
M. tuberculosis Bioinformatics Workshop Documentation

Release 0.1

Ulas Karaoz

Jul 23, 2018

Contents

1	Overview	3
2	Workshop Agenda	5
3	Workshop Computer Labs	7

This workshop was co-organized by Berkeley Lab Ecology Department, UCSF Division of Pulmonary, Critical Care, Allergy and Sleep Medicine, and The Philippine Genome Center (PGC).

Participants were from National Institutes of Health at the University of the Philippines, Research Institute for Tropical Medicine, and The Philippine Genome Center (PGC).



CHAPTER 1

Overview

This five day bioinformatics workshop is intended to open doors to applying bioinformatics for microbial genomics and phylogenomics, with a specific focus on *M. tuberculosis*. The overarching goal is to drive engagement with data analysis by increasing the reliability and quality of data interpretation regarding microbial genomics. The workshop is organized as a mixture of lectures and hands-on practicals.

1.1 Instructor

Dr. Ulas Karaoz (Berkeley Lab)

1.2 Keynote

Midori Kato-Maeda, MD (UCSF): Genomics to improve the diagnosis of drug resistant tuberculosis

2.1 Agenda and Lectures' Slides

2.1.1 Tuesday

PM

Intermediate to Advanced Unix Shell for Bioinformatics

2.1.2 Wednesday

AM

Lecture: Whole-genome sequencing: technologies and data

- Overview of short and long read sequencing technologies
- Project design considerations: library complexity, data quantity and quality

Lecture: Genomics of M. tuberculosis and Genome Assembly Algorithms

- Short intro to genomics and population genomics of M. tuberculosis
- Reference-based, de-novo, and hybrid genome assembly

PM

Hands-on computer lab:

- Short and Long Read Genome Assembly
- Assembly Quality Control
- Genome Annotation

2.1.3 Thursday

AM

Lecture: Read Mapping and variant calling

PM

Hands-on computer lab:

- Read mapping
- Variant calling

2.1.4 Friday

AM

Lecture: Microbial phylogenetics and phylogenomics

Lecture: Reproducible research in high-throughput biology and building bioinformatics pipelines

PM

Hands-on practical:

- *M. tuberculosis* phylogenetics and phylogenomics
- Building computational pipelines with Snakemake

2.1.5 Saturday

AM

Lecture & Practical: Building computational pipelines with python and Snakemake

3.1 Day 1: Unix Shell Refresher for Bioinformatics

Unix Shell Refresher for Bioinformatics

3.1.1 Introductory Unix Shell

Updated June 2018 by Ulas Karaoz

Credits

Original author: *Tracy Teal* for [Data Carpentry](#)

Original contributors: *Paul Wilson, Milad Fatenejad, Sasha Wood and Radhika Khetani* for [Software Carpentry](#)

Additional contributors: *Titus Brown, Karen Word* (<https://github.com/ngs-docs/2015-shell-genomics/>)

Notes

This workshop operates under the [Software Carpentry Code of conduct](#)

Learning Objectives

- What is the shell?
- How do you access it?
- How do you use it and what is it good for?
 - Running commands
 - Storing files in folders

- Manipulating files
 - Automating actions
 - Where are resources where I can learn more?
-

Important: Before starting:

1. Download the test data. Launch your shell terminal, type or copy/paste the following:

```
wget https://s3-us-west-1.amazonaws.com/mtb-bioinformatics-workshop/shell-data.zip
```

2. Unpack the *.zip* archive:

```
unzip shell-data.zip
```

3. We'll be posting code snippets to: https://public.etherpad-mozilla.org/p/mtb_bioinformatics_shell
-

What is the shell?

The *shell* is a program that presents a command line interface which allows you to control your computer using commands entered with a keyboard instead of controlling graphical user interfaces (GUIs) with a mouse/keyboard combination.

Some specific reasons to learn about the shell:

- For most bioinformatics tools, you have to use the shell. There is no graphical interface. If you want to work in metagenomics or genomics you're going to need to use the shell.
- The shell gives you *power*. The command line gives you the power to do your work more efficiently and more quickly. When you need to do things tens to hundreds of times, knowing how to use the shell is transformative.
- To use remote computers or cloud computing, you need to use the shell.

Information on the shell

The challenge with UNIX is that it's not particularly simple - it's a power tool, with its own deep internal logic with lots of details. The joke is that Unix is user-friendly - it's just very selective about who its friends are!

Shell cheat sheets:

- <http://fosswire.com/post/2007/08/unixlinux-command-cheat-sheet/>
- https://github.com/swcarpentry/boot-camps/blob/master/shell/shell_cheatsheet.md

Explain shell - a web site where you can see what the different components of a shell command are doing.

- <http://explainshell.com>
- <http://www.commandlinefu.com>

Starting with the shell

We will spend most of our time learning about the basics of the shell by manipulating some experimental data we downloaded (see above)

Open up the shell and type the command

```
pwd
```

and then hit ENTER - this will give you a directory name.

Once that's done, type:

```
ls
```

and hit ENTER. You should see a listing of files, with 'shell-data.zip' and *data* among them.

Running commands

`pwd` and `ls` are examples of commands - programs you run at the shell prompt that do stuff. `pwd` stands for *print working directory*, while `ls` stands for *list files*.

Another command you'll find yourself using a lot is `cd`, which stands for *change directory*. Try typing:

```
cd data
```

and then:

```
pwd
```

You should see that you're now in the *data*/subdirectory (or folder) underneath the original directory. Type `ls` to see what files are in here.

What's going on? The shell has a concept of *working directory*, which is basically the default location for commands to look when you run them. When you run `ls`, by default it looks in your current working directory; when you run `cd`, it changes your current working directory.

What's the difference between `cd` and *data*? Here, `cd` is the command, and *data* is an *argument* to that command - think of the command as the verb, and the argument as the noun upon which the verb acts.

In this tutorial, commands are shown as I am a command: you can type me in your shell terminal.

Now type:

```
cd ..
```

and type `ls`. You should see at least two entries, *shell-data.zip* and *data*. Here you're using shorthand notation to go back up a directory.

Type:

```
ls data
```

to tell `ls` to look in a different directory than your current working directory. This is equivalent to:

```
cd data
ls
cd ..
```

Files and directories

Go back into the *data* directory and list the files:

```
cd data
ls
```

In here, all mixed up together are files and directories/folders. If we want to know which is which, we can type:

```
ls -F
```

Anything with a */* after it is a directory. Things with a *** after them are programs. If there's nothing there it's an otherwise unremarkable file (e.g. a data file).

You can also use the command:

```
ls -l
```

to see whether items in a directory are files or directories. `ls -l` gives a lot more information too, such as the size of the file.

Command line options

Most programs take additional options (or **arguments**) that control their exact behavior. For example, *-F* and *-l* are **arguments** to `ls`. The `ls` program, like many programs, take a lot of arguments. But how do we know what the options are to particular commands?

Most commonly used shell programs have a manual. You can access the manual using the `man` program. Try entering:

```
man ls
```

This will open the manual page for `ls`. Use the space key to go forward and `b` to go backwards. When you are done reading, hit `q` to quit.

Programs that are run from the shell can get extremely complicated. To see an example, open up the manual page for the `find` program. No one can possibly learn all of these arguments, of course. So you will probably find yourself referring back to the manual page frequently.

The Unix directory file structure

As you've already just seen, you can move around in different directories or folders at the command line. Why would you want to do this, rather than just navigating around the normal way.

When you're working with bioinformatics programs, you're working with your data and it's key to be able to have that data in the right place and make sure the program has access to the data. Many of the problems people run in to with command line bioinformatics programs is not having the data in the place the program expects it to be.

Moving around the file system

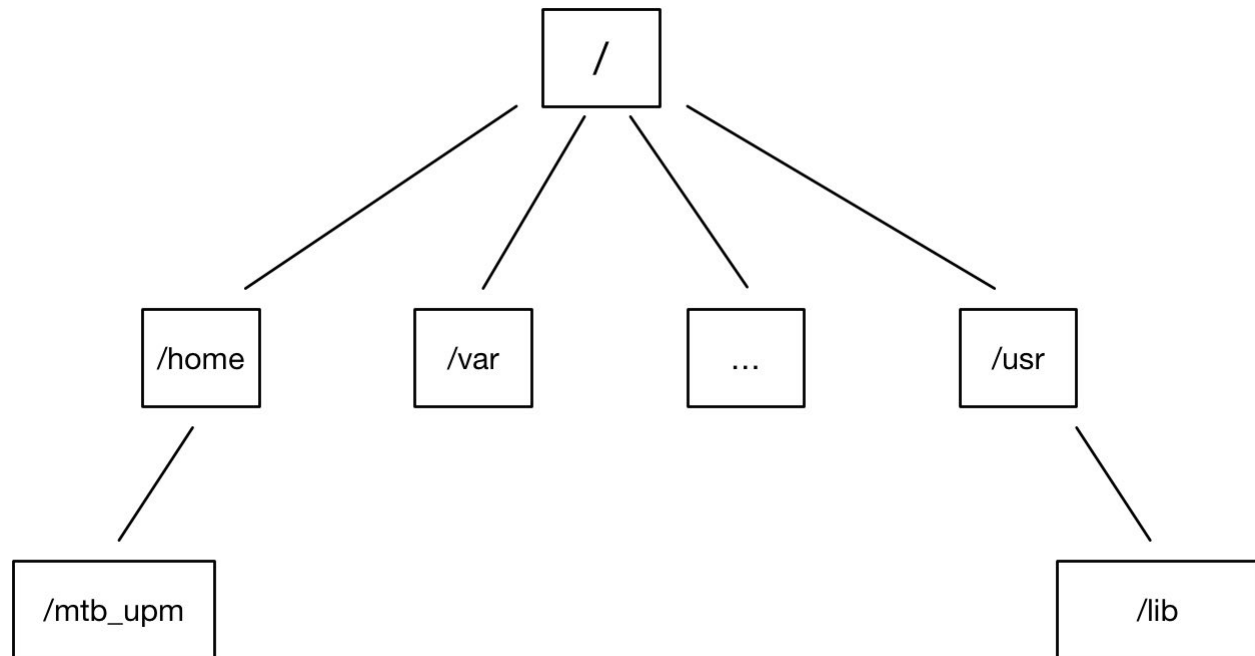
Let's practice moving around the file system a bit.

We're going to work in that *data* directory we just downloaded.

First let's navigate there using the regular way by clicking on the different folders.

First we did something like go to the folder of our username. Then we opened *data*

This is called a hierarchical file system structure, like an upside down tree with root (/) at the base that looks like this.



That (/) at the base is often also called the *top* level.

When you are working at your computer or log in to a remote computer, you are on one of the branches of that tree, your home directory (/home/mtb_upm)

Now let's go do that same navigation at the command line.

Type:

```
cd
```

This puts you in your home directory. This folder here.

Now using `cd` and `ls`, go in to the *data* directory and list its contents.

Let's also check to see where we are. Sometimes when we're wandering around in the file system, it's easy to lose track of where we are and get lost.

Again, if you want to know what directory you're currently in, type:

```
pwd
```

What if we want to move back up and out of the *data* directory? Can we just type `cd home`? Try it and see what happens.

To go **back up a level** we need to use ...

Type:

```
cd ..
```

Now do `ls` and `pwd`. See now that we went back up in to the home directory. `..` means go back up to the enclosing folder level.

Looking folder within folder within folder within...

Try entering:

```
cd data/hidden
```

and you will jump directly to *hidden* without having to go through the intermediate directory. Here, we're telling `cd` to go into *data* first, and then *hidden*.

Then do:

```
cd ../../
```

to go back up two levels. (Try typing `pwd` to see where you are!)

You could put more directories and a file on the end, too; for example,

```
ls data/hidden/tmp1/notit.txt
```

You can do the same thing with any UNIX command that takes a file or directory name.

Shortcut: Tab Completion

Navigate to the home directory. Typing out directory names can waste a lot of time. When you start typing out the name of a directory, then hit the tab key, the shell will try to fill in the rest of the directory name. For example, type `cd` to get back to your home directory, then enter:

```
cd da<tab>
```

The shell will fill in the rest of the directory name for *data*. Now `cd` to *data/IlluminaReads* and try:

```
ls Sl<tab><tab>
```

When you hit the first tab, nothing happens. The reason is that there are multiple directories in the home directory which start with *Sl*. Thus, the shell does not know which one to fill in. When you hit tab again, the shell will list the possible choices.

Tab completion can also fill in the names of programs. For example, enter `e<tab><tab>`. You will see the name of every program that starts with an *e*. One of those is `echo`. If you enter `ec<tab>` you will see that tab completion works.

Full vs. Relative Paths

The `cd` command takes an argument which is the directory name. Directories can be specified using either a **relative path** or a **full path**. The directories on the computer are arranged into a hierarchy. The full path tells you where a directory is in that hierarchy. Navigate to the home directory. Now, enter the `pwd` command and you should see:

```
/home/mtb_upm
```

which is the full name of your **home directory**. This tells you that you are in a directory called *mtb_upm*, which sits inside a directory called *home* which sits inside the very top directory in the hierarchy. The very top of the hierarchy is a directory called `/` which is usually referred to as the **root directory**. So, to summarize: *mtb_upm* is a directory in *home* which is a directory in `/`.

Now enter the following command:


```
cd /home/mtb_upm/data/hidden
```

This jumps to *hidden*. Now go back to the home directory (`cd`). We saw earlier that the command:

```
cd data/hidden
```

had the same effect - it took us to the *hidden* directory. But, instead of specifying the full path (*/home/mtb_upm/data*), we specified a **relative path**. In other words, we specified the path relative to our current directory. A full path always starts with a `/`. A **relative path** does not.

A **relative path** is like getting directions from someone on the street. They tell you to “go right at the Stop sign, and then turn left on Main Street”. That works great if you’re standing there together, but not so well if you’re trying to tell someone how to get there from another country. A full path is like GPS coordinates. It tells you exactly where something is no matter where you are right now.

You can usually use either a **full path** or a **relative path** depending on what is most convenient. If we are in the home directory, it is more convenient to just enter the relative path since it involves less typing.

Over time, it will become easier for you to keep a mental note of the structure of the directories that you are using and how to quickly navigate amongst them.

Saving time with shortcuts, wild cards, and tab completion

Shortcuts

There are some shortcuts which you should know about. Dealing with the home directory is very common. So, in the shell the **tilde character**, `~`, is a shortcut for your home directory. Navigate to the *data* directory:

```
cd
cd data
```

Then enter the command:

```
ls ~
```

This prints the contents of your home directory, without you having to type the full path. The shortcut `..` always refers to the directory above your **current directory**. Thus:

```
ls ..
```

prints the contents of the */home/mtb_upm* directory. You can **chain** these together, so:

```
ls ../../..
```

prints the contents of */home* which is above your home directory. Finally, the special directory `.` always refers to your **current directory**. So, `ls`, `ls ..`, and `ls ../../..` all do the same thing, they print the contents of the **current directory**. This may seem like a useless shortcut right now, but we’ll see when it is needed in a little while.

To summarize, while you are in the *data* directory, the commands `ls ~`, `ls ~/.`, `ls ../../..`, and `ls /home/mtb_upm` all do exactly the same thing. These shortcuts are not necessary, they are provided for your convenience.

Our dataset: FASTQ files

Assume that we whole-genome sequenced 9 bacterial genomes and generated paired-end Illumina reads. We get our data back from the sequencing center as FASTQ files, and we stick them all in a folder called *IlluminaReads*. We want to be able to look at these files and do some things with them.

Wild cards

Navigate to the *data/IlluminaReads* directory (hint: use `cd`). This directory contains our FASTQ files and some other ones we'll need for analyses. If we type `ls`, we will see that there are a bunch of files with long file names. Some of them end with *.fastq*.

The `*` character is a shortcut for “everything”. Thus, if you enter `ls *`, you will see all of the contents of a given directory. Now try this command:

```
ls *fastq
```

This lists every file that ends with a *.fastq*. This command:

```
ls /usr/bin/*.sh
```

Lists every file in */usr/bin* that ends in the characters *.sh*.

We have paired-end sequencing, so for every sample we have two files. If we want to just see the list of the files for the forward direction sequencing we can use:

```
ls *R1*fastq
```

lists every file in the **current directory** whose name contains the number *R1*, and ends with *.fastq*. There are 18 such files which we would expect because we have 9 samples.

