General instructions

Instructions for filling in the worksheets:

- Worksheets include questions (some are for refreshing the knowledge, whereas some will probably require a little bit of literature review) and practical work, where you carefully insert all codes and as much as possible comments.
- Each step of the bioinformatics analysis should be well documented and explained why it was performed and why each parameter/option of the command was used.
- Bioinformatics tools which you will use have several options. I encourage you to explore why they are used for. Make sure that you add this to the report as well.
- Comments regarding the worksheets, how can be improved, what should be added, etc. are welcome.

> Lines starting with `$` are commands that should be run in the terminal. You are meant to run everything that comes after `$`.

Setting up your environment

Virtual environments are great, because they let you have separate environments for separate proejcts. This is advantageous, since one project could rely on a certain package version 3, while some other may require version 4.

Conda

An advantage that **Conda** provides is not only for managing Python libraries, but also command line tools. This can make the tool's instalation process uniform and more generalized for users that don't work on the same systems.

I recommend installing Conda with these instructions:
docs.conda.io/projects/conda/en/latest/user-guide/install/index.html.
Essentially the difference between Miniconda and Anaconda is that with Miniconda you have to install many tools manually. Install whichever you like.

Bioconda

To use certain bioinformatics tools, we need to use the **Bioconda** channel. No installation is needed, only this 3 commands that alter your `.condarc` configuration:

```
$ conda config --add channels bioconda
$ conda config --add channels conda-forge
$ conda config --set channel_priority strict
```

Virtual environment

Finally, we can create an environment for this project using:

```
$ conda create --name ans
```

Answer the prompt with yes to create an environment and then **activate** the environment with `$ conda activate ans`. Activation of the environment is local to a terminal session, so you will have to use this command again if you open another terminal instance.
To deactivate it simply run `$ conda deactivate`.

Finding our data

During this course we were tasked to work on the RNA-seq dataset linked to the article titled *"Wolbachia pipientis modulates germline stem cells and gene expression associated with ubiquitination and histone lysine trimethylation to rescue fertility defects in Drosophila"*.

BioProject accession number

First thing to do is to find this article on **NCBI** or **PubMed**. This can be done with a quick Google search:
https://academic.oup.com/genetics/article/229/3/iyae220/7934994#510949052.

In the article, find the section *Data availability*. There will be an accession number for a **BioProject** (a BioProject is a collection of biological data related to a large-scale research effort).

SRA accession number

We can search for sequencing data related to this BioProject through the command line using **NCBI Entrez Direct tool**. First we install it in our environment with `$ conda install bioconda::entrez-direct`. Then we make a query and save the output in `csv` format.

```
$ esearch -db pubmed -query PRJNA1166928 | efetch -format runinfo >
runinfo.csv
```

If we take a quick look at this file, it has many different columns, while we are only interested in the SRA (sequence read archive) accession numbers. We can extract the first column with `cut` on the output of `tail` that will skip the first line (the header):

```
$ tail -n +2 runinfo.csv | cut -d ',' -f1 > SraAccList.txt
```

- `-n +2` tells `tail` to start displaying from the second line
- The `-d` flag is used to specify a delimiter and the `-f1` tells `cut` to extract the first field.

Questions regarding research and sequencing dataset

What is the aim of the study, described in the scientific article?

Provide some info about the dataset you will work with and which sequencing technology was used.

Which kit was used for sequencing library preparation? Does this kit preserve strand information (stranded library) or not?

What is the advantage of stranded mRNA library preparation compared to non-stranded library?

Downloading data

Each of us students had to choose one accession number. I choose **SRR30833097**. Save this into a file with `$ echo "SRR30833097" > OurAcc.txt`.

Prefetch

Sequencing data is obtained from the SRA database with the SRA Toolkit. It can be installed with `$ conda install -c bioconda sra-tools`.

```
$ prefetch --option-file OurAcc.txt
```

Prefetch is used to obtain *Runs* (sequence files in compressed SRA format). The `--output-file` flag is used to use a file with a list of accession numbers as input. The command creates a directory named after the given accession number, where the downloaded files reside.

The prefetched runs can be converted into FastQ format using `fasterq-dump`, that takes the created directory as input:

```
$ fasterq-dump --skip-technical SRR30833097/
```

`--skip-technical` returns only biological reads.

