Microbiome
Microbiome Analysis
Metagenomics
Metatranscriptomics

Bioinformatics
Genome Assembly & Annotation
SNP & Variant Detection
Microbiome & Metagenomic analysis

Genomics
Genome Sequencing
Next Gen Genetic Mapping
RNA-Seq
Bisulfite Sequencing

CAGEF
University of Toronto
Centre for the Analysis of Genome Evolution & Function

Proteomics
Mass Determination
Protein Identification
Post-translational modification

cagef.utoronto.ca
cagef@utoronto.ca

CSB1021HF LEC0131

FUNDAMENTALS OF GENOMIC DATA SCIENCE

0.0.0 Module 6: Assembly and annotation in the command line

0.1.0 About Fundamentals of Genomic Data Science

Fundamentals of Genomic Data Science is brought to you by the **Centre for the Analysis of Genome Evolution & Function (CAGEF)** bioinformatics training initiative. This course was developed based on feedback on the needs and interests of the Department of Cell & Systems Biology and the Department of Ecology and Evolutionary Biology.

The structure of this course is a “code-along”, hands-on style! A few hours prior to each lecture, materials will be made available for download at QUERCUS (https://q.utoronto.ca/). The teaching materials will consist of a weekly PDF that you can use to follow along with the instructor along with any datasets that you’ll need to complete the module. This learning approach will allow you to spend the time coding and not taking notes!

As we go along, there will be some in-class challenge questions for you to solve. Post lecture assessments will also be available for each module, building upon the concepts learned in class (see syllabus for grading scheme and percentages of the final mark).

0.1.1 Where is this course going?

We’ll take a blank slate approach here to learning genomic data science and assume you know nothing about programming or working directly with next generation sequencing data. From the beginning of this course to the end we want to guide you from potential scenarios like:

- You don’t know what to do with a set of raw sequencing files fresh from a facility like CAGEF.

- You've been handed a legacy pipeline to analyse your data or maintain for the lab, but you don't know what it runs or how.
- You plan on generating high-throughput data but there are no bioinformaticians around to help you out.

and get you to the point where you can:

- Recognize the basic tools in sequence analysis.
- Plan and write your own data analysis pipelines.
- Explain your data analysis methods to labmates, supervisors, and other colleagues.

0.1.2 How do we get there?

In the first half of this course, we'll focus on how to generate analysis pipelines using the Galaxy platform – a user-friendly graphical interface that provides access to common sequence analysis tools. After we are comfortable with these tools, we'll look at life through the lens of a command-line interface. It is here that we will learn the basics of file manipulation and how to program scripts that can carry out multiple tasks for us. From there we'll revisit tools from the first half and learn skills to make your data analysis life easier.

0.2.0 Goals of the module

1. Learn to build and execute commands for quality control, assembly, and annotation.
2. Explore how paired-end information and hash length optimization can improve assemblies.
3. Learn how to use the command-line language awk to quickly filter data.
4. Learn how to check for contamination in an assembly using Blast.
5. Learn how to prepare bash scripts to increase the efficiency of bioinformatics pipelines.

0.3.0 Pre-class modules with Coursera

Each week we strongly encourage you to complete the assigned Coursera modules and/or readings ***before*** class. These are meant to provide you with sufficient background material on each week's module so that we can focus on the act of "doing" something with that data rather than spend a lot of time on the origins of it. You'll find a section outlining the next set of Coursera modules and readings at the end of each module.

0.3.1 Go to www.coursera.org and sign up for an account with your e-mail.

0.3.2 Search the following courses and enroll to audit each course (audit):

- Genomic Data Science with Galaxy, Johns Hopkins University.
- Command Line Tools for Genomic Data Science, Johns Hopkins University.

0.4.0 Setting up your working directory

We suggest that you create a new directory (folder) for this course directly off your root directory called "**FGDS**". Working from your root directory is not necessary, but it will make some of the aspects of the course a little easier to manage. For Mac OS users, we suggest you create this as a subfolder in your **user** directory.

0.4.1 Within this directory, create another directory called "**Module6**". This is where we will store the data used in this week's module.

0.4.2 Create a subdirectory called "**Data**" to store the initial files as we download them before decompressing and working with them in later steps.

1.0.0 Quality control of sequencing reads at the command line

This week we revisit our alignment and annotation workflow from Galaxy. We'll still be working with our H. influenzae data but this time it has been broken into a set of paired reads – forward and reverse. We'll learn to work with this type of raw data throughout our pipeline and we'll see how distinguishing between forward and reverse sequences can affect our results.

1.1.0 Directory and data setup

Before you begin, open your command line and make sure you have a path for ~/**FGDS/Module6/Data**. This is where we will store data as it is generated in today's lecture.

1.1.1 Check for the FGDS/Module6/Data path. If necessary, create it before setting it as your current directory.

```
cd ~/FGDS
ls ./Module6/Data        # This will return an error if the directory doesn't exist
mkdir -p Module6/Data    # -p will make parent folders for you if necessary
cd Module6/Data
```

1.1.2 Obtain the paired-end fastq files for this module from the "Data" directory on our course server u

sing the **`sftp`** command. This is the same data that we used in Module 3 when we performed Assembly and Annotation with Galaxy, however, the forward and reverse reads have been divided into separate files. Instead of treating the reads as single-end reads as we did in Module 3, we will treat these reads as *paired-end reads*.

```
sftp fgds@142.150.215.186
      PW: fall2024
ls
ls Data/Module6/

# Download both of the fastq datasubset files to your Data directory
get Data/Module6/*.gz
exit
```

1.2.0 Run `fastqc` from Anaconda

Now that we have our data in our Module6 directory we can proceed with performing a quality check on our fastq reads.

1.2.1 Before using **`fastqc`**, however, we will need to activate Anaconda (although it should already be active). If you do not see **`(base)`** at the beginning of your command prompt, then activate it and then double-check your available environments.

```
conda activate
conda info -e
```

1.2.2 Review the help menu for **`fastqc`** using the **`--help`** flag. All the tools required for this course should have been installed and added to our Anaconda installation last week, but we'll have to navigate our way through our environments to access them all.

```
fastqc
fastqc --help
```

Note that an abbreviated help menu for most bioinformatics software can also be accessed by simply typing the command without any options or input, but for **`fastqc`** this will launch the GUI version of the software instead.

1.2.3 Now we are ready to build a **`fastqc`** command from command line. Although we are inside an Anaconda shell, these commands are issued just as though they were installed directly into your command line.

```
cd ~/FGDS/Module6/
mkdir fastqc

# Our actual call to fastqc
fastqc --kmers 7 --outdir fastqc Data/HinfKW20_datasubset_for.fastq.gz
      Data/HinfKW20_datasubset_rev.fastq.gz
```

Command	Meaning
`--kmers 7`	search for overrepresented kmers of 7 bases in length
`--outdir fastqc`	send all output to a directory called fastqc (must already exist)
`HinfKW20_datasubset_for.fastq`	Input file one, forward reads.
`HinfKW20_datasubset_rev.fastq`	Input file two, reverse reads.

1.2.4 Review the output of your **`fastqc`** analysis, which will be in your **fastqc** directory. You can either review the results at the command line using text files that are in the *.zip archives or open the HTML outputs. I find it is much more useful to look at the figures and evaluate the read quality using the HTML output, so we will be ignoring the text files today.

