Further Readings for Week 2

`wget` and `curl`

While we are familiar with retrieving webpages and downloading files from the internet through the browser, `wget` and `curl` commands achieve similar tasks but through a terminal. They are used to make HTTP requests to retrieve files from the internet. They are very useful as we'll frequently encounter situations where we have to download files from databases like NCBI SRA, Genbank, UniProt etc. Since the file sizes we as Bioinformaticians deal with are typically huge, it is often impossible to analyze these files on local machines and hence we make use of High Performance Computing(HPC) platforms like PACE, AWS, Azure and so on. `wget` and `curl` make it easy for us to download the required files to the remote systems that we are working on. Both these command-line tools can be used for our purposes but `curl` is a more advanced tool and supports additional functionalities compared to `wget`.

To demonstrate these commands, we will download the reference genome for Pseudomonas *aeruginosa* PAO1 from NCBI found here.

```
$ wget
https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/006/765/GCF_000006765.1_ASM
676v1/GCF_000006765.1_ASM676v1_genomic.fna.gz -O
./data/PAO1_reference.fna.gz
--2024-08-30 19:54:42--
https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/006/765/GCF_000006765.1_ASM
676v1/GCF_000006765.1_ASM676v1_genomic.fna.gz
Resolving ftp.ncbi.nlm.nih.gov (ftp.ncbi.nlm.nih.gov)... 130.14.250.12,
130.14.250.31, 130.14.250.7, ...
Connecting to ftp.ncbi.nlm.nih.gov
(ftp.ncbi.nlm.nih.gov)|130.14.250.12|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1787220 (1.7M) [application/x-gzip]
Saving to: ‘./data/PAO1_reference.fna.gz’

./data/PAO1_reference.fna.gz                 100%
[===========================================================================
==============================>]   1.70M  2.42MB/s    in 0.7s

2024-08-30 19:54:43 (2.42 MB/s) - ‘./data/PAO1_reference.fna.gz’ saved
[1787220/1787220]

$ ls data/
PAO1_reference.fna.gz
```

`-O` option is used to specify the path where the file is downloaded and also rename the downloaded file. Without `-O`, the file is downloaded to the current directory.

```
$ curl
https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/006/765/GCF_000006765.1_ASM
676v1/GCF_000006765.1_ASM676v1_genomic.fna.gz -o data/PAO1_reference.fna.gz
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100 1745k  100 1745k    0     0  1808k      0 --:--:-- --:--:-- --:--:--
1808k

$ ls data/
PAO1_reference.fna.gz
```

Similarly, for `curl`, we use `-o` to specify the path for downloading the file as well as rename the downloaded file.

`md5sum`

The command is used to check the integrity of a file. Once we download a file using `wget` or `curl` as shown above, how do we verify the downloaded file is not corrupted? We can use `md5sum` to calculate the **MD5 checksum**.

Before we get into the command, we will first try to understand a bit about MD5 and hashing. MD5 or Message Digest 5 is a hashing algorithm that takes the content of the file as input and converts it to a string of alphanumeric characters or a **hash**. An important feature of a hashing algorithm is that it always gives the same hash for the same, unchanged input. However, if there is even a single byte or character change in the input, the hash changes. It is this property of hashing algorithms we use to check the integrity of the files. And, it is important to do so because incomplete or corrupt data will significantly impact the insights we draw from the data and it can be tricky to trace back the root cause for these types of issues.

Let's see a demo illustrating this -

```
$ echo "I love tea" > text_file1.txt
$ md5sum text_file.txt
927fbfe887a7732cf222461bfbaab663  text_file1.txt

$ echo "I love tea" > text_file2.txt
$ md5sum text_file2.txt
927fbfe887a7732cf222461bfbaab663  text_file2.txt   <-- same hash as above

$ echo "I love sea" > text_file.txt
$ md5sum text_file.txt
addb17710e7fb287247112e9f4cc2609  text_file.txt    <-- different hash
```

As you can see, the hashes generated on files with same content is always the same. However, even a single letter change from "tea" to "sea" completely changes the hash.

Now let's see a similar demo on a file containing DNA sequences:

```
$ echo -e ">header1\nATCGTATATAGCAAGATGAC\nATGACATATAGATGACACAT" >
sequence.fna
```

```
$ cat sequence.fna
>header1
ATCGTATATAGCAAGATGAC
ATGACATATAGATGACACAT
$ md5sum sequence.fna
bb3030529a93f68efe0b449a9f24f10e  sequence.fna

$ sed -i '0,/G/{s/A/T/}' sequence.fna #change the first occurrence of 'A' to
'T'
$ cat sequence.fna
>header1
TTCGTATATAGCAAGATGAC
ATGACATATAGATGACACAT
$ md5sum sequence.fna
2af208c926320c772d65ba4697b54fc9  sequence.fna
```

Again, we see that even with a single base change, the hash generated is completely different. If you're curious to know the syntax used with `sed` command above, you'll find this thread interesting.

