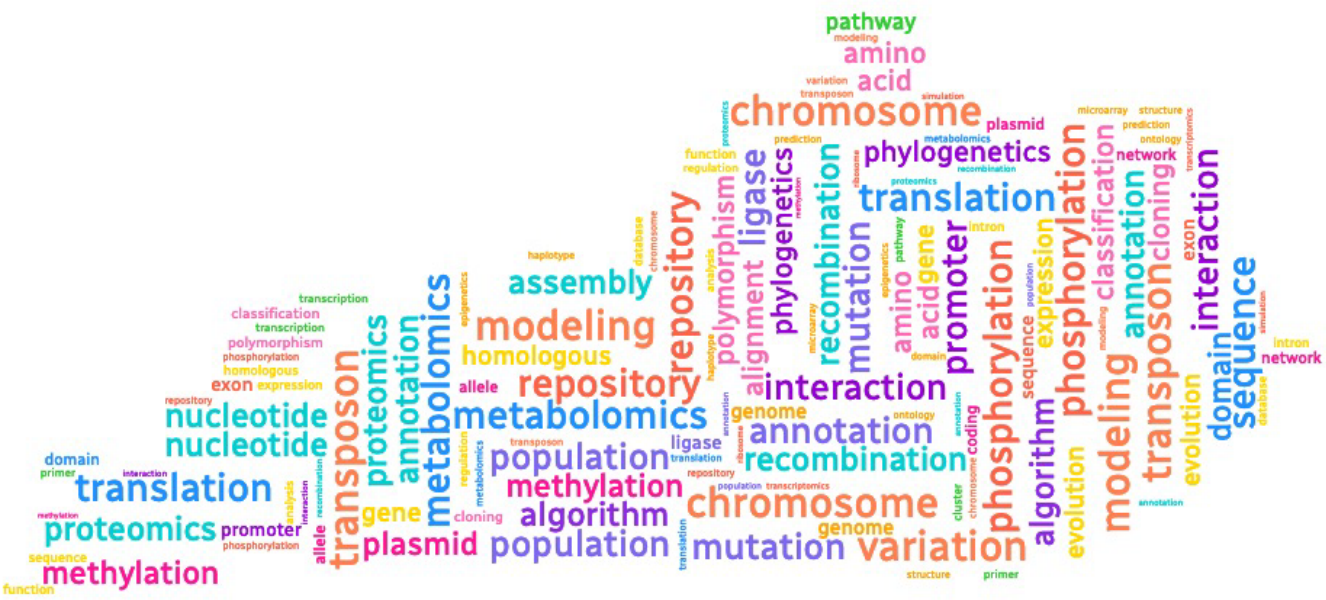
KY-INBRE Essentials of Next Generation Sequencing Workshop 2024



July 22nd to July 25th, 2024

University of Kentucky/University of Louisville

ESSENTIALS OF NEXT GENERATION
SEQUENCING WORKSHOP 2024

Module 1

Part I: Essentials of Unix/Linux

Goal: To gain familiarity with, and comfort using, the Unix command line. The majority of bioinformatics programs are command line tools. Competency with the command line will make navigating these tools and interpreting/re-organizing/filtering their outputs much easier.

1.1 Getting connected and reconnected

The steps for connecting to your virtual machine (VM) will differ depending on what operating system you are running. Follow the steps in the subsection below that corresponds to your system.

On a PC

The instructions below describe two methods of connecting to your VM from a PC—**OpenSSH/PowerShell** and **PuTTY**. You only need to follow the instructions for one method. **OpenSSH** requires no additional installation on computers running the latest version of Windows. **PuTTY** requires installation but may be more user-friendly.

OpenSSH/PowerShell

Open the **PowerShell** program. You can use the OpenSSH client **ssh** (secure shell) to connect to your virtual machine:

☐ Let's try and connect. Type the command below into **PowerShell**, changing myName and Xs to match the credentials you received via email. myName should match the username you received, and Xs should be changed so that the hostname below (the part after the "@" symbol) matches the hostname you received. When you have finished typing the command, hit enter to run the command.

```
• ssh myName@XX.XXX.XXX.XX
```

- ☐ You will be prompted for a password; type the password we sent you in the email. When you start typing your password, you won't see anything appear, but **ssh** is nevertheless reading the characters you type. Hit enter when you are done.
- ☐ After successfully logging in, you should now see a prompt, similar to:

```
myName@ip-XXX-XX-XX-XXX:~$
```

You will see your name instead of **myName**, and you will see new numbers in place of the Xs. You are now "located" in your home directory (often abbreviated as ~ or ~/). The dollar sign $ at the end of prompt simply signifies the end of the prompt.

- ☐ To logout from your VM, just type:

```
• logout
```

This will close the connection, returning you to the **PowerShell** prompt on your PC.

- ☐ Try and log in again. We will assume that you can do this step in future classes without issue.

PuTTY

PuTTY is a free telnet/SSH client for Windows that predates the native Windows OpenSSH client. Although it is not necessary to install a separate SSH client for this workshop, you may prefer **PuTTY** for its user-friendly interface and better terminal emulation. You can download **PuTTY** from https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html

- ☐ Download and install **PuTTY** from the link above.
- ☐ Open **PuTTY**. (If you ran the installer, you can find **PuTTY** in the Start menu.)

  Enter the hostname for your virtual machine into the "Host Name (or IP address)" box. You can find your hostname in the email we sent you with your VM credentials; it should look like `myName@XX.XXX.XXX.XX` where each set of Xs represents between one and three digits. Leave the default port 22 unchanged.

- ☐ Click "Open" to connect to your VM.
- ☐ You will first be prompted for your username. Enter the username from the credentials email. When you start typing your password, you won't see anything appear, but **PuTTY** is nevertheless reading the characters you type. Hit enter when you are done.
- ☐ When prompted for a password, enter the password from the credentials email.
- ☐ After successfully logging in, you should now see a prompt, similar to:

```
myName@ip-XXX-XX-XX-XXX:~$
```

You will see your name instead of **myName**, and you will see new numbers in place of the Xs. You are now "located" in your home directory (often abbreviated as ~ or ~/). The dollar sign $ at the end of prompt simply signifies the end of the prompt.

☐ To exit **PuTTY** just type exit:

```
• exit
```

This will close **PuTTY**.

☐ Try to start **PuTTY** and log in again. We will assume that you can do this step in future classes without issue.

1.1.2 On a Mac

If you are working on a Mac, simply open the program **Terminal**, found in the *Utilities* folder. You can use the **ssh** (secure shell) command to connect to your virtual machine.

☐ Let's try and connect. Type the command below into **PowerShell**, changing myName and Xs to match the credentials you received via email. myName should match the username you received, and Xs should be changed so that the hostname below (the part after the "@" symbol) matches the hostname you received. When you have finished typing the command, hit enter to run the command.

```
• ssh myName@XX.XXX.XXX.XX
```

☐ You will be prompted for a password; type the password we sent you in the email. When you start typing your password, you won't see anything appear, but **ssh** is nevertheless reading the characters you type. Hit enter when you are done.

☐ After successfully logging in, you should now see a prompt, similar to:

```
myName@ip-XXX-XX-XX-XXX:~$
```

You will see your name instead of **myName**, and you will see new digits in place of the Xs. You are now "located" in your home directory (often abbreviated as ~ or ~/). The dollar sign $ at the end of prompt simply signifies the end of the prompt.

☐ To logout from your VM, just type:

```
• logout
```

This will close the connection, returning you to the prompt on your Mac.

☐ Try and log in again. We will assume that you can do this step in future classes without issue.

1.2 Exploring and learning basic commands

Let's try to familiarize ourselves with our VM by using a few commands.

☐ Display your working directory using **pwd** (print working directory).

```
• pwd
```

You will see something like this:

```
myName@ip-xxx-xx-xx-xxx:~$ /home/myName
```

This is your current working directory, meaning you are virtually located here. Any actions you perform will assume you are here and will create files here unless you instruct otherwise.

- ☐ Let's list the current directory to see what goodies we can find here. The command to use is **ls**:

  - `ls`

```
myName@ip-xxx-xx-xx-xxx:~$ ls
assembly  blast  genes  qiime2  rnaseq  sequences  variants
```

- ☐ Let's create a *unix* directory. This is where we will do all our work on our Unix skills.

  - `mkdir unix`

- ☐ List the current directory again to make sure the *unix* directory was successfully created:

```
myName@ip-xxx-xx-xx-xxx:~$ ls
assembly  blast  genes  qiime2  rnaseq  sequences  unix  variants
```

How do we know it's a directory? One way is to tell is that the name is colored blue—more on this later...

- ☐ Change into the new *unix* directory.

  - `cd unix`

```
myName@ip-xxx-xx-xx-xxx:~/unix$
```

- ☐ To illustrate how to navigate the UNIX directory structure, we'll create a new directory inside the current one:

  - `mkdir example`

  This has created a new directory called *example* inside the *unix* directory.

- ☐ List your current directory's contents again. You will now see a new directory called *example* has been added:

```
myName@ip-xxx-xx-xx-xxx:~/unix$ ls
example
```

- ☐ Now, we'll create an empty file using the **touch** command:

  - `touch myFile.txt`

- ☐ List your current directory's contents again. Now we have *myFile.txt*:

```
myName@ip-xxx-xx-xx-xxx:~/unix$ ls
example  myFile.txt
```

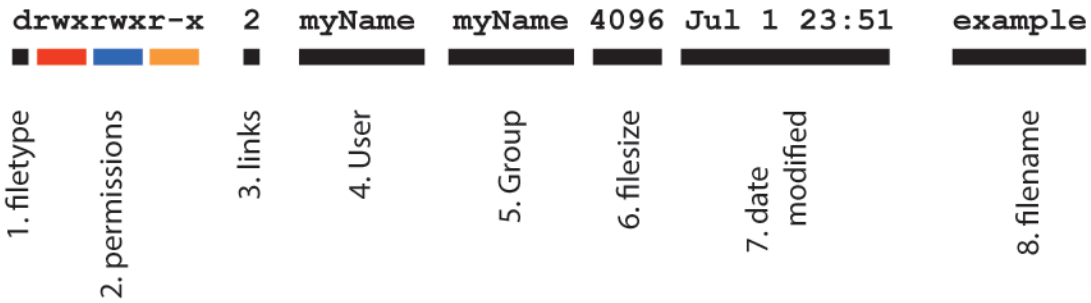
Now we see that files are listed in black. Another way to tell whether you have a directory or a file is to use **ls** with the -l option (lowercase letter L—not the number one). This is the "**l**ong" directory listing that shows a lot more details:

☐ Perform a long directory listing.

- `ls -l`

```
myName@ip-xxx-xx-xx-xxx:~/unix$ ls -l
total 4
drwxrwxr-x 2 ngsadmin ngsadmin 4096 Jul 26 22:06 example
-rw-rw-r-- 1 ngsadmin ngsadmin    0 Jul 26 22:06 myFile.txt
```

What does this all mean? Below is an example key:



For file permissions, "r" means "can read", "w" means "can write", and "x" means "can execute." Notice there are nine columns (or three sets) of permissions in the example; these three sets of permissions designate the settings for the user/owner (in red on the key), the group (blue), and others (everyone else on the machine, in orange on the key).

So, other than by simply remembering it, how do you know what "-l" does?

☐ Show the manual for **ls**:

- `man ls`

Every Linux program has a manual page installed and is accessible via **man** ***program-name***.

☐ Use the spacebar to scroll through the document and find the -l option

☐ When you have finished exploring other possible options, press **q** to quit the **man** program.

1.3 "There's no place like home"

As a beginner, it's easy to get "lost" in the UNIX directory structure until you learn the concept of directory/file paths. However, a quick and easy way to "find" yourself is to return to your home directory. The following is an exercise to illustrate absolute and relative paths and to show you how to get home from any location.

☐ Let's begin learning about directory "paths." First, we need to know where we are in the system. Recall, we can find this out with **pwd** (print working directory).

- `pwd`

Unless you did something unexpected, you should still be in the *unix* directory. **pwd** prints out an **absolute path**:

```
myName@ip-xxx-xx-xx-xxx:~$ /home/myName/unix
```

An absolute path to a file or directory points to a specific location and is unaffected by where you are currently working in the filesystem.

/home	where all individual User home directories are located
/myName	your personal home directory
/unix	the directory in which you are currently working (and where all output files will be created unless you specify otherwise

☐ If we want to find our way home, we can get there in a few different ways. First, let's use an absolute path. (Note: change myName to your username on your VM.)

- `cd /home/myName`

You can tell if this worked because the command line prompt should now say:

```
myName@ip-xxx-xx-xx-xxx:~$
```

☐ Now we're going to jump all the way to the example directory that we created a few minutes ago—this time using a relative path.

- `cd unix/example`

This path is a **relative path** because the meaning of the path depends on the current working directory. The command above says change into the *unix* directory, found inside the current directory, and then change into the *example* directory, which is inside in the *unix* directory.

```
myName@ip-xxx-xx-xx-xxx:~/unix/example$
```

☐ If we want to make absolutely sure of our location, we can print our working directory (**pwd**).

- `pwd`

We should get the absolute path to the *example* directory:

```
/home/myName/unix/example
```

☐ Now, we'll use a relative path to start navigating back to our home directory:

- `cd ..`

This says move up one directory level which, in this case, places us in the *unix* directory:

```
myName@ip-xxx-xx-xx-xxx:~/unix$
```

☐ Let's repeat the previous command to move up one more level, which places us back in our home directory.

- `cd ..`

```
myName@ip-xxx-xx-xx-xxx:~$
```

☐ To help us understand the difference between true */home* and <u>your</u> home directory, let's list the true */home* directory.

- `ls /home`

This will show a listing of all the "home" directories for every user on the system. Look for your own username; it will surely be there. The forward slash in front of *home* simply tells the shell that "true" home is in the system's root directory.

☐ Now list your personal home directory. (Remember that tilde represents your home directory.)

- `ls ~`

This shows the directories and files that are available for your personal use.

☐ Now, we'll use the UNIX equivalent of clicking our heels to go straight to our personal home directory:

- `cd`

We can also specify home using the ~ (tilde) symbol (`cd ~`) for changing directories, but that's just too much unnecessary work!

```
myName@ip-xxx-xx-xx-xxx:~$
```

1.4 Improving productivity - using your command history:

Bioinformatics involves many repetitive tasks. The UNIX system has a number of useful features to assist us. First, it keeps track of all the commands that you have previously issued up to a specified limit. You can iterate back and forth through previous commands by using the ▲ and ▼ keys on your keyboard.

☐ Try it:

- ▲ ▲ ▲ ▲ ▼ ▼ ▼ ▼

☐ Use the up and down arrow keys to navigate to the previous command where you "jumped" to the *example* directory inside the *unix* directory. When you find it, hit return.

```
myName@ip-xxx-xx-xx-xxx:~/unix/example$
```

☐ You can also see your command history as follows:

- `history`

In the last two exercises, we had you move between different directories to teach you about "paths." However, once you settle on a working directory for a given project, there is no need to change directories again. In fact, until you are familiar with the command line environment, you will get into trouble if you change directories unnecessarily. For this reason, we will learn how to work inside one directory and operate on other directories and files by specifying the "paths" to these resources.

BUT FIRST...

1.5 Minimizing errors with tab completion

We cannot overemphasize the importance and utility of learning and using tab completion. Ninety percent or more of command failures can be eliminated simply by using tab completion routinely.

☐ **Make sure you are in your personal home directory**.

☐ Create a file called *testfile.txt* in the *example* directory you created earlier:

- `touch unix/example/testfile.txt`

☐ Type `ls u`, then hit the tab key but DON'T hit return:

- `ls u<tab-key>`

The cursor will complete the second word to "unix" because it knows there is only one directory inside your current working directory that starts with the letter "u."

☐ Now add an "e" after the forward slash and hit tab again:

- `ls unix/e<tab-key>`

☐ Now you can finally hit the return key, and you will get a file listing for the *example* directory that is present in the *unix* directory. It should contain the *testfile.txt* file.

```
ngsadmin@ngs-02:~$ ls unix/example/
testfile.txt
```

What if you had two directories that start with the same letter? To demonstrate, we'll create a second directory that starts with "ex" inside the *unix* directory. Remember, however, we are currently in the *example* directory, so we need to make sure to tell the shell to create our new directory in *unix*.

☐ Create a directory called *extra* inside *unix*. First, we'll use an explicit path.

- `mkdir ~/unix/extra`

Here is an example of where ~ (tilde) comes in useful. Instead of typing the full explicit path, which would be */home/myName/unix/extra*, we can use ~ which is shorthand for */home/myName*.

☐ Now make a second directory in the *unix* directory using a relative path. This would also be a good time to try using the up-arrow key to repeat the previous command after removing the "~/" and adding a "2" the end of "extra".

- `mkdir unix/extra2`

Here you're telling the shell to create the *extra2* directory inside the *unix* directory which can be found in your current working directory.

☐ Let's make sure that the proper directories were created:

- `ls -l unix`

```
myName@ip-xxx-xx-xx-xxx:~$ ls -l unix
total 12
drwxrwxr-x 2 ngsadmin ngsadmin 4096 Jul 18 20:55 example
drwxrwxr-x 2 ngsadmin ngsadmin 4096 Jul 22 06:20 extra
drwxrwxr-x 2 ngsadmin ngsadmin 4096 Jul 22 06:20 extra2
-rw-rw-r-- 1 ngsadmin ngsadmin    0 Jul 26 22:06 myFile.txt
```

☐ Use tab completion to list the *unix/example* directory. You will find that u <tab> will complete as expected, but e <tab> will only complete as far as "ex". This because the system now sees three directories that start with "ex".

☐ Hit tab twice to see the possibilities available for completion.

```
example/  extra/  extra2/
```

Of course, we knew what to expect because we just created these directories, but, in many cases, we will not know a directory's contents, or we will have forgotten. Double tab comes in very useful in these situations.

☐ Type an "a" and then tab, and the command should now be able to complete to *example*. You can hit return to execute the command.

☐ To illustrate the error-checking capabilities of tab completion, try typing **ls unix/E** and hitting tab:

- `ls unix/E <tab-key>`

Completion will fail and your machine may even bloop at you to say something's wrong. This illustrates three things:

i) Unix is case sensitive, and, in this case, it does not see any directories starting with capital "E" inside the directory that is one level up from your current location.

ii) If tab completion is not completing, this means YOU made an error. Either you are trying to point to a directory/file that does not exist in the specified location, OR you made a typo when typing its name. (Here we made a typo— "E" instead of "e".)

iii) If you use tab completion, it is impossible to make typos or to run programs using files that are not in the locations where you think they are.

For the best experience in this workshop, and in the future, you should get in the habit of using tab completion right now and keep using it routinely for all commands.

1.6 Learning a text editor

Most of the text editors with which we are familiar (Word, GoogleDocs, TextEdit, etc.) use symbols for hyphens, quotation marks, and end-of-line markers that are incompatible with many unix programs and can create problems that are VERY hard to diagnose. For this reason, it is useful to learn how to use a UNIX text editor. Most programmers will use sophisticated editors such as **vi**, **vim**, **emacs** or **joe**, but these have steep learning curves and can be confusing for beginners. For this reason, we will learn **nano**, which, as its name implies, is a program with a small but decent set of "bare-bones" functions.

- ☐ **Make sure you are in the *unix* directory**. **We are not going to move out of this directory for the rest of this exercise.**
- ☐ Start **nano**, and we will edit a file called *name.txt*. **nano** will create the file if it does not exist. You can add a directory path before the filename to specify where it should go. (By default, the file would be created in your current working directory.) The following creates and edits a file called *name.txt* within the *example* directory:

```
• nano example/name.txt
```

Remember: we don't need to be in the *example* directory to create a file there. We simply tell the system where we want the file to be created by specifying the path.

- ☐ The **nano** screen has two main sections: a buffer to edit text and a menu of common shortcuts. The ^ character in the shortcuts section represents the control key. For example, ^O means hold the control key and "o" key together. Shortcuts in **nano** do not use the shift key, so ^O really means ^o. WriteOut means save. (Think of it as writing out to disk.)
- ☐ Test **nano** by entering your name.
- ☐ Save your file. (^o).
- ☐ **nano** will prompt you to give a filename for the file you will save. Hit enter to write to that file. By default, this is the file you gave to **nano** (*example/name.txt*) when you first issued the command. You can also start **nano** without specifying a filename.
- ☐ Exit **nano** (^x).
- ☐ Use **nano** to open your file again and verify its contents to convince yourself that the text was saved. **Remember to specify the file path (*example/name.txt*)**.

Pressing ^g (the control key and g together) will display common **nano** shortcuts. You can quit the shortcuts menu by hitting "q". ^c will display a status bar with your current position (line, column, and character numbers).

Copying and pasting can be difficult at first. When using **PuTTY**, highlighting with the mouse selects and copies text; right-clicking pastes text. In **PowerShell**, you can highlight text with your mouse and press the enter key to copy text, and, as in **PuTTY**, you can right click to paste text. With the Mac, you can copy and paste as usual.

Within **nano**, ^k will cut the current line and ^u will paste. You can also copy the current line by pressing the shift key, alt key, and 6 key simultaneously. There might be a noticeable pause when pasting large amounts of data.

You can use **nano** to view a text file safely; if you accidentally edit the file, just exit without saving by answering "no" when prompted.

If you only need to view a document without editing it, you can use the command **cat** (concatenate), for short files.

```
cat example/name.txt
```

Or, for long files, **more** and **less** are commands that will let you page-up and page-down through the file. They are often more useful than **cat** for viewing large files.

```
less example/name.txt
more example/name.txt
```

You can exit from **more** and **less** using **'q'** (and with ctrl-c, for **more**).