Mac Users: Use the **`open`** command to open these files in a web browser in separate tabs.

```
ls fastqc
open fastqc/HinfKW20_datasubset_for_fastqc.html
open fastqc/HinfKW20_datasubset_rev_fastqc.html
```

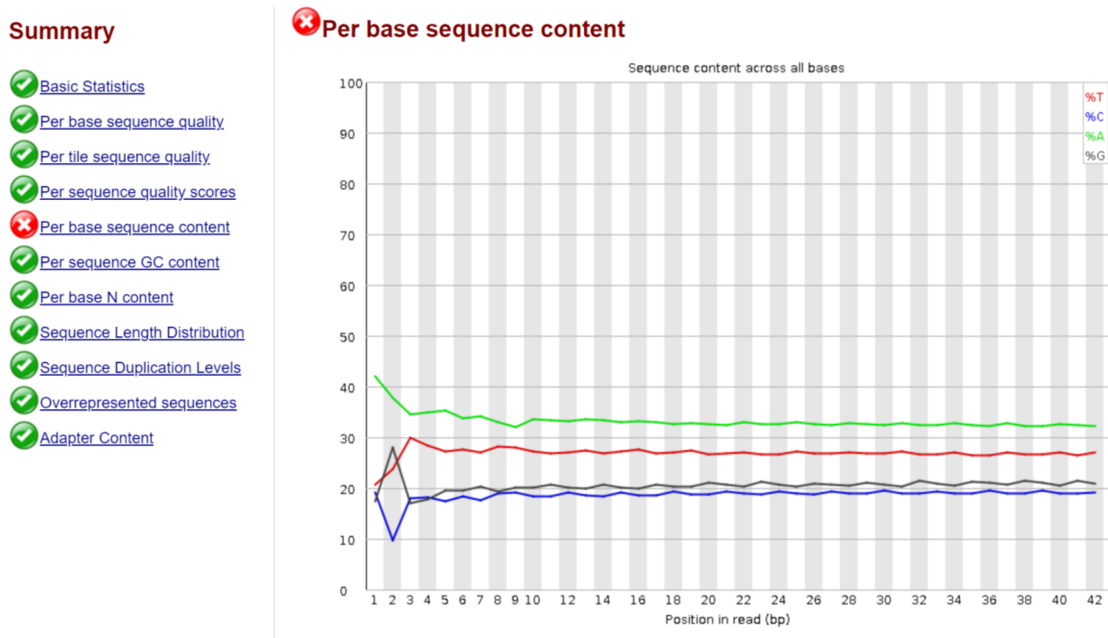
Windows Users: Use the **`explorer.exe`** command to open files from your current working directory in your default web browser.

```
explorer.exe HinfKW20_datasubset_for_fastqc.html
explorer.exe HinfKW20_datasubset_rev_fastqc.html
```

You can also locate the files using your WSL shortcuts you created and double-click on the HTML file. Otherwise, you will need to copy the html files to a windows directory and open them through the Windows Explorer.

Alternatively, you can use the command below. We copy the files directly to the root of your C: in a new directory called “FastQC_Temp”. Once the files are in these directories, open them by double clicking in Windows Explorer.

```
mkdir /mnt/c/FastQC_Temp                # Make a directory in the C drive
cp fastqc/*.html /mnt/c/FastQC_Temp/    # Copy all of the html files there
```



As you might remember from Module 2, our **`fastqc`** analysis yielded some warnings, but none of the tests that failed were concerning. However, that is not the case here, where the "**Per base sequence content**" test fails for the forward reads. This suggests that combining the forward and reverse reads may have masked a smaller issue that is significant when you only look at the forward reads in isolation. In this case, there appears to be relatively strong biases in the first couple bases of the forward reads. We also see that A% and T% are no longer similar with each other with almost a 5-7% difference. As we'll see, however, our main issue lies in the first couple of bases for our reads.

1.3.0 Read trimming with `trimmomatic`

Given our issues with the forward read sequence data, it would be advisable to filter our reads. We will run **`trimmomatic`** to solve these issues.

1.3.1 Double check where your home directory is located. While we have been using the "~" quote often as a shortcut for our home directory, we will need the full path in later steps.

```
pwd          # This will show you your home directory structure for 1.3.3
```

1.3.2 Review the basic command options that are available for **`trimmomatic`.** A more detailed description of [trimmomatic can be found here](#).

```
trimmomatic -h
```

1.3.3 Build and execute a **`trimmomatic`** command to remove any adapters, trim low quality reads, and remove the first two bases from each read.

```
cd ../                        # get back to ~FGDS/Module6/
trimmomatic PE -phred33 Data/HinfKW20_datasubset_for.fastq.gz
      Data/HinfKW20_datasubset_rev.fastq.gz Data/HinfKW20_trimmed_for_paired.fastq
      Data/HinfKW20_trimmed_for_unpaired.fastq
      Data/HinfKW20_trimmed_rev_paired.fastq
      Data/HinfKW20_trimmed_rev_unpaired.fastq
```

```
ILLUMINACLIP:/home/<user>/anaconda3/share/trimmomatic-0.39-
2/adapters/TruSeq2-PE.fa:2:30:10 SLIDINGWINDOW:4:20 HEADCROP:2
```

 # Note your ILLUMINACLIP directory must be altered to match your own home directory

Command	Meaning
`PE`	input data is paired-end
`-phred33`	Phred33 quality scores were used in the input files
`Data/HinfKW20_datasubset_for.fastq.gz`	forward fastq.gz file (input)
`Data/HinfKW20_datasubset_rev.fastq.gz`	reverse fastq.gz file (input)
`Data/HinfKW20_trimmed_for_paired.fastq`	forward paired fastq file (output)
`Data/HinfKW20_trimmed_for_unpaired.fastq`	forward unpaired fastq file (output)
`Data/HinfKW20_trimmed_rev_paired.fastq`	reverse paired fastq file (output)
`Data/HinfKW20_trimmed_rev_unpaired.fastq`	reverse unpaired fastq file (output)
`ILLUMINACLIP:TruSeq2-PE.fa:2:30:10`	trim TruSeq2 adapters off the end of reads **2** (maximum seed mismatch) **30** (palindrome clip threshold) **10** (simple clip threshold)
`SLIDINGWINDOW:4:20`	trim reads when avg quality across **4** bases < Phred**20**
`HEADCROP:2`	trims the first **2** bases off every read

```
(base) mokca@MokData:~/FGDS/Module6$ trimmomatic PE -phred33 Data/HinfKW20_datasubset_for.fastq.gz Data/HinfKW20_datasubset_rev.fastq.gz
 Data/HinfKW20_trimmed_for_paired.fastq Data/HinfKW20_trimmed_for_unpaired.fastq Data/HinfKW20_trimmed_rev_paired.fastq Data/HinfKW20_tr
immed_rev_unpaired.fastq ILLUMINACLIP:/home/mokca/anaconda3/share/trimmomatic-0.39-2/adapters/TruSeq2-PE.fa:2:30:10 SLIDINGWINDOW:4:20 H
EADCROP:2
TrimmomaticPE: Started with arguments:
 -phred33 Data/HinfKW20_datasubset_for.fastq.gz Data/HinfKW20_datasubset_rev.fastq.gz Data/HinfKW20_trimmed_for_paired.fastq Data/HinfKW
20_trimmed_for_unpaired.fastq Data/HinfKW20_trimmed_rev_paired.fastq Data/HinfKW20_trimmed_rev_unpaired.fastq ILLUMINACLIP:/home/mokca/a
naconda3/share/trimmomatic-0.39-2/adapters/TruSeq2-PE.fa:2:30:10 SLIDINGWINDOW:4:20 HEADCROP:2
Using PrefixPair: 'AATGATACGGCGACCACCGAGATCTACACTCTTTCCCTACACGACGCTCTTCCGATCT' and 'CAAGCAGAAGACGGCATACGAGATCGGTCTCGGCATTCCTGCTGAACCGCTC
TTCCGATCT'
Using Long Clipping Sequence: 'AGATCGGAAGAGCGTCGTGTAGGGAAAGAGTGTAGATCTCGGTGGTCGCCGTATCATT'
Using Long Clipping Sequence: 'AGATCGGAAGAGCGGTTCAGCAGGAATGCCGAGACCGATCTCGTATGCCGTCTTCTGCTTG'
Using Long Clipping Sequence: 'TTTTTTTTTTAATGATACGGCGACCACCGAGATCTACAC'
Using Long Clipping Sequence: 'TTTTTTTTTTCAAGCAGAAGACGGCATACGA'
Using Long Clipping Sequence: 'CAAGCAGAAGACGGCATACGAGATCGGTCTCGGCATTCCTGCTGAACCGCTCTTCCGATCT'
Using Long Clipping Sequence: 'AATGATACGGCGACCACCGAGATCTACACTCTTTCCCTACACGACGCTCTTCCGATCT'
ILLUMINACLIP: Using 1 prefix pairs, 6 forward/reverse sequences, 0 forward only sequences, 0 reverse only sequences
Input Read Pairs: 1000000 Both Surviving: 979232 (97.92%) Forward Only Surviving: 9793 (0.98%) Reverse Only Surviving: 6591 (0.66%) Drop
ped: 4384 (0.44%)
TrimmomaticPE: Completed successfully
```

1.3.4 Compare the content of trimmomatic's trimmed paired data output.

```
wc Data/*_paired.fastq          # Can also use: wc ./Data/*_paired.fastq
```

Of the 1,000,000 read pairs that we started with, we still have 97.92% of our reads in the paired-end output. However, when you look at the file information, you can see that while the two paired.fastq files have the same number of lines, they differ in the number of characters. This suggests that the forward paired read file was more aggressively trimmed.