So how does this actually work? Well...when the shell (we are using a particular flavor called **bash shell** for this tutorial) sees a word that contains the `*` character, it automatically looks for filenames that match the given pattern. In this case, it identified four such files. Then, it replaced the **R1*.fastq* with the list of files, separated by spaces.

What happens if you do `ls R1*fastq`?

When wildcards go bad

To discuss: Explain how to deal with filenames that being with - (use `'-'`), have spaces (use quotes/backslashes/tab completion), and/or quotes (use the other kind of quotes/backslashes/tab completion).

Examining Files

We now know how to switch directories, run programs, and look at the contents of directories, but how do we look at the contents of files?

The easiest way to examine a file is to just print out all of the contents using the program `cat`. Enter the following command:

```
cat S190_L001_R1_001.fastq
```

This prints out the contents of the *S190_L001_R1_001.fastq* file.

1. Print out the contents of the *~/data/IlluminaReads/upm.files* file. What does this file contain?
2. Without changing directories, (you should still be in *data*), use one short command to print the contents of all of the files in the */home/mtb_upm/data/IlluminaReads* directory.

Make sure we're in the right place for the next set of the lessons. We want to be in the *IlluminaReads* directory. Check if you're there with `pwd` and if not navigate there. One way to do that would be

```
cd ~/data/IlluminaReads
```

`cat` is a terrific program, but when the file is really big, it can be annoying to use. The program, `less`, is useful for this case. Enter the following command:

```
less S188_L001_R1_001.fastq
```

`less` opens the file, and lets you navigate through it. The commands are identical to the `man` program.

Some commands in “less”

key	action
“space”	to go forward
“b”	to go backwards
“g”	to go to the beginning
“G”	to go to the end
“q”	to quit

`less` also gives you a way of searching through files. Just hit the “/” key to begin a search. Enter the name of the word you would like to search for and hit enter. It will jump to the next location where that word is found. Try searching the *S188_L001_R1_001.fastq* file for “TACGGAGGATGC”. If you hit “/” then “enter”, `less` will just repeat the previous search. `less` searches from the current location and works its way forward. If you are at the end of the file and search for the word “cat”, `less` will not find it. You need to go to the beginning of the file and search.

For instance, let’s search for the sequence *1101:9376:4054* in our file. You can see that we go right to that sequence and can see what it looks like.

Remember, the `man` program actually uses `less` internally and therefore uses the same commands, so you can search documentation using “/” as well.

There’s another way that we can look at files, and in this case, just look at part of them. This can be particularly useful if we just want to see the beginning or end of the file, or see how it’s formatted.

The commands are `head` and `tail` and they just let you look at the beginning and end of a file respectively.

```
head S188_L001_R1_001.fastq
tail S188_L001_R1_001.fastq
```

The `-n` option to either of these commands can be used to print the first or last *n* lines of a file. To print the first/last line of the file use:

```
head -n 1 S188_L001_R1_001.fastq
tail -n 1 S188_L001_R1_001.fastq
```

Searching files

We showed a little how to search within a file using `less`. We can also search within files without even opening them, using `grep`. `Grep` is a command-line utility for searching plain-text data sets for lines matching a string or regular expression. Let’s give it a try.

Let’s search for that sequence *1101:9376:4054* in the *S188_L001_R1_001.fastq* file.

```
grep 1101:9376:4054 S188_L001_R1_001.fastq
```

We get back the whole line that had *1101:14341* in it. What if we wanted all four lines, the whole part of that FASTQ sequence, back instead.

```
grep -A 3 1101:9376:4054 S188_L001_R1_001.fastq
```

Command flags are options that we use to change the behaviour of a program. The **-A flag** stands for “after match” so it’s returning the line that matches plus the three after it. The **-B flag** returns that number of lines before the match.

Creating, moving, copying, and removing

Now we can move around in the file structure and look at files. But what if we want to do normal things like copy files or move them around or get rid of them. Sure we could do most of these things without the command line, but what fun would that be?! Besides it’s often faster to do it at the command line, or you’ll be on a remote server where you won’t have another option.

The *upm.files* file is one that tells us what sample name goes with what sequences. This is a really important file, so we want to make a copy so we don’t lose it.

Lets copy the file using the `cp` command. The `cp` command backs up the file. Navigate to the *IlluminaReads* directory and enter:

```
cp upm.files upm.files_backup
```

Now *upm.files_backup* has been created as a copy of *upm.files*.

Let’s make a *backup* directory where we can put this file.

The `mkdir` command is used to make a directory. Just enter `mkdir` followed by a space, then the directory name.

```
mkdir backup
```

We can now move our backed up file in to this directory. We can move files around using the command `mv`. Enter this command:

```
mv upm.files_backup backup/
```

This moves *upm.files_backup* into the directory *backup/*; the **full path** would be *~/data/IlluminaReads/backup*.

The `mv` command is also how you **rename** files. Since this file is so important, let’s **rename** it:

```
mv upm.files upm.files_IMPORTANT
```

Now the file name has been changed to *upm.files_IMPORTANT*. Let’s **delete** the backup file now:

```
rm backup/upm.files_backup
```

The `rm` file removes the file. Be careful with this command. It doesn’t just nicely put the files in the Trash. **THEY ARE REALLY GONE.**

By default, `rm`, will NOT **delete directories**. You can tell `rm` to delete a directory using the **-r option (flag)**; we could test it out on *backup*, but let’s not... ;)

Writing files

We've been able to do a lot of work with files that already exist, but what if we want to write our own files. Obviously, we're not going to type in a FASTQ file, but you'll see as we go through other tutorials, there are a lot of reasons we'll want to write a file, or edit an existing file.

To write in files, we're going to use the program `nano`. We're going to create a file that contains the favorite `grep` command so you can remember it for later. We'll name this file 'awesome.sh':

```
nano awesome.sh
```

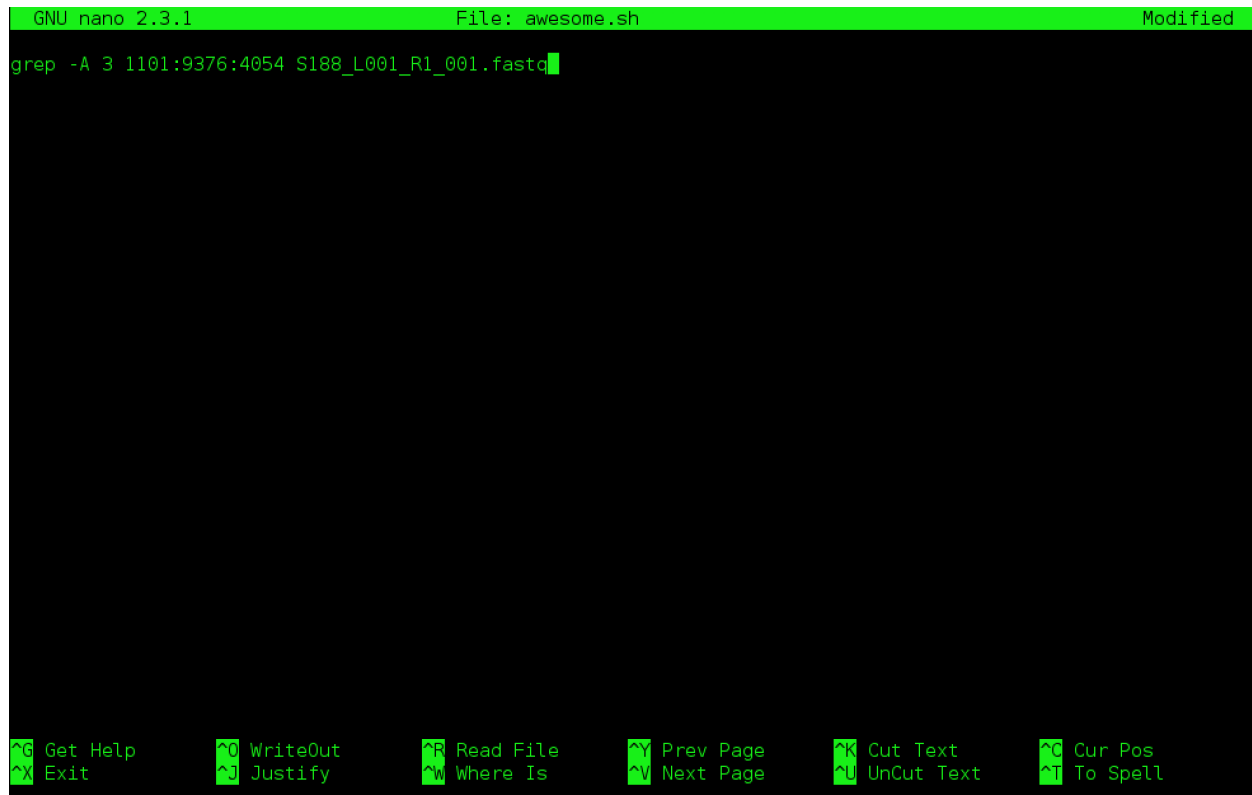
Now you have something that looks like



```
GNU nano 2.3.1 File: awesome.sh
[ New File ]
^G Get Help      ^O WriteOut      ^R Read File     ^Y Prev Page     ^K Cut Text      ^C Cur Pos
^X Exit          ^J Justify       ^W Where Is      ^V Next Page     ^U UnCut Text    ^T To Spell
```

Type in your command, so it looks like

```
GNU nano 2.3.1      File: awesome.sh      Modified
grep -A 3 1101:9376:4054 S188_L001_R1_001.fastq
```



```
^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Page  ^K Cut Text   ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is   ^V Next Page  ^U UnCut Text ^T To Spell
```

Now we want to save the file and exit. At the bottom of nano, you see the “^X Exit”. That means that we use `Ctrl-X` to exit. Type *Ctrl-X*. It will ask if you want to save it. Type *y* for yes. Then it asks if you want that file name. Hit *Enter*.

Now you’ve written a file. You can take a look at it with `less` or `cat`, or open it up again and edit it.

Exercise

Open *awesome.sh* and add “echo AWESOME!” after the `grep` command and save the file.

We’re going to come back and use this file in just a bit.

Running programs, revisited

Commands like `ls`, `rm`, `echo`, and `cd` are just ordinary programs on the computer. A program is just a file that you can **execute**. The program `which` tells you the location of a particular program. For example:

```
which ls
```

will return `/bin/ls`. Thus, we can see that `ls` is a program that sits inside of the `/bin` directory. Now enter:

```
which find
```

You will see that `find` is a program that sits inside of the `/usr/bin` directory.

So ... when we enter a program name, like `ls`, and hit enter, how does the shell know where to look for that program? How does it know to run `/bin/ls` when we enter `ls`. The answer is that when we enter a program name and hit enter, there are a few standard places that the shell automatically looks. If it can’t find the program in any of those places, it will print an error saying “command not found”. Enter the command:

```
echo $PATH
```

This will print out the value of the *PATH* **environment variable**. Notice that a list of directories, separated by colon characters, is listed. These are the places the shell looks for programs to run. If your program is not in this list, then an error is printed. The shell **ONLY** checks in the places listed in the *PATH* **environment variable**.

Navigate to the *data* directory and list the contents. You will notice that there is a program (executable file) called *hello.sh* in this directory. Now, try to run the program by entering:

```
hello.sh
```

You should get an error saying that *hello.sh* cannot be found. That is because the directory */home/mtb_upm/data* is not in the *PATH*. You can run the *hello.sh* program by entering:

```
./hello.sh
```

Remember that *.* is a shortcut for the current working directory. This tells the shell to run the *hello.sh* program which is located right here. So, you can run any program by entering the path to that program. You can run *hello.sh* equally well by specifying:

```
/home/mtb_upm/data/hello.sh
```

Or by entering:

```
~/data/hello.sh
```

When there are no */* characters, the shell assumes you want to look in one of the default places for the program.

(Why doesn't it look at your **current directory** by default? Any ideas?)

Writing scripts

We know how to write files and run scripts, so I bet you can guess where this is headed. We're going to run our own script.

Go in to the *IlluminaReads* directory where we created *awesome.sh* before. Remember we wrote our favorite *grep* command in there. Since we like it so much, we might want to run it again, or even all the time. Instead of writing it out every time, we can just run it as a script.

It's a command, so we should just be able to run it. Give it try.:

```
./awesome.sh
```

Alas, we get `-bash: ./awesome.sh: Permission denied`. This is because we haven't told the computer that it's a program. To do that we have to **make it executable**. We do this by **changing its mode**. The command for that is *chmod* - change mode. We're going to change the mode of this file, so that it's **executable** and the computer knows it's OK to run it as a program.:

```
chmod +x awesome.sh
```

Now let's try running it again:

```
./awesome.sh
```

Now you should have seen some output.

Congratulations, you just created your first shell script!

Challenge:

write a script that:

- resides in the data directory;
 - changes to the IlluminaReads/ subdirectory of the current working directory;
 - makes two subdirectories, “left” and “right”;
 - moves all of the R1 sequencing files into the left directory;
 - moves all of the R2 sequencing files into the right directory;
-

WE ARE DONE!

More resources

- Software Carpentry tutorial - [The Unix shell](#)
- The shell handout - [Command Reference](#)
- [explainshell.com](#)
- <http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html>
- `man bash`
- Google - if you don't know how to do something, try Googling it. Other people have probably had the same question.

Most importantly - learn by doing. There's no real other way to learn this than by trying it out. Open pdfs from the command line, automate something you don't really need to automate...

Some books you should look into –

1. [Practical Computing for Biologists](#)
2. [Bioinformatics Data Skills](#)

3.1.2 Advanced Beginner/Intermediate Shell

Updated June 2018 by [Ulas Karaoz](#)

Credits

Original authors: *Jessica Mizzi, Titus Brown, and Lisa K. Johnson* at Lab for Data Intensive Biology (<http://dib-training.readthedocs.io/en/pub/2016-01-13-adv-beg-shell.html>)

Learning objectives:

- Expose you to a bunch of syntax around shell use & scripting.
- Show you the proximal possibilities of shell use & scripting.
- Give you some useful tricks
- Provide opportunity for discussion, so please ask questions!

Important: Before starting:

1. Make sure you have the test data. Download and unpack:

```
wget https://s3-ap-southeast-1.amazonaws.com/mtb-bioinformatics-workshop-singapore/
↪shell-data.zip
# will unzip a folder called `data`
unzip shell-data.zip
```

2. Set your current working directory to be the top level dir:

```
cd ~/data/
```

3. We'll be posting code snippets to: https://public.etherpad-mozilla.org/p/mtb_bioinformatics_shell
 4. In this tutorial, commands are shown as I am a command: you can type me in your shell terminal.
-

Exploring directory structures

So, if we do `ls`, we see a bunch of stuff. We didn't create this folder. How do we figure out what's in it?

Here, `find` is your first friend:

```
find . -type d
```

This walks systematically (recursively) through all files underneath `.`, finds all directories (*type d*), and prints them (assumed, if not other actions).

We'll come back to `find` later, when we use it for finding files.

Renaming a bunch of files

Let's go into the *IlluminaReads* directory:

```
cd IlluminaReads
```

and take a look with `ls`.

For our first task, let's pretend that we want to change the extension of all of the fastq files from `*.fastq` to `*.fq`. Here, we get to use two commands - `for` and `basename`.

`for` lets you do something to every file in a list. To see it in action:

```
for i in *.fastq
do
    echo $i
done
```

This is running the command `echo` for every value of the variable `i`, which is set (one by one) to all the values in the expression `*.fastq`.

If we want to get rid of the extension `.fastq`, we can use the `basename` command:

```
for i in *.fastq
do
    basename $i .fastq
done
```

Now, this doesn't actually rename the files - it just prints out the name, with the suffix `.fastq` removed. To rename the files, we need to capture the new name in a variable:

```
for i in *.fastq
do
    newname=$(basename $i .fastq).fq
    echo $newname
done
```

What `$(...)` does is run the command in the middle, and then replace the `$()` with the value of running the command.

Now we have the old name (`$i`) and the new name (`$newname`) and we're ready to write the rename command, which is `mv`:

```
for i in *.fastq
do
    newname=$(basename $i .fastq).fq
    echo mv $i $newname
done
```

Question: why did we use 'echo' here?