1.7 Using SCP to transfer files between machines

scp (secure copy) is a command-line tool that can send files to a remote machine or download files from a remote machine. Let's learn how to use **scp** by sending your newly created *name.txt* document to an Administrator Virtual machine whose Desktop we will monitor using a graphical interface.

SCPing to a remote machine involves the following syntax: 1) first specify the file (path-to-file/filename) to be transferred; 2) define the destination and new name (if needed) of the copied file (machine-address:path-to-file/filename):

```
scp <file_to_transfer> username@remote.machine.address:target_directory/filename
```

File transfer will begin upon entry of a valid password.

- ☐ Make sure you are still in the *unix* directory:

  ```
  myName@ip-xxx-xx-xx-xxx:~/unix$
  ```

- ☐ Not only will we send the file to the Administrator machine, but we will change its name in the process. If we don't do this, each time a class participant sends their file to the Admin VM, the new file will simply write over any similarly-named versions uploaded beforehand. Type the following, but replace `myName` with the name you gave in the *name.txt* file that you created using **nano**.

  ```
  scp example/name.txt ngs@3.91.83.190:Desktop/myName.txt
  ```

- ☐ The first time you connect, you will probably get a message similar to the following:

  ```
  The authenticity of host ' myName@ip-xxx-xx-xx-xxx (xx.xxx.xxx.xxx)' can't be
  established.
  ECDSA key fingerprint is SHA256:72tL3r5bj1i3/m+Hh/kwuiy2fbksrLGsRj2XS60WQ6w.
  Are you sure you want to continue connecting (yes/no/[fingerprint])?
  ```

  Type yes and hit return.

- ☐ At the prompt, enter the password: **n9sUs3r23**

This will copy the file *name.txt* across the network to the *Desktop* directory of the administrator machine. In the process, it will rename your file to your name with a *.txt* suffix. We will screen-share with the target computer to confirm the arrival of your file.

SCPing from a remote machine uses the syntax: 1) First, specify the required target file's location (machine-address:path-to-file/filename); 2) indicate where the file is to be saved on the local machine and what its name should be (path-to-file/filename).

```
scp username@remote.machine.address:source_directory/filename
target_directory/filename
```

In each case, transfer will begin upon entry of a valid password.

- ☐ Now let's grab a file from the Admin machine and (secure) copy it to your VM (using the same password as before). In this case, we are going to keep the existing filename, so we just need to tell scp where to put the file on the local machine (the "." in the command is a shortcut that means "here" in your current working directory):

```
scp ngs@3.91.83.190:Desktop/NGS-message.txt .
```

You just downloaded the *NGS-message.txt* file into your current (*unix*) directory.

- ☐ List the contents of the directory to ensure that it arrived:

```
• ls .
```

- ☐ Note that we could have written the command without the period (i.e., **ls**), and we would get the same result. Try it.
- ☐ Use **cat** to read the contents of the *NGS-message.txt* file. (Remember, use tab completion.)

```
• cat example/NGS-message.txt
```

1.8 Transferring files from the VM to your local machine

Some of the tools we will use in this workshop function best when run on our local machine. To use them on files created on our VMs, we will need to create local copies. This can also be accomplished using **scp**. Here, we will copy a sequence file (*MoRepeats.fasta*) from inside our *blast* directory to the PC (Mac) on which you are working:

- ☐ On your VM, list the *blast* directory to check that the file is where it is supposed to be. Here, we use the explicit path to the directory:

```
• ls ~/blast
```

Or, we can use a relative path:

```
• ls ../blast
```

- ☐ Open a new terminal window. On a PC this can be done by typing "powershell" in the Start menu. Click on "Windows Powershell." On a Mac, open Spotlight search with command-spacebar, type Terminal, and then hit return. Or, if you're already using a terminal window, open another one by selecting "Shell" in the top menu bar and then choose "New Window."
- ☐ Copy the *MoRepeats.fasta* file from the *blast* directory on your VM to the current working directory on your local machine:

```
• scp myName@XX.XXX.XXX.XX:blast/MoRepeats.fasta .
```

- ☐ Check that the file has been transferred successfully by listing it:

```
• ls -l ~
```

- ☐ For further confirmation, navigate to the file in your system's native file explorer.

1.9 Downloading data from the internet

Normally, when you want to download a file from the internet, you open a browser, search for the file, and then download it by clinking on a link or an icon. How, then, do we download from the internet to a remote machine when we are not sitting in front of that machine, and we have no browser? Fortunately, there are command line tools that allow us to accomplish this task. Here, we will use a program called **wget** (web get) to download a file from the National Center for Biotechnology Information FTP site.

- ☐ In a web browser, open the page for *Saccharomyces cerevisiae* (bakers' yeast) at the NCBI website: (https://www.ncbi.nlm.nih.gov/genome/?term=Saccharomyces%20cerevisiae[Organism])
- ☐ Click on the top link that says **R64**. This is the main reference genome for *S. cerevisiae*.
- ☐ Click on the link that says FTP and then right-click on the selection that says *GCF_000146045.2_R64_genomic.fna.gz*
- ☐ Go back to your VM's terminal window
- ☐ Make sure you are in the *unix* directory
- ☐ Type "**wget** " (with a space afterwards), paste in the link that you just copied, specify an output filename with the **-O** flag (not a zero; it stands for "output"), and then hit return:

```
• wget https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/146/045/
  GCF_000146045.2_R64/GCF_000146045.2_R64_genomic.fna.gz -O
  example/yeast.nt.gz
```

Note: Our limited page width necessitates that we wrap long commands across multiple lines. **Do not hit return at the end of each line** because return tells the shell to execute whatever comes before it. **Therefore, all commands in boxes should be typed on a single line.**

Successful execution of the **wget** command should yield a runtime message similar to the following:

```
--2023-06-20 11:43:48--
https://ftp.ncbi.nlm.nih.gov/genomes/all/GCF/000/146/045/GCF_000146045.2_R
64/GCF_000146045.2_R64_genomic.fna.gz
Resolving ftp.ncbi.nlm.nih.gov (ftp.ncbi.nlm.nih.gov)... 165.112.9.229,
2607:f220:41e:250::7, 2607:f220:41e:250::11, ...
Connecting to ftp.ncbi.nlm.nih.gov
(ftp.ncbi.nlm.nih.gov)|165.112.9.229|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3843460 (3.7M) [application/x-gzip]
Saving to: 'yeast.nt.gz'

yeast.nt.gz
100%[=============================================================>]
3.67M  1.38MB/s    in 2.7s

2023-06-20     1:43:54 (1.38 MB/s) - 'yeast.nt.gz' saved [3843460/3843460]
```

☐ List the contents of your example directory to make sure you now have the yeast.nt.gz file.

- `ls example/`

```
myName@ip-xxx-xx-xx-xxx:~/unix$ ls example/
Name.txt  testfile.txt  yeast.nt.gz
```

The new file shows up in red because it is a compressed archive.

If you do not see the file, check the download dialog. Were there any errors? Or did you forget to rename the file with the `-O` flag? Did you direct the output into the *example* directory?

A gzipped (compressed) archive was downloaded. It must be decompressed before we can look at it. But first, let's take a quick peek at the contents. (Remember, use tab completion.)

- `head example/yeast.nt.gz`

Note that the contents of gzipped files are not human-readable.

☐ So, let's decompress the file (**use tab completion**).

- `gunzip example/yeast.nt.gz`

This will produce a file called *yeast.nt* in the *example* directory.

☐ List the directory to check that the decompressed file exists.

☐ Let's have peek at the first 10 lines to check the decompression (**use tab completion**).

- `head example/yeast.nt`

☐ You should see a "sequence header" line with information about the sequence, followed by lines with As, Cs, Gs and Ts—the actual DNA sequence data.

1.10 Copying and moving files

☐ Suppose we want to make a copy of our *yeast.nt* file in the same directory. We will need to give it a new name (**use tab completion for the old name**). Of course, the system doesn't yet know the new name, so tab completion won't work for that:

- `cp example/yeast.nt example/yeast_copy.nt`

Note: the first argument of **cp** (copy) is the file you wish to copy, while the second argument is the destination and, if desired, a new name for the copy.

☐ List the *example* directory to check that a copy was made.

☐ Suppose we want to copy this file to a backup in a different directory and keep the same name. We just need to give the location for the copy. (Use tab completion.)

- `cp example/yeast.nt extra/`

☐ List the *extra* directory to check that the file was copied.

To move files from one location to another, we use the aptly named **mv** (move) program.

☐ We'll use **mv** to move the recently created *yeast_copy.nt* file from the *example* directory to the *extra* directory. (Use tab completion.)

- `mv example/yeast_copy.nt extra/`

☐ List the *extra* directory again to check that the file was copied.

☐ List the *example* directory, and you will see that the *yeast_copy.nt* is no longer there.

☐ Now we'll learn another useful feature of **mv**. We're going to move the *yeast_copy.nt* file back into the *example* directory but we're going to rename it at the same time. (Use tab completion for the existing file.)

- `mv extra/yeast_copy.nt example/yeast_backup.nt`

☐ Use **ls** again to check that the intended file was created.

1.11 Renaming files and directories

There isn't a dedicated rename command in UNIX. Instead, we can trick **mv** to do the rename for us. We just tell it to move the file to the same location under a different name.

☐ Use **mv** to rename the *yeast_backup.nt* file to *yeast_copy.fasta*. (Use tab completion for the original file.)

- `mv example/yeast_backup.nt example/yeast_copy.fasta`

☐ Use **ls** to check that the renaming was successful.

1.12 Removing files and directories

☐ Suppose we want to remove our *yeast_copy.fasta* file. We use **rm** (remove). (Use tab completion.)

- `rm example/yeast_copy.fasta`

List the contents of the *example* directory. The *yeast_copy.fasta* file is gone, permanently. There isn't really a concept of a "trash can" when dealing with the terminal/console.

☐ Removing entire directories has the potential to be disastrous because they could contain hundreds of precious files. Fortunately, **rm** won't remove a directory without the `-r` flag (recursive). (Use tab-completion.)

- `rm -r extra2`

☐ List the contents of your current working directory. The *extra2* directory is also now gone.

Part II - The Bioinformatician's UNIX Toolbox

1.13 Redirection

- ☐ We will use some DNA sequence data to learn how to extract information from large files. Make sure you still have the *yeast.nt* file that we downloaded earlier. If not, use **wget** to re-download it from NCBI into your *unix/example* directory and be sure to unzip the file using **gunzip** before continuing.

- ☐ Let's make sure we are working in the *unix* directory.

```
• cd ~/unix
```

The best way to understand redirection is to try it.

- ☐ Let's look at the first ten lines of the *yeast.nt* file.

```
• head example/yeast.nt
```

It is sometimes useful to see more or fewer than the first ten lines of a file. We can specify how many lines we want to see using the -n option to **head**. For example, to show the first 20 lines of our *yeast.nt* file, we could use the command below.

```
head –n 20 example/yeast.nt
```

Or, if you're super lazy, you can also specify the number of lines using just "-":

```
head –20 example/yeast.nt
```

- ☐ Now we'll redirect the output to a file named *output.txt* in the *example* directory.

```
• head example/yeast.nt > example/output.txt
```

- ☐ List the contents of the *example* directory. You should see an *output.txt* file.
- ☐ **cat** the *output.txt* file to display its contents. It contains the first 10 lines of the *yeast.nt* file
- ☐ Now try this:

```
• cat < example/output.txt
```

We just used redirection to direct the *output.txt* file into **cat**. This is a bit silly because **cat** can handle filenames perfectly well without the "<" symbol, but the example serves to illustrate the point that the < and > symbols, respectively, can direct files into and out of UNIX programs. Some (not many) bioinformatics programs send the output to the screen and do not save results by default. In such cases, you can redirect the output to a file using the ">" symbol.

1.14 grep

Grep is a tool that can search for patterns within text files. It is based on regular expressions, a language for representing patterns. Patterns can get quite complex, but we'll stay relatively simple. Before we advance any further, we should make sure we understand the contents of the *output.txt* file which contains DNA sequence information.

- ☐ **cat** the *output.txt* file to display its contents.

The first line,

```
>NC_001133.9 Saccharomyces cerevisiae S288C chromosome I, complete sequence
```

starts with the ">" symbol which tells us that this line contains the metadata for the DNA sequence that follows it. The `NC_001133.9` field is the official accession number for that sequence. The following text in the line provides additional information which tells us that this is the sequence for *Saccharomyces cerevisiae* (baker's yeast) strain S288C, chromosome I. The *output.txt* file does not contain the complete sequence, however, because we only extracted the first nine lines of sequence data.

First, we're going to search for every line of sequence metadata in the *yeast.nt* file. However, as we've just learned, the symbol used to define the metadata (>) just happens to have a special meaning in UNIX—it means output the results of a command to a file. For this reason, we must be very careful when searching for ">" symbols.

The main pattern searching program in UNIX is **grep**; the command has the general format

```
grep <pattern> <filename>
```

which means search for this pattern in this file. If we were to search for the > symbol without doing anything to tell **grep** that we're looking for the actual symbol and not redirecting its output, it would interpret the command

```
grep > example/yeast.nt
```

to mean search for nothing in no files and redirect the output to a file called *yeast.nt*. This will throw an error and, at the same time, overwrite our lovely *yeast.nt* sequence data with an empty output. In other words, it will delete all the existing sequence data from the file!

- ☐ So, instead, we need to inform **grep** to search for the ">" character by placing it in quotation marks:

  - `grep '>' example/yeast.nt`

```
myName@ip-xxx-xx-xx-xxx:~/unix$ grep '>' example/yeast.nt
>NC_001133.9 Saccharomyces cerevisiae S288C chromosome I, complete sequence
>NC_001134.8 Saccharomyces cerevisiae S288C chromosome II, complete sequence
>NC_001135.5 Saccharomyces cerevisiae S288C chromosome III, complete sequence
>NC_001136.10 Saccharomyces cerevisiae S288C chromosome IV, complete sequence
>NC_001137.3 Saccharomyces cerevisiae S288C chromosome V, complete sequence
>NC_001138.5 Saccharomyces cerevisiae S288C chromosome VI, complete sequence
>NC_001139.9 Saccharomyces cerevisiae S288C chromosome VII, complete sequence
>NC_001140.6 Saccharomyces cerevisiae S288C chromosome VIII, complete sequence
>NC_001141.2 Saccharomyces cerevisiae S288C chromosome IX, complete sequence
>NC_001142.9 Saccharomyces cerevisiae S288C chromosome X, complete sequence
>NC_001143.9 Saccharomyces cerevisiae S288C chromosome XI, complete sequence
>NC_001144.5 Saccharomyces cerevisiae S288C chromosome XII, complete sequence
>NC_001145.3 Saccharomyces cerevisiae S288C chromosome XIII, complete sequence
>NC_001146.8 Saccharomyces cerevisiae S288C chromosome XIV, complete sequence
>NC_001147.6 Saccharomyces cerevisiae S288C chromosome XV, complete sequence
>NC_001148.4 Saccharomyces cerevisiae S288C chromosome XVI, complete sequence
>NC_001224.1 Saccharomyces cerevisiae S288c mitochondrion, complete genome
```

☐ Re-run the previous command (use the up arrow to navigate back to it) and redirect the output to a file called *seqHeaderLines.txt* inside the *example* directory:

```
• grep '>' example/yeast.nt > example/seqHeaderLines.txt
```

☐ List the contents of your *example* directory. You should now see the *seqHeaderLines.txt* file.

☐ **cat** the *seqHeaderLines.txt* file to display its contents.

☐ By adding the `-c` option to your **grep** command, you can use **grep** to count the number of lines that matched your search criteria. **Paired with the ">" pattern, this is a quick way to count how many sequence records are in a FASTA file.**

```
• grep -c '>' example/yeast.nt
```

☐ How many sequences does the *yeast.nt* file contain? _____

☐ Try this:

```
• grep '>' example/yeast.nt -A 5 -B 2
```

☐ Look at the output and guess what the -A and -B options do? _____.
If you can't guess, read the manual page for grep to check the various options (man grep).

1.15 Pipes

It is possible to build small-scale workflows by sending the output of one program to the input of another.

☐ Let's grab the sequences header (metadata) lines again (**use the up-arrow key to go back to the earlier command**):

```
• grep '>' example/yeast.nt
```

☐ Now suppose we only want to see the headers of the first five sequences in the file.

```
• grep '>' example/yeast.nt | head -n 5
```

The | symbol is a "pipe" and acts as a middleman between **grep**'s output and **head**'s input. It says, use the output of **grep** as the input to **head**.

Note that grep also has a similar function: `grep -m 5 '>' example/yeast.nt` will accomplish the same thing (-m means max-count of matching lines)

☐ We could also count the number of matching lines by piping the **grep** output using another program called **wc** (word count):

```
• grep '>' example/yeast.nt | wc -l
```

Of course, this is unnecessary because we already learned that **grep** has its own, built-in counter

1.16 awk

awk is a programming language that is typically used for parsing and filtering/selecting text—it is especially useful for working on files with a tabular structure.

☐ Repeat the **grep** command to retrieve the header lines for each sequence in the *yeast.nt* file. The output will have lines that look like this:

```
>NC_001133.9 Saccharomyces cerevisiae S288C chromosome I, complete sequence
```

☐ What if we wanted to filter these search results so that only the first part of the sequence header is displayed (i.e., everything before the first space)? We can create a sequence of piped commands to do this.

- `grep '>' example/yeast.nt | awk '{print $1}'`

```
myName@ip-xxx-xx-xx-xxx:~/unix$ grep '>' example/yeast.nt | awk '{print
$1}'
>NC_001133.9
>NC_001134.8
>NC_001135.5
>NC_001136.10
>NC_001137.3
>NC_001138.5
>NC_001139.9
>NC_001140.6
>NC_001141.2
>NC_001142.9
>NC_001143.9
>NC_001144.5
>NC_001145.3
>NC_001146.8
>NC_001147.6
>NC_001148.4
>NC_001224.1
```

The part of the command in single quotes is **awk** code. This example has one line of code that prints a variable called `$1`. `$1` specifies the first column of each line, as determined by the table's field separators (which by default are spaces and tabs).

☐ Can you guess what substituting $1 with $2, $3, etc. would do? _____

☐ Try the respective commands to find out.

awk programming can get quite complicated. We could spend an entire session on it, but, for now, just be aware that it is useful whenever selection/filtering/reorganization of tabular data is needed.

☐ Now try this:

- `grep -v '>' example/yeast.nt | awk '{print $1}'`

☐ With everything scrolling down the screen so fast, it's hard to see what's going on, isn't it? Hit control-c and try piping (|) the results through **less** (or the related program, **more**) to see one page at a time. (Hit spacebar to advance one page at a time.) Compare the output of

this command with the previous one (and compare the commands) and see if you can determine what the -v option does. Look at the manual page for **grep** to check your answer.

- ☐ Instead of printing out the contents of specific fields, we could instead get the length of each line matched by **grep**.

```
• grep '>' example/yeast.nt | awk '{print length($0)}'
```

- ☐ Or the length of a given field:

```
• grep '>' example/yeast.nt | awk '{print length($1)}'
```

- ☐ One more particularly useful **awk** operation is shown below. (This is especially useful when working with long DNA sequences.) Run this command and look at the output to try and understand what it is doing. You will need to pipe through **less** or **more** to inspect the output one page at a time.

```
• grep –v '>' example/yeast.nt | awk '{print substr($1, 1, 10)}'
```

- ☐ I think the **substr** function does the following: _____
  Test your answer by playing around with the arguments provided to **substr** (try changing the values of $1, 1 and 10)

- ☐ Finally, note that we can search for strings using **awk** itself—we don't have to involve **grep** at all. In its most simple form, the structure of an **awk** command is as follows:

```
awk 'check for some condition {do the following if condition is true}' filename
```

- ☐ In the **awk** code below, `$1 ~ />/` indicates that **awk** should only run the code in braces if the lines match the pattern ">". Note that we do not need to escape the ">" symbol here because the surrounding single quotes ensure that the symbol is interpreted literally.

```
• awk '$1 ~ />/ {print substr($1, 1, 10)}' example/yeast.nt
```

- ☐ If we wanted to run code to search for lines that do not match the ">" pattern, we can use:
  `$1 !~ />/` instead. Try it…

1.17 sed

Let's suppose we want to use the list of sequence headers we got from **awk** as input to a fictional script that only accepts numbers as input. How do we get rid of the >NC_ prefix? We could do this entirely with **awk (**using **substr)**, but we could also pipe the output from **awk** to **sed** instead.

sed is a "stream editor" that relies heavily on pattern matching (much like **grep** does). It is an incredibly powerful tool and takes much practice to master. We will give you some basic exposure to **sed** by showing how to use it for a common task—pattern substitution.

In its simplest form, a **sed** statement looks like this:

```
sed 's/term1/term2/g' filename
```

In the command above, the "s" means substitution. The stuff between the first and second slashes (term1, in this example) is the pattern to match, and the stuff between the second and third slashes (term2) is what to substitute the matched pattern with. The **g** flag at the end means "global" so that every occurrence on a line is substituted. Without this flag, only the first instance on each line would be changed.

- ☐ To solve the problem stated at the start of this section, we can simply substitute the >NC_ prefix with nothing (an empty string).

```
• awk '$1 ~ />/ {print $1}' example/yeast.nt | sed 's/>NC_//'
```

The above examples give only a tiny taste of what **awk** and **sed** can do but will allow you to accomplish many bioinformatics tasks. Among many other things, you now know enough to extract (and rearrange) specific columns from large lists of data (we will practice this in a later module), and how to perform global find and replace in huge files that cannot be read into a typical text editing program (such as Word).