Let's now apply `md5sum` to the reference file we downloaded earlier to find the checksum:

```
$ md5sum PAO1_reference.fna.gz
c859ec5a25506367506bce972788aca8  PAO1_reference.fna.gz
```

Note that most databases from where we download such data also contain a file that provides corresponding checksums. In this case, such a file is present in the same folder as the reference file we downloaded earlier.

```
$ wget
https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/006/765/GCF_000006765.1_ASM
676v1/md5checksums.txt -O ./data/md5_checksums.txt
--2024-08-30 23:29:18--
https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/006/765/GCF_000006765.1_ASM
676v1/md5checksums.txt
Resolving ftp.ncbi.nlm.nih.gov (ftp.ncbi.nlm.nih.gov)... 130.14.250.11,
130.14.250.31, 130.14.250.12, ...
Connecting to ftp.ncbi.nlm.nih.gov
(ftp.ncbi.nlm.nih.gov)|130.14.250.11|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1094 (1.1K) [text/plain]
Saving to: ‘./data/md5_checksums.txt’

./data/md5_checksums.txt                        100%
[============================================================================
================================>]   1.07K  --.-KB/s    in 0s

2024-08-30 23:29:18 (19.5 MB/s) - ‘./data/md5_checksums.txt’ saved
[1094/1094]

$ cat data/md5_checksums.txt #contains md5 checksums for all files in that
```

```
folder
1d74549ecdc78c5c47d882eb82986400  ./annotation_hashes.txt
b97f852090fd9d2eb2959f26c83be082
./GCF_000006765.1_ASM676v1_assembly_report.txt
1864434828c4a0ccd160409842e29494
./GCF_000006765.1_ASM676v1_assembly_stats.txt
4fb91265652d8d89a43a4b0ed3b777d1
./GCF_000006765.1_ASM676v1_cds_from_genomic.fna.gz
e60c6b677f2646d918419b8930217679
./GCF_000006765.1_ASM676v1_feature_count.txt.gz
3690ff1f459da05c9b5bc956d00bc581
./GCF_000006765.1_ASM676v1_feature_table.txt.gz
c859ec5a25506367506bce972788aca8  ./GCF_000006765.1_ASM676v1_genomic.fna.gz
f95d94d13de3ced99021a88158369370  ./GCF_000006765.1_ASM676v1_genomic.gbff.gz
4cabca061c8da3de375bc80ee19101c2  ./GCF_000006765.1_ASM676v1_genomic.gff.gz
3864488c899199f6f7a731cf1a2ae48d  ./GCF_000006765.1_ASM676v1_genomic.gtf.gz
8266b915e806dcf376d4c8322ac0306d  ./GCF_000006765.1_ASM676v1_protein.faa.gz
2e23e3e5d5e4f0bb19f3a2549099502f  ./GCF_000006765.1_ASM676v1_protein.gpff.gz
5bf0b5bba74353db6bf4dcb20abdee98
./GCF_000006765.1_ASM676v1_rna_from_genomic.fna.gz
42aee7c0fecd1f859fba1eceb4ba42dd
./GCF_000006765.1_ASM676v1_translated_cds.faa.gz

$ grep "GCF_000006765.1_ASM676v1_genomic.fna.gz" data/md5_checksums.txt
#grep the filename we downloaded earlier
c859ec5a25506367506bce972788aca8  ./GCF_000006765.1_ASM676v1_genomic.fna.gz
```

As we can see, the checksum in the file is the same as the one we calculated using the `md5sum` so we can be sure of the data integrity of the downloaded reference file. Further, to avoid manual comparison of the checksums or automate checking file integrity, we can make use of the `-c` or `--check` option. An example using this option is shown here.

Apart from checking the integrity of files, there are other interesting applications of hashing in Bioinformatics. Below is one such application that Prof. Collins pointed me to:

There is a growing usage of hashes in sequence typing microbes. Traditional sequence typing involves getting the sequence of a locus or multiple loci and comparing their entire sequences against a database of all previously seen alleles of those loci. This involves quite a lot of data as sequences can be long and so the databases can be quite large. Because we are only really looking for 100% matches in the database, the actual sequence isn't important. We just want to know if it's in the database and what the allele ID is for each locus. Recently, people have started hashing each allele and storing that in their databases rather than actual sequences. Now instead of comparing long sequences to one another, we can compare short hashes for a much quicker process and a smaller database size. This works well because DNA only uses 4 characters, while hashes can use (for example) 36 characters if they use numbers and letters. This allows hashes to store more information in fewer characters, thus saving space and compute power for the string comparisons used in sequence typing. There is a related blog article by Dr. Lee Katz, a senior bioinformatician at CDC, explaining the hashing approach used in the MLST database.