Now that we're pretty sure it all looks good, let's run it for real:

```
for i in *.fastq
do
    newname=$(basename $i .fastq).fq
    mv $i $newname
done
```

We have renamed all the files!

Let's also get rid of the annoying `_001` that's at the end of the files. `basename` is all fine and good with the end of files, but what do we do about things in the middle? Now we get to use another useful command, `cut`.

What `cut` does is slide and dice strings. So, for example,

```
echo hello, world | cut -c5-
```

will give you `o, world`.

`cut` expects to take a bunch of lines of input from a file. By default it is happy to take them in from **stdin** ("standard input"), so you can specify `-` and give it some input via a pipe, which is what we're doing with `echo`:

We're taking the output of `echo hello, world` and sending it to the input of `cut` with the `|` command (**pipe**).

You've probably already seen this with `head` or `tail`, but many UNIX commands take **stdin** and **stdout**.

Let's construct the `cut` command we want to use. If we look at the names of the files, and we want to remove *001* only, we can see that each filename has a bunch of **fields** separated by `_`. So we can ask `cut` to pay attention to the first four **fields**, and omit the fifth, around the **separator** (or **delimiter**) `_`:

```
echo S194_L001_R1_001.fq | cut -d_ -f1-4
```

That looks about right, let's put it into a **for loop**:

```
for i in *.fq
do
    echo $i | cut -d_ -f1-4
done
```

Good, now assign it to a **variable** and **append an ending**:

```
for i in *.fq
do
    newname=$(echo $i | cut -d_ -f1-4).fq
    echo $newname
done
```

and now construct the `mv` command:

```
for i in *.fq
do
    newname=$(echo $i | cut -d_ -f1-4).fq
    echo mv $i $newname
done
```

and if that looks right, run it:

```
for i in *.fq
do
    newname=$(echo $i | cut -d_ -f1-4).fq
    mv $i $newname
done
```

You've renamed all your files.

Let's do something quite useful - subset a bunch of FASTQ files.

If you look at one of the FASTQ files with `head`,

```
head S188_L001_R1.fq
```

you'll see that it's full of FASTQ sequencing records. Often we want to run a bioinformatics pipeline on some small set of records first, before running it on the full set, just to make sure all the commands work. So we would like to subset all of these files without modifying the originals.

First, let's make sure the originals are read-only:

```
chmod u-w *.fq
```

Now, let's make a *subset* directory:

```
mkdir subset
```

Now, to subset each file, we want to run a `head` with an argument that is the total number of lines we want to take. In this case, it should be a multiple of 4, because FASTQ records have 4 lines each. So let's plan to take the first 100 lines of each file by using `head -400`.

The **for loop** will now look something like:

```
for i in *.fq
do
    echo "head -400 $i > subset/$i"
done
```

If that command looks right, run it for real:

```
for i in *.fq
do
    head -400 $i > subset/$i
done
```

We now have our subsets.

— Exercise: Can you rename all of your files in `subset/` to have *subset.fq* at the end?

(Work in small groups; start from working code; there are several ways to do it, all that matters is getting there.)

Backtracking

Variables:

You can use either `$varname` or `${varname}`. The latter is useful when you want to construct a new filename, e.g.:

```
MY${varname}SUBSET
```

would expand `${varname}` and then put `MY .. SUBSET` on either end, while

```
MY$varnameSUBSET
```

would try to put `MY` in front of `$varnameSUBSET` which won't work.

(Unknown/uncreated variables give nothing.)

(Variables are interpreted inside of `""`, and not inside of `'`.)

Pipes and redirection:

To **redirect stdin** and **stdout**, you can use:

```
> - send stdout to a file
< - take stdin from a file
| - take stdout from first command and make it stdin for second command
>> - appends stdout to a previously-existing file
```

stderr (for *errors*) can be redirected:

```
2> - send stderr to a file
```

and you can also say:

```
>& - to send all output to a file
```

Editing on the command line:

Most prompts support *readline-style* editing. This uses `emacs` control keys (*emacs* is a very powerful editor, tough with a bit of steep learning curve)

Type something out; then type *CTRL-a*. Now type *CTRL-e*. Beginning and end...

Up arrows to recall previous command, left/right arrows, etc.

Another useful command along with `basename` is `dirname`. Any idea what it does?

Working with collections of files; conditionals

Let's go back to the *data* directory and play around with loops some more.

```
cd ..
```

if acts on things **conditionally**:

```
for i in *
do
    if [ -f $i ]; then
        echo $i is a file
    elif [ -d $i ]; then
        echo $i is a directory
    fi
done
```

but what is the `[]` notation? That's actually running the `test` command; try `help test | less` to see the docs. This is a

syntax that lets you do all sorts of useful things with files –

We can use it to get rid of empty files:

```
touch emptyfile.txt
```

to create an empty file, and then:

```
for i in *
do
    if [ \! -s $i ]; then
        echo rm $i
    fi
done
```

...and as you can see here, we are using `!` to say 'not'.

Executing things conditionally based on exit status

Let's create two scripts (you can use `nano` here if you want) – in `success.sh`, put:

```
#!/bin/bash
echo mesucceed
exit 0
```

and in `fail.sh`, put:

```
#!/bin/bash
echo mefail
exit 1
```

You can do this with *here documents*. A *here document* (or *heredoc*) is a way of getting text input into a script without having to feed it in from a separate file.

```
cat > success.sh <<EOF
#!/bin/bash
echo mesucceed
exit 0
EOF
cat > fail.sh <<EOF
#!/bin/bash
echo mefail
exit 1
EOF
```

Now make them executable:

```
chmod +x success.sh fail.sh
```

(Somewhat counterintuitively, an exit status of 0 means “success” in UNIX.)

You can now use this to chain commands with `&&` and `||`:

```
./success.sh && echo this succeeded || echo this failed
./fail.sh && echo this succeeded || echo this failed
```

You can do this with R and python scripts too. In R, you set the exit status of a script with `quit(status=0, save='no')` and in Python with `sys.exit(0)`. Any failure of the script due to an exception will automatically set the exit status to non-zero.

The exit status of the previous command can be examined with `$?`:

```
./success.sh
if [ $? -eq 0 ]; then echo succ; fi

./success.sh
if [ $? -ne 0 ]; then echo fail; fi
```

Writing shell scripts

Always put `set -e` at the top.

Sometimes put `set -x` at the top.

You can take in **command line parameters** with `$1`, `$2`, etc. `$*` gives you all of them at once.

Other bits and pieces

- You can use to do line continuation in scripts (in R and Python, too!)
- Scripts exit in a subshell and can't modify your environment variables.

If you want to modify your environment, you need to use `.` or *source*.

- Subshells are ways to group commands with `(...)`.
- `screen` lets you use multiple windows at the same time. It is extremely useful.
- History tricks:

```
!! - run previous command
!-1 - run command-before-previous command (!-2 etc.)
!$ - replace with the last word on the previous line
!n - run the nth command in your 'history'
```

A general approach you can use

- Break your task down into multiple commands
- Put commands in shell scripts, run in serial
- Use intermediate i/o files to figure out what's going on!
- Use echo to debug!

More things you can do with `find`

`find` can be extremely flexible and powerful. Here is more useful things you can do with `find` command:

Print all files:

```
find . -type f
```

Print all files w/details:

```
find . -type f -ls
```

Find all directories in the current directory:

```
find * -prune -type d -print
```

...and get their disk usage:

```
find * -prune -type d -exec du -skh {} \;
```

Here, `-exec` runs the command specified up until the `;`, and replaces the `{}` with the filename.

Same result, different command:

```
find . -depth 1 -type d -exec du -skh {} \;
```

Find all files larger than 100k:

```
find . -size +100k -print
```

Find all files that were changed within the last 10 minutes:

```
find . -ctime -10m
```

(...and do things to them with `-exec` ;).

Run `grep -l` to find all files containing the string 'CGTTATCCGGATTATTGGGTTTA':

```
find . -type f -exec grep -q CGTTATCCGGATTATTGGGTTTA {} \; -print
```

Note, you can use `-a` (and) and `-o` (or), along with `(` and `)`, to group conditions:

```
find . \( \( -type f -size +100k \) -o \( -type f -size -1k \) \) -print
```

Note that you can `exec` a Python, R, or shell script.

Exercise:

How would you copy all files containing a specific string ('CGTTATCCGGATTATTGGGTTTA', say) into a new directory? And what are the pros (and cons) of your approach?

(Work in small groups; start from working code, say, the 'find' command above; there are several ways to do it, all that matters is getting there.)

3.2 Day 2: Microbial Genome Assembly (from short and long reads) and Genome Annotation

In this lab we will perform de novo bacterial genome assembly. We will also annotate the assembled genome. Although we will specifically be assembling a *Mycobacteria tuberculosis* genome, the methodology presented here applies to any bacterial genome. You will be guided through the genome assembly starting with data quality control, through to building contigs and analysis of the results. At the end of the lab you will know:

1. How to perform basic quality checks on the read data
2. How to run a short read assembler on Illumina short reads
3. How to run a long read assembler on Pacific Biosciences long reads
4. How to improve the accuracy of a long read assembly using short reads
5. How to assess the quality of an assembly
6. How to annotate a microbial genome

3.2.1 Data Sets

The datasets we will use are sequencing reads from *M. tuberculosis* genome generated using Illumina and PacBio platforms. For the Illumina platform, there are two sets of reads covering the same library at 20X and 60X coverage. We will also use a reference assembly for *M. tuberculosis* H37Rv (NC_000962.3). H37Rv genome is 4411532 bp long.

The datasets are located under `/home/mtb_upm/mtb_genomics_workshop_data/genome-assembly-annotation/data` directory, which has the following content:

File	Content
MTB_illumina60X_R1.fastq	60X coverage run, Illumina short reads, forward
MTB_illumina60X_R2.fastq	60X coverage run, Illumina short reads, forward
MTB_illumina20X_R1.fastq	20X coverage run, Illumina short reads, forward
MTB_illumina20X_R2.fastq	20X coverage run, Illumina short reads, forward
MTB_pacbio_circular-consensus-sequence-reads.fastq	PacBio long reads, circular consensus sequence
NC_000962.3.fna	Mycobacterium tuberculosis H37Rv reference assembly, 4411532 bp, linear

3.2.2 Copying data and preparing the directories

First let's create a directory named `wgs_assembly` for this practical. And then, under that directory create subdirectories to hold our data and results:

```
mkdir ~/wgs_assembly
mkdir -p ~/wgs_assembly/data
mkdir -p ~/wgs_assembly/results
mkdir -p ~/wgs_assembly/results/qc
mkdir -p ~/wgs_assembly/results/short_long_reads_assembly
```

Now, copy the data into our data directory:

```
cp -r /home/mtb_upm/mtb_genomics_workshop_data/genome-assembly-annotation/data/* ~/wgs_assembly/data
```

Define variables for convenience:

```
MYDATADIR=~/wgs_assembly/data
MYRESULTSDIR=~/wgs_assembly/results
```

3.2.3 Read Quality Control and Trimming

To perform data quality control, we will use `FastQC`, a tool that processes reads and presents visual reports.

First, peak into the first few lines of `.fastq` files for the Illumina reads from 60X sequencing run

```
head -n 10 $MYDATADIR/MTB_illumina60X_R1.fastq
head -n 10 $MYDATADIR/MTB_illumina60X_R2.fastq
```

What can you tell about the headers? What is the read length for this run? Can you quickly count the number of reads (Hint: use `grep` and `wc`)?

Running `fastqc`

We will now run `fastqc`. For most tools used in practicals, you can get help by using `-h` or `-help` flag. Type the following to get help for `fastqc`:

```
fastqc -h
```

Run `fastqc` for forward reads from 60X and 20X runs saving the results under your results directory:

```
fastqc -o $MYRESULTSDIR/qc $MYDATADIR/MTB_illumina60X_R1.fastq
fastqc -o $MYRESULTSDIR/qc $MYDATADIR/MTB_illumina20X_R1.fastq
```

Now run it again for reverse reads:

```
fastqc -o $MYRESULTSDIR/qc $MYDATADIR/MTB_illumina60X_R2.fastq
fastqc -o $MYRESULTSDIR/qc $MYDATADIR/MTB_illumina20X_R2.fastq
```

Repeat for PacBio reads.

```
fastqc -o $MYRESULTSDIR/qc $MYDATADIR/MTB_pacbio_subreads.fastq
fastqc -o $MYRESULTSDIR/qc $MYDATADIR/MTB_pacbio_circular-consensus-sequences.
↪fastq
```

`fastqc` performs a suite of tests and then for each indicates whether the test has been passed (green tick) or failed (red cross). Notice though that `fastqc` has no idea about what your library should look like. All of its tests are based on a completely random library with 50% GC content and therefore it will show the test as failed if your sample doesn't match these assumptions. For instance, you might have reads sampled from a high AT or high GC genome, and it will fail per sequence GC content. You need to be aware of how your library been generated and interpret the `fastqc` reports taking that into account.

Question: For a whole genome shotgun library, what would you expect “per base sequence content” to look like?
Question: What if you had amplicon sequences from a multiplex PCR assay?

Now, open `fastqc` reports in a browser:

```
firefox $MYRESULTSDIR/qc/MTB_illumina60X_R1_fastqc.html
firefox $MYRESULTSDIR/qc/MTB_illumina60X_R2_fastqc.html
firefox $MYRESULTSDIR/qc/MTB_illumina20X_R1_fastqc.html
firefox $MYRESULTSDIR/qc/MTB_illumina20X_R2_fastqc.html
firefox $MYRESULTSDIR/qc/MTB_pacbio_R1_fastqc.html
firefox $MYRESULTSDIR/qc/MTB_pacbio_R2_fastqc.html
```

- **Per base sequence quality:** This is probably one of the most important diagnostics. Inspecting this, you can determine what quality trimming parameters you should use in downstream analysis.
- **Per base sequence content:** For a randomly sampled genome with a GC content of 50%, we would expect that at any given position within a read, there will be an equal probability of finding a A,C,G, or T base. Our library from *M. tuberculosis* is expected to reflect the GC content, which holds here. There seems to be some minor bias at the start of the read, which may be due to PCR duplicates during amplification or during library preparation.
- **Sequence duplication levels:** In a whole genome shotgun library, very few reads are expected to occur more than once. In a shotgun library of a targeted region, a low level of duplication can simply be the result of high level of sampling (i.e. high coverage). In addition, duplication might also indicate some kind of enrichment bias (i.e. PCR overamplification)

<https://sequencing.qcfail.com/> is a good resource where people post about different ways a sequencing run might have failed.

Running `sickle` to quality trim reads:

Now that we have an idea about the quality profile of our reads, we can do quality based trimming of reads. We will use a `Q-score` threshold of 30 (`-q 30`), remove all reads that are shorter than 100 bp after trimming (`-l 100`), and also trim 10 bp from 5' end just to reduce bias. After trimming, singletons reads (reads whose pair dropped off) are save into a separate file:

```
sickle pe -t sanger -q 30 -l 100 -n 10 \
-f $MYDATADIR/MTB_illumina60X_R1.fastq \
-r $MYDATADIR/MTB_illumina60X_R2.fastq \
-o $MYRESULTSDIR/qc/MTB_illumina60X_qced_R1.fastq \
-p $MYRESULTSDIR/qc/MTB_illumina60X_qced_R2.fastq \
-s $MYRESULTSDIR/qc/MTB_illumina60X_qced_single.fastq 1>$MYRESULTSDIR/qc/MTB_
↪illumina60X_sickle.log

sickle pe -t sanger -q 30 -l 100 -n 10 \
-f $MYDATADIR/MTB_illumina20X_R1.fastq \
-r $MYDATADIR/MTB_illumina20X_R2.fastq \
-o $MYRESULTSDIR/qc/MTB_illumina20X_qced_R1.fastq \
-p $MYRESULTSDIR/qc/MTB_illumina20X_qced_R2.fastq \
-s $MYRESULTSDIR/qc/MTB_illumina20X_qced_single.fastq 1>$MYRESULTSDIR/qc/MTB_
↪illumina20X_sickle.log
```

3.2.4 Pre-assembly Quality Control

Before running `sga preqc`, we need to *interleave forward* and *reverse* reads into a single file. *Interleaving* reads mean rearranging *forward* and *reverse* reads into a single file, keeping the original order and for each pair writing *forward* read followed by the *reverse* read.

