i` to `cp`, `mv`, `rm`
many files at once	`*.txt`, `chr?.bed`, `[cn]*`, `*.{txt,bed}`
make a symbolic link	`ln -s TARGET LINKNAME`
where does a link point?	`readlink link`, `realpath link`
long listing	`ls -lh`, `ls -la` (with hidden files)
change permissions	`chmod +x script.sh`, `chmod g+r file`, `chmod 755 folder`
my groups	`groups`, `id`
size of a file or folder	`du -sh path`
my quota	`check_quota home`, `check_quota bigdata`
output to a file / append	`cmd > file` / `cmd >> file`
errors to a file	`cmd 2> err.txt`; both: `cmd > all.txt 2>&1` or `cmd &> all.txt`
discard errors	`cmd 2> /dev/null`
search a file	`grep pattern file`; `-c` count, `-i` any case, `-n` line numbers, `-v` invert, `-w` word
search a compressed file	`zgrep pattern file.gz`
compress / uncompress	`gzip file` / `gunzip file.gz`; read with `zcat`, `zless`
archive a folder	`tar -czf out.tar.gz folder/`; list `-tzf`; extract `-xzf`

Task	Command
download	`curl -L -O URL`, `wget URL`
check a download	`md5sum file`, `md5sum -c checksums.txt`, `sha256sum file`
laptop <-> cluster	`scp file USER@cluster.hpcc.ucr.edu:path`, `rsync -av src/ USER@...:dest/`
running programs	`jobs`, `ps -u $USER`, `top`, `kill PID`, `Ctrl-C`, `Ctrl-Z`, `bg`, `fg`
software	`module avail NAME`, `module load NAME/VERSION`, `module list`, `module purge`
interactive compute node	`srun -p short -c 2 --mem 4G --time 1:00:00 --pty bash -l`
submit a batch job	`sbatch job.sh`, check with `squeue -u $USER`
homework	`git clone URL`, `git status`, `git add f`, `git commit -m "msg"`, `git push`

Pipes: `cmd1 | cmd2` sends the output of `cmd1` into `cmd2`; `cmd1 | tee file | cmd2` also saves a copy in `file`.

Practice exercises

Work in `~/bigdata/gen220/unix2`.

1. **Organize.** Make the folders `ex/data`, `ex/scripts` and `ex/results` with one `mkdir` command (hint: `-p` and `{}`). Copy every `.txt` file from `../GEN220/data` into `ex/data` with one `cp` command. Then rename `ex/data/numbers.txt` to `ex/data/random_numbers.txt`.
2. **Links.** In `ex/data`, make a symbolic link called `yeast.gff3.gz` that points to the yeast GFF3 file in `GEN220_data/genome` (use an absolute path; `realpath` can tell you one). Show that it works by printing its first 3 lines with `zcat`. What does `ls -l` show?
3. **Wildcards.** Using one `ls` command each, list (a) all the files in `GEN220_data/genome` that start with `E`, (b) only the `.pep.gz` files, and (c) the two `E_coli` protein files using `{}`. How many proteins are in all the `.pep.gz` files together?
4. **Permissions.** Make `ex/data/codon_table.txt` read-only for everyone and show that `echo test >> ex/data/codon_table.txt` now fails. Make `ex/results` readable by your group but not by others, and check with `ls -ld ex/results`.
5. **Redirection.** Run `ls ../GEN220/data/codon_table.txt missing.txt` so that the normal output goes to `ls_out.txt` and the error goes to `ls_err.txt`. Then run it again to put both in `ls_all.txt`. Then run it so that you see only the normal output on the screen.
6. **grep.** How many codons in `codon_table.txt` code for Serine? Which line numbers are they on? How many lines of `rice_random_exons.bed` are on chromosome 12 (not 1)? How many sequences are in `GEN220_data/genome/S_enterica.pep.gz`?
7. **Download and check.** Download the yeast protein file `GCF_000146045.2_R64_protein.faa.gz` and `md5checksums.txt` from the NCBI folder used in Did it download correctly?. Check it with `md5sum -c --ignore-missing`, and count the proteins with `zgrep -c`.
8. **Compute node, modules and Git.** Start an interactive session on the `short` partition with 1 CPU and 2 GB of memory for 30 minutes. Run `hostname`, load `samtools`, and print its version (`samtools --version | head -n 1`). Exit. Then write a script `ex/scripts/sizes.sh` that prints the size of every file in `../GEN220_data/genome` sorted by size (hint: `du -h`, `sort -h`). If you have cloned your HW1 repository, practice the `add`, `commit`, `push` cycle with a test file and check that it appears on GitHub.