What if I installed trimmomatic manually? Note that `trimmomatic` is written in java but we have installed it with a package manager so a lot of the background associations have been handled. If, however, you installed it yourself, you must specify the whole path to the jar file to run it with the java interpreter.

So rather than just `trimmomatic` to initiate the command, you would use:

```
java -jar /path/to/Trimmomatic-0.39.jar
```

1.4.0 Rerun `fastqc` on the trimmed dataset

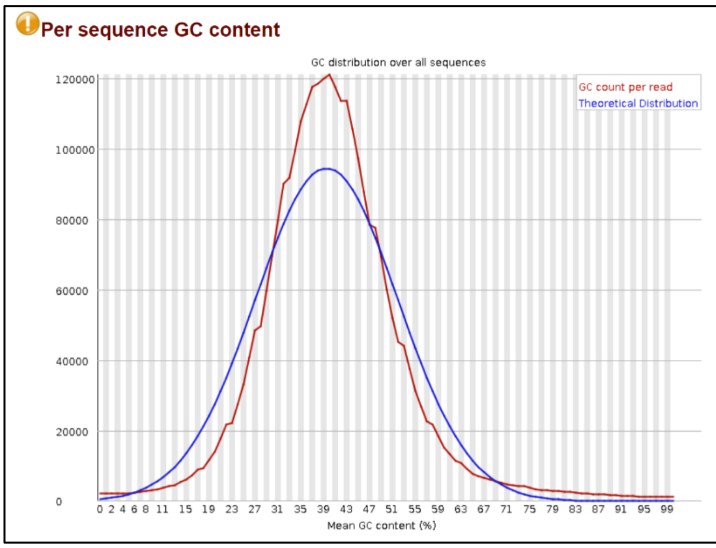
Now that you've trimmed and filtered your data, check to see if any new issues have arisen from the process.

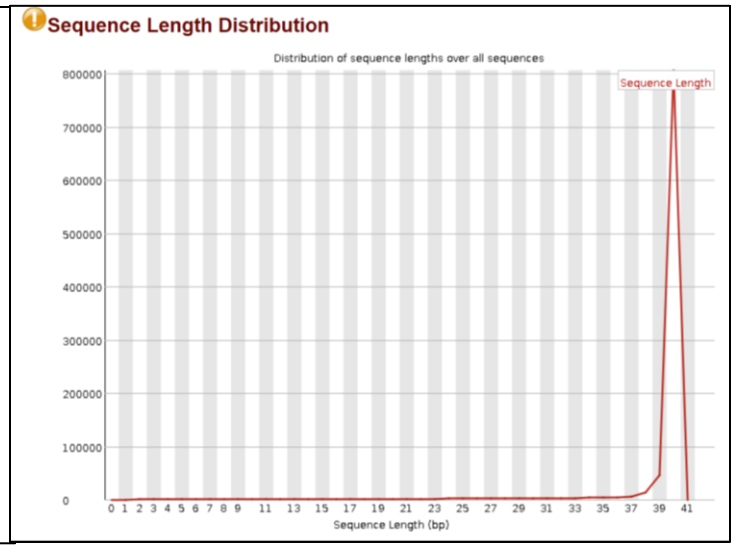
1.4.1 Create a new directory called fastqc_trimmed and save the results of a **`fastqc`** analysis to this directory.

```
cd ~/FGDS/Module6/
mkdir fastqc_trimmed

# Our actual call to fastqc
fastqc --kmers 7 --outdir fastqc_trimmed Data/HinfKW20_trimmed_for_paired.fastq
      Data/HinfKW20_trimmed_rev_paired.fastq
```

1.4.2 Review the results of your fastqc analysis. Depending on your OS, you will either open it directly from your terminal (macOS/Windows) or find it in your windows Explorer and open it from there.





While we have resolved our issues with per base sequence content, we now receive some warnings about our "**Per sequence GC content**" and "**Sequence Length Distribution**". If we were to unzip the data generated by **`fastqc`** we could look directly at the sequence length distribution and see that we now have a broad range of reads as short as 1 bp through to 40 bp. This trimmed read distribution creates the second warning from our analysis.

2.0.0 Paired-end genome assembly with velvet

Now that we have our sequencing reads trimmed and there appear to be no major failures in the data, we can continue the genome assembly process. As we learned in the “Assembly and Annotation with Galaxy” module (Module 2), **`velvet`** assembles genomes in two steps. These steps include building a hash table of kmers (**`velveth`**) and performing the assembly (**`velvetg`**).

2.1.0 Build your k-mer hash table with `velveth`

We’ll build our hash table using the **`velveth`** command. Note that unlike on Galaxy we have a pair of fastq files relating to their forward and reverse reads. We’ll see if velvet can leverage this information to its advantage when assembling the genome.

2.1.1 Ensure you are in your **Module6** directory

```
cd ~/FGDS/Module6
```

2.1.1 Examine the **`velveth`** options. The summary is quite large, so you’ll want to pipe the output to something like less.

```
velveth --help | less
```

Note from the help description that the first 2 inputs do not have a flag but are rather just expected to be the **<output_directory>** and **<hash_length>**. These two parameters must be included as the first two arguments in your **`velveth`** call!

2.1.2 Build your **`velveth`** command taking special note of the parameters. We’ll re-use this code later as part of a bash script to do some empirical analysis of hash lengths. The hash length must be an odd number that is shorter than most reads in the dataset because any reads smaller than the hash size will be ignored. It is generally stated that your hash length should be between 21 bps and the average read length (~40 bps for us) minus 10 bps.

```
velveth velveth_21_out 21 -shortPaired -fastq -separate
    Data/HinfKW20_trimmed_for_paired.fastq Data/HinfKW20_trimmed_rev_paired.fastq
```

Command	Meaning
velveth_21_out	output directory for the velveth command (created automatically)
21	hash length (21 basepairs in this case)
-shortPaired	type of reads in the input files
-fastq	format of reads in the input files
-separate	use separate files for paired reads
HinfKW20_trimmed_for_paired.fastq	input reads (forward *trimmed dataset*)
HinfKW20_trimmed_rev_paired.fastq	input reads (reverse *trimmed dataset*)

2.2.0 Assemble your genome with `velvetg`

Now that our hash table is assembled we can pass the output on to **`velvetg`**. Remember, all our **`velveth`** output is now in the **`~/FGDS/Module6/velveth_21_out`** directory.

2.2.1 Examine the **`velvetg`** command and the basic options for working with it.

```
velvetg --help                # velvetg would also trigger the same result
```

2.2.2 Build your **`velvetg`** command, again taking note of the parameters – especially our minimum contig length.

```
velvetg velveth_21_out -cov_cutoff auto -exp_cov auto -min_contig_lgth 200
```

Command	Meaning
`velveth_21_out`	input/output directory from velveth for velvetg
`-cov_cutoff auto`	automatically determine minimum coverage cutoff for contigs
`-exp_cov auto`	automatically determine expected coverage with probabilistic formula
`-min_contig_lgth 200`	minimum contig length exported to configs.fa file

```
(base) mokca@MokData:~/FGDS/Module6$ velvetg velveth_21_out/ -cov_cutoff auto -exp_cov auto -min_contig_lgth 200
[0.000000] Reading roadmap file velveth_21_out//Roadmaps
[1.031945] 1958464 roadmaps read
[1.042247] Creating insertion markers
[1.162378] Ordering insertion markers
[1.234754] Counting preNodes
[1.373565] 1396600 preNodes counted, creating them now
[2.102853] Sequence 1000000 / 1958464
[2.814440] Adjusting marker info...
[2.964703] Connecting preNodes
[3.005061] Connecting 1000000 / 1958464
[3.262104] Cleaning up memory
[3.262434] Done creating preGraph
[3.262452] Concatenation...
[3.423268] Renumbering preNodes
[3.423311] Initial preNode count 1396600
[3.432407] Destroyed 1309685 preNodes
[3.432442] Concatenation over!
[3.432444] Clipping short tips off preGraph
[3.447344] Concatenation...
[3.504689] Renumbering preNodes
[3.504742] Initial preNode count 86915
[3.505195] Destroyed 81568 preNodes
[3.505212] Concatenation over!
```

Do you remember how to save the standard output to a file?