To interleave reads, we will use `seqtk` tool:

```
seqtk mergepe $MYRESULTSDIR/qc/MTB_illumina20X_qced_R1.fastq $MYRESULTSDIR/qc/MTB_
↪illumina20X_qced_R2.fastq > $MYRESULTSDIR/qc/MTB_illumina20X_qced_R1R2.fastq
seqtk mergepe $MYRESULTSDIR/qc/MTB_illumina60X_qced_R1.fastq $MYRESULTSDIR/qc/MTB_
↪illumina60X_qced_R2.fastq > $MYRESULTSDIR/qc/MTB_illumina60X_qced_R1R2.fastq
```

The `sga` computations will take several hours on a single core for the 60X library. To save time, we won't be doing these computations but simply copy the results from the precalculated directory. Here is how to do these computations (**DONOT RUN**).

First, index the data using `sga index`:

```
sga index -t 4 -a ropebwt $MYRESULTSDIR/qc/MTB_illumina20X_qced_R1R2.fastq
sga index -t 4 -a ropebwt $MYRESULTSDIR/qc/MTB_illumina60X_qced_R1R2.fastq
```

Run `sga preqc`:

```
sga preqc -t 4 $MYRESULTSDIR/qc/MTB_illumina20X_qced_R1R2.fastq > $MYRESULTSDIR/qc/
↪MTB_illumina20X_qced_R1R2.preqc
sga preqc -t 4 $MYRESULTSDIR/qc/MTB_illumina60X_qced_R1R2.fastq > $MYRESULTSDIR/qc/
↪MTB_illumina60X_qced_R1R2.preqc
```

Copy the precalculated results:

```
cp /home/mtb_upm/mtb_genomics_workshop_data/genome-assembly-annotation/precalculated/
↪MTB_illumina20X_qced_R1R2.preqc $MYRESULTSDIR/qc
cp /home/mtb_upm/mtb_genomics_workshop_data/genome-assembly-annotation/precalculated/
↪MTB_illumina60X_qced_R1R2.preqc $MYRESULTSDIR/qc
```

Make the report:

```
sga-preqc-report.py -P genome_size \
-P branch_classification_error -P branch_classification_repeat -P branch_
↪classification_variant \
```

(continues on next page)

(continued from previous page)

```
-P de_bruijn_simulation_lengths -P kmer_distribution -P gc_distribution -P fragment_
↪ sizes \
MTB_illumina20X_qced_R1R2.preqc MTB_illumina60X_qced_R1R2.preqc -o sga-preqc-report.
↪ pdf
```

Open the PDF report and interpret the results.

Questions: * Was the genome size estimated correctly? * What differences do you observe between the 20x (blue) and the 60X (red) datasets? * Which library will be easier to assemble?

3.2.5 Assembly

We can now proceed with the assembly of short and long reads. We will assemble each library (2 short and 1 long reads library) separately and compare the assemblies.

Assembly using Illumina short reads

We will assemble the *M. tuberculosis* genome using the *qced* short reads, separately for 20X and 60X libraries. We will be using SPAdes assembler, which is a de Bruijn graph based short read assembler.

The parameters that go into a short read assembler can be tricky to optimize and tuning them is quite time consuming. The most important parameter is the k-mer size to be used to build the de Bruijn graph. SPAdes makes the parameterization quite easy and will select parameter values automatically.

The assembly for the 20X library will be computed quickly but for the 60X library the computation will take a few hours. **ONLY RUN for 20X, DONOT RUN for 60X.** For 60X, we will use precalculated results.

You can run SPAdes as follows using 4 threads (-t 4):

```
spades.py -o $MYRESULTSDIR/short_long_reads_assembly/MTB_illumina20X_spades -t 4 --12
↪ $MYRESULTSDIR/qc/MTB_illumina20X_qced_R1R2.fastq
spades.py -o $MYRESULTSDIR/short_long_reads_assembly/MTB_illumina60X_spades -t 4 --12
↪ $MYRESULTSDIR/qc/MTB_illumina60X_qced_R1R2.fastq
```

Copy the precalculated assembly directory:

```
cp -r ~/mtb_genomics_workshop_data/genome-assembly-annotation/precalculated/MTB_
↪ illumina60X_spades $MYRESULTSDIR/short_long_reads_assembly/MTB_illumina60X_spades
```

Assembly using PacBio long reads

Now, we will generate an assembly of the *M. tuberculosis* genome using PacBio long reads. Longer reads are better at resolving repeats and therefore will typically give more contiguous assemblies compared to short reads. However, long reads have a significantly higher error rate so consensus base qualities are expected to be much lower.

To assemble long reads, we will use *canu*, an overlap-layour-consensus assembler. Generate the assembly with the following command (this will take about ~30 min.). Again, we will use precomputed results.

```
canu gnuPlotTested=true -p MTB_pacbio_canu -d MTB_pacbio_canu_auto genomeSize=4.4m -
↪ pacbio-raw MTB_pacbio_circular-consensus-sequence-reads.fastq
```

Copy the precomputed assembly: cp -r ~/mtb_genomics_workshop_data/genome-assembly-annotation/precalculated/MTB_pacbio_canu_auto \$MYRESULTSDIR/short_long_reads_assembly/MTB_pacbio_canu_auto

Now you should have 3 separate *M. tuberculosis* assemblies. Copy the fasta files for the assembled contigs into a separate directory for convenience:

```
mkdir $MYRESULTSDIR/short_long_reads_assembly/assemblies
cp $MYRESULTSDIR/short_long_reads_assembly/MTB_pacbio_canu_auto/MTB_pacbio_canu.
↪contigs.fasta $MYRESULTSDIR/short_long_reads_assembly/assemblies/MTB_pacbio_
↪assembly.fasta
cp $MYRESULTSDIR/short_long_reads_assembly/MTB_illumina20X_spades/contigs.fasta
↪$MYRESULTSDIR/short_long_reads_assembly/assemblies/MTB_illumina20X_assembly.fasta
cp $MYRESULTSDIR/short_long_reads_assembly/MTB_illumina60X_spades/contigs.fasta
↪$MYRESULTSDIR/short_long_reads_assembly/assemblies/MTB_illumina60X_assembly.fasta
```

Next, we will look into these assemblies in detail.

3.2.6 Assessing the quality of the assemblies

We will now start assessing the quality of our assembly. Overall, we want an assembly that is in as few contigs as possible, as complete as possible (with respect to a reference), and as accurate as possible (ideally at single base pair resolution). So, when evaluating an assembly, there are 3 factors to consider listed below. Usually, in any assembly strategy there are tradeoffs between these factors.

- **Contiguity:** Long contigs are better than short contigs. Long contigs provide more information about the genomic structure (for instance gene order, synteny)
- **Completeness:** We want the assembly to represent as much of the true genome as possible. The assumed genome size in our case is ~4.4 Mb
- **Accuracy:** We want an assembly that has few large-scale misassemblies and consensus errors (mismatches or indels). The latter is especially critical for identification of nucleotide variants.

To evaluate assemblies, we will use *quast* (Quality Assessment Tool for Genome Assemblies). In this case, we have a high quality reference assembly we can use for groundtruthing. Typically, there will be a lot of short “leftover” contigs that contain repetitive or low-complexity sequence or low-quality reads that cannot be assembled. To avoid biasing our evaluation, we will only use contigs that are at least 500 bp long (default threshold in *quast*). You can *quast* as follows on the 3 assemblies:

DO NOT RUN. We will use precomputed results.

```
# define a variable for the reference genome
REF_GENOME=/home/mtb_upm/mtb_genomics_workshop_data/genome-assembly-annotation/data/
↪NC_000962.3.fna
quast.py -R $REF_GENOME -o $MYRESULTSDIR/short_long_reads_assembly/assemblies/quast_
↪3assemblies_qc \
$MYRESULTSDIR/short_long_reads_assembly/assemblies/MTB_illumina20X_assembly.fasta \
$MYRESULTSDIR/short_long_reads_assembly/assemblies/MTB_illumina60X_assembly.fasta \
$MYRESULTSDIR/short_long_reads_assembly/assemblies/MTB_pacbio_assembly.fasta
```

Copy the precomputed results:

```
cp -r ~/mtb_genomics_workshop_data/genome-assembly-annotation/precalculated/
↪assemblies/quast_3assemblies_qc/ $MYRESULTSDIR/short_long_reads_assembly/assemblies/
```

In *quast_3assemblies_qc* directory, open *report.html* file in a browser.

```
:: firefox $MYRESULTSDIR/short_long_reads_assembly/assemblies/quast_3assemblies_qc/report.html&
```

When you opened the results web page, put your green stickies on. We will discuss these results in detail.

3.2.7 Improving the accuracy of the long read assembly

In the lecture, we have seen that PacBio long reads have a significantly higher error rate compared to Illumina short reads. As a result, the long read assembly will have many errors in the consensus sequence (primarily indels). We can increase the accuracy of the consensus sequences using the Illumina reads, which will provide more coverage and higher quality bases. This process of improving the accuracy of the long reads using high fidelity short reads is sometimes referred to as *polishing*.

To polish PacBio assembly, we will use `pilon` which will generate a new consensus sequence for the assembly. The details of the following commands will be clear in the next lecture. For now, run the following commands:

```
mkdir $MYRESULTSDIR/short_long_reads_assembly/polished_pacbio_assembly
PACBIO_ASSEMBLY=$MYRESULTSDIR/short_long_reads_assembly/assemblies/MTB_pacbio_
↪assembly.fasta
cd $MYRESULTSDIR/short_long_reads_assembly/
bwa index $PACBIO_ASSEMBLY
bwa mem -t 4 -p $PACBIO_ASSEMBLY $MYRESULTSDIR/qc/MTB_illumina60X_qced_R1R2.fastq |_
↪samtools sort -T tmp -o $MYRESULTSDIR/short_long_reads_assembly/polished_pacbio_
↪assembly/MTB_illumina60X_qced.mapped_to_pacbio_contigs.sorted.bam -
samtools index $MYRESULTSDIR/short_long_reads_assembly/polished_pacbio_assembly/MTB_
↪illumina60X_qced.mapped_to_pacbio_contigs.sorted.bam
java -Xmx8G -jar /opt/pilon-1.22.jar --genome $PACBIO_ASSEMBLY --frags $MYRESULTSDIR/
↪short_long_reads_assembly/polished_pacbio_assembly/MTB_illumina60X_qced.mapped_to_
↪pacbio_contigs.sorted.bam --output $MYRESULTSDIR/short_long_reads_assembly/polished_
↪pacbio_assembly/MTB_pacbio_assembly_shortreadcorrected
```

Do assembly qc with 4 assemblies. **DONOT RUN**, we will use precomputed results.

```
quast.py -R ../NC_000962.3.fna -o quast_4assemblies_qc ./MTB_illumina20X_assembly.
↪fasta ./MTB_illumina60X_assembly.fasta ./MTB_pacbio_assembly.fasta ./MTB_pacbio_
↪assembly_shortreadcorrected.fasta
```

Copy precomputed results:

```
cp -r ~/mtb_genomics_workshop_data/genome-assembly-annotation/precalculated/
↪assemblies/quast_4assemblies_qc/ $MYRESULTSDIR/short_long_reads_assembly/assemblies/
```

In `quast_4assemblies_qc` directory, open `report.html` file in a browser. After you open, make sure you turn on the *extended report*.

```
:: firefox $MYRESULTSDIR/short_long_reads_assembly/assemblies/quast_4assemblies_qc/report.html&
```

When you opened the results web page, put your green stickies on. We will discuss these results in detail.

3.2.8 Microbial Genome Annotation

Now that we have assembled the reads into contigs, we will annotate the genome. That means we will predict genes and assign functional annotation to them. *M. tuberculosis* is a very clonal organism and its genome is already well annotated so typically when we assemble a new *M. tuberculosis* isolate, there is no need to do *de novo* annotation. Although we are using *M. tuberculosis* as a working example here, the methodology is applicable to all microbial genomes.

A set of programs are available for genome annotation tasks. Here we will use `Prokka`, a pipeline comprising several open source bioinformatic tools and databases. `Prokka` automates the process of locating open reading frames (ORFs) and RNA regions on contigs, translating ORFs to protein sequences, searching for protein homologs and producing standard output files. For gene finding and translation, `Prokka` uses a program named `prodigal`. Homology

searches are then performed using BLAST and HMMER based on the translated protein sequences as queries against a set of public databases (CDD, PFAM, TIGRFAM) as well as custom databases that are included with Prokka.

Create a directory to hold the annotation results:

```
mkdir -p $MYRESULTSDIR/annotation
```

Define a shell variable for the final contigs:

```
FINALCONTIGS=$MYRESULTSDIR/short_long_reads_assembly/polished_pacbio_assembly/MTB_
↳pacbio_assembly_shortreadcorrected.fasta
```

Run Prokka (will take ~30 minutes for our assembly). **DONOT RUN**, copy the precalculated results.

```
prokka $FINALCONTIGS --outdir $MYRESULTSDIR/annotation/MTB_pacbio_assembly_
↳shortreadcorrected --norrna --notrna --metagenome --cpus 8
```

Prokka produces several types of output:

- a .gff file, which is a standardised, tab-delimited, format for genome annotations
- a Genbank (.gbk) file, which is a more detailed description of nucleotide sequences and the genes encoded in these.

Copy the precalculated results:

```
mkdir -p $MYRESULTSDIR/annotation/MTB_pacbio_assembly_shortreadcorrected/
cp ~/mtb_genomics_workshop_data/genome-assembly-annotation/precalculated/prokka_
↳precalculated/* $MYRESULTSDIR/annotation/MTB_pacbio_assembly_shortreadcorrected/
```

When your dataset has been annotated you can view the annotations directly in the gff file. Peek into the gff file by doing:

```
more -S $MYRESULTSDIR/annotation/MTB_pacbio_assembly_shortreadcorrected/PROKKA_
↳07052018.gff
```

Question: How many coding regions were found by Prodigal? (Hint: use grep) Question: How many of the coding regions were given an enzyme identifier? Question: How many were given a COG identifier?

Visualizing the annotations with Artemis

Artemis is a graphical Java program to browse annotated genomes. Launch Artemis and load gff file:

```
art&
```

Pick one of favorite *M. tuberculosis* genes, and try finding it (Goto→Navigator→Goto Feature with Key Name)

3.3 Day 3: Read mapping and variant calling

In this lab we will map short reads against the reference *Mycobacterium tuberculosis* genome (H37Rv) and call variants based on the mapped reads. This pipeline is sometimes also called “reference based assembly” since you can use the called variants to generate a new genomic sequence.

At the end of the lab you should:

1. know how to index the reference genome and map short reads

2. be familiar with different file formats (.bam, .sam, .vcf)
3. how to query and manipulate .bam files
4. how to assess mapping quality
5. how to deduplicate the alignments and how to do indel realignment
6. how to filter by mapping quality
7. how to call variants

3.3.1 Dataset

The dataset we will be using for today's and tomorrow's practicals is a subset of 1000 *M. tuberculosis* isolates from a Russian population. The results of this study was published by Casali et al.: [Evolution and transmission of drug-resistant tuberculosis in a Russian population](#). These isolates have been whole-genome shotgun sequenced using Illumina platform (2x100 bp). Raw read data has been deposited to European Nucleotide Archive under Study Accession [PRJEB2138](#). For this tutorial, we will focus on a subset of these isolates (n=99 isolates).

