DE NOVO ASSEMBLY PRACTICAL

Introduction

During this section you will learn how to run a *de novo* assembly on sets of paired FASTQ sequences of Hepatitis C Virus (HCV) data. *De novo* assembly is a process by which short reads are 'stitched' together into longer contiguous sequences called 'contigs'. Several algorithmic approaches have been taken, with their common goal being the identification of overlapping read sequences such that chains of reads can be elided into contigs. The Wikipedia page for assemblers provides an overview of the available programs and their varying approaches:

en.wikipedia.org/wiki/De_novo_sequence_assemblers

The practical is divided into three parts:

1. Using 'clean' data
   a. Perform assembly with **SPAdes**
   b. Re-run the assembly with altered **SPAdes** parameters
   c. Using **QUAST** to evaluate assembly outputs
   d. Using **BLAST** to look at contig identities
2. Using less clean data
   a. The importance of trimming (**trimmomatic**)
3. Looking at normalisation
   a. Using a kmer-dependent filtering step to normalise data before assembly

Command prompt

Throughout these practical notes, lines to be typed into the Unix shell are prefixed with `GCV2025:~$` in green. In the practical itself, '`GCV2025`' will be extended by text specific to the training shell, possibly including the current directory. This will not affect the running of the practical.

Results files

If time is running low, and completing all the assembly and analysis is looking unlikely, then there are pre-made results files available. In each directory (**`part1`**, **`part2`** and **`part3`**), there is a zipped directory called **`results.zip`**. Within these are outputs generated by running the commands on the data for that section. You are encourage to run the commands as much as possible however!

To access the pre-run data, whilst in the directory containing the **`zip`** file, run the following command:

```
GCV2025:~$ unzip results.zip
```

All the outputs of the tools used in this practical will be extracted into the directory, with the exception of BLAST results as these are obtained through a web interface.

De novo assemby with SPAdes

The HCV genome is small, at approximately 9.5kb, so the assembly itself should not take very long to perform. When run on much larger organisms such as bacterial or eukaryotic genomes, the time can increase considerably. To avoid the interfering factors mentioned in the lecture, the FASTQ sequences used in this first section are filtered. Less favourable sequence sets are explored in subsequent parts.

The following tools will be used:

SPAdes

A very popular tool used across many different organisms. Improved versions have been developed to address specific needs (e.g. metaSPAdes for metagenomic assemblies).

Bankevich A, et al. J Comput Biol (2012) 19(5):455-77
Nurk S, et al. Genome Research (2017) 27(5):824-34

QUAST

The outputs of SPAdes comprises multiple files, of which one contains the assembled contig list in FASTA format. This will be further analysed by this QC program, which generates a simple breakdown of assembly metrics.

Gurevich A, *et al*. Bioinformatics (2013) 29(8):1072-5

BLAST

This well-known online tool is used to query input sequence(s) against global sequence databases.

Altschul, SF, et al. J Mol Biol (1990) 215(3):403-10
Camacho C, et al. BMC Bioinformatics (2009) 10(1):421-9

A: Analysing 'clean' data

Access the folder **de_novo_assembly/files** and find the first pair of FASTQ files

```
GCV2025:~$ cd denovo_assembly/files
```

This directory contains three sets of FASTQ files. For the first assembly, we will use FASTQ files which have been through QC, trimming and filtering:

A_R1.fastq & A_R2.fastq

SPAdes: default kmers

We are going to perform a *de novo* assembly on this first set of FASTQ files using **SPAdes**. One of the optional parameters for **SPAdes** is a range of kmer sizes. Kmers are sequences of characters of length *k* sampled from one or more longer string(s). The default kmer sizes used by **SPAdes** are 21, 33, and 55, meaning that the program runs its assembly step three times, each time converting the sequence data into strings of a different length. After completion, it merges the results of all kmer sizes into a single output file. Data for each individual kmer are retained in folders within the output directory.

You can view the arguments **SPAdes** can take by typing the name of the tool:

```
GCV2025:~$ spades.py
```

We will run **SPAdes** with the default kmers to begin with:

```
GCV2025:~$ spades.py -t 4 -1 A_R1.fastq -2 A_R2.fastq -o A_output
```

These are explained below (taken from the usage information called up by the **-h** help command above):

-1 <filename>	file with forward paired-end reads
-2 <filename>	file with reverse paired-end reads
-t <int>	number of threads [default: 16]
-o <output_dir>	the name of the output directory

This shouldn't take too long to complete. Notice how **SPAdes** produces a highly verbose output whilst running! Navigate into the output directory (**cd A_output**) and see how **SPAdes** produces a large number of files. Here, we are chiefly interested in the contents of **contigs.fasta**.