Since we have only single-end sequences, it should output a single `.fastq` file in the current directory. You can check that the line count matches 18149460 with:

```
$ wc -l SRR30833097.fastq
```

Generating a quality report

a) Run FastQC over your dataset. Explain the results

Command line FastQC

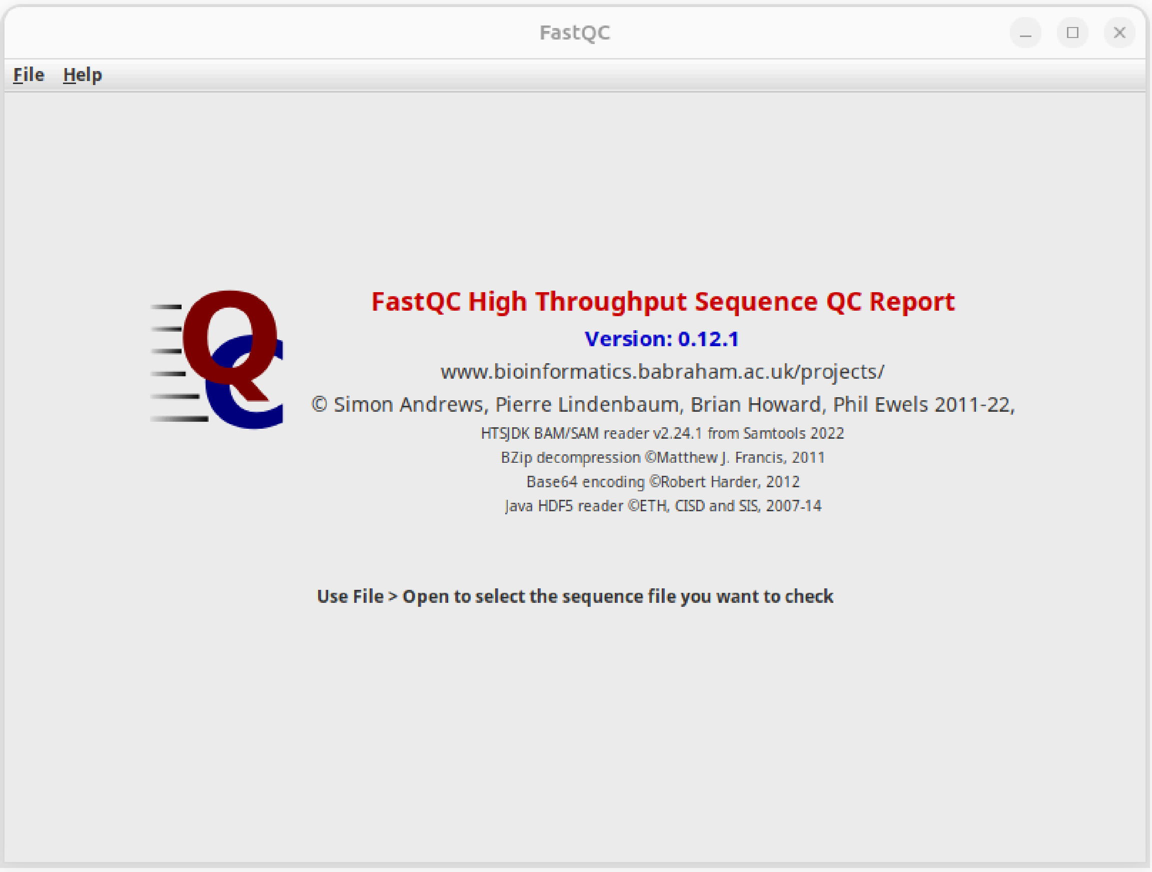
To get a **full report** on all sequences in our dataset, we can use **FastQC tool**. Install it with `$ conda install -c bioconda fastqc` and run it on the `.fastq` file.

```
$ fastqc SRR30833097.fastq
```

The quality report is the generated `.html` file. To view the rendered file, you can open it with your browser (e.g. `opera *.html`).