3.3.2 Data Prep and Read QC

Let's create the directory structure to hold our results and define some variables for convenience:

```
mkdir ~/mapping-variantcalling
MYBASE=/home/mtb_upm/mapping-variantcalling
MYDATA=$MYBASE/data
MYRESULTS=$MYBASE/results
mkdir -p $MYDATA/reads; mkdir -p $MYDATA/ref
mkdir -p $MYRESULTS/fastqc; mkdir -p $MYRESULTS/log; mkdir -p $MYRESULTS/qc; mkdir -p
↪ $MYRESULTS/mapping
```

We will start with a single isolate, *ERS050945* and then scale-up the analysis for all of the 99 isolates. We will define first a variable for the path to the folder with the source data and the precalculated results. We will then copy the data into our folder structure we created above:

```
SRCBASE=/home/mtb_upm/mtb_genomics_workshop_data/mapping-variantcalling
sample=ERS050945
cp $SRCBASE/data/reads/${sample}*.fastq $MYDATA/reads/
cp $SRCBASE/data/ref/* $MYDATA/ref
```

Generate a fastqc report:

```
fastqc $MYDATA/reads/${sample}_R1.fastq
fastqc $MYDATA/reads/${sample}_R2.fastq
```

Inspect the read quality profile and pick quality trimming parameters. Quality trim reads using sickle:

```
sickle pe -t sanger -q 30 -l 50 -n 5 \
-f $MYDATA/reads/${sample}_R1.fastq -r $MYDATA/reads/${sample}_R2.fastq \
-o $MYRESULTS/qc/${sample}_qced_R1.fastq -p $MYRESULTS/qc/${sample}_qced_R2.fastq \
-s $MYRESULTS/qc/${sample}_single.fastq 1> $MYRESULTS/log/${sample}.sickle.log
```

Question: How many paired reads remain after quality trimming? (hint: inspect the *sickle* command above and try to guess where that information might be)

3.3.3 Mapping Reads to the Reference Genome

Now that we have high-quality reads, we will start *mapping* reads to our reference genome. That is, using a “reference” *Mycobacterium tuberculosis* genome, we will determine the likely genomic positions from which the short reads have most likely been sampled.

Note: The canonical reference genome sequence for *M. tuberculosis* is that of strain [H37Rv](#), the first *M. tuberculosis* genome to have been sequenced. As such, *H37Rv* has a high quality genomic sequence and a wealth of information accumulated since then. It is important to realize that the reason *H37Rv* genome is being used as a reference is rather practically grounded.

Note: *Read mapping*, *read alignment*, and *sequence alignment* are often used interchangeably but in fact they are different concepts. The purpose of sequence alignment is to identify the *homologous* positions between evolutionarily related sequences. With read mapping, our objective is to identify the genomic region(s) from which the read sequence might be coming from. A mapping is considered to be correct if it overlaps the true region. Once we identify these regions, then we want align the read, in other words find the detailed placement of each base in the read.

For mapping reads to the reference genome, we will use BWA suite of programs. In this workshop, we will only going to be using two programs within the suite: `bwa index` and `bwa mem`.

BWA ([Burrows Wheeler Aligner](#)) uses Burrows Wheeler transform to rapidly map reads to the reference genome. Importantly, BWA allows for a certain number of mismatches as well as for insertions and deletions.

Before starting mapping, first we first need to index the reference genome. Indexing means arranging the genome into easily searchable chunks. Index the reference genome sequence using *bwa index*:

```
bwa index $MYDATA/ref/NC_000962.3.fna
```

`bwa index` will output some files with a set of extensions (`.amb`, `.ann`, `.bwt`, `.pac`, `.sa`), which the main alignment program (`bwa mem`) knows the format of. They will all end up in the same directory as the reference fasta file. Indexing is done once for the reference sequence. Before starting mapping, you need to make sure that these files have been generated. You don’t need to know the information in each but here is the information each one stores:

- `.amb`: text, records appearance of N (or other non-ATGC) in the reference fasta
- `.ann`: text, to record ref sequences, name, length, etc.
- `.bwt`: binary, the Burrows-Wheeler transformed sequence
- `.pac`: binary, packaged sequence (four base pairs encode one byte)
- `.sa`: binary, suffix array index

Now, we will start mapping reads. Explore the mapping and alignment options that `bwa mem` has by typing:

```
bwa mem
```

Read mapping has many uses in bioinformatics pipelines. Depending on your goals, you might want to change the parameters of the mapper, which primarily determine the sensitivity and specificity of mapping. In this workshop, we will use the default options, which work well for short reads.

```
bwa mem $MYDATA/ref/NC_000962.3.fna $MYRESULTS/qc/${sample}_qced_R1.fastq $MYRESULTS/
↪ qc/${sample}_qced_R2.fastq > $MYRESULTS/mapping/${sample}.mapped.sam
```

which aligns forward and reverse reads against the reference, outputs them to the `stdout` in `.sam` format, which we then save to a file.

Mappers result in either SAM or BAM file (<http://genome.sph.umich.edu/wiki/SAM>). Those are formats that contain the alignment information, where BAM is the binary version of the plain text SAM format. Although we are outputting a .sam file here for illustrative purposes-so we can peek into it-, you will typically want to do generate .bam files for your mapping because .sam files can get quite big and there is really no need to save them.

Inspect the first few lines of the .sam output file with `head` or `more`. It should look as follows:

```
# @SQ SN:NC_000962.3 LN:4411532
# @PG ID:bwa PN:bwa VN:0.7.17-r1188 CL:bwa mem ERS050945/data/ref/NC_000962.3.fasta
↳ERS050945/results1/qc/
```

Question: What does each line with sequences represent? Can you make sense of the information in different columns?

3.3.4 Visualizing the read alignments

We will be visualizing read alignments using the [Integrative Genomics Viewer](#) (IGV), a popular lightweight visualization tool for NGS data.

When you launch IGV, you first need to load a genome. By default, it will load the human genome. To get started, load the reference genome and then open your .bam file.

3.3.5 Inspecting, summarizing, and manipulating the read alignments

We will now inspect and further process the mapping output. For this, we will use `samtools` suite, which as its name suggests let us *massage* the mapping information.

In this section, we will also be playing with the read alignments to have a working understanding of how to get information from and manipulate them.

Converting sam to bam and back

First, let's convert our .sam mapping output to its binary counterpart .bam. The binary format is much easier for computers to process but very difficult for humans to read.

To convert .sam to .bam, we use the `samtools view` command. We must specify that our input is in .sam format (by default it expects .bam) using the `-S` option. We will also explicitly specify that we want the output to be .bam (by default it also produces bam) with the `-b` option. `samtools` follows the UNIX convention of sending its output to the UNIX STDOUT, so we need to use a redirect operator ("`>`") to create a BAM file from the output.

```
samtools view -S -b $MYRESULTS/mapping/${sample}.mapped.sam > $MYRESULTS/mapping/${sample}.mapped.bam
↳{sample}.mapped.bam
```

If you need to convert back .bam to .sam and look at it on the fly without saving to disk, you would do:

```
samtools view $MYRESULTS/mapping/${sample}.mapped.bam | head -n 10
```

Getting mapping stats directly from .bam

We can also determine mapping statistics directly from the bam file. Use for instance the following (which uses FLAG options, `-F` and `-f`, more on that later below):

```
samtools view -c -F 4 $MYRESULTS/mapping/${sample}.mapped.bam
samtools view -c -f 4 $MYRESULTS/mapping/${sample}.mapped.bam
```

What do these numbers outputted to STDOUT represent -invoke `samtools`'s help to answer this-. Do things make sense and add up?

Sorting the read alignments

When you map and align the reads to the reference, the resulting read alignments are in random order with respect to their position in the reference genome. In other words, the `.bam` file is in the order that the sequences occurred in the input `.fastq`. Prove to yourself that this is the case:

```
samtools view $MYRESULTS/mapping/${sample}.mapped.bam | more
```

Doing anything useful downstream such as calling variants or visualizing the alignments requires that the `.bam` is further manipulated. It must be sorted such that the alignments occur in “genome order”. That is, ordered positionally based upon their alignment coordinates on each chromosome (we obviously have a single one here). Sort the `.bam` file:

```
samtools sort $MYRESULTS/mapping/${sample}.mapped.bam -o $MYRESULTS/mapping/${sample}.  
↪sorted.bam
```

After sorting, check the order:

```
samtools view $MYRESULTS/mapping/${sample}.sorted.bam | more
```

Question (easy!): What do you notice?

Note: Sorting can also be done using `picard`. We will be using `picard` for removing duplicate alignments after sorting. When we build our variant calling pipeline, we will do the sorting with `picard` just to keep things consistent with downstream processing. The results should be identical.

Indexing read alignments (i.e. indexing `.bam`)

Indexing a genome sorted `.bam` file allows one to quickly extract alignments overlapping particular genomic regions. Indexing is also required by genome viewers (for instance IGV) so that the viewers can quickly display alignments in each genomic region to which you navigate. This is essential for large genomes.

Index your `.bam` file:

```
samtools index $MYRESULTS/mapping/${sample}.sorted.bam
```

This will create an additional index file -what's that file's extension?-

For instance, now that we have indexed the `.bam` file, we have the flexibility to extract the alignments from a defined genomic region. We can, for example, extract alignments from the 150th kilobase of the genome.

```
samtools view $MYRESULTS/mapping/${sample}.sorted.bam NC_000962.3:150000-151000
```

Question: How many alignments are within `NC_000962.3:150000-151000`?

Detailed inspection of some alignments

Again, let's inspect the first 10 alignments in our `.bam` file in detail:

```
samtools view $MYRESULTS/mapping/${sample}.sorted.bam | head -n 10
```

Let's also inspect just the header. The *header* in a bam file records important information regarding the reference genome to which the reads were aligned, as well as other information about how the BAM has been processed. We can ask the view command to report solely the header by using the `-H` option. As the downstream programs further process the alignments, they will typically add information about what they did to the header. For now, our header contain bare minimum information.

```
samtools view -H $MYRESULTS/mapping/${sample}.sorted.bam
```

The FLAG field in the .bam format encodes several key pieces of information regarding how an alignment aligned to the reference genome. We can use this information to isolate specific types of alignments that we want to use in our analysis. Here is a table for the flags right from the manual:

Bit	Description
0x1	template having multiple segments in sequencing
0x2	each segment properly aligned according to the aligner
0x4	segment unmapped
0x8	next segment in the template unmapped
0x10	SEQ being reverse complemented
0x20	SEQ of the next segment in the template being reversed
0x40	the first segment in the template
0x80	the last segment in the template
0x100	secondary alignment
0x200	not passing quality controls
0x400	PCR or optical duplicate
0x800	supplementary alignment

For example, we often want to call variants solely from paired-end sequences that aligned “properly” to the reference genome. Can you tell why?

To ask the view command to report solely “proper pairs” we use the `-f` option and ask for alignments where the second bit is true (proper pair is true).

```
samtools view -f 0x2 $MYRESULTS/mapping/${sample}.sorted.bam
```

- Question: How many properly paired alignments are there? How does that number compare to the number you got above using `-F`?
- Question: How many improperly paired alignments are there?
- Question: Using what you played with, how would you calculate the fragment size distribution?

3.3.6 QCing and filtering the read alignments

Before we can attempt variant calling, we need to make sure that the alignments that are resulted from mapping artefacts are identified. We also want to get an overall picture on the mapping quality.

Duplicate marking

During the lecture, you heard about library and optical duplicates, which if they exist will bias the variant calling. We will mark these duplicates using `picard`.

First, let's sort the .bam file, this time using `picard`. See the note in the previous section. This should give identical results to `samtools sort`, we are simply doing it this way for pipelining consistency.

```
java -Xmx16g -Djava.io.tmpdir=/tmp -jar /opt/picard.jar SortSam \
I=$MYRESULTS/mapping/${sample}.mapped.bam O=$MYRESULTS/mapping/${sample}.sorted.bam_
↪ SORT_ORDER=coordinate
```

Now, we can mark the duplicates:

```
java -Xmx16g -Djava.io.tmpdir=/tmp -jar /opt/picard.jar MarkDuplicates \
I=$MYRESULTS/mapping/${sample}.sorted.bam O=$MYRESULTS/mapping/${sample}.sorted.nodup.
↪ bam \
REMOVE_DUPLICATES=true ASSUME_SORT_ORDER=coordinate M=$MYRESULTS/mapping/${sample}.
↪ sorted.dupmetrics
```

The metrics file will contain some statistics about the de-duplication. In the resulting bam file, only one fragment from each duplicate group survives unchanged, other duplicate fragments are given a `FLAG 0x400` and will NOT be used downstream. Optimally, detection and marking of duplicate fragments should be done per library, i.e., over all read groups corresponding to a given library. In practice, often it is fine to take a shortcut and is sufficient to do it per lane (read group) to keep things naively *parallelizable*.

Assessing depth of coverage

You might often want to take a peak at the depth of coverage using the read alignments. You can do this with `samtools depth`:

```
samtools depth -a $MYRESULTS/mapping/${sample}.sorted.nodup.bam > $MYRESULTS/mapping/${
↪ sample}.coverage.txt
```

This will give you the depth of coverage for each position in the genome.

Question: How many lines should be in this file? Question: Can you write an `awk` one-liner to quickly compute the average depth of average?

Assessing mapping qualities

Next, we will generate summary statistics on mapping qualities. For this, we will use `qualimap`. `qualimap` has a GUI but we don't want to use it here. To avoid launching the GUI, we do `unset DISPLAY` first.

```
unset DISPLAY;
qualimap bamqc -bam $MYRESULTS/mapping/${sample}.sorted.nodup.bam --java-mem-size=4g -
↪ outformat PDF -outdir $MYRESULTS/mapping/${sample}.bam.qualimap
```

The results will be in a text file named `${sample}.bam.qualimap/genome_results.txt`. Later when we scale our analysis up to multiple isolates, we will parse out these files to get summary statistics for all of the isolates.

Indel realignment and filtering low quality alignments

We will use `pilon` to call our variants. This will give us a candidate list from a single sample, which we can further filter after we merge variants from multiple samples. `pilon` with its `--variant` option will perform *indel realignment*. We can also specify *depth* and mapping quality based filters. Run `pilon`:

```
java -Xmx16g -jar /opt/pilon-1.22.jar --genome $MYDATA/ref/NC_000962.3.fna --bam
↪ $MYRESULTS/mapping/${sample}.sorted.nodup.bam --output $MYRESULTS/mapping/${sample}_
↪ pilon --variant --mindepth 10 --minmq 40 --minqual 20
```

The output will be a `.vcf` file and a `.fasta` file. `.fasta` file has the sequence from this reference based assembly, which we won't be using downstream.

3.4 Day 4: Phylogenetics/Phylogenomics and Building Reproducible Computational Pipelines

3.4.1 Datasets

The dataset is a subset of 1000 *M. tuberculosis* isolates whole genome sequenced by Casali et al.: [Evolution and transmission of drug-resistant tuberculosis in a Russian population](#). In the previous tutorial, we generated an initial set of variants from a single *M. tuberculosis* isolate. In this subsequent tutorial, we will be repeating that for additional isolates. We will then combine all of these variants into a single dataset for further analysis.

File	Content
Mtb_repeats.bed	precalculated, coordinates of the repetitive regions in the reference <i>M.tuberculosis</i> genome

3.4.2 Data Prep and Read QC

Before starting, make sure that the following variables are defined in your shell environment. That means, you will need to type the following when you launch a new shell terminal:

```
MYBASE=/home/mtb_upm/mapping-variantcalling
MYDATA=$MYBASE/data
MYRESULTS=$MYBASE/results
```

Create a separate directory for the `.vcf` files:

```
mkdir $MYRESULTS/vcf
```

3.4.3 Processing and Merging SNVs from multiple isolates to get an MSA

What we need to do next is to get a set of high-quality SNVs across the whole genome. A SNV, single nucleotide variant (variant means substitution here), is defined as a variation in a single locus without any reference to its frequency in the population. A SNP or single nucleotide polymorphism, on the other hand is a variation for which we have evidence that it is fixed in the population. That means we should see it in more than a certain (arbitrarily defined but often >1%) number of individuals/isolates. Therefore, not all the SNVs are SNPs. Although this is the working definition we will be following, there is no consensus on this definition and all single nucleotide variants are called SNPs independent of their frequency in the population.

Previously in the workshop...

`vcf` file we generated using `pilon` contains unfiltered SNVs as well as indels. To remind, `pilon` did a series of filtering and qu

1. indel realignment
2. base quality score filtering
3. mapping quality score filtering
4. depth of coverage filtering.

So all SNVs in our `vcf` file have been, to the best of our ability, stripped off any known read mapping/alignment artifacts, has a reasonably high base quality, depth of coverage, and mapping quality.