To view the **contigs.fasta** file, use **less**:

```
GCV2025:~$ less contigs.fasta
```

There are several things you can do to investigate the output of the assembly:

- Return each header from the contigs file.The grep command finds lines in the input file (`contigs.fasta`) that contain the specified text (“>” in this case, i.e. all headers in a FASTA file):

```
GCV2025:~$ grep ">" contigs.fasta
```

- Return the the total number of contigs by counting the number of occurrences of “>”. Here, the **`-c`** flag is used to make **`grep`** return the *count of the lines* (not the text) instead of the lines themselves.

```
GCV2025:~$ grep -c ">" contigs.fasta
```

- Return the base count of the contigs. This is a more complicated command, so don’t worry if the notation seems unfamiliar[1].

```
GCV2025:~$ grep -v ">" contigs.fasta | wc | awk '{print $3-$1}'
```

Typing **`Ctrl`** and **`r`** together will perform a reverse search of your commands by bringing up the following in your command prompt:

```
(reverse-i-search)`':
```

By typing in, e.g. **`grep`**, and then iterating through the list with **`Ctrl`** and **`r`** together, you can reverse search through all of your previous commands that contained the selected word. This allows you to quickly find previous commands and avoids lots of retyping. Typing **`Ctrl`** and **`g`** together cancels the reverse search.

[1] The **`-v`** flag tells **`grep`** to return lines that *do not match* the search text (i.e. the sequences of a FASTA file as opposed to the headers). The **`|`** symbol is a “PIPE” and tells the shell to pass the output of the first command as the input to the second. Here, the second command is **`wc`**. This is “word count” and by default returns three columns containing the counts of newline characters, words and bytes respectively. In the **IVA** output, the sequence lines are not “wrapped”, i.e. after every 60 bases, a newline has been added. Consequently, the total number of bytes includes both bases *and* newline characters. The second PIPE passes the three columns from **`wc`** to **`awk`**, a low-level programming language that is used to print out the third column (**`$3`**, total bytes), less the number in the first column (**`$1`**, total newlines).

Questions

1. How many contigs and bases are in the file?

2. Are these numbers what you'd expect to see?

Quast: Assembly statistics

We will now investigate the quality of the assembly using a tool called **QUAST**. This tool gives a series of statistics about the contigs generated. This program takes as input the **`contigs.fasta`** files from the assemblies.

```
GCV2025:~$ quast.py contigs.fasta -o quast
```

Navigate into the **`quast`** output directory. Notice how **QUAST** produces several files. Many contain the same information but in various formats. Here we are interested in **`report.txt`**, which provides a breakdown of information regarding the contigs. Open using one of either **`more`**, **`less`** or **`cat`** commands, or view in **`nano`** if you are comfortable with this application.

```
GCV2025:~$ more report.txt
GCV2025:~$ less report.txt
GCV2025:~$ cat report.txt
GCV2025:~$ nano report.txt
```