2.3.0 Converting your `velvet` commands into a bash script

Suppose we wanted to optimize some of the parameters in our genome assembly. For instance, we could vary the hash length with three sets of values: 21, 23, and 25. Depending on your needs, you could just repeat the commands in the command-line, varying the hash length each time we run **`velveth`**. Another approach, however, would be to generate our own bash script where we repeat the commands instead.

2.3.1 Create a new script called **`velvetscript.sh`** in your **`~FGDS/Module6/`** directory. In this example we'll use **`vi`** but you could work with a separate editor as well.

```
vi velvetscript.sh            # Start a new bash script with vi
      i                       # insertion mode
      #!/bin/bash             # Add the shebang so bash knows what this script is
```

2.3.2 Copy our previous **`velveth`** and **`velvetg`** commands as two lines in your script.

```
# velvet commands for hash length 21
velveth velveth_21_out 21 -shortPaired -fastq -separate
      Data/HinfKW20_trimmed_for_paired.fastq Data/HinfKW20_trimmed_rev_paired.fastq

velvetg velveth_21_out -cov_cutoff auto -exp_cov auto -min_contig_lgth 200
```

2.3.3 Repeat the above copied commands two more times but alter the parameters to use a hash length of 23 and 25. Remember to update the output directory names and hash length ***in each command***. The other parameters in our commands will remain the same. Finish the script with these three sets of commands and save it.

```
[esc]                                       # Exit insertion mode
:wq                                         # Save and quite vi
```

```
#!/bin/bash
# This script will run three different versions of velveth by alternating
# hash length between 21, 23, and 25

# velvet commands for hash length 21
velveth velveth_21_out 21 -shortPaired -fastq -separate Data/HinfKW20_trimmed_for_paired.fastq
 Data/HinfKW20_trimmed_rev_paired.fastq
velvetg velveth_21_out -cov_cutoff auto -exp_cov auto -min_contig_lgth 200

# velvet commands for hash length 23
velveth velveth_23_out 23 -shortPaired -fastq -separate Data/HinfKW20_trimmed_for_paired.fastq
 Data/HinfKW20_trimmed_rev_paired.fastq
velvetg velveth_23_out -cov_cutoff auto -exp_cov auto -min_contig_lgth 200

# velvet commands for hash length 25
velveth velveth_25_out 25 -shortPaired -fastq -separate Data/HinfKW20_trimmed_for_paired.fastq
 Data/HinfKW20_trimmed_rev_paired.fastq
velvetg velveth_25_out -cov_cutoff auto -exp_cov auto -min_contig_lgth 200
```

2.3.4 Update your bash script permissions and run it.

```
# Remember you cannot run an executable without permission!
chmod 744 velvetscript.sh

rm -r velveth_21_out/          # Remove our previous results

bash velvetscript.sh           # Run our bash script
```

Tired of the default vi/vim colour scheme? If you're feeling tired of the default colour scheme in vi or vim, you can make choose one using with any of the following commands:

```
:colorscheme + [space] + [tab]

:color <color_scheme>        # ie :color elflord
```

To make your change more permanent you can make a .vimrc (vim run commands) file in your home directory containing a similar command code when it starts up:

```
echo 'colorscheme desert' >> ~/.vimrc
```

After that, vi will always implement the colour scheme you've chosen when it starts up.

2.4.0 Compare your results between your different `velvet` assemblies

Each **`velvet`** assembly generated includes a **`Log`** file that details information about the number of accepted contigs, the n50 (median contig length), and the maximum contig length. We can use these statistics to compare our three sets of assemblies.

2.4.1 View the **`Log`** information for our assembly based on a hash length of 21. We know that the genome should be approximately 1.83 Mbps in length based on the completed genome that we downloaded in Module 1. A good genome assembly will also have a high N50 value.

```
less ./velveth_21_out/Log
```

```
Mon Dec  6 10:26:35 2021
 velvetg velveth_21_out -cov_cutoff auto -exp_cov auto -min_contig_lgth 200
Version 1.2.10
Copyright 2007, 2008 Daniel Zerbino (zerbino@ebi.ac.uk)
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
Compilation settings:
CATEGORIES = 4
MAXKMERLENGTH = 191
OPENMP
LONGSEQUENCES

Median coverage depth = 17.170077
Final graph has 619 nodes and n50 of 8478, max 31262, total 1678412, using 1799211/1958464 reads
```

2.4.2 Use **`grep`** to help you to summarize your files. Rather than check each individual **`Log`** file, we can take advantage of our name scheme to view all three log files simultaneously. To accomplish this, we use a **`grep`** call to parse through our **`Log`** files and display the resulting matches.

```
grep -e "Median" -e "Final" ./velveth_*_out/Log
# -e signals a regex pattern will follow
```

```
(base) mokca@MokData:~/FGDS/Module6$ grep -e "Median" -e "Final" ./velveth_*_out/Log
./velveth_21_out/Log:Median coverage depth = 17.184466
./velveth_21_out/Log:Final graph has 609 nodes and n50 of 8411, max 35794, total 1677130, using 1798619/1958464 reads
./velveth_23_out/Log:Median coverage depth = 15.890110
./velveth_23_out/Log:Final graph has 492 nodes and n50 of 10265, max 46907, total 1701249, using 1787872/1958464 reads
./velveth_25_out/Log:Median coverage depth = 14.076246
./velveth_25_out/Log:Final graph has 539 nodes and n50 of 7058, max 23927, total 1617393, using 1725893/1958464 reads
```

Based on our output, which assembly is the most appropriate to continue with? In terms of all three metrics, we see that setting hash=23 has the fewest contigs and largest N50 and max values, suggesting that it is the most appropriate assembly to continue with under these parameters. Is there, however, another way to evaluate the quality control of our assemblies?

What makes a good N50 value? While intuitively the idea of an N50 suggests that higher values mean better assemblies, this is not always the case. Since the size of genomes can vary, using the N50 between assemblies of different species is not very comparable. Thus, while there may be some correlation between high N50 and high quality assemblies, this is not always the case.

Instead, if you know the target size of your genome, you can calculate other metrics like NG50. Similar in concept to N50, the calculation of NG50 requires grouping the largest contigs that sum to 50% of the estimated genome size. The smallest contig in this set determines the NG50 value.

3.0.0 Assembly quality control with BLAST

Another way that we can assess the quality of our assembly and filter the dataset of contaminant reads is by blasting our output contigs against a reference database using **`blast`**. In the interest of time, we skipped this QC step when we were working on Galaxy. Start by reviewing the **`blastn`** command help menu, which allows you to query nucleotide sequences in a nucleotide database.

3.1.0 Use the `awk` command to parse data

The **`awk`** command searches files for text, much like **`grep`**, and allows you to perform specific actions on that text. Named after its creators (*Aho, Weinberger, and Kernighan*), **`awk`** allows a programmer to write simple programs that search for patterns within each line of an input file except it can also take a specific action when a match is found. This form of pattern scanning, *and* processing can be used to generate reports or perform simple command-line operations. The general syntax of an **`awk`** command is:

```
awk options 'BEGIN {action} /selection_criteria/ {action1;action2} END {action}'
      input-file > output-file
```

Syntax	Meaning
`awk`	Initiates the call to `awk`
`options`	The list of options (ie -v var=value or var2=$variable)
`' '`	Aside from the options, all the selection/action commands are stored here
`/selection_criteria/`	The use of the slash (`/`) denotes the beginning and end of your regular expression
`{action1;action2}`	The use of curly braces `{ }` enclose the action(s) you wish to perform. This could be dependent upon a selection criteria or run without any criteria. You can perform multiple actions within a set of `{ }` and separate them using the semicolon (`;`) character.
`BEGIN {action}`, `END {action}`	These are specific actions you wish to take before the first record is read (BEGIN) or after the last record has been read (END). This is a good place to set a variable value!

The **`awk`** command also has internal features and variables. For each record (line) of a file or input, **`awk`** will use whitespace as a delimiter to separate each "word" (field) into **`$n`** variables. For instance, if a line has the following text: "`The_current      example record.`", you can access each individual field by its position:

awk variable	Field string
`$1`	The_current
`$2`	example
`$3`	record.
`$0`	The_current example record.