DAY
2

Module 2

Sequence Data: Quality assessment and trimming

Goals: Learn how to assess sequence quality and trim/filter sequence reads to generate improved datasets for downstream analyses

2.1 Background

Before we start, we need to be sure we have a good grasp of the concepts of sequence reads, paired-ends, their relationships to the DNA fragments being sequenced, and to the original DNA samples. Figure 1 serves as a reminder of how the various entities are related to one another.

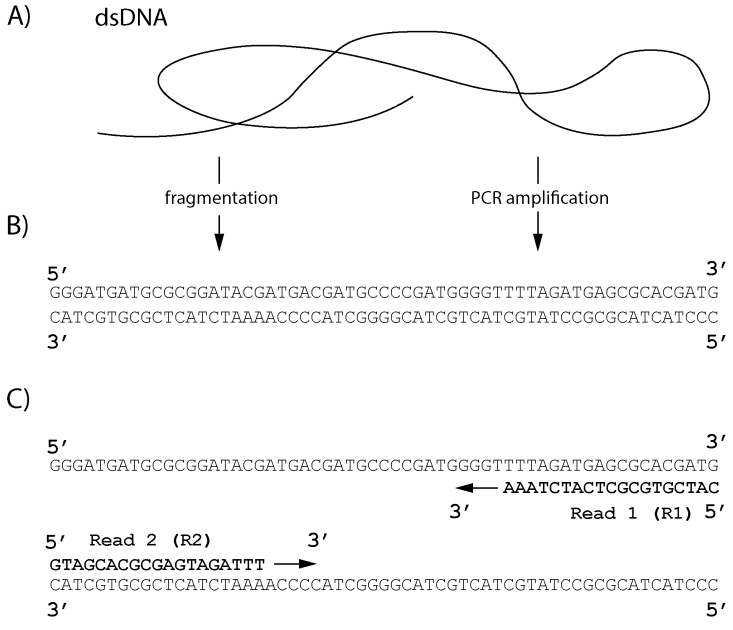


Figure 1. Summary of the essential steps involved in sequencing.

A) shows a double-stranded (ds) DNA molecule extracted from an organism, environmental sample, or a fossil.

B) The DNA is fragmented into amplifiable pieces (~300 bp–1.6 kb), or a specific target fragment is amplified, and adaptors (not shown) are added.

C) The two DNA strands are separated, and sequence is read from each strand in the 5' to 3' direction, producing two reads, R1 and R2. These are called "paired-end" reads because they are derived from opposite ends of a single dsDNA fragment.

2.2 Connecting to your VM with X11 forwarding

In this class, we will be using X11 forwarding to run a graphical user interface from the virtual machine on your local machine. To use X11 forwarding, you may need to install an X server first on your local machine.

2.2.1 On a PC

For PC users, we will be using an application called **VcXsrv** as an X server. If you are using a lab machine on campus, you already have this installed and can skip to the "Once VcXsrv is installed" instructions below.

Downloading VcXsrv

- ☐ Follow this link and click the green download button to download **VcXsrv**.

  https://sourceforge.net/projects/vcxsrv/

- ☐ Open the downloaded file and follow the installation steps.

Once VcXsrv is installed

- ☐ After **VcXsrv** is successfully installed, start XLaunch by clicking on its icon on the Desktop, or by opening the Windows start menu (accessible by clicking the Windows icon) and typing **XLaunch**
- ☐ In the window that appears, click next through all the pages then click finish. Leave all the settings at their defaults

From PowerShell

- ☐ Before connecting to your VM via **ssh**, set your DISPLAY environment variable so that your VM will know how to connect to **VcXsrv**. (Note that this is **PowerShell** syntax for setting an environment variable on your local machine, not the **bash** syntax we use on the VM.)

  ```
  • $env:DISPLAY = "localhost:0"
  ```

- ☐ **ssh** into your virtual machine using the -Y option, which allows graphical programs to run over the **ssh** connection.

  ```
  • ssh -Y myName@XX.XXX.XXX.XX
  ```

- ☐ Check that X11 forwarding is working by opening a graphical program called **xclock**:

  ```
  • xclock
  ```

  You should see a clock appear in a popup window. X11 forwarding is sometimes slow, so you may need to wait several seconds for the window to appear. Additionally, X11

applications sometimes start minimized with **VcXsrv**, so you may need to restore the **xclock** window from the taskbar at the bottom of the screen.

- ☐ Close the clock

Each time you wish to use graphical applications on your VM with X11 forwarding, you will need to have **XLaunch** (**VcXsrv**) started on your local machine. You will also need to re-set the DISPLAY environment variable and log on using the -Y option.

For PuTTY users

- ☐ Open **PuTTY**. Look on the left panel. Under Connection, click on the plus next to SSH to expand that section. Then click on X11. On the right, check the box for "Enable X11 forwarding."
- ☐ On the left, click on "Session" at the top. On the right, click on "Default Settings" and click the "Save" button. This will remember the setting you just changed.
- ☐ Now connect as you did before in Class 1, entering in your username and hostname information and clicking "Open."
- ☐ After successfully logging in, you should see the familiar prompt.

```
myName@myName:~$
```

- ☐ Check that X11 forwarding is working:

```
• xclock
```

You should see a clock appear. X11 forwarding is sometimes slow, so you may need to wait several seconds for the window to appear. Additionally, X11 applications sometimes start minimized with **VcXsrv**, so you may need to restore the **xclock** window in your taskbar.

- ☐ Click "OK" in the dialog box.

Each time you wish to use graphical applications on your VM with X11 forwarding, you will need to have **XLaunch** (**VcXsrv**) started. If you changed the default settings in **PuTTY** as instructed, you should not need to go through the step of enabling X11 forwarding again.

2.2.2 On a Mac

For Mac users, we will be using an application called **XQuartz** as an X server. This application does not come with macOS, so we will need to download **XQuartz** before continuing.

- ☐ Open **Terminal** and use **cd** to navigate to a directory on your local machine (not your VM) where you will be able to find the downloaded file. For example, you could run the command below to change to your *Desktop* directory.

```
• cd ~/Desktop
```

- ☐ Download **XQuartz**. (We've used GitHub's URL shortener to make this one easier to type.)

```
curl -OLJ https://git.io/JWTZq
```

- ☐ Open the *XQuartz-2.8.1.dmg* file you just downloaded. You can do this graphically (e.g., by finding the file in **Finder**), or you can run the following command to open the file from the command line.

```
open XQuartz-2.8.1.dmg
```

- ☐ In the window that appears, click on the *XQuartz.pkg* file.
- ☐ Use the installer to install **XQuartz**. If you receive security prompts as the installer continues, click "Allow."
- ☐ You must log out of your Mac to complete the installation. Save any work you have and click the installer's "Log Out" button. (This should log you out of your Mac, but be aware that some programs may prevent logging out. If you are not logged out by clicking the "Log Out" button in the installer, you will need to log out manually.)
- ☐ Log back in.
- ☐ Start **XQuartz** (e.g., by searching for "XQuartz" in Spotlight Search.) You can access Spotlight Search by hitting Command + Spacebar.
- ☐ From **Terminal** or the **xterm** window (created when starting **XQuartz**), **ssh** into your virtual machine using the -Y option, which allows graphical programs to run over the **ssh** connection. Don't forget to change your name and hostname.

```
ssh -Y myName@XX.XXX.XX.XXX
```

- ☐ Check that X11 forwarding is working:

```
xclock
```

  You should see a clock appear on your screen. X11 forwarding is sometimes slow, so you may need to wait several seconds for the window to appear.

- ☐ Click "OK" in the dialog box.

If **xclock** did not work above, you may not have logged out of your Mac. If you are sure you logged out, and **xclock** still did not work, try restarting instead. Note that you needed to logout or restart your Mac—not your VM.

Each time you wish to use graphical applications on your VM with X11 forwarding, you will need to have **XQuartz** started and will need to use the -Y option.

2.3 Assess overall sequence quality with FastQC

- ☐ Make sure you are logged into your VM.
- ☐ Change (**cd**) into the *sequences* directory

☐ **Mac users only** – use this command to ensure proper rendering of the forwarded graphics:

```
• export _JAVA_OPTIONS='-Dsun.java2d.xrender=false'
```

☐ Open the **FastQC** graphical user interface (GUI):

```
• fastqc &
```

☐ Click on the File menu and select "Open". (Alternatively, you can just type Ctrl-O.)

☐ Select the *sequences* folder and click "Open"

☐ Select both of the Br80 *fastq* datasets (shift click each one) and click "Open". These sequences were generated using the Illumina MiSeq platform with a run kit that allows acquisition of 2 x 300 bp reads. Once the analysis is complete, a summary page will display. Note the left-hand panel which shows the various statistics that are available

☐ Look at **Basic Statistics**: This provides file name, type, encoding, number of sequences, sequence length range, sequences to be filtered, and overall %GC content.

☐ How many sequence reads were in for the R1 and R2 datasets? R1: _____,R2: _____

☐ What are the ranges in sequence lengths? low: _____,high: _____

☐ Click on **Per base sequence quality**: For each data point, the central red line is the median value; the yellow box represents the inter-quartile range (25-75%). The upper and lower whiskers represent the 10% and 90% points, and the blue connecting line represents the mean quality. A warning (yellow "!") is issued if the lower quartile for any base is less than 10, or if the median for any base is less than 25. This tab will signal failure (red "X") if the lower quartile for any base is less than 5 or if the median for any base is less than 20.

☐ Click on **Per tile sequence quality**: Illumina sequencing flow cells consist of "tiles" which are discrete areas that are imaged at each round of sequencing. This metric uses the flow cell coordinates in each read identifier to assess whether there might have been any systemic sequencing issues. Quality is represented using a color scale, with cold colors indicating quality higher than average for the given base and warm colors indicating quality lower than average. Base position is plotted on the x-axis and tile number is on the y-axis. (Note: 9 tiles were imaged with the MiSeq V2 chemistry used for the present run, and they are represented by indexes 1101 through 1109; 19 are now imaged with V3 chemistry.) A warning is issued if, at any base, a tile has a mean Phred score below 2 less than the mean quality across all tiles at that base. A failure is raised if, at any base, a tile has a mean Phred score below 5 less than the mean quality across all tiles at that base.

☐ Click on **Per sequence quality scores**: This plots a frequency distribution for average quality across the entire sequence. A warning is raised if the most frequently observed mean quality is below 27 (this equates to a 0.2% error rate). An error is raised if the most frequently observed mean quality is below 20 (1% error rate).

☐ Click on **Per base sequence content**: the relative frequency of each base should reflect the overall frequency of these bases in the acquired sequence data. Strong position-specific biases usually indicate an overrepresented, possibly contaminating sequence. Consistent bias across all positions either indicates true genome bias (i.e., AT-/GC-richness) or signals a

systematic sequencing problem. This module issues a warning if the difference between A and T, or G and C is greater than 10% in any position. This module will indicate an error if the difference between A and T, or G and C is greater than 20% in any position.

- ☐ Click on **Per sequence GC content**: the overall GC content should reflect the GC content of the underlying genome. This module issues a warning if more than 15% of the reads deviate from the normal distribution; and a "failure" if the deviation involves more than 30% of the reads. An unusually shaped distribution could indicate the presence of AT-/GC-rich genomic "islands" or a contaminated library (e.g. fungal DNA contaminated with bacteria).
- ☐ Click on **Per base N content**: the % of reads with Ns at each base position. This module raises a warning if any position shows an N content of >5%. This module will raise an error if any position shows an N content of >20%.
- ☐ Click on **Sequence length distribution**: generates a graph showing the distribution of fragment sizes in the file under study. This module will raise a warning if all sequences are not the same length or an error if any of the sequences have zero length.
- ☐ Click on **Sequence duplication levels**: Plots the % of reads that occur as exact duplicates. Due to the supposedly random nature of shearing/tagmentation, most reads should have slightly different sequence start positions. Large numbers of exact sequence duplicates implicate PCR amplification as a culprit. A warning is given at >20% duplication, and an error is thrown at >50%.
- ☐ Click on **Overrepresented sequences**: Lists specific sequence(s) that are highly overrepresented in the dataset. All of the sequences which make up more than 0.1% of the total are reported. This module will issue a warning if any sequence is found to represent more than 0.1% of the total. This module will issue an error if any sequence is found to represent more than 1% of the total.
- ☐ Click on **Adapter content:** Occasionally, adapters may be missed by the de-multiplexing program, sometimes due to sequencing errors and sometimes because the fragment being sequenced is much shorter than the read length (typically 250 bp or 300 bp for MiSeq) which leads to the 3' adapter being sequenced. **FastQC** looks for matches to four common adapters within reads. The module will issue a warning when a sequence is found in more than 5% of reads and will report failure when a sequence is found in more than 10% of all reads.

Note that the quality of the sequences drop off dramatically as the sequence reads extend past 280 bp. This is not surprising because these data were generated using the 600 cycle MiSeq flowcell, which was fairly new technology at the time these data were generated.

Possible problems with biased sequence composition were detected at the starts and ends of the reads. Front-end bias is probably due to the tagmentation process. (In practice, the transposon that cuts the DNA and adds adapter sequences operates preferentially in certain genomic contexts.) This is unlikely to cause problems during assembly (after all, this represents "true" genomic sequence) and can therefore be ignored. On the other hand, the sequence bias at the ends of the R2 reads point to frequent occurrence of homopolymer tracts due to poor sequence resolution. Therefore, the R2 reads should be quality-trimmed prior to downstream analyses.

- ☐ Now run **FastQC** on the "Lh88405" sequences in the same directory.

Here, the Lh88405 dataset has exceptional sequence quality throughout the sequence reads, which is not surprising given that only 150 bp of data were acquired from each DNA strand. However, a significant proportion of reads have adapter contamination at their 3' ends, and this contamination must be trimmed before sequence assembly.

Question: Thinking about the various processes involved in NGS library construction, what would be the primary reason that so many reads have adaptor sequences at their ends? _____

_____

2.4 Trimming/filtering poor quality sequence data with Trimmomatic

High quality sequence data is important for all types of downstream analyses. **FastQC** provides a good overview of sequence quality and alerts us to potential problems with our data, such as short sequence lengths, unusual base composition indicative of sequence adaptor contamination and quality decline near the ends of the sequences. Now we will use the free utility, **Trimmomatic,** to filter and trim reads to address these quality issues.

Bolger et al. (2014) Trimmomatic: A flexible trimmer for Illumina sequence data. Bioinformatics doi: 10.1093.

http://www.usadellab.org/cms/uploads/supplementary/Trimmomatic/TrimmomaticManual_V0.32.pdf

Usage:

```
java –jar <path to trimmomatic.jar> PE [–threads <threads>] [–phred33 |
–phred64] [–trimlog <logFile>] <input 1> <input 2> <paired_output_1>
<unpaired_output_1> <paired_output_2> <unpaired_output 2> <trim_options>
```

In the above command, we introduce square and angle brackets for the first time. In this example, the text enclosed in square brackets ("[" and "]") are "options" that specify various user-specified parameters that affect program behavior. The brackets are omitted from the actual command. As implied by their name, these parameters are "optional" and if they are not specified, the program, assumes specific default settings. Parameters in angle brackets are required, and in this example show where we must insert specific inputs to **Trimmomatic**. In the actual command, you will omit the brackets and substitute the terms inside them with your specific filenames. This should become clear when you compare the provided usages with the actual commands we will be using.

We will use **Trimmomatic**'s nifty paired-end feature (PE mode) that separates reads based on whether corresponding forward and reverse sequences are available. This is useful because some downstream analysis programs (e.g., the **MaSuRCA** assembler) will only allow paired-end reads as input, or they process paired-ends and unpaired reads separately.

Because we are running Trimmomatic in paired-end mode, we will need to provide both forward (R1) and reverse (R2) read filenames as arguments to the program

☐ Use **Trimmomatic** to trim poor quality sequences from the Br80 dataset. **Remember, enter the entire command on a single line (without the backslashes). Type very carefully and use tab completion where possible.**

```
• java –jar trimmomatic–0.38.jar PE –threads 2
  –phred33 –trimlog Br80_errorlog.txt
  Br80_S1_L001_R1_001.fastq
  Br80_S1_L001_R2_001.fastq
  Br80_S1_L001_R1_paired.fastq
  Br80_S1_L001_R1_unpaired.fastq
  Br80_S1_L001_R2_paired.fastq
  Br80_S1_L001_R2_unpaired.fastq
  CROP:280 SLIDINGWINDOW:20:20 MINLEN:150
```

`PE`	run in paired-end mode (create paired-read and unpaired-read output files)
`–trimlog`	creates a log of all read trimmings
`–phred33`	use the phred33 quality scale
`CROP`	clip read to specified length by removing reads from 3' end
`SLIDINGWINDOW`	clip read once mean quality in window of specified size falls below target value (average Phred score ≤ 20 in 20 nucleotide window)
`MINLEN`	discard reads that are shorter than the specified length

You can tell if you issue the correct commands if the program reports that TrimmomaticPE was started with a set of arguments, no errors (exceptions) are reported, and the command line prompt does not immediately reappear (indicating that the program is processing data). Once the trimming is complete, the program will report some summary statistics indicating how many reads were trimmed/retained after the trimming/filtering steps.

☐ Now run **FastQC** on the four trimmed output files (the ones with *paired* and *unpaired* in their names) to assess how well **trimmomatic** performed in removing poor quality sequence.

2.5 Trimming adaptor sequences

Next, we will use **Trimmomatic** to clip off sequences that correspond to adaptors that were added onto our DNA fragments during library construction. To do this we provide **Trimmomatic** with a fasta file that contains the sequences of possible adaptors.

☐ Use the command line to take a quick look at the contents of the *adaptors.fa* file (it should be in your working directory).

Note that this file does not contain all possible adaptor sequences and additional ones may need to be added to the file depending on the specific type/manufacturer of the kit that was used for NGS library production.

☐ Use **Trimmomatic** to remove adaptor contamination (and poor quality regions) from the Lh88405 dataset. **Remember, enter the entire command on a single line (without the backslashes). Type very carefully and use tab completion when typing names of input files.**

```
• java -jar trimmomatic-0.38.jar PE -threads 2
  -phred33 -trimlog Lh88405_errorlog.txt
  Lh88405_1.fq.gz
  Lh88405_2.fq.gz
  Lh88405_1_paired.fq
  Lh88405_1_unpaired.fq
  Lh88405_2_paired.fq
  Lh88405_2_unpaired.fq
  ILLUMINACLIP:adaptors.fa:2:30:10 SLIDINGWINDOW:20:20 MINLEN:100
```

`ILLUMINACLIP` clip off sequences that match adaptors in the provided reference file

☐ Run **FastQC** on the four trimmed output files to assess how well **trimmomatic** performed in removing the adaptor contamination.

2.6 Optional exercises

Using the command line to convert between sequence formats:

The FASTQ format is similar to the FASTA format, but, unlike FASTA lines, which begin with ">", FASTQ lines begin with "@". FASTQ files also have after each sequence a line containing "+" followed by a line containing the Phred quality data for each base encoded as ASCII characters.

☐ Take a look at one of the *.fastq* files in your *sequences* directory to familiarize yourself with the format.

☐ Generate a *.fasta* file from a *.fastq* file using only Linux commands. (Don't just find a program that already does this conversion for you.) See if you can accomplish this in one line, using piped commands.

Hint: You already know commands (**grep, awk, sed**) that will allow you to do accomplish this task. Use the **man** pages (or Google/ChatGPT) to find out how to extend the functions of these programs to generate the necessary output.

Running FASTQC using the command line:

Above we used a graphical user interface (GUI) to interact with FASTQC. However, it can also be run through the command line. Try it:

```
• fastqc GG11_S2_L001_R1_001.fastq GG11_S2_L001_R2_001.fastq
```

Runtime messages will print to the screen to show you progress on each set of data files.

☐ When complete, list the current directory. You will see two html files and two zipped archives.

☐ Copy the html files to your local machine and double-click on them to open in an html browser.

Module 3

Genome Assembly

Goal: 1) Learn how to use the command line program **Velvet** to generate genome assemblies
2) Assemble the genome of the bacterium *Bacillus cereus*
3) Visualize the assembly layout

Input(s): R1 and R2 reads of *B. cereus*:
`Bcereus_S1_L001_R1_001.fastq.gz`
`Bcereus_S1_L001_R2_001.fastq.gz`

Output(s): `Bcereus_interleaved.fastq` file of interleaved reads
`Bcereus_velvet` directory of Velvet assembly results

Most genome assembly programs are run in a similar manner to one another, with the main differences usually being the types of sequence reads for which they are optimized. For this particular exercise, we will use the **Velvet** assembler because it is a good all-round genome assembler for Illumina sequence data.

Velvet 1.2.10

Velvet is a short read aligner and easily handles the assembly of large datasets comprising reads ranging from 50 to 300 bp in length.

Zerbino DR, Birney E. (2008) Velvet: algorithms for *de novo* short read assembly using de Bruijn graphs. Genome Res. 18:821–829.

https://github.com/dzerbino/velvet/tree/master

The **Velvet** package contains two main programs that perform separate functions. **Velveth** creates a table that lists all of the subsequences of a pre-chosen length, k ("k-mers" *sensu* monomer, dimer, etc.), that are found within the sequence reads along with the number of times each k-mer occurs. **Velvetg** then builds and traverses the de Bruijn graph using the k-mer table as its guide. Before we start running either program, however, we must first get the sequence reads into a suitable format.