Questions

1. How long is the largest contig?

2. How many contigs are over 1,000 bases long?

3. What is the total length of the contigs when only considering ones whose length is ≥1000 nucleotides?

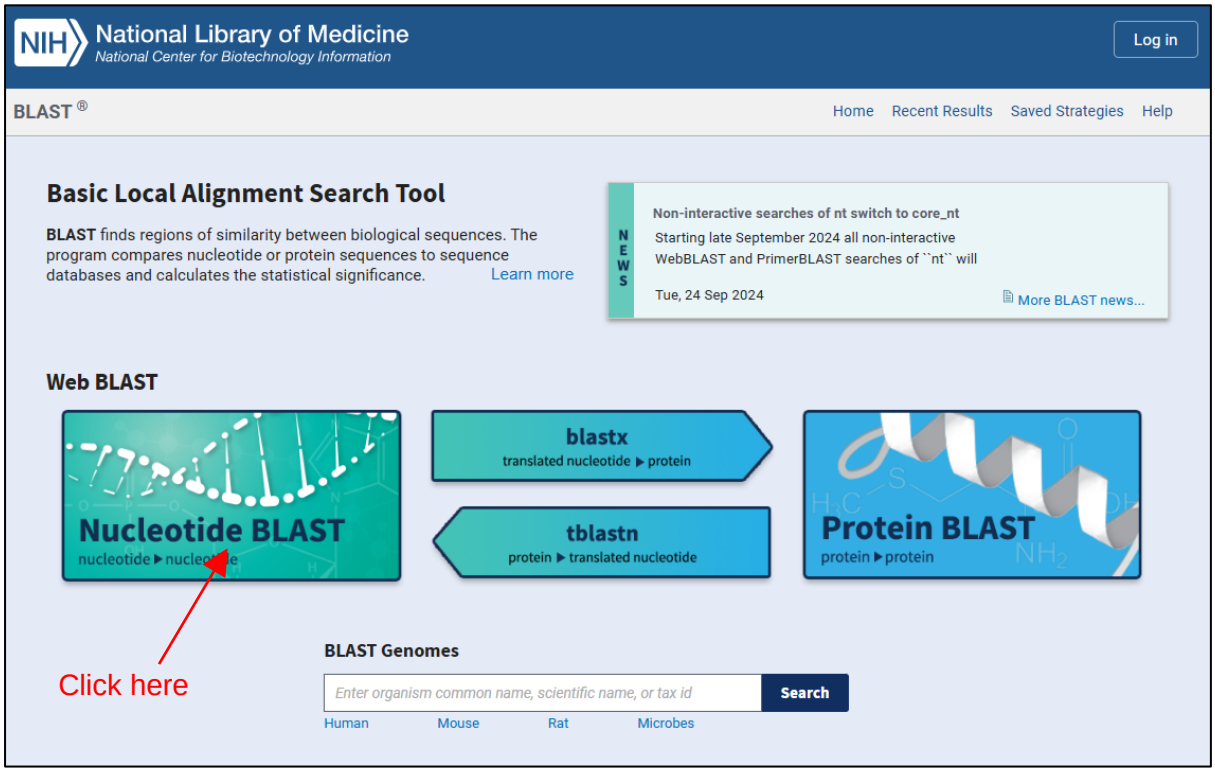
BLAST: Interrogating contig identity

After generating contigs, it is common to investigate their identity. Depending upon the source and treatment of the FASTQs submitted to the assembler, the identity of each contig could be

- a genome or sub-genomic region of a target of interest ✓✓✓
- the same, but from a non-pathogenic target of little interest ✓
- host genome or exome sequence ×
- a contaminant ×××

By using BLAST, we can investigate the identity of contig sequences. This can determine whether your contigs are HCV as expected or contamination. We can use blast by following this link https://blast.ncbi.nlm.nih.gov/Blast.cgi in a web browser.

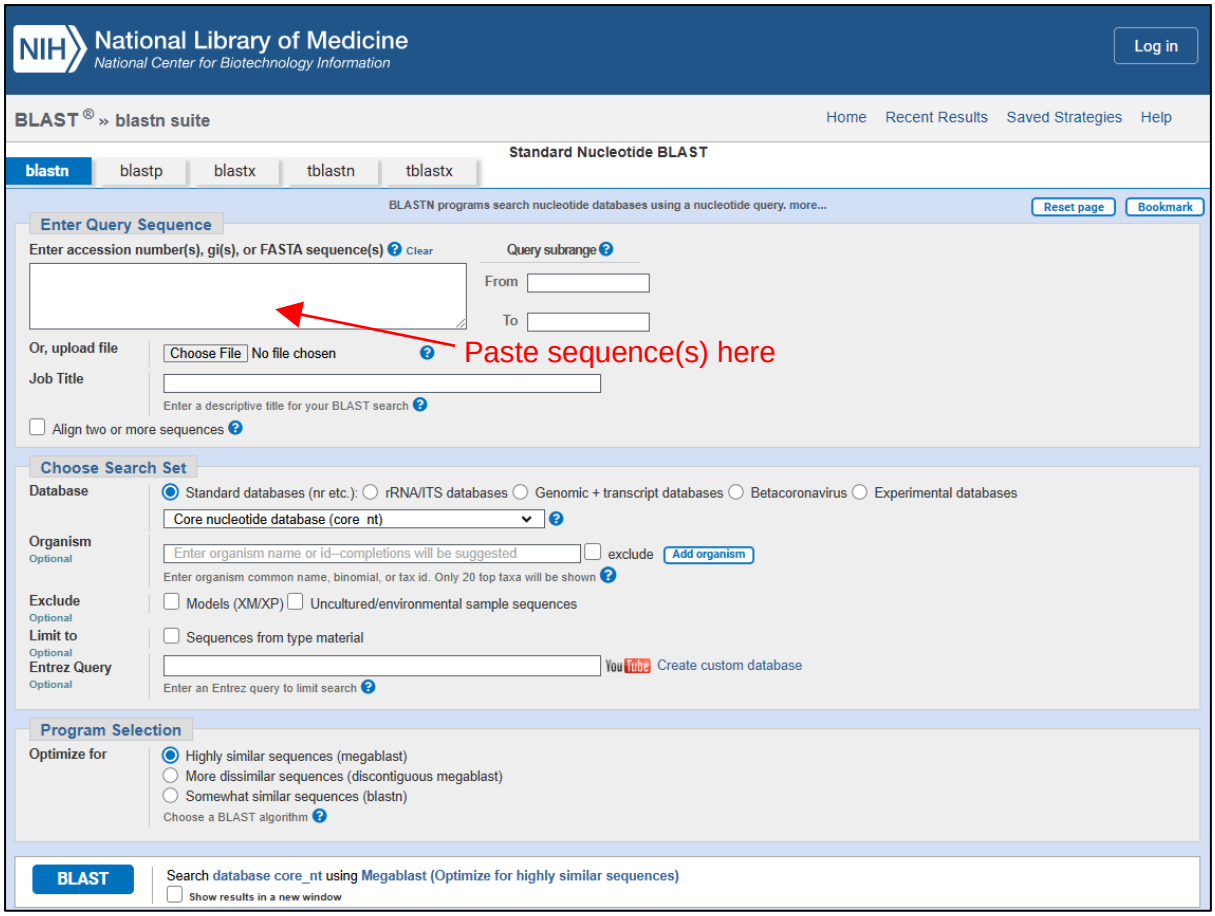
Click on **Nucleotide BLAST** on the homepage:



Open the **`contigs.fasta`** file from the first **SPAdes** run (this can be found in the **`A_output`** directory), either using a text editor or on the command line using **`cat`**.

```
GCV2025:~$ cat contigs.fasta
```

Copy & paste the first 5 contigs (i.e. the longest contigs) into the '***Enter Query Sequence***' box on **Nucleotide BLAST** (also known as **blastn**), and then hit **BLAST** at the bottom of the screen.



Once **BLAST** has completed, near the top of the page is a drop down menu in the '***results for***' section. This allows you to click the **BLAST** output for each of the 5 contigs.

Questions

1. Are these what you would expect?

2. What are the alignment scores like for the top hits (look at **`Query`** cover and **`Ident`** % for clues)?

SPAdes: long kmers

With many bioinformatics tools, the user can specify many operational parameters. It is worth spending a little time learning about these and experimenting with how they affect outputs. *De novo* assembly is no exception, and finding the correct parameters to suit your application is important.

Re-run the **SPAdes** assembly but this time change the kmer sizes to **`99`** and **`127`** as follows:

```
GCV2025:~$ spades.py -k 99,127 -t 4 -1 A_R1.fastq -2 A_R2.fastq -o A_output2
```

The usage of the kmer argument is as follows:

`-k <int,int,...>` comma-separated list of k-mer sizes (must all be odd and less than 128) [default: 'auto']

Run **QUAST** on the **`contigs.fasta`** file, and look at this **`report.txt`** output.

Question

1. What are the differences between the **QUAST** results using the altered kmer settings of the **SPAdes** runs?

2. Are there any differences when the "long kmers" **SPAdes** output is submitted to **BLAST**?

What do the other metrics in the **QUAST** report tell us?

`Total length`	Combined length of all contigs ≥*500* nucleotides in length.
`N50 / N75`	Collectively, all the contigs of this length or longer contain at least[2] 50% or 75% of all bases in the contig set.
`L50 / L75`	The number of contigs with length greater than or equal to **`N50`** or **`N75`**.

Some tools return an **`N95`** score. This is the same as **`N50`** / **`N75`** but at the 95th percentile.

[2] As there will be a single contig that tips the cumulative base count over 50%, the **`N50`** does not divide contigs into two halves with identical base counts.

B: Analysing less clean data

For this section, we will use a second pair of FASTQ files. Running them through **SPAdes** before and after 'trimming' will illustrate the value of cleaning the data.

SPAdes: raw data

Start by looking at the raw FASTQs:

B_R1.fastq
B_R2.fastq

Notice how the file sizes are considerably larger than the last sets. This means that the **SPAdes** command will take longer to complete. However, it shouldn't take longer than 5-10 minutes or so, even with the extensive set of kmer sizes here:

```
GCV2025:~$ spades.py -k 21,33,55,77,99,127 -t 4 -1 B_R1.fastq -2
B_R2.fastq -o B_output
```

Once completed, analyse the **contigs.fasta** file using **QUAST** as before

```
GCV2025:~$ cd B_output
GCV2025:~$ quast.py -o quast contigs.fasta
```

As before, open the contigs file, highlight the top 5 contigs, copy-and-paste these into **BLAST** and run as before.

Questions

1. How many contigs are there?

2. How big is the largest contig?

3. Note down the `N50`, `N75`, `L50` & `L75` scores.

   N50: **N75**: **L50**: **L75**:

4. To what organisms are the contigs **BLAST**ing?

   Contig 1:

   Contig 2:

   Contig 3:

   Contig 4:

   Contig 5:

5. Is this useful?

Trimmomatic: removing low quality read data

Navigate back to the **files** directory and run the following command:

```
GCV2025:~$ trimmomatic PE -threads 4 B_R1.fastq B_R2.fastq -baseout
Btrim.fastq LEADING:30 TRAILING:30 MINLEN:50
```

The trimming tool is very fast, and should complete in no time. Four files are produced:

Btrim_1P.fastq
Btrim_1U.fastq
Btrim_2P.fastq
Btrim_2U.fastq

The *second* and *fourth* contain unpaired forward and reverse reads respectively, whereas it is the *first* and *third* of these that contain the retained paired-end reads we will be carrying forward into the next assembly.