Furthermore, there are additional built-in variables to help keep track of your record position within a file and how to read or output fields and records.

awk variable	Meaning
`NR`	The current count of input records (usually lines) that have been seen
`NF`	The number of fields with the current input record
`FS`	The field separator character used to divide fields. The default is whitespace but this can be reassigned to another character. Note: this starts with the text to the left of the separator!!!
`RS`	The current record separator character. The default is a newline character, hence each line being a new record. Note: this starts with the text to the left of the separator!!!

OFS	The output field separator determines how **awk** separates fields when you print them. A blank space is the default.
ORS	The output record separator determines how **awk** separates records when you print them. A newline character is the default.

Lastly, the **awk** command has, amongst other things, access to I/O statements. These can allow you to do things like:

Statement	Meaning
`print`	Print the current record to standard output.
`getline`	Set $0 from the next input record.
`system(cmd-line)`	Execute a command (cmd-line) as though you were in the command line. It will return the exit status value (did it work (0) or fail etc?)
`var_name=value`	Set a var_name to a specific variable.
`"literal_string"`	We can represent literal strings (ie text) inside double-quotations (" ") and concatenate strings and variables by positioning them together without separators.

Note that the above descriptions are a non-exhaustive list of variables and statements that are available from **awk**. It is a full-fledged programming language that is extremely useful for quickly manipulating your text files and command outputs. Let's use it in our next section to address some of our questions.

3.2.0 Locating your installation of blast with `awk`

As you may recall, we did not explicitly install **blast** to our system last class, but it may already be presently installed in one or more of our environments. The question is, what is the most efficient method to identify which environment it is in? With the correct commands, we can use the power of **awk** to search through all of our environments for us.

3.2.1 Identify the environments in our Anaconda installation and turn them into a single set of input.

```
conda info -e                  # This presents us with a set of environment names
conda info -e | awk '{print $1}'    # Print the first word/field on each line
```

3.2.2 Use a regular expression as the selection criteria for printing the first field. We have managed to capture the first part of each line from our environment list but we still have additional comment lines that are making it through.

```
# This will get us all of the comment lines
conda info -e | awk '/^#/ {print $1}'

# Now we will get the opposite of our match criteria
conda info -e | awk '!/^#/ {print $1}'
```

3.2.3 We have now successfully printed out our environment names to standard output. Recall that we can search through a specific Anaconda environment's list of packages using:

`conda list -n <env_name> <package>`.

Let's combine our previous **awk** command with this information.

```
# Search the base environment for the blast package
conda list -n base blast
```

```
# Use awk to capture an env name and make a command for conda
conda info -e | awk '!/^#/ {cmd="conda list -n "$1" blast"; print cmd}'

# We made a command but we still need to run it using system!
conda info -e | awk '!/^#/ {cmd="conda list -n "$1" blast"; system(cmd)}'
```

```
(base) mokca@MokData:~/FGDS/Module6$ conda info -e | awk '!/^#/ {cmd="conda list -n "$1" blast"; system(cmd)}'

EnvironmentLocationNotFound: Not a conda environment: /home/mokca/anaconda3/envs/blast

# packages in environment at /home/mokca/anaconda3:
#
# Name                    Version                   Build  Channel
# packages in environment at /home/mokca/anaconda3/envs/alignersENV:
#
# Name                    Version                   Build  Channel
# packages in environment at /home/mokca/anaconda3/envs/prokkaENV:
#
# Name                    Version                   Build  Channel
blast                     2.16.0               hc155240_3    bioconda
blast-legacy              2.2.26               h9ee0642_4    bioconda
# packages in environment at /home/mokca/anaconda3/envs/quastENV:
#
# Name                    Version                   Build  Channel
blast                     2.16.0               hc155240_3    bioconda
# packages in environment at /home/mokca/anaconda3/envs/r_432ENV:
#
# Name                    Version                   Build  Channel
# packages in environment at /home/mokca/anaconda3/envs/sambcftoolsENV:
#
# Name                    Version                   Build  Channel

EnvironmentLocationNotFound: Not a conda environment: /home/mokca/anaconda3/envs/blast
```

3.2.4 As you can see from the output, the command is nearly perfect! We now see that blast is installed in **`prokkaENV`** as well as **`quastENV`**! However, we get a small error at the end saying that the environment "**`blast`**" is not found. A little digging would show us that our initial call to **`conda info -e`** creates an additional blank line at the beginning and end of its output.

We can use the **`NF`** variable as criteria to our **`awk`** command to clean it up. An empty line will have no fields (ie NF=0). Using boolean logic, we can add the following criteria: for a line to be processed, it must not start with a **`#`** AND it cannot be empty. We use the logical AND (**`&&`**) to convey this requirement.

```
# Make a small fix to filter away empty lines
conda info -e | awk '!/^#/ && NF {cmd="conda list -n "$1" blast"; system(cmd)}'
```

Of course, now that we have our command, it seems quite long. However, it sure beats checking each individual environment separately – especially if you have more than a few.

3.3.0 Create a subset of your contigs with `awk`

Now that we've verified the installation of blast, we can go ahead and activate **`prokkeENV`** to use one of **`blast`**'s commands to analyse the quality of our contigs.

This type of **`blast`** can be quite time consuming because you're querying a large database with multiple long contigs. Therefore, as an example for how a **`blast`** QC on an assembly would work, we're only going to use the first three contigs from our assembly today. In practice, you would want to query all your contigs to ensure that none of them were formed from contaminant reads.

3.3.1 Capture a subset of our contigs – the first 3 to be precise – using **`awk`** and put it into a new file **`contigs_3.fa`**.

```
cd velveth_23_out                       # change directories
awk 'BEGIN {n=0} /^>/ {n++} n>3 {exit} {print}' contigs.fa > contigs_3.fa
```

Statement	Meaning
`Awk`	Begin the `awk` command
`/^>/ {n++}`	Look for a `>` at the beginning of a line. When you find a match, increment a counter (`n`) by 1. Therefore, every time we encounter a contig header, the counter will increment.
`n>3 {exit}`	If the value of `n` becomes greater than 3, then you will exit `awk`. Otherwise, go to the next command.
`{print}`	Print the record to standard output.
`contigs.fa`	This is our input file – a set of contigs made by `velvetg`
`> contigs_3.fa`	Write our standard output to `contigs_3.fa`

3.3.2 Confirm that we have only taken the first 3 entries from our contig file

```
head -5 contigs_3.fa                  # Look at the first 5 lines of your file
grep -c "^>" contigs_3.fa             # count the occurrences of ">"
```

3.4.0 Use `blastn` to examine your contig subset

We now have a subset of our data with just 3 contig entries. We'll build a **`blast`** command and use this as input instead of our full set of contigs.

3.4.1 Activate **`prokkaENV`** and review the **`blastn`** information. There is quite a bit of information on the help page so you'd better pipe the output to **`less`**.

```
conda activate prokkaENV                 # Activate our environment
blastn -help | less
```

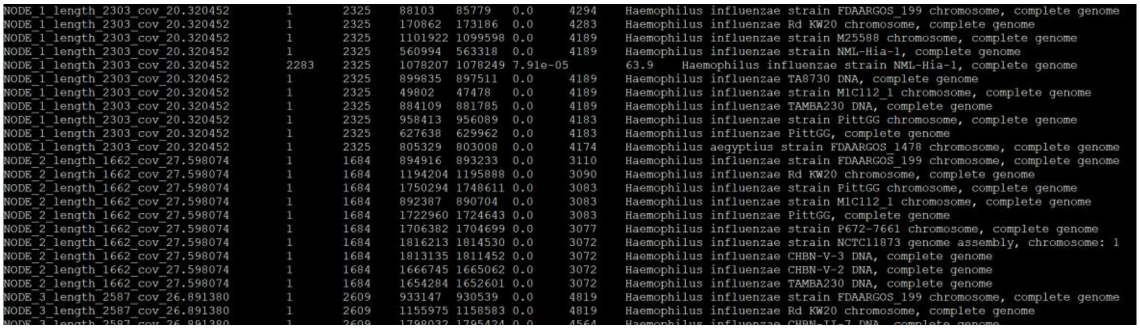
3.4.2 Build and run a **`blastn`** command to query the non-redundant (**`nr`**) database with the three contigs that you pulled from your genome assembly. There are a lot of options available here for you to review, but we want mostly defaults. These commands are designed to run from your **velveth_23_out** directory.

```
blastn -query contigs_3.fa -db nr -out velvet_23_contigs_blast.out -max_target_seqs
      10 -remote -outfmt '6 qseqid qstart qend sstart send evalue bitscore stitle'
```