3.1 Analyze the quality of the sequence reads

- ☐ First, make sure you are in the home directory of your virtual machine.
- ☐ Change into the *assembly* directory.
- ☐ List the directory contents. You will see that it contains two sequence files, *Bcereus_S1_L001_R1_001.fastq.gz* and *Bcereus_S1_L001_R2_001.fastq.gz*. The sequence filenames contain useful information about the sequences. "S1" means that this was the first barcode on the flow cell. "L001" indicates the data came from flow cell lane 1. "R1" means forward read (while "R2" means reverse). The "001" means this is dataset 1 from this lane of the flow cell. "fastq" indicates the sequence format, and "gz" tells us that the data were compressed with gzip.
- ☐ Before we start the assembly, let's examine the data quality using **FastQC**. Refer to Lab 2 for a refresher on using the program. If you have any difficulty launching **FastQC**, make sure you have followed the instructions for connecting to your VM with X11 forwarding.
- ☐ There are possible problems with the input data. What are these problems and how would you solve them?

1. Problem: _____; Solution: _____
2. Problem: _____; Solution: _____
3. Problem: _____; Solution: _____
4. Problem: _____; Solution: _____

3.2 OPTIONAL: Trim/filter the reads to remove poor quality sequences

Only perform the following trimming steps if you are super proficient with the command line.

- ☐ Use **Trimmomatic** to trim/filter the data to remove poor quality sequence. Refer to the instructions in Lab 2 for a reminder of required arguments and their values (modify the values as needed for this dataset).
- ☐ Record the **Trimmomatic** arguments that you used so that we can compare assemblies generated with trimmed/non-trimmed data:

_____

_____

3.3 Identify a suitable k-mer length

Because **Velvet** is a de Bruijn graph assembler, it breaks the sequencing reads down into subsequences known as k-mers. The choice of k-mer length is very important and will affect the quality of the assembly. A k-mer that is too long will produce an incomplete assembly, while one that is too short will yield a fairly complete assembly but will have a large number of very short contigs. To assist us in the choice of a suitable k-mer length, we will use an online tool called Velvet Advisor:

http://dna.med.monash.edu.au/~torsten/velvet_advisor/

- ☐ Visit the URL in the link. (**Important: Note that there is an underscore _ between the words velvet and advisor**).
- ☐ Fill in the relevant fields with information on the sequencing reads and estimated genome size (5.5 Mb). **When selecting the "paired-end reads" option, you should enter the number of reads in either the R1 or R2 datasets—not both.** Use a target k-mer coverage of 20. (Note that assemblies typically improve as k-mer coverage increases to 20 and then reach a plateau beyond this value.) Inspect the highlighted "Answer" field to find the recommended k-mer length. Note, this is only a guideline, and the best length is determined empirically by running **Velvet** iteratively, using a range of k-mer values around the one suggested by **Velvet Advisor**.
- ☐ What is the suggested k-mer length for the *B. cereus* genome assembly? _____

Before we move on, let's get a sense of how the choice of k-mer length is affected by the size of the genome under investigation:

- ☐ Fungal sequencing project: 30 million reads; 150 bp PE; genome size 45 Mbp

  Suggested k-mer length: _____
- ☐ Human sequencing project: 600 million reads; 150 bp PE; genome size 3.1 Gbp

  Suggested k-mer length: _____

3.4 Using screen

In this lab and others, we will be running programs that take a lot of time to run. While these programs are running, you may need to leave, or your SSH connection may drop unexpectedly. This is problematic because closing an SSH connection generally terminates your shell process and any processes you are running from that shell. We do not want the program to stop running whenever we disconnect; fortunately, there are ways to allow our programs to persist even when we are not connected.

One way to solve this problem would be to use a terminal emulator in a desktop environment on the virtual machine. We could access the desktop of the virtual machine using a remote desktop program (such as **NoMachine**), and since the desktop persists after closing the program on the client, we could continue running our program in the terminal emulator after stopping the session. Although this approach is useful for keeping some programs running (especially graphical applications), it is usually not ideal for keeping command-line programs running because accessing a terminal through a terminal emulator with a remote desktop program is usually slower than accessing a terminal through SSH.

Fortunately, various programs exist to solve this issue without a desktop environment. We will be using **screen**, a popular terminal multiplexer. As its name suggests, **screen** allows us to create virtual "screens" on which to view terminals. Like actual screens we might access with remote desktop software, those created by **screen** persist even if we close our connection.

We can create a **screen** session by simply running **screen** with no arguments. However, it is common to give names to new **screen** session, as it makes it easier to access them later. We will name the session for this class "velvet" by providing **screen** with a the -S argument.

☐ Create a new **screen** session named "velvet".

- `screen –S velvet`

You should see that the screen immediately seems to clear. We have created a new **screen** session named "velvet" and are now attached to that screen. The new **screen** session starts a new **bash** process, which runs the *.bashrc* file upon startup and inherits any exported variables (such as PATH) from the shell you used to create the session.

☐ Run a few commands, such as **ls**, in the new screen.

☐ After running a few commands, detach from the current screen by typing Ctrl-a then d.

You will see that you are in the same shell you were using before you created the new screen. We can now easily reattach the screen we just created to resume working in that session.

☐ Reattach the named screen session created earlier:

- `screen –r velvet`

Unless you cleared the screen before detaching, you should see the output from the commands you ran in the **screen** earlier.

Although here we knew the name of the **screen** session we wanted to reattach, this may not always be the case (especially if we create a **screen** without specifying a name). Fortunately, we can list existing screen sessions by running:

```
screen –ls
```

The output of this command looks something like this:

```
There is a screen on:
      5868.velvet (02/02/2019 12:42:22 AM) (Detached)
1 Socket in /var/run/screen/S–acta225
```

The first column starting on the second line lists the process ID followed by a dot and the name of the screen. You can reattach the screen using either.

We will mainly be using **screen** for its ability to continue running programs after SSH connections are closed. We can test this by disconnecting from the VM without terminating the screen session, reconnecting, and reattaching the screen.

☐ Close the SSH connection (e.g., by closing the PuTTY/terminal emulator window or by detaching from the screen again and executing "**exit**"). Do not use the **exit** command within the screen. **exit** will terminate the **screen** session, but we only want to detach the screen from the current terminal.

☐ Reconnect to the VM via SSH.

- ☐ Reattach to the screen we created. (Note: If you closed the terminal window, your screen may still be attached. Add the -d argument to the command we used above to detach the screen before attaching it again.)
- ☐ Check that the output from the commands you ran earlier is still present in the screen.

Although we can terminate a **screen** session by running the **exit** command with the screen attached, it may sometimes be useful to terminate a screen without attaching it first. We can do this by running

```
screen –X –S <screen_name> quit
```

It is also possible to terminate an attached **screen** session without typing the **exit** command in the shell. The command Ctrl-a \ will also terminate the attached **screen** session

3.5 Run velveth using the suggested k-mer value

Before we begin, let's recap how **Velvet's** implementation of de Bruijn Graph assembly works: i) **velveth** breaks reads up into sets of overlapping k-mers and builds a temporary library that records the sequences and positions of k-mers within the reads; ii) it then identifies overlapping reads by comparing the k-mer profile for each read against the library to find the reads with the largest overlap; iii) these overlaps are represented as a graph that serves as "roadmaps" for guiding the de Bruijn process; iv) **velvetg** breaks the reads into k-mers again and implements a traditional Eulerian path traversal to form a PreGraph v) this preliminary graph is then threaded through the roadmaps, so as to take advantage of the additional information contained in the reads (i.e. links between k-mers that are lost when you generate a k-mer list); vi) finally, **velvetg** tries to resolve repeats and removes bubbles caused by sequencing errors.

We are now ready to run our genome assembly. First, we will use velveth to build the Roadmaps (a PreGraph) that describe connections between reads.

Usage:

```
velveth <output-directory> <kmer-length> <read-type> <read-format> <r1> <r2>
```

- ☐ Run **velveth** on the *B. cereus* sequence reads using the suggested k-mer value obtained from **velvet advisor**:

```
• velveth Bcereus_velvet1 <kmer-value> –shortPaired –fastq.gz
  –separate Bcereus_S1_L001_R1_001.fastq.gz
  Bcereus_S1_L001_R2_001.fastq.gz
```

- ☐ Assembly should take about a minute and will produce a new directory (*Bcereus_velvet1*) that contains the following files:

Log	describes the compilation settings for the program
Sequences	the uncompressed sequences used for assembly
Roadmaps	a graph file summarizing overlaps between reads based on shared k-mers

You could look at the *Roadmaps* file, but you will find that the information contained therein is not very useful to us (it just helps **velvetg** to perform its tasks).

3.6 Run velvetg to perform the assembly

Velvetg performs the graph assembly, works to remove "bubbles" caused by sequencing errors, and resolves repeated sequences.

Usage:

```
velvetg [options] <velvet-output-directory>
```

We'll run **velvetg** using the default parameters to generate a "baseline" assembly.

☐ Run **velvetg** to generate our first assembly.

- `velvetg Bcereus_velvet1`

☐ When the run has completed, list the output directory. There should be seven output files:

Log	describes the compilation settings for the program and summarizes the final assembly
stats.txt	summarizes information about each contig
Sequences	contains the input reads
PreGraph	graph file, not intended to be read by end user (but see later)
Graph	graph file, not intended to be read by end user (but see later)
Graph	graph file, not intended to be read by end user (but see later)
contigs.fa	fasta file of contigs longer than 2k (where k is the k-mer length used in **velveth**)

☐ Interrogate the end of the **velvetg** runtime message or the resulting *Log* file to determine the following metrics for the assembly:

Genome size (total): _____

Number of scaffolds (nodes)*: _____

Longest scaffold (max): _____

N50 value: _____

*Note: Scaffolds are referred to as "nodes" in the *Log* file but they are actually scaffolds because we used paired-end reads to stitch adjoining "nodes" together.

3.7 Run velvet using a range of k-mer values

To obtain an optimal genome assembly, we could run **Velvet** multiple times using different k-mer lengths surrounding our suggested value. Fortunately, the nice folks at the Victorian Bioinformatics Consortium have written a script that automates this process. All we have to do is provide it a range of k-mer values to test and the step size between each value.

Usage:

```
velvetoptimiser [options] –f '<velveth input line>'
```

☐ **Make sure you are in the velvet screen**.

☐ Run **velvetoptimiser** using the k-mer range 121 to 201. (Be patient. **velvetoptimiser** takes around 20 minutes to run to completion. Also note that there is no tab completion on the arguments to **velveth**, so be careful not to make typos!)

```
• velvetoptimiser –s 121 –e 201 –x 10 –d Bcereus_velvet_optimal
  –f '–shortPaired –fastq.gz –separate
  Bcereus_S1_L001_R1_001.fastq.gz
  Bcereus_S1_L001_R2_001.fastq.gz' –t 1
```

`–s`	The starting (lower) k-mer value (default '19')
`–e`	The end (higher) k-mer value (default '201')
`–x`	The step in k-mer search, min 2, no odd numbers (default '2')
`–d`	The name of the directory to put the final output into
`–f`	The file section of the **velveth** command line (default '0').
`–shortPaired`	Type of reads in assembly
`–fastq.gz`	Sequence read format
`–separate`	The sequence reads are paired-end with the pairs being provided in separate files (Important: the R2 file must contain paired-ends for all the reads in the R1 file and vice versa, and corresponding pairs must be in the same order in the two files)
`–t`	The number of parallel threads to run with. (default: number of processors)

☐ When the run has completed, examine the output directory. Now there should be eight output files: the seven that were produced in the first run, and an eighth that starts with a date and time and ends with "*Logfile.txt*"—the actual name depends on the date and time of your run. This last file is one created by **velvetopimiser**, and it summarizes all of the assemblies and optimization steps that were performed. Metrics for the optimized assembly are reported at the bottom.

☐ Find the information on the final optimized assembly from the end of the **velvetoptimiser** runtime message, or from the *XX-XX-XXXX-XX-XX-XX_Logfile.txt* file and summarize the following metrics:

Genome size (total bases in contigs): _____

Number of scaffolds (contigs)*: _____

Longest scaffold: _____

N50: _____

Note: Scaffolds are referred to as "contigs" in the **velvetoptimiser log file but they are actually scaffolds because we used paired-end reads to stitch adjoining "contigs" together.

What characterizes an optimal assembly? It depends on how you plan to utilize your genome. For example, if you wish to identify and characterize specific genomic regions, you might want to have maximal contig lengths (which means a high N50) to increase the chance of having the target sequence(s) on a single contig. On the other hand, if you plan to use the whole genome in subsequent analyses, then having a small number of total contigs might be desirable. (Note that the two metrics, though often correlated, don't always fully correspond.) In **velvetoptimiser**, the optimization function can be tuned to maximize different variables:

LNbp	The total number of nucleotides in large contigs
Lbp	The total number of base pairs in large contigs
Lcon	The number of large contigs
max	The length of the longest contig
n50	The n50 value
ncon	The total number of contigs
tbp	The total number of base pairs in contigs

Examples of optimization functions include:

'Lbp'	Just the total base pairs in contigs longer than 1kb
'n50*Lcon'	n50 times the number of long contigs
'n50*Lcon/tbp+log(Lbp)'	n50 times the number of long contigs divided by the total bases in all contigs, plus the log of the number of bases

3.8 Further optimize the assembly (optional)

- ☐ Use the optimized parameters established in the above **velvetoptimiser** run as a guide to try and further improve the *Bacillus cereus* genome assembly. Specifically, select new start and end k-mer sizes that bracket the optimal value obtained from the first run. **Remember to use odd starting k values (to avoid palindromes) and an even step size to maintain k "oddness" throughout the run. (2 is a good step size.)**
- ☐ Interrogate the end of the runtime message, or the new *XX-XX-XXXX-XX-XX-XX_Logfile.txt* file, to see how the final metrics compare with the first round of optimization:

Genome size (total bases in contigs): round 2:_____ round 1:_____

Number of scaffolds (contigs): round 2:_____ round 1:_____

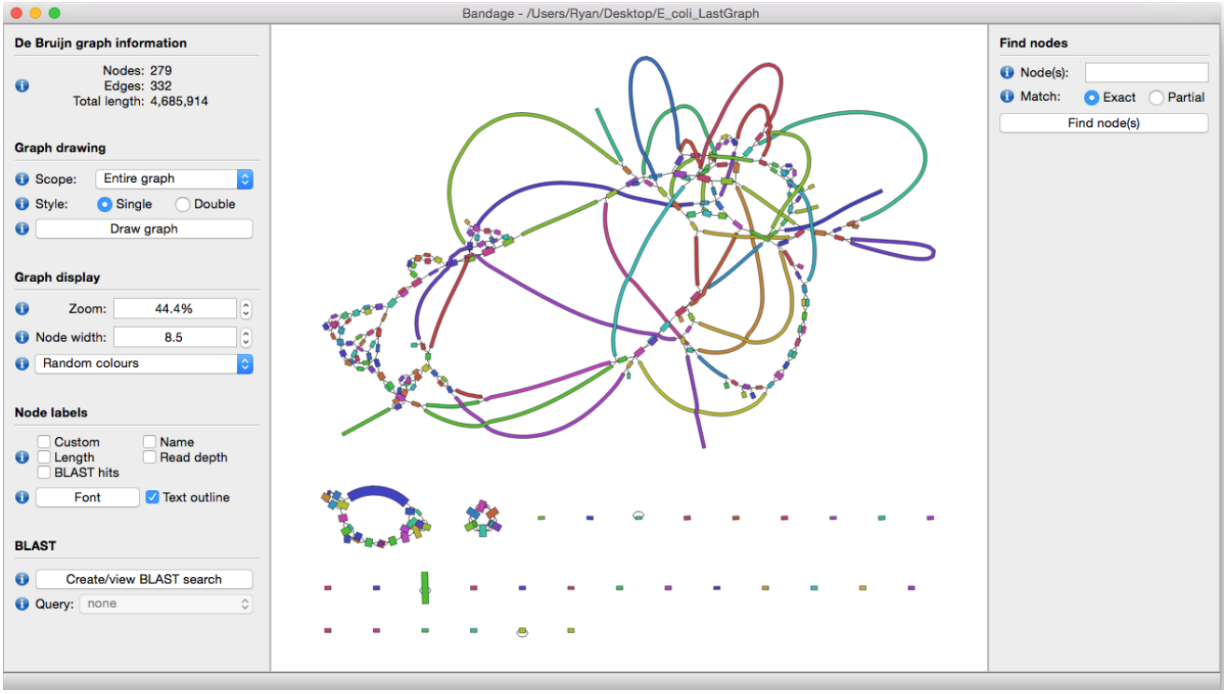
Longest scaffold: round 2:_____ round 1:_____

N50: round 2:_____ round 1:_____

Optional exercise: Visualizing assemblies using Bandage

Bandage is a graphical tool that allows one to interrogate an assembly graph to examine contiguity and resolve ambiguities. It also incorporates a **blast** search engine for homing in on sequences of interest. One can view entire assemblies or select individual nodes and their neighbors.

https://rrwick.github.io/Bandage/



- ☐ Visit the above link and download and install the Bandage executable on your local machine (Mac or PC)
- ☐ Use **scp** to download the *Graph2* file for one of your **velvet** assemblies to your local machine.
- ☐ Open Bandage and under the **File** menus, select "**Load graph**".
- ☐ In the file selection dialog, navigate to the downloaded *Graph2* file.
- ☐ Select the *Graph2* file and click "**Open**".
- ☐ Click the "**Draw graph**" button on the left-hand panel.

Bandage will now display the entire assembly graph, consisting of one major cluster of interconnected nodes (i.e., contigs) in a highly reticulated network, a number of minimally branched graphs, and mostly single linear nodes. Overall, this assembly looks quite good because there aren't too many "snarled" networks, and the number of nodes is fairly small. **Note that this graph has many more nodes than the final optimized assembly. This is because the VelvetOptimiser wrapper that invokes the velvetg processes appears to prevent the last run from writing the LastGraph file.**

☐ Click on any line (single, or within a branched network). Observe that the identity of the selected node is displayed in the right-hand panel, along with its length and average depth of coverage.

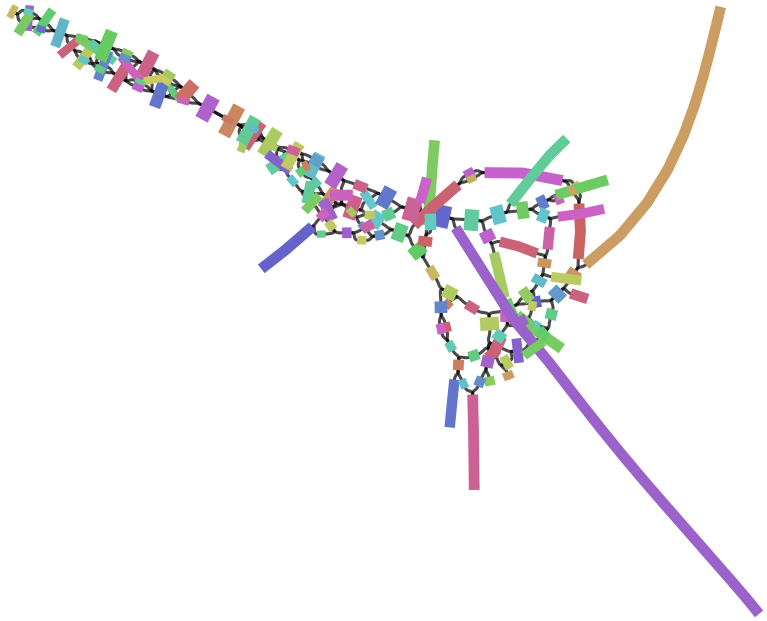


Figure 1. Example of a complex layout in a part of the *B. cereus* assembly

Now let's find out how to zoom in on a specific node in the layout. First let's find a suitable node to work on.

☐ Click on any node in a region with a complex layout. An example of such a region is shown in Figure 1.

☐ Under **Find nodes** in the right-hand panel, type the number of the node about which you'd like information, then click "**Find node(s)**"

The viewer will zoom in on the node(s) you selected within the context of the entire layout graph. As you can see, the layout is quite complex and hard to interpret. We will need to zoom in more closely on regions for more clarity.

If we wish to view the environs of our selected node with more clarity, we can change the **Scope** under "**Graph drawing**" (left-hand panel).

☐ Select "**Around node**(s)" using the **Scope** pull-down menu. Type your node number into the "**Node(s)**" box that appears, increase **Distance** to 4, and then click "**Draw Graph**". This will show your favorite node at the center of the graph and linked nodes extending up to 4 nodes away.

This ability to "see" complex layouts is a great benefit when trying to resolve genomic regions that are poorly assembled. Plus, it can tell you if your sequence of interest is in a poorly assembled region. **Bandage can even perform BLAST searches to find out in which node(s) any given query sequence resides.**


DAY
3

Module 4

Sequence Search and Alignment: BLAST

Goal:	Be able to install and use the Basic Local Alignment Search Tool (BLAST) to align and compare sequences. - Search the NCBI non-redundant (nr) BLAST database with a local query file - Create a BLAST database from a local sequence - Search the local database with a local sequence

Input:	`blast/MoTeR_retrotransposons.fasta`
	`blast/MoRepeats.fasta`
	`blast/magnaporthe_oryzae_70-15_8_supercontigs.fasta`

Output:	`blast/MoTeRs.nrBLASTn` `blast/MoRepeats.Moryzae_genomeBLASTn1`

4.1 Run a web blastn search

To demonstrate web BLAST, we are going to use two sequences from the genome of the fungus *Magnaporthe oryzae* and then search for related sequences in the NCBI DNA sequence database. The queries, known as MoTeR1 and MoTeR2, occur in multiple copies in the *M. oryzae* genome and are telomeric retrotransposons that code for proteins that operate on RNA copies of the MoTeR elements; they insert the sequences into new chromosome locations via reverse transcription. MoTeRs are special transposons because they insert specifically into telomeres, which are the sequences that form the ends of linear chromosomes.