Firstly, run the **ll** command. The file sizes of the **trimmed_1P & _2P** FASTQ files are indistinguishable from the original **raw_R1** and **_R2** FASTQ files. Looking at the **trimmed _1U** and **_2U** files, we see that these file sizes are very small by comparison, and they are measured in kilobytes rather than megabytes.

By using the following **grep** and **awk** scripts, we can count the number of reads and bases in the FASTQ files, to see how trimming has affected the datasets. Substitute the trimmed filenames for the raw one to compare outputs[3].

```
GCV2025:~$ grep -c ^+$ B_R1.fastq
GCV2025:~$ awk 'NR%4==2{n+=length($0)}END{print n}' B_R1.fastq

GCV2025:~$ grep -c ^+$ Btrim_1P.fastq
GCV2025:~$ awk 'NR%4==2{n+=length($0)}END{print n}' Btrim_1P.fastq
```

Run **SPAdes**, followed by **QUAST** and **BLAST** (as per the previous page – in addition to changing the input filenames, don't forget to direct the **SPAdes** output to a new, unique folder such as **B_output2** or it will overwrite the first output!).

[3] The same *read counts* should be obtained from both the forward and reverse reads files, as they represent paired-end reads, i.e. two sequences per read. Some programs do not ensure read number equality after they've processed files – be very careful as unequal read counts will stop a lot of downstream programs from working.
For more information about the **awk** script, see the end of this file.

Questions

Repeat questions 1-4 from the raw reads run.

1. How many contigs are there?

2. How big is the largest contig?

3. Note down the `N50`, `N75`, `L50` & `L75` scores.

   `N50`: **`N75`**: **`L50`**: **`L75`**:

4. To what organism(s) are the contigs **BLAST**ing?

   `Contig 1`:

   `Contig 2`:

   `Contig 3`:

   `Contig 4`:

   `Contig 5`:

5. Was using **`trimmomatic`** of value here?

6. How many *reads* were discarded?

7. How many *bases* were discarded?

8. What percentages do these represent?

9. Why might more bases be being trimmed from the R2 than the R1 FASTQ file?

C: When trimming is not enough

Although only a tiny fraction of reads and bases were lost, the impact upon assembly was dramatic. Thinking about how *de novo* assembly works, it makes sense that at the termini of reads, where the overlap matching between reads takes place, erroneous bases have a major impact upon the assembler's ability to link sequences.

In the **`files`** directory is a third pair of FASTQ files from yet another HCV sample. This section aims to show how normalisation is a very useful tool when preparing datasets for assembly.

SPAdes: Raw data

Running **SPAdes** on the raw samples is likely to give suboptimal results. This can be confirmed by running **SPAdes** followed by **QUAST**ing and **BLAST**ing the output (note the kmer sizes). The command[4] below takes a few minutes to run, as these raw FASTQs are large.

```
GCV2025:~$ spades.py -k 99,127 -t 4 -1 C_R1.fastq -2 C_R2.fastq -o
C_output &> /dev/null
```

As expected, the contigs are large in number (over two thousand), short in length (max 658nt), and although four of the top five contigs **BLAST** to HCV, deriving a genome is clearly not possible.

SPAdes: trimmed data

To obtain trimmed reads, run Trimmomatic as before:

```
GCV2025:~$ trimmomatic PE -threads 4 C_R1.fastq C_R2.fastq -baseout
Ctrim.fastq LEADING:30 TRAILING:30 MINLEN:50
```

Run SPAdes again with the trimmed reads (or if time is short, **`unzip`** the results files as above, replacing '**`_raw`**' suffixes with '**`_trimmed`**'):

```
GCV2025:~$ spades.py -k 99,127 -t 4 -1 Ctrim_1P.fastq -2
Ctrim_2P.fastq -o C_output2 &> /dev/null
```

[4] The '**`&> /dev/null`**' bit at the end suppresses the (extremely) verbose screen output, dumping it in a folder that acts as a sort of rubbish bin for data. All the output is still available, should you be interested, in the **`spades.log`** file inside the output directory.