Command	Meaning
`-query`	input file in fasta format (`contigs_3.fa`)
`-db`	database you want to query (`nr`)
`-out`	output file (`velvet_23_contigs_blast.out`)
`-max_target_seqs`	the maximum number of subjects (hits per target) you want to include in the output
`-remote`	execute search on the NCBI server
`-outfmt`	output format (`6`=tabular output, where we have specified output columns to include)

3.4.3 Check your **`blast`** output to ensure that there is a significant hit for *H. influenzae* in the top ten hits for each of your three contigs. All of our hits should be in an *H. influenzae* strain because it is a very well characterized species from which multiple strains have been sequenced.

```
less velvet_23_contigs_blast.out
```



```
NODE_1_length_2303_cov_20.320452	1	2325	88103	85779	0.0	4294	Haemophilus influenzae strain FDAARGOS_199 chromosome, complete genome
NODE_1_length_2303_cov_20.320452	1	2325	170862	173186	0.0	4283	Haemophilus influenzae Rd KW20 chromosome, complete genome
NODE_1_length_2303_cov_20.320452	1	2325	1101922	1099598	0.0	4189	Haemophilus influenzae strain M25588 chromosome, complete genome
NODE_1_length_2303_cov_20.320452	1	2325	560994	563318	0.0	4189	Haemophilus influenzae strain NML-Hia-1, complete genome
NODE_1_length_2303_cov_20.320452	2283	2325	1078207	1078249	7.91e-05	63.9	Haemophilus influenzae strain NML-Hia-1, complete genome
NODE_1_length_2303_cov_20.320452	1	2325	899835	897511	0.0	4189	Haemophilus influenzae TA8730 DNA, complete genome
NODE_1_length_2303_cov_20.320452	1	2325	49802	47478	0.0	4189	Haemophilus influenzae strain M1C112_1 chromosome, complete genome
NODE_1_length_2303_cov_20.320452	1	2325	884109	881785	0.0	4189	Haemophilus influenzae TAMBA230 DNA, complete genome
NODE_1_length_2303_cov_20.320452	1	2325	958413	956089	0.0	4183	Haemophilus influenzae strain PittGG chromosome, complete genome
NODE_1_length_2303_cov_20.320452	1	2325	627638	629962	0.0	4183	Haemophilus influenzae PittGG, complete genome
NODE_1_length_2303_cov_20.320452	1	2325	805329	803008	0.0	4174	Haemophilus aegyptius strain FDAARGOS_1478 chromosome, complete genome
NODE_2_length_1662_cov_27.598074	1	1684	894916	893233	0.0	3110	Haemophilus influenzae strain FDAARGOS_199 chromosome, complete genome
NODE_2_length_1662_cov_27.598074	1	1684	1194204	1195888	0.0	3090	Haemophilus influenzae Rd KW20 chromosome, complete genome
NODE_2_length_1662_cov_27.598074	1	1684	1750294	1748611	0.0	3083	Haemophilus influenzae strain PittGG chromosome, complete genome
NODE_2_length_1662_cov_27.598074	1	1684	892387	890704	0.0	3083	Haemophilus influenzae strain M1C112_1 chromosome, complete genome
NODE_2_length_1662_cov_27.598074	1	1684	1722960	1724643	0.0	3083	Haemophilus influenzae PittGG, complete genome
NODE_2_length_1662_cov_27.598074	1	1684	1706382	1704699	0.0	3077	Haemophilus influenzae strain P672-7661 chromosome, complete genome
NODE_2_length_1662_cov_27.598074	1	1684	1816213	1814530	0.0	3072	Haemophilus influenzae strain NCTC11873 genome assembly, chromosome: 1
NODE_2_length_1662_cov_27.598074	1	1684	1813135	1811452	0.0	3072	Haemophilus influenzae CHBN-V-3 DNA, complete genome
NODE_2_length_1662_cov_27.598074	1	1684	1666745	1665062	0.0	3072	Haemophilus influenzae CHBN-V-2 DNA, complete genome
NODE_2_length_1662_cov_27.598074	1	1684	1654284	1652601	0.0	3072	Haemophilus influenzae TAMBA230 DNA, complete genome
NODE_3_length_2587_cov_26.891380	1	2609	933147	930539	0.0	4819	Haemophilus influenzae strain FDAARGOS_199 chromosome, complete genome
NODE_3_length_2587_cov_26.891380	1	2609	1155975	1158583	0.0	4819	Haemophilus influenzae Rd KW20 chromosome, complete genome
```

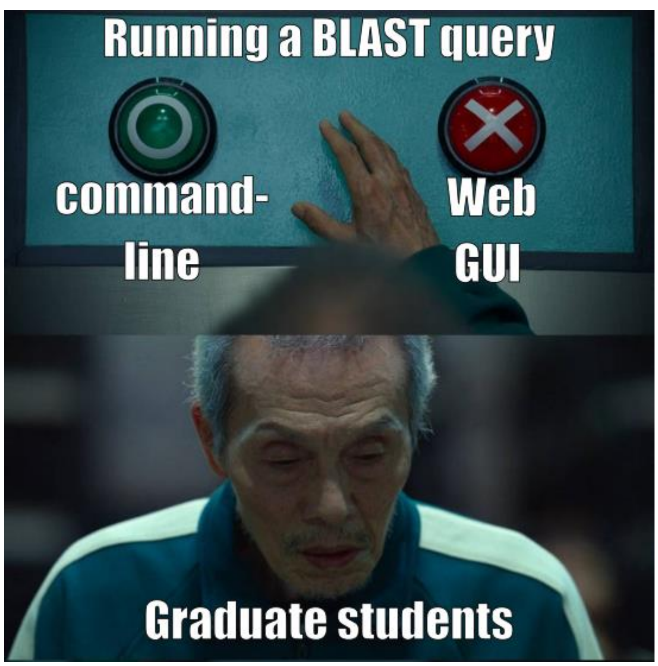
You should expect to see the following columns:

- Query ID
- Query Start
- Query End
- Subject Start
- Subject End
- Expect-value (Number of expected hits of similar quality by chance, which depends on the database size)
- Bit-score (Measure of sequence similarity that is independent of database size, but dependent on the length of the query)
- Subject Title (Full Header Line)

It appears that we retrieved a lot of hits to other *Haemophilus influenza* genomes which suggests, for at least our first 3 contigs, that we have a fairly successful assembly. The remaining contigs, however, would still need to be checked.

Still uncomfortable with the command line? You may still prefer to stay with the more familiar GUI-based blast running off the web. If that is the case, you can also run your blast analyses using the online platform.

You can also find more details on running the BLAST+ command line applications online via NCBI.



3.5.0 Use quast to examine your assembly

Much like we did in Module 2, we'll now use QUAST to help examine the quality of our assembly. It will generate reports to help us ascertain how well our assembly matches up to a known reference assembly. Again, this is only possible IF you have a reference assembly to a close relative or a wild-type strain etc.

3.5.1 Activate **quastENV** and review the **quast** help information. There is quite a bit of information on the help page so you'd better pipe the output to **less**.

```
conda activate quastENV              # Activate our environment
quast --help | less
```

3.5.2 Build and run a **quast** command to compare your **velveth_23_out** contig results. These commands are designed to run from your **velveth_23_out** directory but you might recall that we'll also need a reference genome, which should be found in **Module 1**.