- ☐ Use **scp** (or **WinSCP**) to transfer the *MoTeR_retrotransposons.fasta* file from the *blast* directory on your VM to your local machine. See the instructions in the Assembly lab for a reminder on how to transfer files from your VM to your local PC.
- ☐ One the file is saved, navigate to the directory where you saved it and double-click to open the downloaded file.

- ☐ Copy the two sequences contained in the file.
- ☐ Go to https://blast.ncbi.nlm.nih.gov/Blast.cgi.
- ☐ Select the link Nucleotide BLAST. This will open up the **blastn** search page
- ☐ Paste the copied sequences into the "**Enter Query Sequence**" window.
- ☐ For the Database, make sure "**Nucleotide collection (nr/nt)**" is selected, and "**Organism**" is empty.
- ☐ Click the **BLAST** button.
- ☐ On the Results descriptions page that shows up when the search is finished, find the field where it tells you the of length(s) of the query sequence(s) that was submitted.
- ☐ What is the length of the MoTeR1 query sequence? ________________

Note: if you ever "lose" the Results page, you can navigate to the Recent Results tab at the top of any BLAST page to revisit your results (without having to redo the search).

- ☐ Click on the Graphic Summary tab and examine the output. Mouse over the colored lines. Note the name of the respective BLAST hit pops up. Click on the top line, then select **Alignment**.
- ☐ You will see a DNA sequence alignment that will look like that shown in Figure 2:

Magnaporthe oryzae clone FH10L7 non-LTR retrotransposon MoTeR1 reverse transcriptase gene, complete cds; and non-LTR retrotransposon MoTeR2, complete sequence

Sequence ID: JQ747490.1 Length: **48758** Number of Matches: **9**

Range 1: 42992 to 48025 GenBank Graphics ▼ Next Match ▲

Score	Expect	Identities	Gaps	Strand
9297 bits(5034)	0.0	5034/5034(100%)	0/5034(0%)	Plus/Minus

```
Query  1      cccgaacccgaacccaaacccaaacccaaacccaaacccaaaccc
              ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  48025  CCCGAACCCGAACCCAAACCCAAACCCAAACCCAAACCCAAACCC

Query  61     cccGGAGGGTTCCCAAGTCGCCTAAACCCGAAGGGTTTAGGATAT
              ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  47965  CCCGGAGGGTTCCCAAGTCGCCTAAACCCGAAGGGTTTAGGATAT

Query  121    AATTGGATAATTATTTACCCCTGTTGGACAGGGGGGTTGCAGGGG
              ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  47905  AATTGGATAATTATTTACCCCTGTTGGACAGGGGGGTTGCAGGGG

Query  181    TTATTATTTATGCGCCGTTTATTTGTTTAcccccccAAATATTAT
              ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  47845  TTATTATTTATGCGCCGTTTATTTGTTTACCCCCCCAAATATTAT
```

Figure 2. Blast alignment between our query sequence and one of several "hits" in the NCBI database.

The "header" lines contain information about the matching NCBI entry. This is followed by metrics regarding the alignment: score, e-value (likelihood that this is not a true match), identity, number of gaps that needed to be created to maintain alignment, and which DNA strands were aligned. The next lines show the portions of the query sequence and the database entry that were aligned. By convention, the query is always represented first and is always drawn in the plus orientation (i.e., corresponding to the strand that we used to search with).

Look at the longest alignment and answer the following questions:

- ☐ What is the NCBI identifier (ID) for the subject (database) sequence that best matches the MoTeR1 query? ________________
- ☐ What is the total length of the query sequence (not just the alignment)? ________________
- ☐ What is the total length of the subject sequence? ________________________________
- ☐ Does the MoTeR1 query match the forward or reverse complement sequence of the database entry (subject sequence)?

____

- ☐ Use arrowed lines to draw the entire query and subject sequences (to approximate scale) to show how the MoTeR1 sequence aligns relative to the full sequence of the database entry. Show the coordinates of the alignments. (Note: this is important because we should always try and conceptualize how sequences align with one another.) An example is shown below.

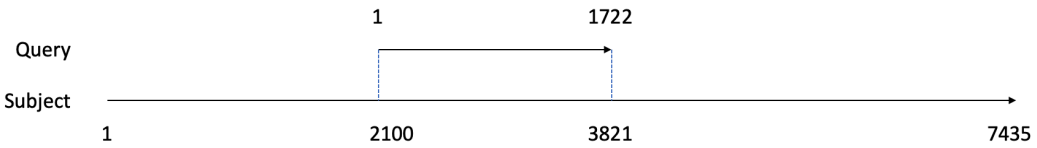


Figure 3. Graphical representation of BLAST alignment

- ☐ Navigate to the top of the page and use the "**Results for:**" pulldown menu to select the results for the MoTeR2 query sequence:
- ☐ Repeat the previous steps for the second query sequence.

Answer the same questions as above for MoTeR2:

- ☐ What is the NCBI identifier (ID) for the subject (database) sequence that best matches the MoTeR2 query? _____
- ☐ What is the total length of the query sequence? _____
- ☐ What is the total length of the subject sequence? _____
- ☐ Does the MoTeR2 query match the forward or reverse complement sequence of the database entry (subject sequence)? _____
- ☐ Use lines to draw the entire query and subject sequences (to approximate scale) to show how the MoTeR2 sequence aligns relative to the full sequence of the database entry.

4.2 Run a web blastx search (nucleotide query/protein DB)

- ☐ At the top of the blast results page, click on the "**Edit Search**" link
- ☐ Copy the sequence in the "**Enter accession number(s)**" field (click in the sequence, Select All and copy) and click the **blastx** tab above.
- ☐ Paste the copied sequences back into the "**Enter accession number(s)**" window.
- ☐ For the Database, make sure "**Non-redundant protein sequences (nr)** is selected, and "**Organism**" is empty.
- ☐ Click the **BLAST** button.
- ☐ When the result pops up, examine the Graphic Summary. Click on the top line. Then, select "**Alignment**".

Carefully examine the alignment for the MoTeR1 protein and answer the following questions:

- ☐ What type of protein does the query sequence encode? _____
- ☐ At what nucleotide position in the query does the start codon occur? _____

☐ At what nucleotide position in the query does the stop codon occur? _____

☐ How many amino acids does the encoded protein contain? _____

4.3 Run a local → remote BLAST search

☐ Change to your *blast* directory.

☐ Use your locally installed **blastn** program to search the NCBI database using the query file *MoTeR_retrotransposons.fasta*:

```
• blastn -remote -db nr -query MoTeR_retrotransposons.fasta
  -evalue 1e-20 -outfmt 0 -out MoTeRs.nrBLASTn0
```

`-remote`	tells the program to search a remote (NCBI) database
`-db`	specifies the database to be searched (we will use the NCBI "nr" database)
`-query`	specifies the local query sequence file (path to file must be included)
`-out`	name of output file
`-evalue`	tells program to only report matches with ≤ specified value
`-outfmt`	specifies format of output (values can range from 0 to 11). Possible formats are listed below: 0 = pairwise 1 = query-anchored showing identities 2 = query-anchored no identities 3 = flat query-anchored, show identities 4 = flat query-anchored, no identities 5 = XML Blast output 6 = tabular 7 = tabular with comment lines 8 = Text ASN.1 9 = Binary ASN.1 10 = Comma-separated values 11 = BLAST archive format (ASN.1)

☐ When the search is complete (or even before) you can use **less** to inspect the resulting output file, *MoTeRs.nrBLAST0*

You will hopefully recognize some of the matches, as the alignments should be equivalent to the ones that you obtained earlier (unless the database has been populated with a new, matching sequence since that time).

4.7 Search a custom BLAST database

The latest versions of **ncbi-blast+** convert the "subject "sequence into a BLAST-searchable database "on the fly," so we can just give **blastn** the name of the subject that we want to search**.**

- ☐ Let's run a **blastn** search using the sequence in *MoRepeats.fasta* as the query and your genome as the database (subject):

```
• blastn –subject magnaporthe_oryzae_70-15_8_supercontigs.fasta
  –query MoRepeats.fasta –evalue 1e-20 –outfmt 0 –out
  MoRepeats.Moryzae_genomeBLASTn0
```

The *MoRepeats.fasta* file contains the sequences of most of the transposons in the *Magnaporthe* genome. Because these elements can copy themselves, they usually exist in multiple copies in the genome.

- ☐ Examine your output file with **less.**
- ☐ Now is a good time to learn about the different output format options (0 through 11). Try them all and try and understand how to interpret each output. Keep track of formats in the output file names. As an example, you could record the results for `–outfmt 6`, as shown below. **For efficiency, use the up arrow on your keyboard to pull up the previous command and then change the numerical value for -outfmt and in the output file before hitting return.**

```
• blastn –subject magnaporthe_oryzae_70-15_8_supercontigs.fasta
  –query MoRepeats.fasta –evalue 1e-20 –outfmt 6 –out
  MoRepeats.Moryzae_genomeBLASTn6
```

- ☐ Use **less** to inspect at the various output formats and make sure you understand how each one represents the alignment information. You can find more information about formats 6 and 7 at http://www.metagenomics.wiki/tools/blast/blastn-output-format-6

Part II. BLAST and the command line

Goal:	Learn how to use the command line to: - extract specific information from blast reports - reconfigure blast reports for easy interpretation
Inputs:	`blast/MoRepeats.Moryzae_genomeBLASTn6 file`
Outputs:	various metrics and reconfigured blast reports

4.8 Using BLAST to answer biologically relevant questions

Up to this point we have used BLAST to i) Find MoTeR-like sequences in the **nr** database; ii) determine if MoTeRs code for proteins and learn a little bit about those proteins (length, function, etc.); and iii) find repeated sequences in the *Magnaporthe oryzae* genome. Taking the last search as an example, if we consider the **blast** results for the MAGGY retrotransposon against the *M. oryzae* genome, the simple discovery of hits is not all that informative because we already knew that most *Magnaporthe* strains contain MAGGY sequences before genome sequencing was even possible.

On the other hand, let's say we know that MAGGY is a highly repeated sequence, but we wish to find out how many full-length, uninterrupted MAGGY sequences are present in the *M. oryzae* genome. This would be new information. **More importantly, however, there aren't any existing tools to answer this question, so we must think of ways to extract this information from our blast search using our existing bioinformatics knowledge.**

It turns out that you already know command line programs that will allow you answer these questions using **blastn** output files. The key is to identify suitable program(s), generate one or more appropriately filtered **blast** outputs, and then apply the tools to extract the desired information.

After completing the following section, you should be able to answer the above question (and many more).

4.9 Filtering BLAST results based on specific criteria

Sometimes we use BLAST simply to find out what other sequences match the input(s) to the search, and that can provide us information about our query sequence(s). However, quite often we use BLAST to address higher level questions. Above, we consider the problem of finding how many full length MAGGY elements are present in the *Magnaporthe oryzae* genome. Answering this question requires that we use

additional programs to filter the BLAST results. Here we will learn some useful command line tools that allow us to extract information from BLAST reports based on different criteria.

- ☐ Before we get started, make sure you understand BLAST output format 6 by reading the information below.

The columns in the blast format 6 output are as follows:

1. query sequence ID
2. subject sequence ID
3. percent sequence identity
4. length of alignment
5. number of mismatches in alignment
6. number of gaps in alignment
7. start position of alignment in query
8. end position of alignment on query
9. start position of alignment in subject
10. end position of alignment in subject
11. alignment score
12. e-value (probability of false alignment, based on database composition)

You can read more about this format- or even how to define your own output format - at the following link:

http://www.metagenomics.wiki/tools/blast/blastn-output-format-6

4.10 Extracting specific lines of results

Before we tackle the question about full length MAGGY copies in the *M. oryzae* genome, let's start filtering BLAST reports using different criteria. First, suppose we wish to know which repeats are found on chromosome 8.7 (chromosome 7 in genome assembly version 8).

- ☐ Change into the *blast* directory.
- ☐ Make sure you that you have a copy of the *MoRepeats.Moryzae_genomeBLASTn6* file that you should have created earlier. If not, perform the relevant **blastn** search to create it now.
- ☐ We can use grep for this task, but we have to be careful. First, try this:

```
• grep 8.7 MoRepeats.Moryzae_genomeBLASTn6
```

Did the command you ran return results for just chromosome 8.7? No, it didn't. This is because "." has a special meaning in regular expressions, the "language" that **grep** and many other programs use to represent patterns in text. More specifically, "." matches any single character. Hence, our current query will result in matches to 8.7, 807, 817, 8a7, etc.

- ☐ We need to "escape" the special meaning of "." so that **grep** interprets the character literally. We can do this by using two backslashes before the period - try it:

```
• grep 8\\.7 MoRepeats.Moryzae_genomeBLASTn6
```

☐ Note how this search also returned some hits to numerical values in other columns. We can solve this by expanding our search term:

```
• grep Chromosome_8\\.7 MoRepeats.Moryzae_genomeBLASTn6
```

☐ Or we can use **awk** to specify matches only in the second column.

```
• awk '$2 ~/8.7/' MoRepeats.Moryzae_genomeBLASTn6
```

Note that the above command actually says find lines with "8, followed by anything, followed by 7" in column 2 but it's not necessary to escape the period because the only possible occurrence in column 2 is in the term "chromosome_8.7."

☐ If we wanted to find the term "8.7" in column three we would need to escape the period but, here, we can use a single backslash because the **awk** command is in single quotation marks.

```
• awk '$3 ~/8\.7/' MoRepeats.Moryzae_genomeBLASTn6
```

☐ Try the above command with and without the backslash.

4.11 Sorting results

Let's imagine we wish to understand the linear order of repeat sequences on chromosome 8.7. This is not easy to visualize using the default output for **blast** because the alignments are ordered according to highest score - which is based on alignment length and accuracy, as opposed to chromosomal position.

So, to uncover the order of repeat sequences on chromosome 8.7, we will first need to extract results for the chromosome and then **sort** them according to chromosome position.

☐ Try the following:

```
• awk '$2 ~/8\.7/' MoRepeats.Moryzae_genomeBLASTn6 | sort -k9n
```

`-k` sort according to a key (in this case column 9)

`-n` sort numerically

You should see that the results are now sorted based on chromosomal positions*.

☐ Try adding the option "r" to the end of the previous command (-k9nr) . How did that affect the results? _____

***Note that the above use of sort did not provide a perfect ordering of matches. This is because the start and end positions of each alignment depend on whether the query sequence matches the chromosome on the forward or reverse strand. If we want, we can create a blast database from the *MoRepeats.fasta* file and use the *M. oryzae* genome as a query. Then, we would be able to sort the alignments in perfect order.**

4.12 More filtering examples

We considered earlier how to identify and count full-length MAGGY copies in the *M. oryzae* genome. To accomplish this, we need to evaluate the length of each **blast** match. Conveniently, we can do this by using grep to extract lines with MAGGY matches and then piping the results through **awk.**

☐ Try this (why do you think we use the value 5638 in the **awk** evaluator?):

- `grep MAGGY MoRepeats.Moryzae_genomeBLASTn6 | awk '$4 >= 5638'`

Remember **awk** typically uses the following syntax:

```
awk –F <field separator to use> 'condition to fulfill {action if condition satisfied}'
```

Note that, because spaces and tabs are the default delimiters, and printing the whole line is the default action, we can abbreviate the command to just the conditional statement.

☐ How about if we wish to identify alignments involving MAGGY copies that are at least 90% of full-length?

- `grep MAGGY MoRepeats.Moryzae_genomeBLASTn6 | awk ‘$4 >= 5638*0.9’`

☐ Similarly, we can use **awk** to extract **blast** matches in specific ranges. Let's look for MAGGY matches that occur between positions 2,000,000 and 3,000,000 on chromosome 8.7 (and, for fun, let’s sort them according to their position on the chromosome too!).

- `grep MAGGY MoRepeats.Moryzae_genomeBLASTn6 | awk '$2 ~ /8\.7/ && $9 > 2000000 && $9 < 3000000' | sort –k9n`

note how we use && to specify "and" (|| would mean "or")

☐ Now suppose we wish to have a list of the repeats that occur on chromosome 8.7. First, we need to extract alignments that involve chromosome 8.7. Then, we want to print out a non-redundant list of the entries found in the query column (#1).

- `awk '$2 ~ /8\.7/ {print $1}' MoRepeats.Moryzae_genomeBLASTn6 | uniq`

Note: Often, we have to **sort** before using **uniq** because the latter program only works on a sorted input. In this case, we do not need to sort because **blast** already sorts the query sequences for us when we select output format 6.

☐ How would you use the command line to determine if chromosome 8.7 contains at least one copy of each of the repeat sequences present in the *MoRepeats.fasta* file?

_____

4.13 Re-ordering and sub-setting output data

Sometimes, we wish to retain only certain information from a data file. Not surprisingly, we can use **awk** to perform the necessary steps.

- ☐ Let's suppose we wish to delete the last two columns and then swap the positions of the columns that refer to the query and subject sequences. We can tell **awk** to print the columns in any order we wish:

```
• awk '{OFS="\t"; print $2, $1, $3, $4, $5, $6, $9, $10, $7, $8}'
  MoRepeats.Moryzae_genomeBLASTn6
```

In the command above, the OFS (Output Field Separator) variable is set so that **awk** will insert a tab between the output columns. By default, it uses spaces.

- ☐ Try running the above command using a comma to separate output values (or omit the OFS command to insert spaces).

4.14 Practice questions

Have a go at answering the questions below and, for each answer, describe: a) which blast output format(s) would be needed/most suitable; b) which command line tools you could employ; and c) in general terms, how would you apply these tools to extract the desired information? Try and generate working code to generate the answers to the below questions.

The Farman lab studies telomeres which are the sequences that define the ends of linear chromosomes. In the fungus, *Magnaporthe oryzae*, the telomeres comprise tandem repeats of the hexanucleotide motif, CCCTAA.

- ☐ Use a text editor to create a new query sequence containing ten, consecutive copies of the telomere repeat motif, CCCTAA. Use this query to search the *M. oryzae* genome. Y**ou will need to give a new option to blast, -dust no. Otherwise, the program filters out alignments for highly repeated sequence motifs.**
- ☐ How many chromosomes contain telomere sequences? _____
- ☐ Are all the matches near to chromosome ends? _____
- ☐ How many telomeres have been captured in this assembly? _____
- ☐ To date, every *M. oryzae* strain analyzed has least seven chromosomes. Based on the answers above, what conclusions can we make about the completeness of the *magnaporthe_oryzae_70-15_8_supercontigs.fasta* assembly? _____
- ☐ Suggest possible reasons _____
- ☐ Think about possible solutions _____

APPENDIX: Installing BLAST

Download the latest version of BLAST

We will download the BLAST binaries directly from the NCBI website. Note: we will be installing **blast** on our virtual machine. However, if you wish to install the program on your own computer, you could download the Mac or Windows versions.

- ☐ Go the NCBI homepage at http://www.ncbi.nlm.nih.gov/
- ☐ Click the "**Data and Software**" link (in the left-hand panel).
- ☐ Click the "**BLAST (Stand-alone)**" link.
- ☐ Click the **ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/LATEST/** link under "**BLAST+ executables**".
- ☐ Find the latest executable for Linux (e.g. *ncbi-blast-2.X.X.0+-x64-linux.tar.gz*). **Note: if the link opens an ftp client on your local machine, navigate to the latest Linux executable and then right-click to copy the URL.**

If we were to click on this link, it would download the file to the machine that we are working on—not the Linux server where the program needs to be installed.

Instead, we will copy the link to the file to make downloading it via the command line easier. We have a 64-bit ("x64") Linux system, so right-click the *ncbi-blast-2.X.X.0+-x64-linux.tar.gz* link and select "Copy Link Address" or the equivalent in your browser.

We will now use the command line to download the latest **BLAST** executables from the NCBI server straight into to the home directory on your virtual machine. Remember, the program we use for downloading from the web is **wget**.

- ☐ Change into the *blast* directory.
- ☐ Download **blast** with **wget** using the link you copied. (Remember that you can paste text in **Terminal** with Command + V. In **PowerShell** and **PuTTY**, you can right click.)
- ☐ After the download finishes, extract the file. (Replace "`X.X`" below with the version of appropriate version number for the file you downloaded.)

```
• tar zxvpf ncbi-blast-2.X.X+-x64-linux.tar.gz
```

  - `-z` decompress using **gzip**
  - `-x` extract files to disk
  - `-v` verbose mode—writes progress to screen
  - `-p` preserve file permissions
  - `-f` read archive from specified file

- ☐ Verify that the *ncbi-blast-2.X.X+* directory has been extracted from the tar file.

Install the BLAST executables

We've downloaded the latest NCBI BLAST package, but how do we run the new version of **blastn**? Let's look for the **blastn** executable under the *ncbi-blast-2.X.X+* directory.

- ☐ List the *ncbi-blast-2.X.X+* directory. (Make sure to change *X.X* to the reflect the version you just downloaded.)

You should see two directories and a few files that provide some useful information about the software, but **blastn** is not present in that directory. We could manually search through the two directories, but that could be time consuming if the directory structure is deeply nested. Fortunately, we can locate our files using **find**, a tool built to search for files by recursively traversing directory trees.

- ☐ Use **find** to search for **blastn** under the *ncbi-blast-2.X.X+* directory. (Again, don't forget to change *X.X*.)

```
• find ncbi-blast-2.X.X+/ -name 'blastn'
```

find should return a single result—a **blastn** file under the *bin* directory. In this case, we might have guessed the executable was in the *bin* directory without using **find**—on Linux, binary executables are conventionally located in a directory named *bin*. Nevertheless, **find** is useful to know any time you are unsure where a file is located.