Questions

1. How many bases and reads were removed from the FASTQs by the trimming process?

	READS	BASES
R1		
R2		

2. Has trimming had any effect on the assemblies from this FASTQ set?

3. Are **`the`** **QUAST** scores noticeably different between **`raw`** and **`trimmed`**?

   `raw`

   `N50`: **`N75`**: **`L50`**: **`L75`**:

   `trimmed`

   `N50`: **`N75`**: **`L50`**: **`L75`**:

4. To what might the number following '**`cov`**' in the **SPAdes** contig names refer, and how might that information be useful?

SPAdes: normalised data

The trimming seems to have made little difference in this instance.

Now try again, using the same trimmed reads, but normalise them first:

```
GCV2025:~$ normalise.sh Ctrim_1P.fastq Ctrim_2P.fastq
```

This calls an in-house **shell** script that in turn acts as a wrapper for three **Python** script sfrom the **khmer** package:

interleave-reads.py
normalize-by-median.py
split-paired-reads.py.

The second of these is the meat of the process (see here for more information as to how this works) – the two flanking scripts merge paired end read files into a single read file and then un-merge it after normalisation. It produces two output files:

normalised_R1.fastq
normalised_R2.fastq

Using any of our read-counting commands (**grep**, **wc**, etc.) will reveal that over 95% of reads and bases have been removed when compared to the original C**trim_1P.fastq** and C**trim_2P.fastq** files. **SPAdes** should take less than a minute to run!

```
GCV2025:~$ spades.py -k 99,127 -t 4 -1 normalised_R1.fastq -2
normalised_R2.fastq -o C_output3 &> /dev/null
```

Run QUAST on the **contigs.fasta** file in **norm_output**, and **BLAST** the top contig(s) to investigate their identity.

Questions

1. Has normalising the data had any effect on the assemblies from this FASTQ set?

2. How do the **QUAST** scores look when compared to those of the previous two assemblies?

   normalised

   N50: **N75**: **L50**: **L75**:

3. What did **BLAST** reveal?

4. Should we consider the '**cov**' values to be as informative in this run as they would be in the two previous runs and why?

Further work

If you have finished all of the tasks, then try repeating the **BLAST**ing steps of this exercise, but using the **blastx** tool (upper middle of the **BLAST** home page) instead of the standard nucleotide **BLAST** (**blastn**).

1. Do the results differ?
2. Check the alignments below the list of matches. What has **blastx** done?
3. Is this useful for an HCV genomic sequence? How about an HIV one? When might **blastx** be more useful than **blastn** when looking at nucleotide sequences (think about non-targeted NGS)?

Why use *de novo* assembly, and when?

Reasons for running an assembly may include:

- A suitable reference sequence isn't always available
- High target variability means reference mapping isn't accurate enough. This is often the case in diverse RNA viruses such as HIV and HCV, where envelope and E1 sequences respectively are very diverse. Gaps in the mapped alignment across these regions are often caused by bioinformatics rather than by poor sequencing *per se*.
- We don't know what the target might be. Metagenomic approaches interrogate all nucleic acids in a sample, looking for one of many known and/or unknown pathogenic agents. *De novo* assembly allows short reads ostensibly from the same source to be collated into larger fragments, making organism identification and characterisation considerably easier.

In many pipelines, a *de novo* assembly precedes one or more reference mapping steps, with the output of the former acting as reference sequence(s) for the latter. Furthermore, in some pipelines a reference mapping can precede a *de novo* assembly. For example, in metagenomic datasets, a mapping step is often used to remove reads derived from host genetic material. A number of curated *Homo sapiens* sequences are publicly available for this sort of approach (e.g. GRCh37).

What affects its success?

A confounding factor in some assembly processes is the prevalence of closely related sequences spanning the same region of the target – a common feature of many RNA virus genome datasets as they are derived from *in vivo* quasispecies. The presence of many variant sequences in a dense sequence dataset from the same target region can confound an assembler's ability to find the correct reads to extend a contig chain. Rare variants can be discarded (i.e. minority superinfecting strains), or in the worst cases, the assembler is unable to generate a single genome-spanning contig and returns a long list of short assemblies.

A related phenomenon is adventitious similarity between two local regions, one in the virus target and one in the host or other source – the assembler can mistakenly link two similar sequences that were in fact derived from different targets. A preceding reference mapping to 'soak up' host reads can be beneficial in these situations. However, there is always a risk of losing on-target information, and a balance must be struck.

As with primer design for PCR, where terminal mismatches can inhibit amplification, with contig building, the frequently erroneous bases at the termini of reads have disproportionately confounding effects. To this end, it is imperative that the data is cleaned prior to assembly. The QC and trimming steps covered in a previous session are pre-requisites.

More about `awk`

`Awk` is a very old (1977!) programming language from the early days of Unix. It is very simple and useful for constructing quick scripts to interrogate repetitive, particularly tabular data. Essentially, an input file is read, line by line, and the program checks if a *pattern* is matched and if it is, it performs a *procedure* on the data contained within the line.