Even after these steps, we expect a relatively high false positive rate within this list so we will need to do some further filtering. The assumption is that we are leaning towards minimizing false positive rate. We might be missing some true variants during this strict filtering steps. However, realize that we want to make sure that what we have as a final list (for instance to infer phylogeny) contain only true variants. That is we don't care so much if we do miss few true variants here and there (that is higher false negative rate is somewhat acceptable).

Why are we doing all this?

The end point for this analysis is to get an MSA (multiple sequence alignment) of the variable sites, which we can then use to infer phylogeny. Keep that in mind as you dive in.

Separating SNVs and indels and filtering SNVs within highly repetitive regions

First, we will need to get rid of the indels (insertion/deletion). As you've heard during the lecture, read mappers don't do a particularly good job with indels. Any called indel by any mapper, you should be very suspicious of. We want to remove them for phylogenetic analysis -we should already have lots of variation to infer phylogeny- as they will bias the phylogenetic inference. In the next step, we will use `vcftools` to make 2 separate `vcf` files, one that contains only SNVs and the other indels. Concurrently, we will also remove any variant that sits within a highly repetitive region. Next, a few words on why we want to do this, particularly in analyzing *M. tuberculosis* genomes.

Note: Repetitive regions in *M. tuberculosis* genome: A significant portion of the genome of *Mycobacterium tuberculosis* contains highly repetitive regions (every microbial genome will have some level of tandem and more complex repetitive regions, regions with low sequence complexity). Short-read sequencing isn't sufficient to make accurate SNV calls within these regions. Therefore we want to remove these regions from further analysis as they will confuse downstream phylogenetic analysis. We have precalculated the repetitive regions, which is provided in the `Mtb_repeats.bed` file in `bed` format. `bed` is a file format used to specify genomic intervals. Overall, the total length of repetitive regions specified in `Mtb_repeats.bed` is ~572 Kb (~13% of the genome). The majority of these regions (~10% of the genome) belong to two large unrelated gene families (PE and PPE families) clustered in the genome. They code for glycine-rich proteins that are often based on multiple copies of the polymorphic repetitive sequences referred to as PGRSs **restructured text**. (****P****olymorphic guanine cytosine-rich (GC-rich) repetitive sequences). Despite their prominence in the genome, we almost know nothing about the biological significance of these regions.

Run `vcftools` to separate SNVs/indels and filter repetitive regions:

```
# get rid of "_pilon" in the filename
mv ${sample}_pilon.vcf ${sample}.vcf
vcftools --gzvcf ${sample}.vcf --exclude-bed $SRCBASE/data/ref/Mtb_repeats.bed --
→remove-indels --recode --recode-INFO-all --out $sample
vcftools --gzvcf ${sample}.vcf --exclude-bed $SRCBASE/data/ref/Mtb_repeats.bed --keep-
→only-indels --recode --recode-INFO-all --out ${sample}.indels

# take a peek at the indels file! We will be leaving it behind from this point on
more ${sample}.indels.recode.vcf
# and the vcftools log file
more ${sample}.indels.log
```

Note: `vcf` files are typically huge (~0.5 GB even for a single microbial genome). Compressing `vcf` files with `bgzip` and indexing them with `tabix` is the standard way `vcf` files are stored. Once that is done, further processing of the

binary files is done with `bcftools`.

Compress and index the vcf file. Notice that we won't use the vcf file for indels for this analysis.

```
bgzip ${sample}.recode.vcf
tabix -p vcf ${sample}.recode.vcf.gz
```

We need to change the sample header (which is “SAMPLE”, the very last column within the data fields) in the vcf file to specify that these variants are from `$sample`. This is to ensure data provenance: we will soon merge vcf files from multiple samples and they all have the same header at this point. You can do this with `bcftools` which requires the header name to be stored in a text file, so we write the sample name into a file called “sample” first.

```
echo $sample > sample
bcftools reheader -s sample ${sample}.recode.vcf.gz > m.${sample}.recode.vcf.gz # we
↪add this "m" just to make the filename different in this temp file
mv -f m.${sample}.recode.vcf.gz ${sample}.recode.vcf.gz

# reindex
tabix -f -p vcf ${sample}.recode.vcf.gz

# make sure nothing that ``pilon`` didn't let it "PASS" doesn't get in
bcftools view --include 'FILTER="PASS"' ${sample}.recode.vcf.gz -O z -o temp

mv -f temp ${sample}.recode.vcf.gz
# reindex and we are done
tabix -f -p vcf ${sample}.recode.vcf.gz
```

All we did here was modifying sample “header” and some simple “data massaging”. If you want, you can unzip the `${sample}.recode.vcf.gz` file, peek into it, and do some other quick sanity check:

```
# OPTIONAL
gunzip ${sample}.recode.vcf.gz
# pay attention to the column name for the very last column, does it look ok?
more ${sample}.recode.vcf
wc ${sample}.vcf
wc ${sample}.recode.vcf
```

Merge vcf files from multiple isolates

Assuming that we did read mapping etc. for multiple isolates (which you will learn how to do using `snakemake`), at this point we can merge all the vcf files from multiple samples into a single vcf file. We will use that single vcf to define *SNP*s that be used for phylogenetic analysis:

```
# the way this following command is constructed is intentionally explicit so you can
↪see what is happening
# notice that this list of 99 isolates contains our prototyping isolate, "ERS050945",
↪and 98 additional ones
bcftools merge \
ERS050923.recode.vcf.gz \
ERS050924.recode.vcf.gz \
ERS050925.recode.vcf.gz \
ERS050926.recode.vcf.gz \
ERS050927.recode.vcf.gz \
ERS050928.recode.vcf.gz \
ERS050929.recode.vcf.gz \
```

(continues on next page)

(continued from previous page)

```
ERS050930.recode.vcf.gz \
ERS050931.recode.vcf.gz \
ERS050932.recode.vcf.gz \
ERS050933.recode.vcf.gz \
ERS050934.recode.vcf.gz \
ERS050935.recode.vcf.gz \
ERS050936.recode.vcf.gz \
ERS050937.recode.vcf.gz \
ERS050938.recode.vcf.gz \
ERS050939.recode.vcf.gz \
ERS050940.recode.vcf.gz \
ERS050942.recode.vcf.gz \
ERS050943.recode.vcf.gz \
ERS050944.recode.vcf.gz \
ERS050945.recode.vcf.gz \
ERS050946.recode.vcf.gz \
ERS050947.recode.vcf.gz \
ERS050948.recode.vcf.gz \
ERS050949.recode.vcf.gz \
ERS050950.recode.vcf.gz \
ERS050951.recode.vcf.gz \
ERS050953.recode.vcf.gz \
ERS050954.recode.vcf.gz \
ERS050955.recode.vcf.gz \
ERS050956.recode.vcf.gz \
ERS050957.recode.vcf.gz \
ERS050958.recode.vcf.gz \
ERS050959.recode.vcf.gz \
ERS050960.recode.vcf.gz \
ERS050961.recode.vcf.gz \
ERS050962.recode.vcf.gz \
ERS050963.recode.vcf.gz \
ERS050964.recode.vcf.gz \
ERS050965.recode.vcf.gz \
ERS050966.recode.vcf.gz \
ERS050968.recode.vcf.gz \
ERS050969.recode.vcf.gz \
ERS050970.recode.vcf.gz \
ERS050971.recode.vcf.gz \
ERS050972.recode.vcf.gz \
ERS053603.recode.vcf.gz \
ERS053604.recode.vcf.gz \
ERS053605.recode.vcf.gz \
ERS053606.recode.vcf.gz \
ERS053607.recode.vcf.gz \
ERS053608.recode.vcf.gz \
ERS053609.recode.vcf.gz \
ERS053610.recode.vcf.gz \
ERS053611.recode.vcf.gz \
ERS053612.recode.vcf.gz \
ERS053613.recode.vcf.gz \
ERS053614.recode.vcf.gz \
ERS053615.recode.vcf.gz \
ERS053616.recode.vcf.gz \
ERS053617.recode.vcf.gz \
ERS053618.recode.vcf.gz \
ERS053619.recode.vcf.gz \
```

(continues on next page)

(continued from previous page)

```

ERS053620.recode.vcf.gz \
ERS053621.recode.vcf.gz \
ERS053622.recode.vcf.gz \
ERS053623.recode.vcf.gz \
ERS053624.recode.vcf.gz \
ERS053625.recode.vcf.gz \
ERS053626.recode.vcf.gz \
ERS053627.recode.vcf.gz \
ERS053628.recode.vcf.gz \
ERS053629.recode.vcf.gz \
ERS053630.recode.vcf.gz \
ERS053631.recode.vcf.gz \
ERS053632.recode.vcf.gz \
ERS053633.recode.vcf.gz \
ERS053634.recode.vcf.gz \
ERS053635.recode.vcf.gz \
ERS053636.recode.vcf.gz \
ERS053637.recode.vcf.gz \
ERS053638.recode.vcf.gz \
ERS053639.recode.vcf.gz \
ERS053641.recode.vcf.gz \
ERS053642.recode.vcf.gz \
ERS053643.recode.vcf.gz \
ERS053644.recode.vcf.gz \
ERS053651.recode.vcf.gz \
ERS053656.recode.vcf.gz \
ERS053659.recode.vcf.gz \
ERS053661.recode.vcf.gz \
ERS053663.recode.vcf.gz \
ERS053667.recode.vcf.gz \
ERS053672.recode.vcf.gz \
ERS053684.recode.vcf.gz \
ERS096323.recode.vcf.gz \
ERS181351.recode.vcf.gz \
ERS181603.recode.vcf.gz \
-O v -o merged-99RussianIsolates.vcf

```

Merging step requires a reasonable amount of memory and takes some time. We will be using precalculated file. Now, what we essentially have is a vcf file that contains all the information to make an MSA of the variable position in the genome. Since everything is already defined with respect to a reference genome, we don't need to build an MSA, per se, it will already just fall off once we get rid of the non variable sites. That is what `snp-sites` does.

Warning: Merging multiple vcf files, the way it has been implemented here, is highly compute heavy. It requires high memory instances when you have 100s to 1000s of isolates.

In what follows, we first convert vcf to a fasta file using `vcf2phylip.py` (despite its name, it also does this) before calling SNPs. **compute heavy, DONOT RUN**

```

# -m: keep all SNVs independent of their frequency, typically with a larger number of
↪ isolates
# we would want to impose a threshold
vcf2phylip.py --fasta --phylip-disable -m 1 --input merged-99RussianIsolates.vcf

# -m: output fasta, -v: output VCF, -p: output PHYLIB
snp-sites -m merged-99RussianIsolates.min1.fasta -o 99RussianIsolates.msa.fasta

```

99RussianIsolates.msa.fasta contains our multiple sequence alignment.

3.4.4 Building Bioinformatics Pipelines with Snakemake

A workflow management system (WMS) is a piece of software that sets up, performs and monitors a defined sequence of computational tasks (i.e. “a workflow”). Snakemake is a WMS that was developed in the bioinformatics community, and as such it has some features that make it particularly well suited for creating reproducible and scalable data analyses.

- The language you use to formulate your workflows is based on Python, which is a language with strong standing in academia. However, users are not required to know how to code in Python to work efficiently with Snakemake.
- Workflows can easily be scaled from your desktop to server, cluster, grid or cloud environments. This makes it possible to develop a workflow on your laptop, maybe using only a small subset of your data, and then run the real analysis on a cluster.
- Snakemake has several features for defining the environment which each task is carried out in. This is important in bioinformatics, where workflows often involve running a large number of small third-party tools.
- Snakemake is primarily intended to work on `_files_` (rather than for example streams, reading/writing from databases or passing variables in memory). This fits well with many fields of bioinformatics, notably next-generation sequencing, that often involve computationally expensive operations on large files. It’s also a good fit for a scientific research setting, where the exact specifications of the final workflow aren’t always known at the beginning of a project.
- Lastly, a WMS is a very important tool for making your analyses reproducible. By keeping track of when each file was generated, and by which operation, it is possible to ensure that there is a consistent “paper trail” from raw data to final results. Snakemake also has features which allow you to package and distribute the workflow, and any files it involves, once it’s done.

The basics

Activating snakemake environment within the VM

In the tutorial virtual machine, snakemake is installed within a `python3.5` environment. Activate the environment as follows:

```
source activate snakemake
# check if snakemake is available
snakemake -h
```

Create a directory to hold our work:

```
mkdir -p ~/mtb_genomics_workshop/snakemake/example
cd ~/mtb_genomics_workshop/snakemake/example
```

In this part of the tutorial we will create a very simple workflow from scratch, in order to show the fundamentals of how Snakemake works. The workflow will take two files as inputs, *a.txt* and *b.txt*, and the purpose is to convert the text in the files to upper case and then to concatenate them.

Run the following shell commands. The first one will make an empty file named *Snakefile*, which will later contain our example workflow. The second and third commands generate two files containing some arbitrary text.

```
touch Snakefile
echo "This is a.txt" > a.txt
echo "This is b.txt" > b.txt
```

Then open *Snakefile* in a text editor. You can use [SublimeText](#), which you can invoke from command line with `subl`:

```
subl Snakefile
```

Rules

A Snakemake workflow is based on rules which take some file(s) as input, performs some type of operation on them, and generate some file(s) as outputs. Here is a very simple rule that takes *a.txt* as an input and produces *a.upper.txt* as an output. Copy this rule to your *Snakefile* and save it.

```
rule convert_to_upper_case:
    input:
        "a.txt"
    output:
        "a.upper.txt"
    shell:
        "tr [a-z] [A-Z] < {input} > {output}"
```

Important: Indentation is important in Snakefiles, so make sure that you have the correct number of spaces before *input/output/shell* and their respective subsections. The number of spaces per level doesn't matter as long as you're consistent. Here we use four, but you could just as well use two for a more compact look. Don't use tabs (unless your editor automatically converts them to spaces).

- A rule has a name, here it's *convert_to_upper_case*. Make an effort to name your rules in a way that makes it easy to understand. The purpose of the rule, as rule names, are one of the main ways to interact with the workflow.
- The *shell* section contains the shell commands that will convert the text in the input file to upper case and send it to the output file. In the shell command string, we can refer to elements of the rule via curly brackets. Here, we refer to the output file by specifying *{output}* and to the input file by specifying *{input}*. This particular command can be read like “send the contents of *a.txt* to the program *tr*, which will convert all characters in the set [a-z] to the corresponding character in the set [A-Z], and then send the output to *a.upper.txt*”.

Now let's run our first Snakemake workflow. When a workflow is executed Snakemake tries to generate a set of target files. Target files can be specified via the command line (or, as you will see later, in several other ways). Here we ask Snakemake to make the file *a.upper.txt*. It's good practice to first run with the flag `--dryrun` (or `-n`), which will show what Snakemake plans to do without actually running anything. You can also use the flag `--printshellcmds` (or `-p`, for showing the shell commands that it will execute, and the flag `--reason` (or `-r` for showing the reason for running a specific rule. `snakemake --help` will show you all available flags.

Do a dry-run:

```
snakemake --dryrun --reason --printshellcmds a.upper.txt
```

```
Building DAG of jobs...
Job counts:
  count  jobs
   1     convert_to_upper_case
   1
```

(continues on next page)

(continued from previous page)

```
rule convert_to_upper_case:
    input: a.txt
    output: a.upper.txt
    jobid: 0
    reason: Missing output files: a.upper.txt

tr [a-z] [A-Z] < a.txt > a.upper.txt
Job counts:
  count  jobs
    1    convert_to_upper_case
    1
```

You can see that Snakemake plans to run 1 job: the rule *convert_to_upper_case* with *a.txt* as input and *a.upper.txt* as output. The reason for doing this is that it's missing the file *a.upper.txt*.

Now execute the workflow without the *--dryrun* flag and check that the contents of *a.upper.txt* is as expected. Then try running the same command again. What do you see? It turns out that Snakemake only reruns jobs if **one of the input files is newer than one of the output files, or if one of the input files will be updated by another job**. This is how Snakemake ensures that everything in the workflow is up to date.

Wildcards

We are going to next introduce the concept of wildcards in snakemake. What if we ask Snakemake to generate the file *b.upper.txt*?

```
$ snakemake --dryrun --reason --printshellcmds b.upper.txt
Building DAG of jobs...
MissingRuleException:
No rule to produce b.upper.txt (if you use input functions make sure that they don't
↳raise unexpected exceptions).
```

That didn't work. We could copy the rule to make a similar one for *b.txt*, but that would be a bit cumbersome. Here is where named wildcards come in; one of the most powerful features of Snakemake. Simply change the input from *input: "a.txt"* to *input: "{some_name}.txt"* and the output to *output: "{some_name}.upper.txt"*.