- ☐ Change to the *ncbi-blast-2.X.X+/bin* directory.
- ☐ Check which version of **blastn** is running.

```
• blastn -version
```

You should see that we are still getting the same result as before, version 2.9.0+. That's because the system copy is in a location that the OS searches first when it looks for executables. To execute the new **blastn**, we will need to tell the system specifically to run the one in the current directory.

- ☐ Run the version of **blastn** we just downloaded.

```
• ./blastn -version
```

Remember that ./ is the current directory.

We could specify a path to the **blastn** every time we want to run the command, but that would require much typing. It would be convenient if we could let the system know to run our new **blastn** by default when we type **blastn**.

How does the system know where to find **blastn** when we run the command without ./? When executing a program without a location specified, the shell (the program that interprets your commands) looks for the program in the locations specified in the PATH environment variable.

- ☐ Show the contents of PATH. Variables in a command (denoted by a dollar sign prefix) are replaced by their actual values.

```
• echo $PATH
```

The command above should print a colon-separated list of paths to directories containing executables. The system prioritizes directories earlier in the list when searching for a program.

PATH isn't the only environment variable set on your system. You can see the others using the **env** command.

We can ask where a program will be found using the **which** command.

☐ Determine where the **blastn** command is found.

```
• which blastn
```

You should see that **blastn** is located at */usr/bin/blastn*.

We also could have used the **locate** command, which searches for files throughout the system using string matching.

☐ Find files containing the string "blastn" in their paths.

```
• locate blastn
```

We get many hits with **locate.** There are multiple versions of **blastn** (and other similarly named programs) on the system, but */usr/bin/blastn* is the first (and only) one on the PATH.

We can also use **whereis**, which looks for binary executables (and some other kinds of files) on the PATH and various common locations.

☐ Find **blastn** with **whereis**.

```
• whereis blastn
```

whereis shows two locations—the location in */usr/bin* and the a location in */opt/rmblast-2.10.1/bin*; the latter **blastn** is needed indirectly for **MAKER**, a genome annotation pipeline we will discuss in a later lab.

Let's add the directory with the executables to your PATH environment variable.

☐ Show your current PATH again.

```
• echo $PATH
```

```
/opt/miniconda3/bin:/opt/miniconda3/condabin:/usr/local/sbin:/usr/local
/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/opt/pup
petlabs/bin:/opt/miniconda3/bin
```

☐ Now change the PATH to add *~/blast/ncbi-blast-2.X.X+/bin*. The "export" keyword ensures that the programs we run from the shell see changes to the PATH as well. (We could omit "export", but then the programs we run would see the old PATH.)

```
• export PATH=~/blast/ncbi-blast-2.X.X+/bin:$PATH
```

Here, we told the system to search *~/blast/ncbi-blast-2.X.X+/bin* for executables BEFORE it searches any other stored paths. That way, we can be sure it finds (and uses) the **blastn** version that we just installed first.

- ☐ View PATH again:

```
• echo $PATH
```

```
/home/myName/blast/ncbi-blast-2.X.X+/bin/:/opt/miniconda3/bin:/opt/mini
conda3/condabin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin
:/bin:/usr/games:/usr/local/games:/opt/puppetlabs/bin:/opt/miniconda3/b
in
```

- ☐ Check which **blastn** we would run and verify that we are using the **blastn** we downloaded.

```
• which blastn
```

Although it is often convenient to make changes to the PATH this way, the change we made will not persist after we log out. Let's verify that.

- ☐ Log out of your VM and log back in.
- ☐ Verify that we are using the old version of **blastn** again.

To make our change to the PATH persistent, we can have **bash**, the shell that interprets our commands, to run the **export** command each time we log in.

- ☐ Create a new *.bash_profile* file in the **nano** text editor.

```
• nano .bash_profile
```

bash expects the *.bash_profile* file to be a **bash** script—i.e., a file containing commands for **bash** to execute. *.bash_profile* is read and executed by **bash** every time you start a new **bash** "login shell"—this happens, for example, when you login via SSH. You could put any commands you like in this file to be executed each time you login. Typically, *.bash_profile* contains custom settings that allow you to configure the way your shell looks and behaves. In this case, we will tell the shell to add a new file path to the ones where it normally searches for executables by setting the PATH variable.

- ☐ In your *.bash_profile*, type the following, replacing "`yourusername`" with your actual username and replacing X.X with the appropriate version number.

```
export PATH="/home/yourusername/blast/ncbi-blast-2.X.X+/bin:$PATH"
if [ -f /home/yourusername/.bashrc ]; then
    source /home/yourusername/.bashrc
fi
```

The first command we type sets PATH tells the shell to search the *ncbi-blast-2.X.X+/bin* directory before it searches the default path. The rest of the script tells **bash** to also run your *.bashrc* file if it exists. *.bashrc* is similar to *.bash_profile*, but it runs for interactive non-login shells like the ones we usually create when we make a new **screen** session.

- ☐ Close the file by hitting Control + x. Accept the modifications by typing "y" and hit enter.
- ☐ Tell the shell to read the new profile without having to open a new terminal.

  ```
  • source .bash_profile
  ```

  Make sure you don't see any errors when running the command above.
- ☐ Check to see if the new **blast** program is in your path.

  ```
  • blastn –version
  ```

You should get a response indicating that the current version is now 2.X.X+, the new version we installed. If so, let's get blasting…

If you notice that **ls** no longer produces colorized output when you next log in, you may have made a mistake in your *.bash_profile*. Check that your *.bash_profile* matches the example given above and that you have replaced "yourusername" and X.X with the correct values.

Module 5

Part I. Gene Prediction

Goal: Learn to utilize the command line programs **SNAP** and **AUGUSTUS** to search genomes for predicted genes.

5.1.1 SNAP

SNAP is an *ab initio* gene-finding program that uses a hidden Markov model to search for regions of a genome that are likely to be genes. **SNAP** and other hidden Markov model gene-finders use statistical properties of the DNA sequence to infer features such as start codons, splice junctions, and untranslated regions.

Korf, I. Gene finding in novel genomes. *BMC Bioinformatics* 5:59, 2004.

https://github.com/dzerbino/velvet/tree/master

To predict genes well, these methods need a description of the statistical parameters of the model: that is, what do genes look like in the organism under study?

SNAP ships with a number of parameter files (models) for existing organisms and can be trained on other organisms.

☐ If we run in screen we need to provide a special argument that tells screen to begin **bash** as a login shell. This is required so that environment variables needed by the programs we will be using are set correctly (**note the bash option is a lowercase L**).

```
• screen -S genes bash -l
```

☐ First, we'll list the available models. The environment variable `ZOE` says where **SNAP** is installed:

```
• echo $ZOE
```

☐ Look inside that directory:

- `ls /usr/share/snap`

☐ The HMM directory contains the parameter files:

- `ls /usr/share/snap/HMM`

☐ Look at the contents of one of the parameter files:

- `less /usr/share/snap/HMM/C.elegans.hmm`

SNAP parameter files typically have the *.hmm* extension, although, as you can see from files such as *fly*, *rice*, and *thale*, the extension is not necessary. It doesn't look like any of the parameter files are particularly closely related to our organism (a fungus), so we will have to train our own.

5.1.2 Preparing training data

Goal:	Use annotations from a related genome to train **SNAP** to recognize *M. oryzae* genes.
Input):	`FH_V2_maker.gff (a list of gene feature coordinates in a previously annotated genome of M. oryzae)`
Output):	`Moryzae.hmm`

For our training data, we will use gene annotations from the closely related FH_V2 genome. These annotations were generated using the **MAKER** gene annotation pipeline, which we will discuss in detail later.

We will begin by moving to a new directory to work in because **SNAP** training produces many files, and we want to avoid clutter.

☐ Change to the *snap* directory under ~/*genes* where we'll keep our intermediate results and output.

We have a FASTA file with the genome sequences and a GFF file from **MAKER** with both gene annotations and genome sequences. However, **SNAP** requires the training genes be in a custom format (used only by **SNAP**) called "ZFF". Fortunately, **MAKER** comes with a script to convert its annotations into ZFF format.

☐ Convert the **MAKER** annotations to ZFF for **SNAP:**

- `maker2zff FH_V2_maker.gff`

The command above creates files *"genome.ann"* (ZFF format) and *"genome.dna"* (FASTA format) in the current directory. The *.ann* file represents the positions of exons and genes within each contig, while the *.dna* file contains simply the contig sequences.

- ☐ Now we can use **fathom**, a sequence analysis and extraction tool that comes with **SNAP**, to extract the gene sequences from these two files.

Usage: `fathom <annotations> <contigs> –<subcommand> [options]`

The **fathom** tool does not require specific file extensions; however, the annotations file must be in ZFF format, and the contigs file must be in FASTA format.

A number of subcommands are available; we will use only a few. For more information, you can run

`fathom –help | less`.

- ☐ First, get some information on the annotations with `–gene–stats`. This subcommand outputs to the screen the number of sequences (contigs), the number of genes annotated, GC content, average intron and exon lengths, and more.

  - `fathom genome.ann genome.dna –gene–stats`

  Note: It is expected to see error messages about one or two gene models; this can occasionally happen when the annotations contain overlapping genes on different strands.

- ☐ Now we will extract the genome regions containing unique genes using the `–categorize` subcommand. This subcommand creates a number of pairs of ZFF and FASTA files: one pair for regions with errors, one for regions with overlapping genes, and so on. We will ask for up to 1000 base pairs of intergenic sequence on both sides of each gene to help train the HMM about what sequences are likely to occur near genes.

  - `fathom genome.ann genome.dna –categorize 1000`

  Note: Once again, it is expected to see error messages about one or two gene models.

- ☐ Look at the contents of the directory; there are now a number of pairs of *.ann* (ZFF format) and *.dna* (FASTA format) files. Look at `fathom –help` again to see what these categories mean.

- ☐ Training **SNAP** usually works best with unique, non-overlapping genes without alternative splicing; these annotations may be found in *uni.ann*, and the accompanying sequences in *uni.dna*. Let's look at their statistics again:

  - `fathom uni.ann uni.dna –gene–stats`

- ☐ Next, we will use the `–export` subcommand to extract the genome, transcript, and protein sequences from these genes. We must tell `–export` as well to keep 1000 base pairs of context and also (with `–plus`) to flip genes that are on the reverse strand (that is, with their 3' end nearer the beginning of the contig than their 5').

- `fathom uni.ann uni.dna –export 1000 –plus`

The output is in four files:

export.ann	ZFF annotations for the exons of each gene.
export.dna	DNA sequence for each gene, including introns and flanking regions.
export.tx	DNA sequence for each transcript.
export.aa	Protein sequence for each gene.

☐ Use `–gene–stats` again to review the statistics of the *export.ann* and *export.dna* files again; the statistics should be very close, if not identical, to those for *uni.ann*.

☐ Finally, we have the data in suitable format and are ready to train the HMM. The **forge** tool does this part of the process.

- `forge export.ann export.dna`

☐ After a few seconds, you will have a large number of *.model* and *.count* files representing the model parameters: probabilities and frequencies of different types of nucleotide sequences and gene features. List the directory to check that the files were created.

☐ The number of files is too unwieldy for most uses, so we will use one final **SNAP** tool, **hmm-assembler.pl** to condense everything into a single file for use with runs of **SNAP**.

- `hmm–assembler.pl Moryzae . > Moryzae.hmm`

The first argument, `Moryzae`, is the name to use in the header of the output, mostly for identification purposes; it should be a sequence of letters. The second argument is the directory in which to find all of the *.model* and *.count* files—here, the current directory. The program writes the model to standard output, so we use ">" to redirect it to a file.

5.1.3 Gene calling with SNAP

Goal:	Predict genes using **SNAP**.
Input(s):	`magnaporthe_oryzae_70–15_8_supercontigs.fasta` `Moryzae.hmm`
Output(s):	`magnaporthe_oryzae_70–15_8_single_contig–snap.gff2` `magnaporthe_oryzae_70–15_8_single_contig–snap.zff`

Now that we have our parameter file *Moryzae.hmm*, we can get to the business of predicting genes. We'll test **SNAP** out on just a single contig.

☐ Extract a single contig from your *Magnaporthe oryzae* assembly located in the *~/blast* directory into a file. The instructions following the creation of the single contig file (see below) assume

that the file is in *magnaporthe_oryzae_70-15_8_single_contig.fasta*, but you are welcome to use a different name or location.

- `samtools faidx ~/blast/magnaporthe_oryzae_70-15_8_supercontigs.fasta Chromosome_8.7 > magnaporthe_oryzae_70-15_8_single_contig.fasta`

After specifying the filename and location, we specify the id of the sequence we want to extract. The id of our single contig is >`Chromosome_8.7.` **Note** that the ">" at the beginning of the sequence ID is not included in the command.

Finally, we redirect the output to *magnaporthe_oryzae_70-15_8_single_contig.fasta* using ">".

☐ To run **SNAP**, give it the name of your parameter file and your FASTA file. Output goes to standard output, so you probably want to redirect it:

- `snap-hmm Moryzae.hmm magnaporthe_oryzae_70-15_8_single_contig.fasta > magnaporthe_oryzae_70-15_8_single_contig-snap.zff`

☐ Look at the resulting *magnaporthe_oryzae_70-15_8_single_contig-snap.zff*. Use **fathom** again to compute its statistics, using the sequence file *magnaporthe_oryzae_70-15_8_single_contig.fasta*.

- `fathom magnaporthe_oryzae_70-15_8_single_contig-snap.zff magnaporthe_oryzae_70-15_8_single_contig.fasta -gene-stats`

☐ The default output is in ZFF format, which can be used as input for training **SNAP**, but must be converted to work with most other programs. It is also possible to generate a GFF file in the older GFF2 format:

- `snap-hmm Moryzae.hmm magnaporthe_oryzae_70-15_8_single_contig.fasta -gff > magnaporthe_oryzae_70-15_8_single_contig-snap.gff2`

☐ Look at the resulting file: *magnaporthe_oryzae_70-15_8_single_contig-snap.gff2*. Compare the format to that of the GFF3 file: *FH_V2_maker.gff*. Note that it is very similar but column 9 has a lot more information in the GFF3 file.

5.2.1 AUGUSTUS

Goal:	Predict genes using **AUGUSTUS**.
Input(s):	`magnaporthe_oryzae_70-15_8_single_contig.fasta`
Output(s):	`magnaporthe_oryzae_70-15_8_single_contig-augustus.gff3`

AUGUSTUS is another gene finder, similar in principle to **SNAP**. It uses a similar, but distinct, hidden Markov model to predict genes. Although we will not explore this use, **AUGUSTUS** is capable of incorporating information such as protein alignments into its model's parameters. **AUGUSTUS** is free for all uses and comes with a rather large collection of parameter files, trained on various species.

M. Stanke , O. Schöffmann , B. Morgenstern, S. Waack (2006) Gene prediction in eukaryotes with a generalized hidden Markov model that uses hints from external sources. BMC Bioinformatics 7:62.

http://bioinf.uni-greifswald.de/augustus/

5.2.2 Running AUGUSTUS

- ☐ Change to the **AUGUSTUS** directory in ~/*genes*/*augustus*, which is where we'll keep our intermediate results and output.
- ☐ First, let's look at the help:

```
• augustus --help 2>&1 | less
```

Here we are simply piping the output of **augustus** --help into the **less** text viewer. So, why do we need the `2>&1` in the command above? Some programs, including, **augustus** send certain messages to a stream known as stderr instead of stdout, and only stdout can be redirected by a normal pipe. The 2>&1 allows us to redirect the stderr message (2) to stdout, which we can then pipe into **less**.

- ☐ Let's look at the list of parameter files.

```
• augustus --species=help 2>&1 | less
```

We see that among the listed organisms is *Magnaporthe grisea*. This organism is very closely related to ours, so we won't have to retrain **AUGUSTUS**.

- ☐ Make sure you are still in a screen

```
• echo $STY
```

If you are in a screen session, this will list with a number in front of the name of the active screen:

```
630678.genes
```

☐ If you do not see a screen listed start a new one named augustus:

```
• screen –S augustus bash –l
```

☐ Running **AUGUSTUS** is similar to running other gene finders, although, as usual, the details are different. Rather than specifying a parameter file explicitly, you use the name of one of the included species listed by the above commands.

```
• augustus --species=magnaporthe_grisea --gff3=on
  --singlestrand=true --progress=true
  ../snap/magnaporthe_oryzae_70-15_8_single_contig.fasta
  > magnaporthe_oryzae_70-15_8_single_contig-augustus.gff3
```

This step takes around 6-8 minutes. (If you are not already, you may want to use **screen** to protect against disconnection.)

We give **AUGUSTUS** several parameters here:

`--species=magnaporthe_grisea`	Specifies the parameter file to use. Run `augustus --species=help` for a list.
`--gff3=on`	Produce GFF3-format output. The default is GTF.
`--singlestrand=true`	Predict genes on each strand separately; this option allows **AUGUSTUS** to find genes which overlap on opposite strands.
`--progress=true`	Show a progress bar for each step of the gene finding process. There are two steps per contig, or twice that with the option `--singlestrand=true`. **AUGUSTUS** breaks up contigs larger than about 100 kb and considers each portion separately, which may result in more steps.
`../snap/magnaporthe_oryzae_70-15_8_single_contig.fasta`	The last argument is the name of the genome sequence FASTA file.
`> magnaporthe_oryzae_70-15_8_single_contig-augustus.gff3`	Like many programs, **AUGUSTUS** writes its results to standard output, so you most likely want to redirect to a file. The progress bars and error messages will still be visible.

- ☐ Look at the output file to see the results. In addition to describing the locations and relationships of predicted genes and their features, this GFF file also includes the inferred protein sequence for each predicted gene.

- ☐ Take a look at the GFF files created by **SNAP** and **AUGUSTUS**.

Note how **SNAP** produces a rather basic gff2 report which simply lists initial exons (Einit), internal exons (exon), or terminal exons (Eterm). On the other hand, **augustus** produces an output that is much more feature-rich and more easily interpretable.

Note: If you are interested in training AUGUSTUS for your own organism, the easiest way is to use the AUGUSTUS web server:

http://bioinf.uni-greifswald.de/webaugustus/training/create

Select "AUGUSTUS training submission" and follow the directions; a tutorial and sample data are provided on the web site. Training requires a genome FASTA file, and either a list of cDNA or protein sequences from your organism, or a GenBank-format file with gene annotations. When the run completes, you will receive a file *parameters.tar.gz* containing the parameter files for your species. You may then download and build AUGUSTUS, and extract this file into the *config/species/* directory; if it worked correctly, your species will appear in the output of `augustus --species=help`.

Part II. Combining evidences with MAKER

Goal:	Use **MAKER** to prepare and execute a gene prediction and annotation workflow.

Input(s):	`magnaporthe_oryzae_70-15_8_single_contig.fasta`
	`genes/snap/Moryzae.hmm`
	`genes/maker/Moryzae_RNAseq.gtf`
	`genes/maker/genbank/ncbi-cds-Magnaporthe_organism.fasta`
	`genes/maker/genbank/ncbi-protein-Magnaporthe_organism.fasta`

Output(s):	`maker-annotations.gff3` `maker_proteins.fasta`

MAKER is a gene annotation pipeline that integrates the results of various gene finding programs, along with EST and protein sequences, to produce a single consistent set of annotations that takes both predictions and evidences into account.

Cantarel BL, Korf I, Robb SM, Parra G, Ross E, Moore B, Holt C, Sánchez Alvarado A, Yandell M. (2008) MAKER: an easy-to-use annotation pipeline designed for emerging model organism genomes. Genome Res. 18: 188–196.

http://www.yandell-lab.org/software/maker.html

5.3 Running MAKER

MAKER has a very large assortment of options—too many to specify everything on the command line. Instead, we can control **MAKER** through a set of configuration files. The first step in running **MAKER** is to create the configuration files:

- ☐ Change to your *~/genes/maker* directory.
- ☐ List the contents of the directory. You will see a *genbank* directory, a gff file and a merged
- ☐ Create the **MAKER** configuration files:

```
• maker -CTL
```

This last command generates three files:

maker_exe.ctl	lists the locations of the programs **MAKER** uses.
maker_bopts.ctl	sets thresholds for accepting alignments of EST and protein evidence
maker_opts.ctl	Describes the data to be used as input to **MAKER**, the steps to perform, and options for those steps.

The first two files do not change much from one run to another; we will not be changing them, though it doesn't hurt to take a look at their contents. All of our configuration will take place in *maker_opts.ctl*.

☐ Open *maker_opts.ctl* with a text editor such as nano.

Do not be alarmed by the several pages of options; we will be changing only a few of them. Each option has the format:

```
Option=value
```

Number signs ("#") mark the beginning of a comment that extends to the end of the line. Each option has a short comment explaining its meaning. When editing it doesn't matter whether you keep the comments or not but keeping them might be helpful if you later need to remember what the option does.

There are a few options we can set now.

☐ Find the lines for the following options and edit them as shown, changing directory and file names as appropriate to match the file paths in your workspace.