The two lines used in the Trimming section work as follows. The first script will count how many lines consist of only a plus sign – i.e. the third line of every FASTQ. The symbols **`/^+$/`** indicate that the *pattern* is a regular expression looking for the sequence "start-of-line (**`^`**), plus sign (**`+`**) and then end-of-line (**`$`**)". Searching for plus signs more generally (e.g. using **`/+/`**) will find the ones in some of the quality strings as well as the separator lines, hence overestimating the true read count. The *procedure* **`n++`** increments a counter, stored in a variable called *n*, by 1, but only when the expression is matched. Anything after the END statement only takes place once every line in the file has been scanned, i.e. after all the instances of a FASTQ read have incremented the counter, its final number is printed to the screen.

The second script looks for lines whose number within the file (i.e. 1, 2, 3, 4, 5…) when divided by 4 leaves a remainder of 2. In a FASTQ file, these are the lines containing the sequence strings. The *length* is added to the running variable *n*, which as in the first script, is printed at the end.

There are lots of online resources for learning a bit of **`awk`**, and it is a very useful and instructive tool for budding bioinformaticians! Try these:

www.grymoire.com/Unix/Awk.html
www.tutorialspoint.com/awk
www.tecmint.com/category/awk-command
gregable.com/2010/09/why-you-should-know-just-little-awk.html
en.wikibooks.org/wiki/An_Awk_Primer

More about Trimmomatic

As seen in its command line, our implementation of this tool specified three criteria:

LEADING:30

All bases at the start of the read are removed until the first base with a quality score ≥30 is encountered.

TRAILING:30

A similar process is performed at the 3' end of the read. More bases are lost here as on average, quality scores decline over the length of a read.

MINLEN:50

Any read with fewer than 50 bases remaining is discarded. If only one end fails this criterion, the partner is discarded into either the '`_1U`' and '`_2U`' output file.

These are performed in order, so that the **MINLEN** check follows any removal of **LEADING** and **TRAILING** bases. There are a number of other criteria that can be applied; see the **trimmomatic** manual webpage for a complete breakdown:

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

More about normalisation

The **`normalise.sh`** script used to prepare the third read set in Part 3 runs **`normalise-by_median.py`**, available as part of the **khmer** package. Its aim is to remove two sorts of read:

1 Redundant

In a FASTQ set with a high proportion of reads deriving from a short genome (or PCR amplicon), many of the reads contain very similar sequence information, and add very little extra information over those already seen in other reads from the same FASTQ file.

Whilst each read may contain vital minority variant information affecting e.g. drug resistance, for *de novo* assembly, this is superfluous data and slight variations can lead to too many branches being created that the assembler cannot reconcile.

2 Erroneous

Single base errors during sequencing (or during library preparation / PCR amplification) can also confound assemblers by generating multiple unproductive branches in the assembly graph. Reads containing errors can be discarded at this stage, but some applications aim to correct them at a later step. Obviously, this can be dangerous when minority variants and their frequencies are important.

The ‘kmer’ concept needs expanding upon a little to understand this process. Essentially, kmers are ‘strings’ of length *k*. A string is a computer science term for an ordered sequence of characters – it can be useful to equate them to *words*. In our case, the characters are IUPAC-encoded bases, and the strings are sequences of length *k*.

NGS analysis tools often look at the set of kmers within a FASTQ set. For example, some metagenomic tools like **centrifuge** and **kraken** look up each kmer in each read in one or more databases of those found uniquely in the genomes of different species, genera, families, or higher taxonomic divisions, in order to ‘bin’ each read according to its likely source organism. After scanning an entire read set, it may be discovered that a large proportion of reads have been binned together, suggesting that an instance of that organism group may have been present in the original sample. As well as metagenomic analysis, this type of tool is often used to automatically confirm the declared identity of sample types in high-throughput sequencing facilities. In our application, all kmers in the readset are counted rather than compared to a database.

It is important to note that the kmers are derived by stepping along the read sequence one base at a time, i.e. the first kmer encountered is the subsequence of the first read from positions 1 to *k*. The *second* kmer is not that from *k* + 1 to 2*k*, rather it is from 2 to *k* + 1. The third kmer is from 3 to *k* + 2, and so on, until *n* – *k* + 1 kmers have been derived from the read, where *n* is the length of the sequence. As might be imagined, the data structure to store this counting can be large, particularly for data from metagenomes or higher eukaryotic genomes, where the kmer diversity can be enormous. And although the time taken to count millions of kmers per sample ought to be considerable, there are some very clever memory allocation tricks that applications such as **jellyfish** use, and it can be extraordinarily rapid.