Your updated *Snakefile* should look as follows:

```
rule convert_to_upper_case:
    input:
        "{some_name}.txt"
    output:
        "{some_name}.upper.txt"
    shell:
        "tr [a-z] [A-Z] < {input} > {output}"
```

Now try asking for *b.upper.txt* again:

```
$ snakemake b.upper.txt
Building DAG of jobs...
Using shell: /usr/bin/bash
Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
  count  jobs
```

(continues on next page)

(continued from previous page)

```

1      convert_to_upper_case
1

rule convert_to_upper_case:
    input: b.txt
    output: b.upper.txt
    jobid: 0
    wildcards: some_name=b

Finished job 0.
1 of 1 steps (100%) done
Complete log: /home/mtb_upm/mtb_genomics_workshop/snakemake/example/.snakemake/log/
↳ 2018-07-05T134652.700927.snakemake.log

```

What happens here is that Snakemake looks at all the rules it has available (actually only one in this case) and tries to assign values to all wildcards so that the targeted files can be generated. In this case it was quite simple, you can see that it says that *wildcards: some_name=b*, but for large workflows and multiple wildcards it can get much more complex. Named wildcards is what enables a workflow (or single rules) to be efficiently generalized and reused between projects or shared between people and make scaling up computational workflows feasible.

It seems we have the first part of our workflow working, now it's time to make the second rule for concatenating the outputs from *convert_to_upper_case*. The rule structure will be similar; the only difference is that here we have two inputs instead of one. This can be expressed in two ways, either with named inputs like this:

```

input:
    firstFile="...",
    secondFile="..."
shell:
    some_function {input.firstFile} {input.secondFile}

```

Or with indices like this:

```

input:
    "...",
    "..."
shell:
    some_function {input[0]} {input[1]}

```

Important: If you have multiple inputs or outputs they need to be delimited with a comma (as seen above). This is a very common mistake when writing Snakemake workflows. The parser will complain, but sometimes the error message can be difficult to interpret.

Now try to construct this rule yourself and name it *concatenate_a_and_b*. The syntax for concatenating two files in Bash is *cat first_file second_file > output_file*. Call the output *c.txt*. Run the workflow in Snakemake and validate that the output looks as expected.

Wouldn't it be nice if our workflow could be used for *any* files, not just *a.txt* and *b.txt*? We can achieve this by using named wildcards (or in other ways as we will discuss later). As we've mentioned, Snakemake looks at all the rules it has available and tries to assign values to all wildcards so that the targeted files can be generated. We therefore have to name the output file in a way so that it also contains information about which input files it should be based on. Try to figure out how to do this yourself.

Rename the rule to *concatenate_files* to reflect its new more general use.

```
rule concatenate_files:
    input:
        "{first}.upper.txt",
        "{second}.upper.txt"
    output:
        "{first}_{second}.txt"
    shell:
        "cat {input[0]} {input[1]} > {output}"
```

We can now control which input files to use by the name of the file we ask Snakemake to generate.

```
$ snakemake a_b.txt
Building DAG of jobs...
Using shell: /usr/bin/bash
Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
   count  jobs
     1    concatenate_files
     1    convert_to_upper_case
     2
rule convert_to_upper_case:
    input: a.txt
    output: a.upper.txt
    jobid: 2
    wildcards: some_name=a

Finished job 2.
1 of 2 steps (50%) done

rule concatenate_files:
    input: a.upper.txt, b.upper.txt
    output: a_b.txt
    jobid: 0
    wildcards: second=b, first=a

Finished job 0.
2 of 2 steps (100%) done
Complete log: /home/mtb_upm/mtb_genomics_workshop/snakemake/example/.snakemake/log/
↳ 2018-07-05T135145.499632.snakemake.log
```

What we have learned so far: * How a simple Snakemake rule looks * How to define target files when executing a workflow * How to use named wildcards for writing generic and flexible rules.

Tip: You can name a file whatever you want in a Snakemake workflow, but you will find that everything makes much more sense if the filename reflects the file's path through the workflow, e.g. *sample_a.trimmed.deduplicated.sorted.bam*.

Visualization, logging and workflow management

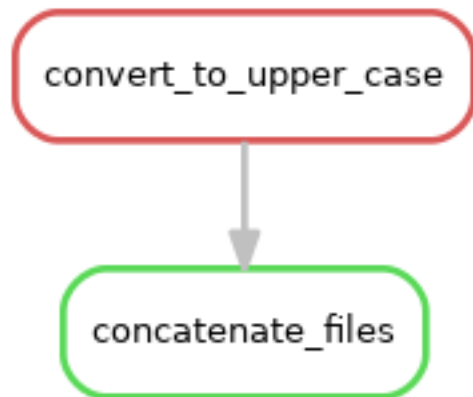
What we've done so far could have been quite easily done in a simple shell script that takes the input files as parameters. Let's now take a look at some of the features where Snakemake really adds value compared to a more straightforward approach. One such feature is the possibility to visualize your workflow.

Snakemake can generate two types of graphs:

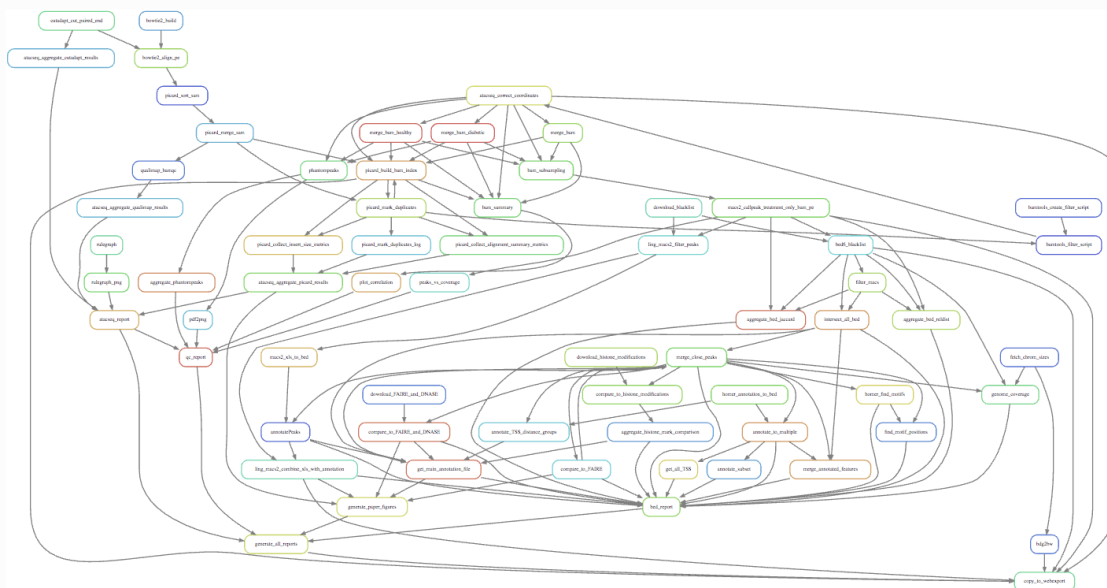
1. rules graph that shows how the rules are connected
2. jobs graph that shows how the jobs (i.e. an execution of a rule with some given inputs/outputs/settings) are connected.

Rule graph: First we look at the rule graph. The following command will generate a rule graph in the dot language and pipe it to the program `dot`, which in turn will save a visualization of the graph as a png file (if you're having troubles displaying png files you could use `svg` or `jpg` instead).

```
snakemake --rulegraph a_b.txt | dot -Tpng > rulegraph.png
```

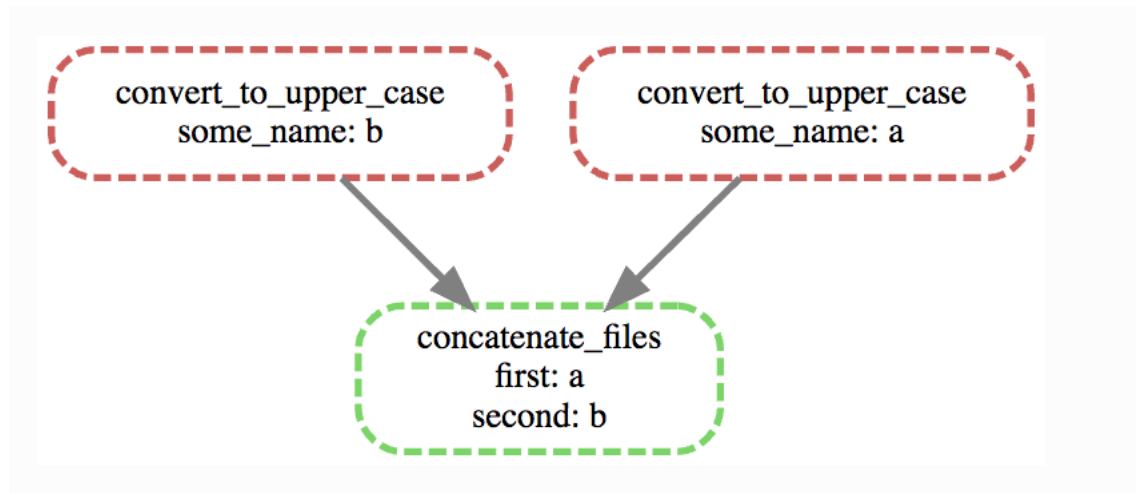


All the graph says is that the output from the rule *convert_to_upper_case* will be used as input to the rule *concatenate_files*. For a more typical bioinformatics project it can look something like this:



Job graph: The second type of graph is based on the jobs, and looks like this for our little workflow (use `-dag` instead of `-rulegraph`).

```
snakemake --dag a_b.txt | dot -Tpng > jobgraph.png
```

The main difference here is that now each node is a job instead of a rule. You can see that the wildcards used in each job are also displayed. Another difference is the dotted lines around the nodes. A dotted line is Snakemake's way of indicating that this rule doesn't need to be rerun in order to generate *a_b.txt*. Validate this by running *snakemake -n -r a_b.txt* and it should say that there is nothing to be done.

We've discussed before that one of the main purposes of using a WMS is that it automatically makes sure that everything is up to date. This is done by recursively checking that outputs are always newer than inputs for all the rules involved in the generation of your target files.

Now try to change the contents of *a.txt* to some other text and save it. What do you think will happen if you run *snakemake -n -r a_b.txt* again?

```

$ snakemake -n -r a_b.txt

rule convert_to_upper_case:
  input: a.txt
  output: a.upper.txt
  jobid: 2
  reason: Updated input files: a.txt
  wildcards: some_name=a

rule concatenate_files:
  input: a.upper.txt, b.upper.txt
  output: a_b.txt
  jobid: 0
  reason: Input files updated by another job: a.upper.txt
  wildcards: first=a, second=b

Job counts:
  count  jobs
    1    concatenate_files
    1    convert_to_upper_case
  ...

```

Also generate the job graph and compare to the one generated above. What's the difference?

Now rerun without *-n* and validate that *a_b.txt* contains the new text. Note that Snakemake doesn't look at the contents of files when trying to determine what has changed, only at the timestamp for when they were last modified.

We’ve seen that Snakemake keeps track of if files in the workflow have changed, and automatically makes sure that any results depending on such files are regenerated. What about if the rules themselves are changed? It turns out that there are multiple ways to do this, but the most straightforward is to manually specify that you want to rerun a rule (and thereby also all the steps between that rule and your target). Let’s say that we want to modify the rule *concatenate_files* to also include which files were concatenated.

```
rule concatenate_files:
    input:
        "{first}.upper.txt",
        "{second}.upper.txt"
    output:
        "{first}_{second}.txt"
    shell:
        "echo 'Concatenating {input}' | cat - {input[0]} {input[1]} > {output}"
```

If you now run the workflow as before you should get “Nothing to be done”, because no files involved in the workflow have been changed. Instead we have to force Snakemake to rerun the rule by using the ‘-R’ flag. Let’s try a dry-run.

```
snakemake a_b.txt -r -n -R concatenate_files
```

Note that the reason for the job is now “Forced execution”. You can target files as well as rules, so you would get the same result with *-R a_b.txt*. Whenever you’ve made changes to a rule that will affect the output it’s good practice to force re-execution like this. Still, there can be situations where you don’t know if any rules have been changed. Maybe several people collaborate on the same workflow but are using it on different files for example. Snakemake keeps track of how all files were generated (when, by which rule, which version of the rule, and by which commands). You can export this information to a tab-delimited file like this:

```
snakemake a_b.txt -D > summary.tsv
```

The contents of *summary.tsv* is shown in the table below (scroll to see the full table).

output_file	date	rule	version	log-file(s)	input-file(s)	shellcmd	status	plan
a_b.txt	Thu Nov 16 12:03:11 2017	concatenate_files			a.upper.txt, b.upper.txt	a.upper.txt b.upper.txt > a_b.txt	rule implementation changed	no update
a.upper.txt	Thu Nov 16 12:03:11 2017	convert_to_upper_case			a.txt	tr [a-z] [A-Z] < a.txt > a.upper.txt	ok	no update
b.upper.txt	Thu Nov 16 12:03:11 2017	convert_to_upper_case			b.txt	tr [a-z] [A-Z] < b.txt > b.upper.txt	ok	no update

You can see in the second last column that the rule implementation for *a_b.txt* has changed. The last column shows if Snakemake plans to regenerate the files when it’s next executed. None of the files will be regenerated because Snakemake doesn’t regenerate files by default if the rule implementation changes. From a reproducibility perspective maybe it would be better if this was done automatically, but it would be very computationally expensive and cumbersome if you had to rerun your whole workflow every time you fix a spelling mistake in a comment somewhere. So, it’s up to us to look at the summary table and rerun things as needed. You can get a list of the files for which the rule implementation has changed, and then force Snakemake to regenerate these files with the *-R* flag.

```
snakemake a_b.txt -R $(snakemake a_b.txt --list-code-changes)
```

There are a number of these *-list-xxx-changes* flags that can help you keep track of your workflow. You can list all options with *snakemake -help*. Run with the *-D* flag again to make sure that the summary table now looks like expected.

You might wonder where Snakemake keeps track of all these things? It stores all information in a hidden subdirectory called *.snakemake*. This is convenient since it's easy to delete if you don't need it anymore and everything is contained in the project directory.

By now you should be familiar with the basic functionality of Snakemake, and you can build advanced workflows with only the features we have discussed here. There's a lot we haven't covered though, in particular when it comes to making your workflow more reusable. In the following section we will start with a workflow for calling variants.

We've learned:

- How to use *-dag* and *-rulegraph* for visualizing the job and rule graphs, respectively.
- How to force Snakemake to rerun relevant parts of the workflow after there have been changes.
- How logging in Snakemake works.

Reference: Koster, J. and Rahmann, S. (2012). Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19), 2520–2522.

Resources:

- The Snakemake [documentation](#)
- Here is the *official* [tutorial](#)
- Further questions? Check out stack [overflow](#).
- There is a snakemake [Googlegroups](#).

Credit: NBIS Reproducible research course.

3.5 Day 5: Building Reproducible Computational Pipelines

Building Reproducible Computational Pipelines using python and snakemake

3.5.1 Building Bioinformatics Pipelines with Snakemake

A workflow management system (WMS) is a piece of software that sets up, performs and monitors a defined sequence of computational tasks (i.e. “a workflow”). Snakemake is a WMS that was developed in the bioinformatics community, and as such it has some features that make it particularly well suited for creating reproducible and scalable data analyses.

- The language you use to formulate your workflows is based on Python, which is a language with strong standing in academia. However, users are not required to know how to code in Python to work efficiently with Snakemake.
- Workflows can easily be scaled from your desktop to server, cluster, grid or cloud environments. This makes it possible to develop a workflow on your laptop, maybe using only a small subset of your data, and then run the real analysis on a cluster.
- Snakemake has several features for defining the environment which each task is carried out in. This is important in bioinformatics, where workflows often involve running a large number of small third-party tools.