Hint: To navigate quickly in nano, use Ctrl-W followed by the option name to find the proper option to be edited. For example, for the first option type Ctrl-W `genome` to see all the places where the word genome is located and find the correct option.

genome=*/home/yourusername/genes/snap/magnaporthe_oryzae_70-15_8_single_contig.fasta*

The FASTA file of genome contigs to annotate. We will use just a single contig for our run. You must specify the full path to the file, including your home directory.

model_org= must be set to blank

The organism to use for repeat masking. We want to leave this option blank because masking can take a long time, so we will remove the default "all" setting, leaving the value blank, to turn off this step. Note that the equal sign is still required!

repeat_protein= must be set to blank

If this value is not already blank, disable searching for repeat proteins by removing anything that follows "=." Otherwise, this will greatly increases the running time.

snaphmm=*/home/yourusername/genes/snap/Moryzae.hmm*

The parameter file to use for **SNAP**.

augustus_species=magnaporthe_grisea

The model organism (parameter file) to use for **AUGUSTUS**.

est2genome=1

Infer predictions from protein homology, 1 = yes, 0 = no

keep_preds=1

This option keeps *ab initio* gene predictions in the output, even if there is no EST or related-organism evidence for those predictions.

☐ Save your updated *maker_opts.ctl.*

We will also use evidence from our organism and related ones to help **MAKER** improve its annotations. "Evidence" here refers to new Expressed Sequence Tags (ESTs/RNAseq data) and

protein sequences as well as pre-computed EST alignments. **MAKER** accepts sequences in FASTA format and gene annotations in GFF version 3 format.

- ☐ We will provide **MAKER** with a .gtf file that contains the reference gene set (i.e., the best guess at gene predictions for *M. oryzae),* which has been enriched with RNAseq data. These data not only provide support for most of the predicted genes, but they also reveal chromosome regions that are expressed but were not previously predicted (i.e. cryptic genes).
- ☐ Convert the *merged.gtf* file in the current directory into GFF3 format. The script **gtf2gff3.pl** can do this for us:

```
☐ gtf2gff3.pl merged.gtf > cufflinks.gff3
```

- ☐ Inspect the resulting file to check that it has the correct GFF3 format. The resulting GFF file describes the locations of the splice junctions in a form that **MAKER** can understand. In a moment, we will configure **MAKER** to use this file as evidence for improving gene predictions. Before we do so, we will obtain some additional data from related organisms.
- ☐ Let's explore how to download existing information on *M. oryzae* genes from the National Center for Biotechnology Information (NCBI). Visit the NCBI-GenBank site at http://www.ncbi.nlm.nih.gov/genbank/
- ☐ At the top of the page, select "Nucleotide" from the drop-down, enter "Magnaporthe [organism]" in the search box, and click search. This returns a list of gene sequences that have "Magnaporthe" as part of their organism name. Click on the name of a sequence to look at the GenBank entry for that gene, or click the "FASTA" link below the result to view the gene sequence in FASTA format. With over 15,000 results, it is infeasible to download the sequences one-by-one. Fortunately, GenBank allows downloading the full search results.
- ☐ Return to the search result page; click "Send to" near the top. A drop-down will appear; click "File" there. Then, under "Format" select "FASTA". If you now click "Create File", your download will begin. However, we do not want to overload the network in the computer lab, so we have already placed the results in your directory. Cancel your download if you have begun.

We can now configure **MAKER** to use the evidences we have gathered:

- ☐ Make sure you're in your *~/genes/maker* directory.
- ☐ Use **nano** to open up *maker_opts.ctl* for editing again.
- ☐ Change the following settings:

est_gff=*/home/yourusername/genes/maker/cufflinks.gff3*

This option provides EST evidence in the form of GFF3 annotations. We are providing the set of splice junctions detected with **cufflinks** (a program related to **stringtie** that will be covered in Lab 6); **MAKER** will use these data to refine the sometimes-incorrect predictions from the gene finder programs.

protein=*/home/yourusername/genes/maker/genbank/ncbi-protein-Magnaporthe_organism.fasta*

This option provides protein sequences from this or related organisms, in FASTA format. These sequences are mapped to the genome using **exonerate**

protein2genome, which acts like **tblastn** (protein-versus-DNA) but pays more attention to potential splice junctions in the genome sequence.

☐ Save and close the *maker_opts.ctl* file.

Now that we have added these settings, we can begin the run of **MAKER**. **MAKER** produces a lot of output to the screen, so it makes sense to redirect it to a log file. On the other hand, we might want to keep an eye on what is going on. We can pipe output to the UNIX **tee** command to send the output to both places.

☐ Let's try **tee**.

```
• ls –l ~ | tee listing.txt
```

☐ Look at the resulting file *listing.txt*.

We are now ready to run **MAKER**. However, it will take a fairly long time to complete, so it is good practice to run it using **screen**. That way, MAKER will continue processing if our network connection is dropped, or we can detach from the screen if the process has not completed before class is over.

☐ Ensure you are in **screen**.

☐ Since we have specified all of our options in the configuration file, there is no reason to provide any on the command line; we do want to log errors and not just normal output, so we use `2>&1` to send errors to the pipe. Finally, we'll append an ampersand to run the process in the background, because it can take a long time to complete (approximately 20 minutes).

```
☐ maker 2>&1 | tee maker.log
```

As **MAKER** executes, you can see the various programs that it runs: **Exonerate**, **SNAP**, and so on. This execution will take approximately 20 minutes.

Note: if **MAKER** finishes much earlier than 20 minutes, there is probably an error in your input, even if no error message appears.

☐ After completion, list the contents of the directory and take a look at the results. The output has gone to the directory *magnaporthe_oryzae_70-15_8_single_contig.maker.output*. Take a look at its contents as well.

☐ There is a *magnaporthe_oryzae_70-15_8_single_contig_datastore* directory containing copies of all the configuration files (renamed with .log file extensions), and a few additional files. One of these files contains a list of all the contigs examined, and the names of the subdirectories in which the results for that contig may be found.

☐ Change back to your *maker* directory and look at the log file. Remember to use tab completion.

```
☐ cat magnaporthe_oryzae_70-15_8_single_contig.maker.output/
  magnaporthe_oryzae_70-15_8_single_contig_master_datastore_index.log
```

Each contig has its own subdirectory within *magnaporthe_oryzae_70-15_8_single_contig_master_datastore*, containing results of all the various programs **MAKER** used.

☐ Merge everything together into one GFF file:

```
☐ gff3_merge -d
  magnaporthe_oryzae_70-15_8_single_contig.maker.output/
  magnaporthe_oryzae_70-15_8_single_contig_master_datastore_index.log
  -o maker-annotations.gff3
```

`-d datastore_index.log` (file path shortened for formatting)	Look for GFF files in the directories mentioned in the log file
`-o maker-annotations.gff3`	Write results to this GFF file

☐ Use **less** or **more** to look at the resulting file *maker-annotations.gff3*.

☐ To understand the GFF format, it is necessary to know what information is contained within the various columns. These are as follows:

- seqname - name of the chromosome or scaffold; chromosome names can be given with or without the 'chr' prefix. Important note: the seqname must be one used within Ensembl, i.e. a standard chromosome name or an Ensembl identifier such as a scaffold ID, without any additional content such as species or assembly. See the example GFF output below.
- source - name of the program/project/database that generated the feature.
- feature - feature type name, e.g. Gene, Variation, Similarity.
- start - Start position of the feature, with sequence numbering starting at 1.
- end - End position of the feature, with sequence numbering starting at 1.
- score - A floating point value.
- strand - defined as + (forward) or - (reverse).
- frame - One of '0', '1' or '2'. '0' indicates that the first base of the feature is the first base of a codon, '1' that the second base is the first base of a codon, and so on..
- attribute - A semicolon-separated list of tag-value pairs, providing additional information about each feature

☐ Once you understand this format, you should be able to pick a gene and draw its structure, including start codon, exons, introns, and stop codons. Draw to approximate scale and note coordinates of each feature on the "virtual" genome.

What if we want to extract the sequences of the various features identified (transcripts, proteins)?

☐ We can do this using the **fasta_merge** tool (do not include a space in the file path):

```
☐ fasta_merge -d
  magnaporthe_oryzae_70-15_8_single_contig.maker.output/
  magnaporthe_oryzae_70-15_8_single_contig_master_datastore_index.log
  -o magnaporthe_oryzae_70-15_8_single_contig
```

☐ Use grep to count the number of genes predicted for Chromosome 8.7: _____

Note: Although we will not be doing so, it is possible to use the results of this **MAKER** run to re-train **SNAP** and run **MAKER** again, producing more and more accurate predictions in an iterative process.

We have only begun to scratch the surface of **MAKER**. For more information, visit the **MAKER** website at gmod.org, where many helpful links and resources are available:

http://gmod.org/wiki/MAKER

http://www.yandell-lab.org/software/maker.html

DAY 4

Module 6

RNAseq

Goal: Learn how to use various tool to extract information from RNAseq reads.

Input(s):
```
magnaporthe_oryzae_70-15_8_supercontigs.fasta
Moryzae_70-15_liquid_culture{1-3}.fastq.gz
Moryzae_FR13_inPlanta{1-3}.fastq.gz
magnaporthe_oryzae-70-15_8_transcripts.gtf
```

Output(s):
```
MoryzaeHS{1-8}.ht2
Mo_70-15_LC{1-3}_accepted_hits.bam
Mo_FR13_IP{1-3}_accepted_hits.bam
```

6.1 Information to be gained from RNASeq analysis

For this exercise, we will analyze six RNASeq datasets. Three were generated from RNA isolated from fungal mycelium growing in a liquid culture broth (three biological replicates). Three were generated from excised lesions containing the fungus growing in rice leaves. For the first step in our analysis, we will use **HISAT2** to align the RNASeq reads to a genome assembly of one of the fungal strains from which the RNAseq reads were derived (strain 70-15).

Kim D, Langmead B and Salzberg SL. **HISAT: a fast spliced aligner with low memory requirements**. *Nature Methods* 2015

https://daehwankimlab.github.io/hisat2/

Mapping RNASeq reads to a reference genome can tell us several things:

i) It will identify genomic regions that code for RNAs (i.e., regions that are genes).

ii) The number of reads mapping to each gene will tell us how strongly they were expressed in the two conditions from which RNA was extracted (mycelium growing in culture, infectious hyphae growing *in planta*).

iii) Gaps in the mappings will reveal the presence and locations of introns in the genes.

iv) Differences in gap positions will identify variable splice junctions.

v) Nucleotide differences can be used for variant calling (when the reads come from a genome that is different to the reference—which is true in this case).

6.2 Create an indexed reference genome

- ☐ Make sure you are in the *rnaseq* directory.
- ☐ Start a screen called "rnaseq".

First, we need to create an index of the reference genome that will allow faster searching. We will accomplish this using **hisat2-build**.

Usage: `hisat2-build <sequence-to-be-indexed> <prefix-for-index-name>`

- ☐ Index the reference genome.

```
• hisat2-build ~/blast/magnaporthe_oryzae_70-15_8_supercontigs.fasta
  MoryzaeHS
```

6.3 Mapping RNASeq reads to a reference genome

HISAT2 is based on the **Bowtie2** alignment engine but uses a more efficient indexing system for faster searching of RNAseq read alignments to the genome assembly. HISAT2 outputs *.sam* format files that we will need to convert to *.bam* for downstream processing.

Usage:

```
hisat2 [options] –o <output_dir> -x <path-to-index> <input-file(s)>
```

- ☐ Create a new directory called *alignments* to receive the output files from **HISAT2**
- ☐ Use **HISAT2** to map the RNAseq reads for the *liquid_culture1* dataset to the HISAT index (note: the program won't give you any indication it is running unless you made an error during entry).

```
• hisat2 -p 2 -x MoryzaeHS -U
  Moryzae_70-15_liquid_culture1_R1.fastq.gz --max-intronlen 2000
  --summary-file alignments/LC1_summary.txt --dta-cufflinks |
  samtools sort - -@ 2 -O
  bam -o alignments/Mo_70-15_LC1_accepted_hits.bam
```

`-p`	number of processors to use. **Note: you only have two processors available to you for this exercise, but normally you would run as many as are available.**
`-x`	name of index to align against
`-U`	comma-separated list of files containing unpaired reads to be aligned to reference. (we provide only one file here.)
`--max-intronlen`	maximum length of intron (500000 if left blank)
`--summary-file`	write alignment summary to this file
`--o`	name of output file

By default, **HISAT2** generates a .sam alignment file, which needs to be converted to .bam format and then sorted according to genome coordinates. This can be conveniently accomplished using a single pipe into the **samtools sort** program.

```
samtools sort - -@ 2 -O bam -o output.bam
```

`-`	use the output of the prior program as input
`-@`	number of processors to use
`-O`	output format (bam/sam)
`-o`	output file path

6.4 Examine the RNASeq mapping results

- ☐ First, let's use our command line to look at the *LC1_summary.txt* alignment summary file to get a sense of how much useable data we have.
- ☐ How many total reads were present in the input file? _____
- ☐ How many reads mapped to the reference genome? _____
- ☐ Use **head** to take a quick look at the first 10 lines of results in the alignments file.

  - `head alignments/Mo_70-15_LC1_accepted_hits.bam`

- ☐ Does the output make any sense? No? That's because the file is in a binary format. Let's use **samtools** to convert the *.bam* file into the human-readable *.sam* format:

  - `samtools view alignments/Mo_70-15_LC1_accepted_hits.bam`

- ☐ Whoa! Did you catch all that? Quit the process (^c) and try piping the results through **more** (or **less**). Next, look at the first alignment record; refer to the information in the .sam/.bam alignment format description c.
- ☐ What is the alignment length? _____
- ☐ What is the lowest quality value for the read represented by this alignment (in PHRED format)?

  _____

- ☐ Does this read align on the forward or the reverse strand of the reference? Use the SAM flag tool decoder to find out (https://broadinstitute.github.io/picard/explain-flags.html).

  _____

- ☐ Space bar through a few pages to view a few more alignments and then quit the program (^c) when you're done.

6.5 Align the remaining RNAseq datasets

- ☐ Repeat the alignment commands process for the *liquid_culture2* and *liquid_culture3* datasets. You can use the up arrow to speed up the command line entry, but remember to change both the input and output file identifiers. (Remember, there are two output files.) For example,

```
hisat2 -p 2 -x MoryzaeHS -U Moryzae_70-15_liquid_culture2_R1.fastq.gz
--max-intronlen 2000 --summary-file alignments/LC2_summary.txt
--dta-cufflinks | samtools sort - -@ 2 -O bam
-o alignments/Mo_70-15_LC2_accepted_hits.bam
```

etc... After each run is complete, list the *alignments* directory contents to make sure that the expected files were created.

☐ Now you have probably discovered that sitting and waiting for long alignment jobs to finish can make repetitive tasks very frustrating. Let's learn to appreciate the power of the command line for automation. We will perform the alignments for all three inPlanta datasets using a single command to loop through the three files, one at a time:

```
for f in {1..3}; do echo Working on dataset
Moryzae_FR13_inPlanta${f}_R1.fastq.gz; hisat2 -p 2
-x MoryzaeHS -U Moryzae_FR13_inPlanta${f}_R1.fastq.gz
--max-intronlen 2000 --dta-cufflinks
--summary-file alignments/IP${f}_summary.txt | samtools sort -
-@ 2 -O bam -o alignments/Mo_FR13_IP${f}_accepted_hits.bam; done
```

Here the **for** loop sequentially assigns the values 1, 2, and 3 to the variable `f`. We then substitute the variable `${f}` for the numerical identifiers in our input and output files. The semicolons (";") allow us to run several commands in a single line sequentially without passing the input of one command to the next (unlike a pipe). For example, we first run **echo** and then **hisat2**.

☐ When the three runs are completed, list the *alignments* directory contents to make sure that all of the expected files were created.

6.6 Assembling transcripts from RNAseq data

We will use **StringTie** to build complete transcripts from the individual RNAseq read mappings.

Pertea et al. (2015) StringTie enables improved reconstruction of a transcriptome from RNA-seq reads. Nature Biotechnology 33(3):290-295.

http://ccb.jhu.edu/software/stringtie/index.shtml

The first step in differential gene expression analysis is to identify the gene from which each sequence read is derived. **StringTie** examines the RNA-seq read mapping results produced by a read aligner such as **TopHat** or **HISAT2** and attempts to reconstruct the complete transcripts, identify transcript isoforms, and estimate transcript abundance. **StringTie** does this by clustering reads and constructing a splice graph for each cluster from which it will identify transcripts. A flow network is then constructed for each identified transcript to estimate the abundance of the transcript via a maximum flow algorithm. StringTie can accept an existing gene annotation file as a guide for constructing transcripts.

Usage:

```
stringtie <path/to/accepted_hits.bam> [options]
```

☐ Take a quick peek at all the **StringTie** options that are available.

```
stringtie --help
```

☐ Make sure you are in the *rnaseq* directory.

☐ First, take a look at the existing annotation file, *magnaporthe_oryzae_70-15_8_transcripts.gtf*, to be sure you understand its format.

Note that the GTF format is very similar to the GFF3 format used in the *.gff3* file that you produced in the gene finding exercise. Each feature entry has an MGG gene_id which identifies the gene to which the feature corresponds. Note also that each gene has one or more corresponding transcript_ids, with T0 denoting the first transcript isoform, T1, the second, etc.

☐ Create a directory named *transcripts*.

☐ Now we are ready to run **StringTie**, and we will provide the *.gtf* file as a reference transcriptome:

```
• stringtie alignments/Mo_70-15_LC1_accepted_hits.bam
  -p 2 -G magnaporthe_oryzae_70-15_8_transcripts.gtf
  -o transcripts/Mo_70-15_LC1_transcripts.gtf
```

`-p`	number of processors to use
`-G`	tells **StringTie** to use the provided reference annotation to guide transcript assembly but also to report novel transcripts/isoforms
`-o`	name of output directory

Notes:

A) Omitting the `-G` option (and accompanying *.gtf* file specification) from the above command would tell the program to generate a *de novo* transcript assembly. Alternatively, one can use `-G` with an additional `-e` flag, which will tell the program to assemble only those reads that correspond to previously identified genes/transcripts.

B) It is recommended that you assemble your replicates individually, i) to speed computation; and ii) to simplify junction identification. Therefore, you will need to run **StringTie** separately for each of your *.bam* files.

☐ Make sure that the *Mo_70-15_LC1_transcripts.gtf* output file was created indicating that the command completed successfully.

☐ Look at the first few lines of the *Mo_70-15_LC1_transcripts.gtf* in order to answer the questions below (use the information below to understand what the different fields mean).

The final column of a GTF file is a semicolon-separated list of tag-value pairs providing additional information about each feature. Depending on whether an instance is a transcript or an exon and on whether the transcript matches the reference annotation file provided by the user, the content of the attributes field will differ. The following list describes the possible attributes shown in this column.

a. gene_id: A unique identifier for a single gene and its child transcript and exons based on the alignments' file name.

b. transcript_id: A unique identifier for a single transcript and its child exons based on the alignments' file name.

c. exon_number: A unique identifier for a single exon, starting from 1, within a given transcript.

d. reference_id: The transcript_id in the reference annotation (optional) that the instance matched.

e. ref_gene_id: The gene_id in the reference annotation (optional) that the instance matched.

f. ref_gene_name: The gene_name in the reference annotation (optional) that the instance matched.

g. cov: The average per-base coverage for the transcript or exon.

h. FPKM: Fragments per kilobase of transcript per million read pairs. This is the number of pairs of reads aligning to this feature, normalized by the total number of fragments sequenced (in millions) and the length of the transcript (in kilobases).

i. TPM: Transcripts per million. This is the number of transcripts from this particular gene normalized first by gene length, and then by sequencing depth (in millions) in the sample.

☐ **For the first gene reported for chromosome 8.1:**

a. Does this gene correspond to a previously annotated *Magnaporthe* gene (does it have an MGG identifier), or is it novel? previously annotated: _____; novel: _____

b. How long is the gene? _____ bp

c. Does it contain any introns? yes: _____; no: _____.

d. How abundantly is it expressed in fragments per kilobase per million reads? _____

☐ Now we will complete the rest of the transcript assemblies by using another kind of **for** loop.

```
• for f in $(ls alignments/*bam | grep -v LC1); do
  echo Working on $f; g=${f/*\//}; stringtie $f -p 2 -G
  magnaporthe_oryzae_70-15_8_transcripts.gtf
  -o transcripts/${g/accepted*/}transcripts.gtf; done
```

Here we are running the loop a little differently. We list all bam files in the alignments directory, and, after omitting the file that we've already processed (using `grep -v LC1`), we assign the results to the variable `f`. Then, for each value (i.e., each file) in `f`, we strip off everything before and including the first forward slash (`${f/*\//}`) and save the result to the variable `g`. Then, we run **stringtie** on each value of `f` (*alignments/Mo_70-15_LC1_accepted_hits.bam*, etc.) and output the results to the transcripts directory under a filename that we create by cutting off the *accepted_hits.bam* suffix from what is stored in the variable (`${g/accepted*/}`) and placing the remainder (e.g., "Mo_70-15_LC2_") in front of "transcripts.gtf".

6.7 Merging transcript assemblies

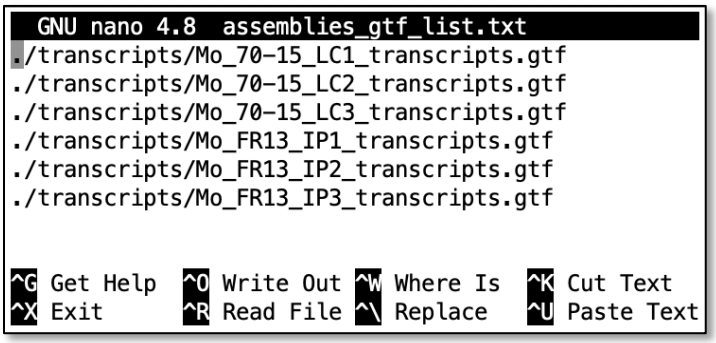
We will use **StringTie** with the `–merge` option to generate a "super-assembly" of transcripts based on the mapping information from all six RNAseq datasets. This will allow **StringTie** to identify overlaps between alignment data for different datasets. In this way, it can assemble complete transcripts for genes whose expression levels are too low to allow full transcript reconstruction from a single sequencing lane.