For normalisation, once the counting is complete, the aim is to retain at most only a few of each of the kmers in the FASTQ set, in whatever way they may be distributed amongst the

individual reads. With the **khmer** script , a **-C** flag sets this parameter (5 in our practical exercise, but it is often set to higher or lower values, depending upon the application). The script goes through each FASTQ file, one read at a time, and looks up the frequency of every kmer in the read sequence before determining the *median* frequency, i.e. the kmer frequency where 50% of kmers in that read have a higher frequency, and 50% have a lower frequency. If this number, multiplied by a random number between 0 and 1, exceeds the parameter defined by **C**, it is discarded.

Reads containing a large number of high-frequency kmers will have a high median kmer frequency, and thus the result of multiplying a high frequency by a random fraction is more likely to be higher than that specified by **-C**. Conversely, reads with rare kmers are likely to be retained as their median frequency will be closer to **-C**.

Two considerations are very important:

1 Normalisation significantly distorts the composition of the read set (intentionally!), and normalised FASTQ sets must not be used for variant frequency calling or any other quantitative analysis.

2 Some implementations of normalisation (not **khmer** though) use a random number generator for each read, such that they are 'heuristic' rather than 'deterministic', i.e. every time such a script is run, slightly different outputs are expected. Consequently, repeatability may be compromised.

   Note that this random number seed can be a feature of other applications, e.g. complex phylogenetic analyses. If reproducibility is important (hint: it always is!), it is usually possible to 'seed' the random number generator when calling the program, thus making the random numbers repeatable.

ANSWERS TO QUESTIONS

Please note that the exact numbers may vary slightly owing to changes in software versions and online BLAST databases between the setting of the questions and the course itself.

A: Analysing 'clean' data

SPAdes: default kmers

1. **59** contigs, **15,994** bases
2. **No**, they show that the HCV genome has not been assembled correctly

Quast: Assembly statistics

1. **4,014** bases
2. **3**
3. **9,061** bases

BLAST: Interrogating contig identity

1. **Yes** – HCV genomes & polyprotein sequences. The subtype of the matches is 1b.
2. Close to **100%** coverage and **92-97%** identity

SPAdes: long kmers

1. A single contig has been generated **9,474** bases in length.
2. Similar **accession** numbers, and **coverage** & **identity** scores.

B: Analysing less clean data

SPAdes: raw data

1. **741**
2. **1,584** bases
3. **906**, **749**, **7**, **11**
4.
    1) *Homo sapiens*
    2) HCV
    3) *Homo sapiens*
    4) *Homo sapiens / Gorilla gorilla (!)*
    5) HCV
5. Not really. Whilst it does confirm the presence of HCV in the sample dataset, there is not a complete genome, and there is plenty of human (and maybe gorilla!) sequence too.

Trimmomatic: removing low quality read data

1. **32**
2. **9,413**

3. **9,413**, **1,205**, **1**, **4**
4.
    a. HCV (subtype 1a)
    b. *Homo sapiens*
    c. *Homo sapiens*
    d. *Gorilla gorilla*
    e. *Homo sapiens*
5. **Yes**, a complete genome of HCV was achieved (contig #1)
6. **205**: **132,685** reads in the raw data, reduced to **132,480** in the trimmed data
7. R1: **34,688**: **19,173,487** bases in the raw, **19,138,799** in the trimmed
   R2: **71,022**: **18,625,291** bases in the raw, **18,554,269** in the trimmed
8. Reads: **0.155%**
   R1 bases: **0.181%**
   R2 bases: **0.381%**
9. Because quality declines across the sequencing process, R2 reads have lower quality scores overall than those in R1.

C: When trimming is not enough

SPAdes: trimmed data

1. Reads: 334,401 – 334,291 = **110** (**0.033%**)
   R1 bases: **116,762** (**0.234%**)
   R2 bases: **189,795** (**0.381%**)
2. **Almost none** - both have a very large number of short contigs. In fact, trimming seems to have made the metrics slightly worse.
3. **No**

	N50	**N75**	**L50**	**L75**
Raw	609	556	4	5
Trimmed	590	556	2	3

4. ‘cov’ refers to the depth of coverage of that contig, the average number of assembled reads covering each base position. A higher number indicates more reads were assembled per unit length. Look back at the contigs from Part 2 and see how the HCV contigs compare to the *Homo sapiens* contigs in this instance.

SPAdes: normalised data

1. **Yes**, there now only **15** contigs, with one of **9,489** bases.
2. **Better all round**.

	N50	**N75**	**L50**	**L75**
Normalised	9,539	9,539	1	1

3. A **full HCV genome** has been assembled (subtype 3a).
4. **No** – normalisation has profoundly disrupted the relationship between cov and the corresponding read density in the original FASTQ.