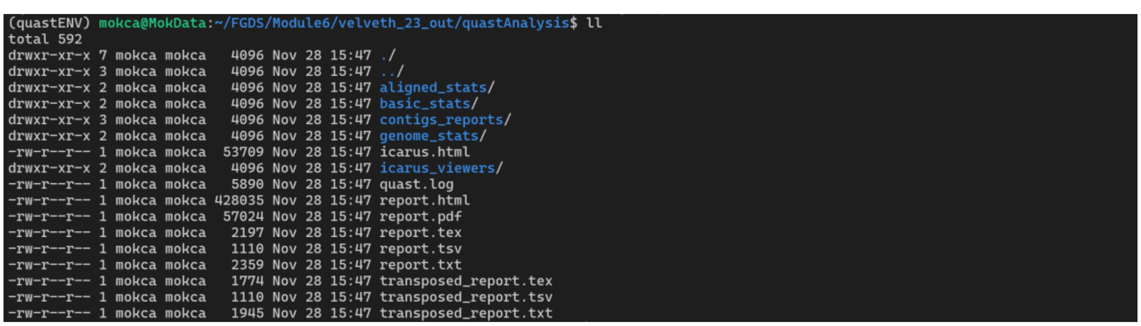
```
cd ~/FGDS/Module6/velvet_23_out/
quast -m 200 -o quastAnalysis -r ~/FGDS/Module1/downloads/HinfKW20_genomic.fna
      contigs.fa
```

Command	Meaning
-m 200	The minimum contig size cutoff.
-o quastAnalysis	The output directory of our analysis.
-r ~ /FGDS/Module1/downloads/HinfKW20_genomic.fna	Location of your genomic reference file.
contigs.fa	The file containing the contigs of your assembly

3.5.3 You can proceed to view your quast results by finding them in the quastAnalysis folder. You can navigate to the file and then use the **explorer.exe** (WSL2) or **open** (MacOS) commands.

```
cd quastAnalysis/icarus_viewers/
explorer.exe alignment_viewer.html          #MacOS: open alignment_viewer.html
```

Additional files on can be found in the subdirectories of **quastAnalysis** including missassembly information in the **contigs_reports** folder.



4.0.0 Genome annotation with Prokka

As was the case when we completed our assembly on Galaxy, we now want to finish up by annotating all the coding features in the assembled genome. This can also be a useful tool for quality control if you know approximately how many coding regions you expect to identify. Remember that **`prokka`** is just one of many tools that can be used for genome annotation but is particularly optimized for speed and simplicity. It has been shown to perform very well for annotation of bacteria, archaea, and viruses.

4.1.0 Annotate your velvet genome with `prokka`

We'll continue working from the **`~/FGDS/Module6/velveth_23_out/`** directory

4.1.1 Start by reviewing the **`prokka`** options and checking the version you are using just so we can all be on the same page.

```
cd ~/FGDS/Module6/velveth_23_out
conda activate prokkaENV
prokka --help
prokka --version
```

```
Computation:
  --cpus [N]          Number of CPUs to use [0=all] (default '8')
  --fast              Fast mode - only use basic BLASTP databases (default OFF)
  --noanno            For CDS just set /product="unannotated protein" (default OFF)
  --mincontiglen [N]  Minimum contig size [NCBI needs 200] (default '1')
  --rfam              Enable searching for ncRNAs with Infernal+Rfam (SLOW!) (default '0')
  --norrna            Don't run rRNA search (default OFF)
  --notrna            Don't run tRNA search (default OFF)
  --rnammer           Prefer RNAmmer over Barrnap for rRNA prediction (default OFF)
(prokkaENV) mokca@LAPTOP-7LF6OG94:~/FGDS/Module6/velveth_23_out$ prokka --version
prokka 1.14.6
(prokkaENV) mokca@LAPTOP-7LF6OG94:~/FGDS/Module6/velveth_23_out$
```

4.1.2 Build a basic **`prokka`** command that will annotate all of the coding and gene features in your reference genome using the bacterial database. You may recognize this command from last week when we tested our **`prokka`** installation.

```
prokka --kingdom Bacteria --outdir HinfKW20_annotation --prefix HinfKW20 --locustag
    HinfKW20Gene --compliant --force --quiet contigs.fa &
```

Command	Meaning
`--kingdom`	database to use for annotation (`Bacteria`)
`--outdir`	output directory name (`HinfKW20_annotation`). This will generate the new directory for us.
`--prefix`	output file name prefix
`--locustag`	output locus tag prefix
`--compliant`	force contig and gene compliance with Genbank
`--force`	write over output directory if one already exists
`--quiet`	do not print all progress to the screen while prokka is running
`&`	Run this process in the background. This frees us to do other commands.

Note we have excluded tags for `--genus`, `--species`, and `--strain` which we used in our annotation commands on Galaxy.

```
(prokkaENV) mokca@LAPTOP-7LF6OG94:~/FGDS/Module6/velveth_23_out$ prokka --kingdom Bacteri
a --outdir HinfKW20_annotation --prefix HinfKW20 --locustag HinfKW20Gene --compliant --fo
rce --quiet contigs.fa &
[1] 14558
```

4.1.3 When you have a process running in the background, you can check on how it is progressing using the command `top`. You'll notice that our call to `prokka` produced a number as output. This is the process ID, and we can use this to find the process when we run `top`.

```
top                         # this runs an interactive program like less
```

```
14786 mokca     20   0   10656    700    660 S  0.3   0.0   0:00.01 sh
14789 mokca     20   0   10656    704    668 S  0.3   0.0   0:00.01 sh
14793 mokca     20   0   10656    704    664 S  0.3   0.0   0:00.01 sh
14794 mokca     20   0   10656    700    660 S  0.3   0.0   0:00.01 sh
    1 root      20   0   10044    480    356 S  0.0   0.0  17:21.35 init
26543 root      20   0   10044     96     48 S  0.0   0.0   0:00.03 init
26544 mokca     20   0   18576   3372   3256 S  0.0   0.0 615:28.54 bash
14558 mokca     20   0   42016  25832  25796 S  0.0   0.2   0:00.99 perl
14625 mokca     20   0       0      0      0 Z  0.0   0.0   0:00.00 sh
14693 mokca     20   0       0      0      0 Z  0.0   0.0   0:00.01 sh
14695 mokca     20   0   10656    700    652 S  0.0   0.0   0:00.03 sh
14755 mokca     20   0   10656    712    668 S  0.0   0.0   0:00.00 sh
14761 mokca     20   0   10656    708    668 S  0.0   0.0   0:00.00 sh
14780 mokca     20   0   10656    700    660 S  0.0   0.0   0:00.00 sh
```

A closer look at our process tells us that `prokka` is actually running as a `perl` script! Use "q" to quit `top`. With this command-line utility, you can keep track of your processes and see when they may be stuck.

4.1.4 Review all of the `prokka` output files (12 total) in your `HinfKW20_annotation` directory.

```
ls -la HinfKW20_annotation
less HinfKW20_annotation/HinfKW20.txt
```

```
organism: Genus species strain
contigs: 285
bases: 1701273
CDS: 1592
gene: 1634
rRNA: 2
tRNA: 39
tmRNA: 1
HinfKW20.txt (END)
```

Learn more about Prokka! You can learn more details about running prokka at the command line with advanced settings like genus specific databases and personalized protein databases at the following link: https://github.com/tseemann/prokka

5.0.0 Bash (to) basics: creating a bash pipeline

Through a series of similar steps as Galaxy we've generated our own command-line version of assembly and annotation. Recall our velvetscript.sh that we created back in section **2.3.0**. It was focused on running multiple versions of velvet for assembly. We'll take this a step further to make an assembly and annotation pipeline.

5.1.0 Download assemblyAndAnnotation.sh

An updated version of this script has been created to run additional commands. To work with it, you must first download and update its permissions.

5.1.1 Download assemblyAndAnnotation.sh from the course server.

```
cd ~/FGDS/Module6/                    # Set your working directory
history | grep sftp                   # to rerun a command use: !XXX
sftp fgds@142.150.215.186             # Or reuse this command
      PW: fall2024
ls Data/Module6/

# Download the bash script to the Module6 directory
get Data/Module6/assemblyAndAnnotation.sh
exit
```

5.1.2 Set permissions for execution of your script

```
chmod 744 assemblyAndAnnotation.sh
```

5.2.0 Review the description and header comments of assemblyAndAnnotation.sh

Before we proceed to editing and updating the script with a few flourishes, we can note a few important pieces of information provided in the initial comment lines of the script.

5.2.1 View the contents of the bash script

```
less assemblyAndAnnotation.sh
```

You'll see a two main sections in this area denoting information on the Input and Output of the script. They can be summarized as follows:

Input: there are 4 arguments at the time of running that denote the partial path to the paired (forward and reverse) fastq files, the hash size used for velveth, and an organism name used as a prefix for the annotation step. The order in which these are supplied is extremely important and cannot be changed.

Output: there are 5 main sets of files produced including a log file and 4 additional folders containing data from the various steps in our pipeline. Each is described on a separate comment line.