Usage:

```
stringtie ––merge [options] <list_of_gtf_files>
```

☐ **Make sure you are in the *rnaseq* directory.**

☐ Open a text editor, such as **nano**, and create a text file with a list of the .gtf files that will be incorporated into the super-assembly. The list should have the following format:



Here is an example of where a file created by a standard text editor such as **Word** will not be read properly by the **stringtie** program and would produce an error.

☐ Save the file using the name *assemblies_gtf_list.txt*.

☐ A quicker way to do this would be to use another **for** loop:

```
• for f in $(ls transcripts/*gtf); do echo ./$f >>
  assemblies_gtf_list_loop.txt; done
```

>> means redirect to a file and append new entries to the same file

☐ Run **stringtie** with the **--merge** option.

```
• stringtie --merge -p 2 -o merged_asm/merged.gtf
  -G magnaporthe_oryzae_70-15_8_transcripts.gtf
  assemblies_gtf_list.txt
```

`-p` number of processors to use

`-o` output file name for the merged transcripts. Here, **stringtie** will create the *merged_asm* directory if it doesn't already exist (this is not the case for many programs)

`-G` include the reference annotation in the merging operation

☐ Examine the *merged.gtf* file produced by `stringtie --merge` inside of the *merged_asm* directory.

☐ Use command line tools to interrogate the *merged.gtf* file to identify novel transcripts that have not been previously identified. Note: novel transcripts will lack MGG identifiers.

You will note that the gene-id attribute for each transcript has a **stringtie**-specific prefix (MSTRG). This is a bit of a problem because, if we use this file for downstream analyses such as differential gene expression analysis, this will be the gene ID that is displayed, even if there is already an established MGG identifier.

☐ Use the **Inherit_IDs.pl** script to convert the gene-id field to its corresponding MGG identifier (if available).

```
• perl Inherit_IDs.pl merged_asm/merged.gtf
  > merged_asm/cufflinks.gtf
```

☐ What is the total number of genes that show at least one alternative transcription start site and/or alternative splice junction? (Recall that the first transcript isoform is given a T0 identifier.)

_____

6.8 Differential gene expression analysis

The **HISAT2/StringTie** pipeline normally feeds into the **Ballgown** program for differential expression analysis (Frazee AC et al. 2015. Nat Biotechnol. 33:243-246). However, this would require many of us to learn the R programming language and environment, which itself could take a whole week. Therefore, we will use the legacy **cuffdiff** program to determine which genes are differentially expressed in one of the RNAseq datasets.

http://cole-trapnell-lab.github.io/cufflinks/cuffdiff/

For our purposes, we will have **cuffdiff** use the *cufflinks.gtf* file which combines the prior gene annotations with the new information (novel transcripts, isoforms, etc.) generated from our RNAseq data. **cuffdiff** then uses the alignment data (in the *.bam* files) to calculate and compare transcript abundances.

Usage:

```
cuffdiff [options] <transcripts.gtf>
      <condition1.repl1.bam,condition1.replicate2.bam…
      <condition2.replicate1.bam,condition2.replicate2.bam…>
```

Note: experimental replicates are separated with commas; datasets being compared are separated by a space (i.e.: Condition1_rep1,Condition21_rep2 (SPACE) Condition2_rep1,Condition2_rep2).

Since **bash** doesn't know about the way **cuffdiff** separates the experimental replicate files (and we have not provided any custom code to **bash** for tab-completing **cuffdiff**), tab completion will not work for an argument after you type the first comma. We can get around this limitation by initially separating with spaces and then replacing the appropriate spaces with commas.

For our experiment, we will compare transcript abundance in fungus grown in liquid culture (three replicates) versus fungus growing *in planta* (three replicates).

☐ Run **cuffdiff** as follows (**DO NOT PUT A SPACE BETWEEN THE COMMAS AND THE FLANKING DATASETS—SEE ABOVE COMMENT IN RED BEFORE RUNNING**):

```
•  cuffdiff -o diff_out -p 2 -L culture,inPlanta
   -u merged_asm/cufflinks.gtf
   alignments/Mo_70-15_LC1_accepted_hits.bam,
   alignments/Mo_70-15_LC2_accepted_hits.bam,
   alignments/Mo_70-15_LC3_accepted_hits.bam   (include a space here)
   alignments/Mo_FR13_IP1_accepted_hits.bam,
   alignments/Mo_FR13_IP2_accepted_hits.bam,
   alignments/Mo_FR13_IP3_accepted_hits.bam
```

`-o` output directory where results will be deposited

`-p` number of processors to use

`-L` Labels to use for the two conditions being compared. These labels will appear at the top of the relevant columns in the various output files.

`-u` Tells **cufflinks** to do an initial estimation procedure to more accurately weight reads mapping to multiple locations in the genome

☐ The gene expression differences are written to the file named *gene_exp.diff* inside the defined output folder. View the header of this file and see if you can determine what information is contained in the various columns. If necessary, look at the description of the output columns in the online **cuffdiff** manual:

http://cole-trapnell-lab.github.io/cufflinks/manual/

- ☐ Practice using the command line to produce an output that shows only those genes that **cuffdiff** predicts to be differentially expressed between the two samples.
- ☐ Use **awk** to print the lines of results for genes that show a more than ten-fold higher expression *in planta* versus in liquid culture.
- ☐ Use the command line to produce a list that contains the identities of the genes that show significant differences in their expression levels (only the names of the genes and nothing else). Write this list to a file.

Hint: You will need to use **grep** and/or **awk**.

Module 7

Variant Calling

Goal: Learn how to use various tools to identify variants after re-sequencing

Input(s): `Mo_FR13_IP{1-3}_accepted_hits.bam`

Output(s): `Mo_FR13_alignments_merged.bam`
`Moryzae_FR13_alignments.bcf`
`Moryzae_FR13_alignments.vcf`

Three of the RNA samples used for RNAseq analysis were generated from fungal strain FR13, which is expected to show several sequence differences relative to the 70-15 strain that was used to generate the reference genome. Therefore, we will use the RNAseq alignments to search for nucleotide differences between FR13 and the 70-15 reference strain.

7.1 Merge the FR13 alignment (.bam) files

Having confidence in variant calls requires one to have multiple reads aligned across the region where there is a genetic difference between the sample and the reference. When sequencing DNA, this is easily accomplished by ensuring that we have enough sequence data to provide several-fold coverage of the genome (≥ 20x). However, with RNAseq data, we have no control over coverage because this is determined by the expression level of each gene at the time and place where the RNA was extracted. Therefore, to maximize coverage across each gene for the purpose of our analysis, we will merge the alignments file for the three *in planta* RNA replicates.

☐ We will use the **samtools** merge tool. (Note that bam files must be sorted by coordinates before they are merged. We already sorted the files in the *rnaseq* lab, so we can use them directly here.)

- `samtools merge Mo_FR13_alignments_merged.bam ../rnaseq/alignments/*FR13*bam`

- ☐ Take a quick look at the *.bam* file. Is it in binary format? You can take a look at it by using **samtools** again:

```
• samtools view Mo_FR13_alignments_merged.bam
```

- ☐ Whoa, can you read the output? Quit the process (control-c) and re-run the command with a pipe into another program that will enable you to look at a few lines at a time.
- ☐ Does the output still look like it contains sam alignment data? If so, proceed to the following.
- ☐ We must provide **mpileup** with the reference genome that was used for alignment. To speed up its computations, **mpileup** uses an indexed version of the genome. This index is different to the one generated with **bowtie2**, and we will use **samtools** to generate it.

```
• samtools faidx
  ../blast/magnaporthe_oryzae_70-15_8_supercontigs.fasta
```

  This will create an index file named *magnaporthe_oryzae_70-15_8_supercontigs.fasta.fai.*

- ☐ Check to make sure the index was created. (Note that the file is created in the same directory as the reference genome.)

7.2 Perform the variant calling

We will use the **bcftools mpileup** utility to extract information on nucleotide variations (substitutions, insertions, deletions) between our sequence sample and a reference genome. **mpileup** does this by using information contained in the CIGAR (Concise Idiosyncratic Gapped Alignment Report) string and the BTOP (back track operations) string in the MD:Z: field of the *.sam* (or *.bam*)file.

Usage:

```
bcftools mpileup [options] -f <reference_genome> <BAM alignment(s)>
```

- ☐ Make sure you are using **screen**.
- ☐ Run **mpileup** on the sorted file.

```
• bcftools mpileup --threads 2 -f
  ../blast/magnaporthe_oryzae_70-15_8_supercontigs.fasta
  Mo_FR13_alignments_merged.bam
```

`--threads` number of processors to use

`-f [FILE]` name of reference genome

bcftools, like **samtools**, outputs results to the screen, making the output kind of hard to read. However, at least you can see that the program is doing something interesting.

- ☐ Stop the process (using control-c) so you don't have to wait a LONG time for the output to finish printing to the screen.

- ☐ Re-run the previous command but redirect the results to a file called *Mo_FR13_alignments.vcf*. (**Note**: the program will take several minutes to complete, during which time you will receive no progress updates—all program outputs are directed to the specified output file.)
- ☐ Inspect the *Mo_FR13_alignments.vcf* file. This summarizes variant statistics for every position in the reference genome for which there are aligned reads. However, it does contain variant calls.
- ☐ We will use **bcftools** to call the variants.

```
• bcftools call -v -c --ploidy 1 Mo_FR13_alignments.vcf
```

`-v`	output potential variant sites only (i.e., skip monomorphic ones)
`-c`	call variants using the original method implemented in **samtools mpileup**
`--ploidy`	set to 1 because *Magnaporthe* is haploid, and we only expect one copy of each gene unless the sequence is repeated

- ☐ Note that **bcftools** is like **samtools** in that it sends results to the screen. Quit the process (control-c) and redirect the output to a file named *Mo_FR13_alignments_called.vcf*.

Unfortunately, the header does not provide much information on the overall structure of the ***.vcf*** file. The main fields in the tab-delimited section are as follows:

Column 1: Chromosome number

Column 2: Nucleotide position

Column 3: SNP ID (if previously characterized and named)

Column 4: Nucleotide in reference genome

Column 5: Alternate allele(s) identified in sequence reads

Column 6: Quality of SNP call

Column 7: Filtering information (“.” = no filter; “Low Qual”; or “PASS”)

Column 8: SNP information (see list of INFO fields in file header)

Columns 9 & 10: SNP formats (see the list of FORMAT fields in file header)

A complete description of the VCF format is in the VCFv4.1.pdf file on the Canvas site.

7.3 Examine the resulting variant calls

☐ Inspect the *Mo_FR13_alignments_called.vcf* file.

At the head of the file is some information on how to interpret the various fields. Below, each line provides information on a predicted variant at a specific nucleotide position within the chromosome. Here you should be able to recognize data that make sense.

Column 10 in the *Mo_FR13_alignments_called.vcf* file contains information that informs us about the likelihood that a given SNP (or INDEL) is valid. First, to have confidence in the call, we want to have several reads that support it, and because we are working with a haploid fungus, ideally we want all reads that overlap a given site to have the alternate (variant) allele. Unfortunately, variant callers are not very smart when operating in haploid mode (they were mostly designed for human variant calling). As a result, they often call variants at sites that are clearly heterozygous for the reference/alternate alleles. Heterozygosity in a haploid organism indicates that there are two or more copies of the site being interrogated but, more importantly, it tells us that the SNP is likely to be false. There are many instances of heterozygosity for variant calls in the *Mo_70-15_FR13.vcf* file, so we need to filter the file to remove these suspect calls.

☐ First, use **more** to look at a few pages of the vcf file and pay attention to the DP4= field in column 10. (Note: this is a w-i-d-e column!) There are four values that follow DP4=: forward strand reads supporting ref allele, reverse strand reads supporting ref allele, forward strand reads supporting the alt allele, reverse strand reads supporting the alt allele. For a true SNP, there should be no reads at all supporting the ref allele.

☐ Use grep to count the total number of SNP calls (don't count INDELs). Next, count the number of low quality calls where there are some reads supporting the reference allele, position, and DP4= values for one such suspect record.
Total SNP calls: _____; Low quality (suspect) SNPs: _____

In addition, many of the SNPs that were called were supported by very few reads, and so we can't be confident that these variant calls aren't due to sequencing errors. Therefore, we will filter the SNP calls to retain only those calls that are supported by at least 10 reads, with no reads containing the ref allele.

☐ There are specific tools for this purpose, but why waste our time trying to remember the necessary commands and options when we can easily do this with **grep**?

```
• grep DP=[1-9][0-9] Mo_FR13_alignments_called.vcf | grep DP4=0,0,
  > Mo_FR13_alignments_filtered.vcf
```

`grep DP=[1-9][0-9]` look for "`DP=`" followed by any number between 1 and 9, followed by any number between 0 and 9. (This is how we make sure that `DP=` is followed by a number that is equal to, or greater than, 10.)

Examine the resulting filtered file containing "high quality" variant calls, and use the suggested tools with pipes to answer the following questions:

☐ How many SNPs were identified on Chromosome 8.1? (**grep**)_____

☐ How many INDELs were found on Chromosome 8.1? (**grep**) _____

☐ Are there any indels that are greater than one nucleotide in length? (**awk {print length(...)}, sort, uniq)**" yes__; no __

- ☐ What is the greatest depth of coverage (DP=) for a SNP in the filtered dataset? (**awk**, **sed**, **sort**, **uniq**) _____

Module 8

IGV for Data Visualization and Exploration

Goal: Utilize **Integrative Genomics Viewer (IGV)** to visualize predicted genes, transcript assembling and RNAseq read alignments

Background

The **Integrative Genomics Viewer (IGV)** is a high-performance visualization tool for interactive exploration of large, integrated genomic datasets. It supports a wide variety of data types, including array-based and next-generation sequence data, and genomic annotations.

Helga Thorvaldsdóttir, James T. Robinson, Jill P. Mesirov**.** Integrative Genomics Viewer (IGV): high-performance genomics data visualization and exploration. Briefings in Bioinformatics 2012.

James T. Robinson, Helga Thorvaldsdóttir, Wendy Winckler, Mitchell Guttman, Eric S. Lander, Gad Getz, Jill P. Mesirov. Integrative Genomics Viewer. Nature Biotechnology 29, 24–26 (2011)

8.1 Installing IGV

- ☐ Go to http://www.broadinstitute.org/software/igv/home
- ☐ On the left-hand, side click on "Downloads".
- ☐ Select and install the version appropriate for your system. Unless you have a specific reason to use an existing Java installation, you should select a version with Java included.

8.2 Prepare annotations

Input(s):	`magnaporthe_oryzae_70-15_8_single_contig.fasta`
	`Mo*accepted_hits_Chr7.bam`
	`cufflinks.gtf`
	`maker-annotations.gff3`
	or `maker-preview.gff` (if your **MAKER** run did not complete)

Output(s):	`Mo*accepted_hits_Chr7.bam`
	`Mo*accepted_hits_Chr7.bam.bai`
	Tracks and annotations visible in IGV

- ☐ Connect to your virtual machine.
- ☐ Change to the *rnaseq* directory.

We will only explore Chromosome 8.7 in the browser, so let's extract the alignment data for just that chromosome. This will significantly speed up the data transfer to our local machines.

- ☐ First, we need to index our bam files:

```
• for f in alignments/*.bam; do samtools index $f; done
```

- ☐ Check that the requisite index (.bai) files were created in the *alignments* directory.
- ☐ Now let's perform the extraction of just the Chromosome_8.7 records:

```
• for f in alignments/*.bam; do
  samtools view -b $f Chromosome_8.7 > ${f/hits/hits_Chr7}; done
```

- ☐ List the *alignments* directory (`ls -lrt`) to make sure six files with *_Chr7.bam* suffixes were created.
- ☐ Now, we need to index our Chromosome_8.7-specific bam files:

```
• for f in alignments/*Chr7.bam; do samtools index $f; done
```

- ☐ Again, list the directory (**ls -lrt**) to make six new (*.bai*) files were created.

8.3 Transfer annotations to local machine

- ☐ Use **scp** to transfer the following files from your VM to your local machine:

  ~/rnaseq/alignments/Mo_70-15_LC{1-3}_accepted_hits_Chr7.bam

  ~/rnaseq/alignments/Mo_FR13_IP{1-3}_accepted_hits_Chr7.bam

  ~/rnaseq/alignments/Mo_70-15_LC{1-3}_accepted_hits_Chr7.bam .bai

  ~/rnaseq/alignments/Mo_FR13_IP{1-3}_accepted_hits_Chr7.bam.bai

  ~/rnaseq/merged_asm/cufflinks.gtf

  ~/genes/maker/maker-annotations.gff3 **or** *maker-preview.gff* if your **MAKER** run did not complete.

  ~/genes/snap/magnaporthe_oryzae_70-15_8_single_contig.fasta

You can use the following command to transfer the .bam and .bai files en masse:

```
• scp myName@xx.xxx.xxx.xx:rnaseq/alignments/*Chr7* .
```

8.4 Navigating IGV

The best way to learn how to navigate a genome browser is simply to play with it by clicking on buttons/features and clicking and dragging in the navigation pane. However, to help you get going, you can refer to this video tutorial:

IGV | Data Navigation Basics

8.5 Load genome into the genome browser

- ☐ Now, open **IGV** on your local machine by double-clicking on the program's icon.
- ☐ In the menu, select **Genomes** > **Load Genome from File...**
- ☐ Browse to the folder where you secure-copied files from the server, and select *magnaporthe_oryzae_70-15_8_single_contig.fasta*.

8.6 Load MAKER annotations into the genome browser

- ☐ First, we will examine the genes that we predicted yesterday. Use **File** > **Load from File...** to import the *maker-annotations.gff3* file.
- ☐ Right-click on the *maker-annotations.gff3* name in the left-hand panel again and select "expanded" view. This will allow you to visualize potentially overlapping genes on the opposing DNA strands.
- ☐ Using several clicks of the "+" button at the top right, zoom in far enough to see the structures of the individual genes.

You will see that each feature is labeled with an identifier that tells you if it was predicted by **snap** and/or **augustus**, if it is a gene model created by **maker**, and/or matched a protein in the NCBI database. Features with an "XP" prefix exhibited matches to proteins at NCBI.

- ☐ When you are done, collapse the track by right-clicking on "maker-annotation.gff3" in the left panel and select "Collapsed."

8.7 Load transcript assemblies into the genome browser

- ☐ Now we will use **File** > **Load from File...** again to import the transcript assembly produced by the `perl Inherit_IDs.pl` command that we ran earlier.
- ☐ Using several clicks of the "+" button at the top right, zoom in far enough to see the individual transcript forms.
- ☐ Right click on the *cufflinks.gtf* name in the left-hand panel again and select "expanded" view. This will allow you to visualize alternative transcript forms identified by **cufflinks**, as well as overlapping transcripts on each strand of the DNA.

The **MAKER** run was informed by prior RNAseq data generated from fungus grown in liquid culture as well as from spores. The RNAseq data we used today incorporated new data from fungus growing *in planta*, and also contained MANY more reads from liquid grown cultures than were analyzed previously. Thus, it is possible that these new data might identify novel genes that escaped prediction by **SNAP** and **Augustus** and which were not picked up by the similarity searches performed by **MAKER**.

- ☐ Start exploring the single contig by clicking and dragging in one of the browser tracks, by dragging the red square that represents the current view window, or by clicking on the left and right arrowheads at each end of the window size indicator.

- ☐ Find a gene that is present in the *cufflinks.gtf* track but was not predicted by **MAKER**. Note that large genes are more likely to be valid genes than smaller ones which might be represented by just a handful of RNAseq reads. List the coordinate(s) of a few potentially novel gene(s):

- ☐ When you are done, collapse the track by right-clicking on "cufflinks.gtf" in the left panel and select "Collapsed."

8.8 Load RNAseq alignments into genome browser

- ☐ Next, we will load the .bam alignment files by selecting **File > Load from File...**
- ☐ Open all of the downloaded **accepted_hits_Chr7.bam* files.

Three plots should open up for each track. The first shows depth of RNAseq read coverage for each position on the chromosome; the second shows splice junctions; and the third shows alignments for each RNAseq read mapping to the chromosome region in question. Depending on the Zoom level, you may or may not see anything at first because information about the RNAseq read alignments only show up after you have zoomed in far enough. If necessary, zoom in so that you can start seeing the read alignments in the main window.

- ☐ Because we have so many reads, it may be necessary to "squish" the view so that more alignments can be displayed in each browser "track." Right-click over each of the filenames in the left-hand panel and select "squished" view in the menu that pops up. If necessary, close the "Coverage" tracks by clicking on the name in the left-hand panel, hitting the delete key, and confirming if prompted.

Note that the 70-15 RNAseq reads were much longer than those generated for FR13 (250 nt vs. 50 nt).

- ☐ Revisit the locations where your transcript assemblies found potentially novel genes and examine the RNAseq data tracks to determine which growth condition (liquid culture and/or *in planta*) produced the reads that supported the respective transcript assemblies.

  gene coordinate:_____; condition: _____

  gene coordinate:_____; condition: _____

  gene coordinate:_____; condition: _____

- ☐ Now, we'll manually examine the RNAseq data tracks for differential expression. For improved visualization, close the cufflinks and maker-annotations tracks and re-open when you need to see gene IDs.

- ☐ Navigate along the contig and find a gene that was expressed at a high level *in planta* but showed no expression in liquid culture:

  gene_id:_____

- ☐ Now find a gene that was expressed at a high level in liquid culture but showed no expression *in planta*:

  gene_id:_____