`ps`

This command was briefly introduced in the handout for demonstrating shebang but let us see another scenario where we might use this command. To give some context, every time we run a command, the UNIX operating system creates a process that is just a running instance of a program or a command. To keep track of the processes, every process is assigned a unique process ID or a PID. Further, a process can be run in foreground(all commands we have executed in class so far; this is the default behavior) or background. For processes that take a long time to execute(which we encounter very often while working on huge files), the command prompt is stuck until it completes execution. To avoid this, we can send such time-intensive processes to the background(in other words, detach it from the current session) by simply adding a `&` at the end of the command. It will execute quitely in the background and we can continue running our next set of commands. This is where `ps` command comes in handy.

`ps` command lists all the running processes and their attributes including PID(1st column), corresponding command being executed(in the last column) and others which we will not worry about. A demonstration of how `ps` can be used in situations described above is shown below by using `sleep` command as the placeholder for a time-intensive process:

```
$ ps
    PID TTY          TIME CMD
 349413 pts/5    00:00:01 bash
 418060 pts/5    00:00:00 ps

$ sleep 10 #sleeps for 10 seconds and the prompt is returned only after
completion

$ sleep 10 & #sends the process to background, gives its PID and returns the
prompt immediately
[1] 419229

$ ps
    PID TTY          TIME CMD
 349413 pts/5    00:00:01 bash
 419229 pts/5    00:00:00 sleep  <-- sleep running in background
 419245 pts/5    00:00:00 ps

$ ps 419229 #sleep executing
    PID TTY      STAT   TIME COMMAND
 419229 pts/5    S      0:00 sleep 10

$ ps 419229 #after successful completion of sleep, PID is removed from the
output
[1]+  Done                    sleep 10
    PID TTY      STAT   TIME COMMAND
```

As shown above, `ps <PID>` gives information only about the process with that PID. The `-A` option lists all the processes including processes run by different users or on a different terminal window.

If you wish to learn more about foreground and background processes and manage them using `fg` and `bg` commands, here is a good blog article.

`kill`

This command is used to interrupt or terminate a process that is suspiciously long-running or otherwise stuck using its PID. We frequently use `ctrl+z` or `cmd+z` to suspend a process(to resume it later) or `ctrl+c` or `cmd+c` when a process is stuck, and we want to abort that process to get the command prompt back. When we use these keystrokes, the operating system sends a ***suspend and interrupt signal*** (with `z` and `c` respectively) to stop that process.

There are times when `ctrl+c` or `cmd+c` cannot kill a process and we need a stronger ***kill signal*** to forcefully kill the process using `kill` command. Details about the types and working of signals are beyond the scope for our purposes but if you are interested, you can refer to this page. In case you are wondering how to issue the `kill` command when a process is stuck and we don't have the prompt, note that kill command is often used to interrupt background processes. Similarly, `ctrl+z` or `cmd+z` and `ctrl+c` or `cmd+c` are useful when working with foreground processes. To kill a foreground process with kill command, you would first need to open a new terminal, get the process ID of the process you want to interrupt using `ps -A` and then issue the `kill` command.

`kill -20 <PID>` suspends the process, `kill -2 <PID>` interrupts the process and `kill -9 <PID>` forcefully kills the process.

```
$ sleep 100 &
[1] 448850

$ ps 448850 #note the status in third column is 'S'
   PID TTY      STAT   TIME COMMAND
448850 pts/4    S      0:00 sleep 100

$ kill -20 448850 #equivalent to ctrl+z or cmd+z

[1]+  Stopped                 sleep 100

$ ps 448850 #note the status in the third column changes to 'T' meaning it
received a stop signal
   PID TTY      STAT   TIME COMMAND
448850 pts/4    T      0:00 sleep 100
```

```
$ sleep 100 &
[1] 421752

$ ps 421752
    PID TTY      STAT   TIME COMMAND
 421752 pts/5    S      0:00 sleep 100

$ kill -2 421752  #equivalent to ctrl+c or cmd+c
[1]+  Interrupt               sleep 100

$ ps 421752 #process is killed and removed from output
    PID TTY      STAT   TIME COMMAND
```

```
$ sleep 100 &
[1] 422356

$ ps 422356
    PID TTY      STAT   TIME COMMAND
 422356 pts/5    S      0:00 sleep 100

$ kill -9 422356 #forcefully kill the process
[1]+  Killed                  sleep 100

$ ps 422356 #process is killed and removed from output
    PID TTY      STAT   TIME COMMAND
```