An additional consideration with this script is the fact that it is built solely to work with paired-end short read data made using Illumina TruSeq2 adapters. Changing your data type would require you to take a closer look at these steps. Also note that at each step, a quick message is directed to the standard output with the **`echo`** command. This will allow us to loosely keep track of the script's progress – a very useful feature!

Don't forget! Bash scripts are just a large copy of your commandline calls. That means that you CANNOT separate or break a single command across multiple lines – unless you use a specific syntax!

5.3.0 Update and complete the assemblyAndAnnotation.sh

Now that you have downloaded and reviewed the proto-script, it will need to be edited with some additional commands so that it can run properly in the command line. Look for "…" to help identify places to alter.

5.3.1 Open up the script for editing

```
vi assemblyAndAnnotation.sh
:set number                    # this will add line numbers to your display
i                              # begin insertion mode
```

5.3.2 Look at the beginning of the script. It has no **shebang** identifying its type so you must add one.

```
#!/bin/bash                                                  # Line 1
```

5.3.3 In order to work with Anaconda commands from within a bash script, you will need to initialize it using the following command under the "`Preparation`" section:

```
source ~/anaconda3/etc/profile.d/conda.sh                    # Line 23
```

5.3.4 Update the **fastqc** command which is missing the input files. Here we will concatenate the file names located in the variable **$1** and **$2** with the expected extension **.fastq**. We can concatenate variables and text using double quotes (`" "`). Your complete line 20 should look like:

```
fastqc --kmers 7 --outdir fastqc "$1.fastq.gz" "$2.fastq.gz"      # Line 30
```

5.3.5 Recall that the command we used for **trimmomatic** is rather long. To make it clearer in our bash script, we can split a single command into multiple lines with the back slash (\). For this to work correctly, there can be NO whitespace following the \.

On the next line, the command will concatenate where the text begins again so you can use indentation to denote the same command is continuing. We'll use two spaces as indentation at the beginning of our continued command here. Your trimmomatic command at line 37 will now span to line 41 with the last `ILLUMINACLIP` command spanning line 40:

```
trimmomatic PE -phred33 "$1.fastq.gz" "$2.fastq.gz" \
  "$1_trimmed_paired.fastq" "$1_trimmed_unpaired.fastq" \
  "$2_trimmed_paired.fastq" "$2_trimmed_unpaired.fastq" \
  ILLUMINACLIP:/home/<your_user_name>/anaconda3/share/trimmomatic-0.39-
2/adapters/TruSeq2-PE.fa:2:30:10 SLIDINGWINDOW:4:20 HEADCROP:2 \
&>> AAscript.log
```

5.3.6 Prior to running **prokka** in our script we need to remember that it is installed in its own environment. We'll need to activate **prokkaENV** to gain access to our **prokka** package. At line 58 add the following:

```
conda activate prokkaENV                                     # Line 58
```

5.3.7 Save the changes to your script and exit **vi**.

```
[esc]
:wq
```

5.4.0 Run your first bash script pipeline

Now that we have completed our bash script, we can proceed with running it for the first time. Take a closer look at the code structure you can see that it is best to run this script within the final output folder you would like to use. Keep in mind that all of the output directories have been generated based on relative paths.

5.4.1 Make a folder for your output and change the working directory to this new folder.

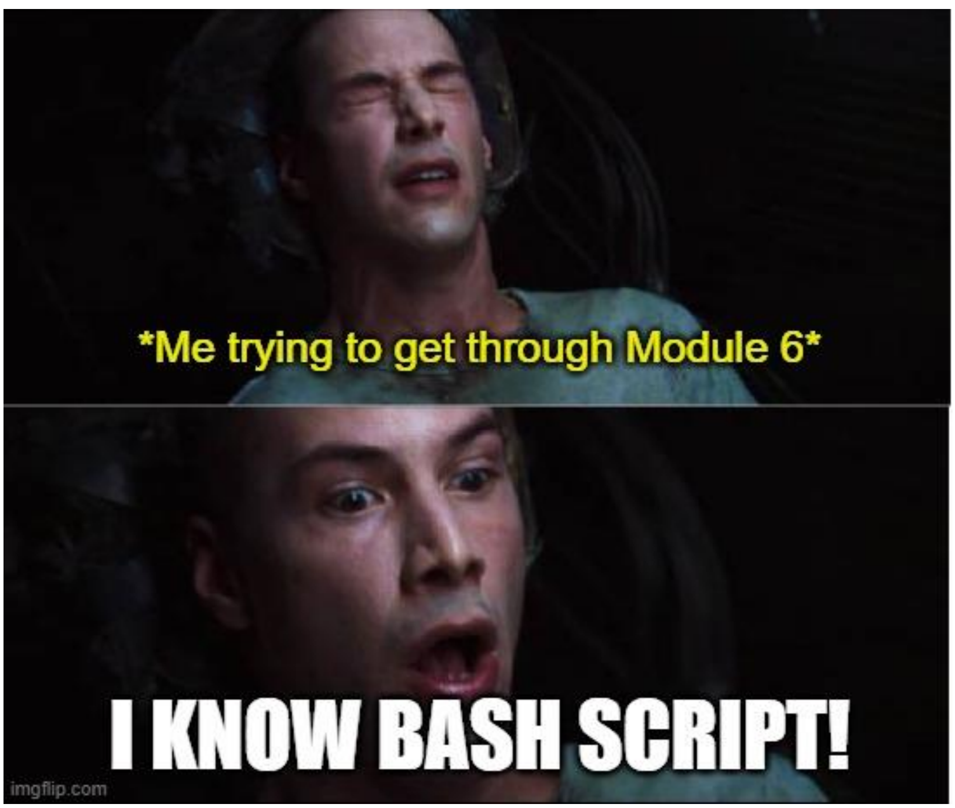
```
cd ~/FGDS/Module6
mkdir pipeline
cd pipeline
```

5.4.2 Run the pipeline bashscript. Recall that the file for this script is located in ~/FGDS/Module6/

5.4.3 When your script eventually completes, you'll be able to view the various output. You can actually do so while **`prokka`** is running as well.

```
less AAscript.log
```

```
ls -la /HinfKW20_annotation
```



That's it for today! Good luck with your pipelines!

6.0.0 Class summary

That concludes our sixth and penultimate lecture bringing you up to speed on running our own assembly and annotation pipeline through the command line interface. Next module we'll focus on building our RNAseq pipeline and learning a little bit more programming. Altogether we've explored the following in this module:

- Quality checking, filtering, assembling and annotating from raw fastq reads in the command line.
- Generating `awk` commands to parse and process data.
- Creating custom bash scripts to run a simple bioinformatics pipeline.

6.1.0 Post-lecture assessment (9% of final grade)

Soon after this lecture, a homework assignment will be made available on Quercus in the assignment section. It will build on the ideas and/or data generated within this lecture. Each homework assignment will be worth 9% of your final mark. If you have assignment-related questions, please try the following steps in the order presented:

- Check the internet for a solution – read forums and learn to navigate for answers.
- Generate a discussion on Quercus outlining what you've tried so far and see if other students can contribute to a solution.
- Contact course teaching assistants or the instructor.

6.2.0 Suggested class preparation for Module 7

Next week we will complete our exploration of bioinformatics tools through the lens of the command line. We'll cover variant calling and RNASeq as well as explore some additional basic programming concepts. To prepare for this, we suggest the following Coursera Modules:

- Command Line Tools for Genomic Data Science, ***Lilana Florea, PhD***:
  - Module 3: Alignment and Sequence Variation (52 mins)
  - Module 4: Tools for Transcriptomics (1hr, 10mins)

6.3.0 Acknowledgements

This course was originally conceived and produced by Dr. David Guttman and Dr. Marcus Dillon.

- **Revision 1.0.0**: edited and prepared by David Guttman, Ph.D. and Marcus Dillon, Ph.D.
- **Revision 2.0.0**: edited and prepared for *CSB1021HF LEC0131, 10-2021* by Calvin Mok, Ph.D., Education and Outreach, CAGEF.
- **Revision 2.0.1**: edited and prepared for *CSB1021HF LEC0131, 10-2023* by Calvin Mok, Ph.D., Education and Outreach, CAGEF.
- **Revision 2.0.2**: edited and prepared for *CSB1021HF LEC0131, 10-2024* by Calvin Mok, Ph.D., Education and Outreach, CAGEF.

CAGEF | University of Toronto
Centre for the Analysis of
Genome Evolution & Function