- Snakemake is primarily intended to work on `_files_` (rather than for example streams, reading/writing from databases or passing variables in memory). This fits well with many fields of bioinformatics, notably next-generation sequencing, that often involve computationally expensive operations on large files. It's also a good fit for a scientific research setting, where the exact specifications of the final workflow aren't always known at the beginning of a project.
- Lastly, a WMS is a very important tool for making your analyses reproducible. By keeping track of when each file was generated, and by which operation, it is possible to ensure that there is a consistent "paper trail" from raw data to final results. Snakemake also has features which allow you to package and distribute the workflow, and any files it involves, once it's done.

The basics

Activating `snakemake` environment within the VM

In the tutorial virtual machine, `snakemake` is installed within a `python3.5` environment. Activate the environment as follows:

```
source activate snakemake
# check is snakemake is available
snakemake -h
```

Create a directory to hold our work:

```
mkdir -p ~/mtb_genomics_workshop/snakemake/example
cd ~/mtb_genomics_workshop/snakemake/example
```

In this part of the tutorial we will create a very simple workflow from scratch, in order to show the fundamentals of how Snakemake works. The workflow will take two files as inputs, *a.txt* and *b.txt*, and the purpose is to convert the text in the files to upper case and then to concatenate them.

Run the following shell commands. The first one will make an empty file named *Snakefile*, which will later contain our example workflow. The second and third commands generate two files containing some arbitrary text.

```
touch Snakefile
echo "This is a.txt" > a.txt
echo "This is b.txt" > b.txt
```

Then open *Snakefile* in a text editor. You can use [SublimeText](#), which you can invoke from command line with `subl`:

```
subl Snakefile
```

Rules

A Snakemake workflow is based on rules which take some file(s) as input, performs some type of operation on them, and generate some file(s) as outputs. Here is a very simple rule that takes *a.txt* as an input and produces *a.upper.txt* as an output. Copy this rule to your *Snakefile* and save it.

```
rule convert_to_upper_case:
    input:
        "a.txt"
    output:
        "a.upper.txt"
    shell:
        "tr [a-z] [A-Z] < {input} > {output}"
```

Important: Indentation is important in Snakefiles, so make sure that you have the correct number of spaces before *input/output/shell* and their respective subsections. The number of spaces per level doesn't matter as long as you're consistent. Here we use four, but you could just as well use two for a more compact look. Don't use tabs (unless your editor automatically converts them to spaces).

- A rule has a name, here it's *convert_to_upper_case*. Make an effort to name your rules in a way that makes it easy to understand. The purpose of the rule, as rule names, are one of the main ways to interact with the workflow.
- The *shell* section contains the shell commands that will convert the text in the input file to upper case and send it to the output file. In the shell command string, we can refer to elements of the rule via curly brackets. Here, we refer to the output file by specifying *{output}* and to the input file by specifying *{input}*. This particular command can be read like "send the contents of *a.txt* to the program *tr*, which will convert all characters in the set [a-z] to the corresponding character in the set [A-Z], and then send the output to *a.upper.txt*".

Now let's run our first Snakemake workflow. When a workflow is executed Snakemake tries to generate a set of target files. Target files can be specified via the command line (or, as you will see later, in several other ways). Here we ask Snakemake to make the file *a.upper.txt*. It's good practice to first run with the flag *--dryrun* (or *-n*), which will show what Snakemake plans to do without actually running anything. You can also use the flag *--printshellcmds* (or *-p*, for showing the shell commands that it will execute, and the flag *--reason* (or *-r* for showing the reason for running a specific rule. *snakemake --help* will show you all available flags.

Do a dry-run:

```
snakemake --dryrun --reason --printshellcmds a.upper.txt
```

```
Building DAG of jobs...
Job counts:
  count  jobs
   1      convert_to_upper_case
   1

rule convert_to_upper_case:
  input: a.txt
  output: a.upper.txt
  jobid: 0
  reason: Missing output files: a.upper.txt

tr [a-z] [A-Z] < a.txt > a.upper.txt
Job counts:
  count  jobs
   1      convert_to_upper_case
   1
```

You can see that Snakemake plans to run 1 job: the rule *convert_to_upper_case* with *a.txt* as input and *a.upper.txt* as output. The reason for doing this is that it's missing the file *a.upper.txt*.

Now execute the workflow without the *--dryrun* flag and check that the contents of *a.upper.txt* is as expected. Then try running the same command again. What do you see? It turns out that Snakemake only reruns jobs if **one of the input files is newer than one of the output files, or if one of the input files will be updated by another job**. This is how Snakemake ensures that everything in the workflow is up to date.

Wildcards

We are going to next introduce the concept of wildcards in `snakemake`. What if we ask `Snakemake` to generate the file `b.upper.txt`?

```
$ snakemake --dryrun --reason --printshellcmds b.upper.txt
Building DAG of jobs...
MissingRuleException:
No rule to produce b.upper.txt (if you use input functions make sure that they don't
↳raise unexpected exceptions).
```

That didn't work. We could copy the rule to make a similar one for `b.txt`, but that would be a bit cumbersome. Here is where named wildcards come in; one of the most powerful features of `Snakemake`. Simply change the input from `input: "a.txt"` to `input: "{some_name}.txt"` and the output to `output: "{some_name}.upper.txt"`.

Your updated *Snakefile* should look as follows:

```
rule convert_to_upper_case:
    input:
        "{some_name}.txt"
    output:
        "{some_name}.upper.txt"
    shell:
        "tr [a-z] [A-Z] < {input} > {output}"
```

Now try asking for `b.upper.txt` again:

```
$ snakemake b.upper.txt
Building DAG of jobs...
Using shell: /usr/bin/bash
Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
  count  jobs
   1      convert_to_upper_case
   1
rule convert_to_upper_case:
  input: b.txt
  output: b.upper.txt
  jobid: 0
  wildcards: some_name=b

Finished job 0.
1 of 1 steps (100%) done
Complete log: /home/mtb_upm/mtb_genomics_workshop/snakemake/example/.snakemake/log/
↳2018-07-05T134652.700927.snakemake.log
```

What happens here is that `Snakemake` looks at all the rules it has available (actually only one in this case) and tries to assign values to all wildcards so that the targeted files can be generated. In this case it was quite simple, you can see that it says that *wildcards: some_name=b*, but for large workflows and multiple wildcards it can get much more complex. Named wildcards is what enables a workflow (or single rules) to be efficiently generalized and reused between projects or shared between people and make scaling up computational workflows feasible.

It seems we have the first part of our workflow working, now it's time to make the second rule for concatenating the outputs from `convert_to_upper_case`. The rule structure will be similar; the only difference is that here we have two inputs instead of one. This can be expressed in two ways, either with named inputs like this:

```
input:
    firstFile="...",
    secondFile="..."
shell:
    some_function {input.firstFile} {input.secondFile}
```

Or with indices like this:

```
input:
    "...",
    "..."
shell:
    some_function {input[0]} {input[1]}
```

Important: If you have multiple inputs or outputs they need to be delimited with a comma (as seen above). This is a very common mistake when writing Snakemake workflows. The parser will complain, but sometimes the error message can be difficult to interpret.

Now try to construct this rule yourself and name it *concatenate_a_and_b*. The syntax for concatenating two files in Bash is *cat first_file second_file > output_file*. Call the output *c.txt*. Run the workflow in Snakemake and validate that the output looks as expected.

Wouldn't it be nice if our workflow could be used for *any* files, not just *a.txt* and *b.txt*? We can achieve this by using named wildcards (or in other ways as we will discuss later). As we've mentioned, Snakemake looks at all the rules it has available and tries to assign values to all wildcards so that the targeted files can be generated. We therefore have to name the output file in a way so that it also contains information about which input files it should be based on. Try to figure out how to do this yourself.

Rename the rule to *concatenate_files* to reflect its new more general use.

```
rule concatenate_files:
    input:
        "{first}.upper.txt",
        "{second}.upper.txt"
    output:
        "{first}_{second}.txt"
    shell:
        "cat {input[0]} {input[1]} > {output}"
```

We can now control which input files to use by the name of the file we ask Snakemake to generate.

```
$ snakemake a_b.txt
Building DAG of jobs...
Using shell: /usr/bin/bash
Provided cores: 1
Rules claiming more threads will be scaled down.
Job counts:
   count  jobs
     1    concatenate_files
     1    convert_to_upper_case
     2
rule convert_to_upper_case:
    input: a.txt
    output: a.upper.txt
```

(continues on next page)

(continued from previous page)

```

    jobid: 2
    wildcards: some_name=a

Finished job 2.
1 of 2 steps (50%) done

rule concatenate_files:
    input: a.upper.txt, b.upper.txt
    output: a_b.txt
    jobid: 0
    wildcards: second=b, first=a

Finished job 0.
2 of 2 steps (100%) done
Complete log: /home/mtb_upm/mtb_genomics_workshop/snakemake/example/.snakemake/log/
↳2018-07-05T135145.499632.snakemake.log

```

What we have learned so far: * How a simple Snakemake rule looks * How to define target files when executing a workflow * How to use named wildcards for writing generic and flexible rules.

Tip: You can name a file whatever you want in a Snakemake workflow, but you will find that everything makes much more sense if the filename reflects the file's path through the workflow, e.g. *sample_a.trimmed.deduplicated.sorted.bam*.

Visualization, logging and workflow management

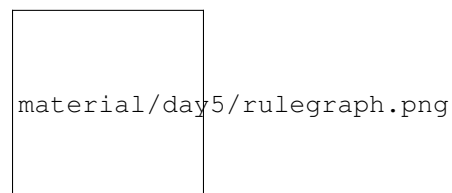
What we've done so far could have been quite easily done in a simple shell script that takes the input files as parameters. Let's now take a look at some of the features where Snakemake really adds value compared to a more straightforward approach. One such feature is the possibility to visualize your workflow.

Snakemake can generate two types of graphs:

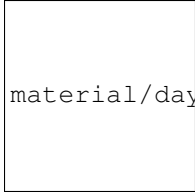
1. rules graph that shows how the rules are connected
2. jobs graph that shows how the jobs (i.e. an execution of a rule with some given inputs/outputs/settings) are connected.

Rule graph: First we look at the rule graph. The following command will generate a rule graph in the dot language and pipe it to the program `dot`, which in turn will save a visualization of the graph as a png file (if you're having troubles displaying png files you could use svg or jpg instead).

```
snakemake --rulegraph a_b.txt | dot -Tpng > rulegraph.png
```



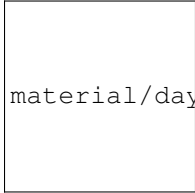
All the graph says is that the output from the rule *convert_to_upper_case* will be used as input to the rule *concatenate_files*. For a more typical bioinformatics project it can look something like this:



```
material/day5/rulegraph_complex.png
```

Job graph: The second type of graph is based on the jobs, and looks like this for our little workflow (use `-dag` instead of `-rulegraph`).

```
snakemake --dag a_b.txt | dot -Tpng > jobgraph.png
```



```
material/day5/jobgraph.png
```

The main difference here is that now each node is a job instead of a rule. You can see that the wildcards used in each job are also displayed. Another difference is the dotted lines around the nodes. A dotted line is Snakemake's way of indicating that this rule doesn't need to be rerun in order to generate *a_b.txt*. Validate this by running `snakemake -n -r a_b.txt` and it should say that there is nothing to be done.

We've discussed before that one of the main purposes of using a WMS is that it automatically makes sure that everything is up to date. This is done by recursively checking that outputs are always newer than inputs for all the rules involved in the generation of your target files.

Now try to change the contents of *a.txt* to some other text and save it. What do you think will happen if you run `snakemake -n -r a_b.txt` again?

```
$ snakemake -n -r a_b.txt

rule convert_to_upper_case:
  input: a.txt
  output: a.upper.txt
  jobid: 2
  reason: Updated input files: a.txt
  wildcards: some_name=a

rule concatenate_files:
  input: a.upper.txt, b.upper.txt
  output: a_b.txt
  jobid: 0
  reason: Input files updated by another job: a.upper.txt
  wildcards: first=a, second=b

Job counts:
  count  jobs
  1      concatenate_files
  1      convert_to_upper_case
  ...
```

Also generate the job graph and compare to the one generated above. What's the difference?

Now rerun without `-n` and validate that `a_b.txt` contains the new text. Note that `Snakemake` doesn't look at the contents of files when trying to determine what has changed, only at the timestamp for when they were last modified.

We've seen that `Snakemake` keeps track of if files in the workflow have changed, and automatically makes sure that any results depending on such files are regenerated. What about if the rules themselves are changed? It turns out that there are multiple ways to do this, but the most straightforward is to manually specify that you want to rerun a rule (and thereby also all the steps between that rule and your target). Let's say that we want to modify the rule `concatenate_files` to also include which files were concatenated.

```
rule concatenate_files:
    input:
        "{first}.upper.txt",
        "{second}.upper.txt"
    output:
        "{first}_{second}.txt"
    shell:
        "echo 'Concatenating {input}' | cat - {input[0]} {input[1]} > {output}"
```

If you now run the workflow as before you should get “Nothing to be done”, because no files involved in the workflow have been changed. Instead we have to force `Snakemake` to rerun the rule by using the `-R` flag. Let's try a dry-run.

```
snakemake a_b.txt -r -n -R concatenate_files
```

Note that the reason for the job is now “Forced execution”. You can target files as well as rules, so you would get the same result with `-R a_b.txt`. Whenever you've made changes to a rule that will affect the output it's good practice to force re-execution like this. Still, there can be situations where you don't know if any rules have been changed. Maybe several people collaborate on the same workflow but are using it on different files for example. `Snakemake` keeps track of how all files were generated (when, by which rule, which version of the rule, and by which commands). You can export this information to a tab-delimited file like this:

```
snakemake a_b.txt -D > summary.tsv
```

The contents of `summary.tsv` is shown in the table below (scroll to see the full table).

output_file	date	rule	version	log-file(s)	input-file(s)	shellcmd	status	plan
a_b.txt	Thu Nov 16 12:03:11 2017	concatenate_files			a.upper.txt, b.upper.txt	a.upper.txt b.upper.txt > a_b.txt	rule implementation changed	no update
a.upper.txt	Thu Nov 16 12:03:11 2017	convert_to_upper_case			a.txt	tr [a-z] [A-Z] < a.txt > a.upper.txt	ok	no update
b.upper.txt	Thu Nov 16 12:03:11 2017	convert_to_upper_case			b.txt	tr [a-z] [A-Z] < b.txt > b.upper.txt	ok	no update

You can see in the second last column that the rule implementation for `a_b.txt` has changed. The last column shows if `Snakemake` plans to regenerate the files when it's next executed. None of the files will be regenerated because `Snakemake` doesn't regenerate files by default if the rule implementation changes. From a reproducibility perspective maybe it would be better if this was done automatically, but it would be very computationally expensive and cumbersome if you had to rerun your whole workflow every time you fix a spelling mistake in a comment somewhere. So, it's

up to us to look at the summary table and rerun things as needed. You can get a list of the files for which the rule implementation has changed, and then force Snakemake to regenerate these files with the `-R` flag.

```
snakemake a_b.txt -R $(snakemake a_b.txt --list-code-changes)
```

There are a number of these `-list-xxx-changes` flags that can help you keep track of your workflow. You can list all options with `snakemake -help`. Run with the `-D` flag again to make sure that the summary table now looks like expected.

You might wonder where Snakemake keeps track of all these things? It stores all information in a hidden subdirectory called `.snakemake`. This is convenient since it's easy to delete if you don't need it anymore and everything is contained in the project directory.

By now you should be familiar with the basic functionality of Snakemake, and you can build advanced workflows with only the features we have discussed here. There's a lot we haven't covered though, in particular when it comes to making your workflow more reusable. In the following section we will start with a workflow for calling variants.

We've learned:

- How to use `-dag` and `-rulegraph` for visualizing the job and rule graphs, respectively.
- How to force Snakemake to rerun relevant parts of the workflow after there have been changes.
- How logging in Snakemake works.

Reference: Koster, J. and Rahmann, S. (2012). Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19), 2520–2522.

Resources:

- The Snakemake [documentation](#)
- Here is the *official* [tutorial](#)
- Further questions? Check out stack [overflow](#).
- There is a snakemake [Googlegroups](#).

Credit: [NBIS Reproducible research course](#).