GUI FastQC

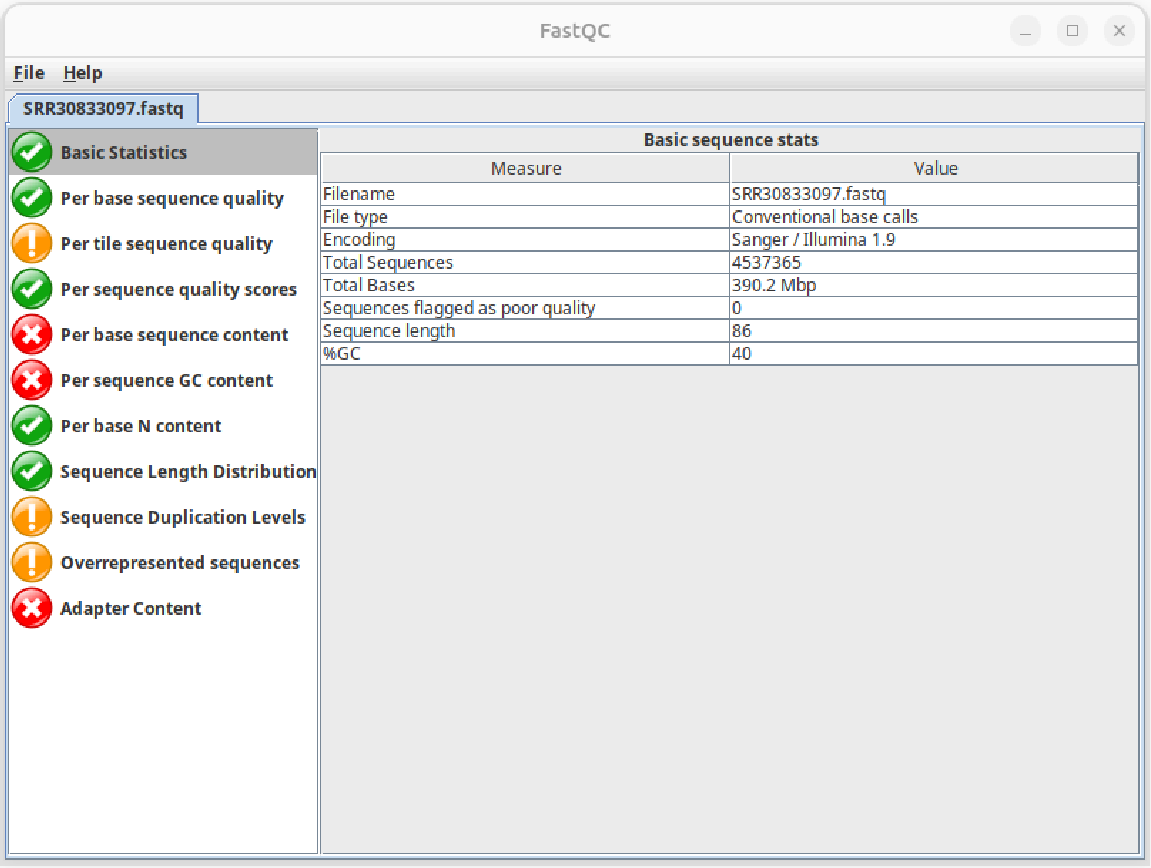
Alternativelly, you can open a **graphical user interface** of FastQC tool by executing only `$ fastqc`. This will open a new window where you can open your `.fastq` file and generate and view the report.



FastQC High Throughput Sequence QC Report
Version: 0.12.1
www.bioinformatics.babraham.ac.uk/projects/
© Simon Andrews, Pierre Lindenbaum, Brian Howard, Phil Ewels 2011-22,
HTSJDK BAM/SAM reader v2.24.1 from Samtools 2022
BZip decompression ©Matthew J. Francis, 2011
Base64 encoding ©Robert Harder, 2012
Java HDF5 reader ©ETH, CISD and SIS, 2007-14

Use File > Open to select the sequence file you want to check

Click on `File > Open` and select your file. When it's finished, you should see the report.



- ✓ Basic Statistics
- ✓ Per base sequence quality
- ! Per tile sequence quality
- ✓ Per sequence quality scores
- ✗ Per base sequence content
- ✗ Per sequence GC content
- ✓ Per base N content
- ✓ Sequence Length Distribution
- ! Sequence Duplication Levels
- ! Overrepresented sequences
- ✗ Adapter Content

Basic sequence stats

Measure	Value
Filename	SRR30833097.fastq
File type	Conventional base calls
Encoding	Sanger / Illumina 1.9
Total Sequences	4537365
Total Bases	390.2 Mbp
Sequences flagged as poor quality	0
Sequence length	86
%GC	40

Explaining the results

b) Exchange FastQC results with your colleagues and run MultiQC to get a joined report for all datasets

Checking quality of sequences

Questions

a) How is a fastq file composed?

FastQ files are text files that combine **FASTA formatted sequences** with their **quality scores** and is the standard format for storing the output of high-throughput sequencing instruments.

Since FastQ files bundle quality scores of sequences, we can take a quick look at the first stored sequence with `$ head -n 4 SRR30833097.fastq`. This will output

```
@SRR30833097.1 NB500947:1144:HM5GMBGXH:1:11101:4455:1091 length=86
GGCGGTCGAGTGCCTCACAGTGTATCAAGGGTNGGCCACGNTCCTNACTAATNGNGGCTNNTTGCGCCATCGTC
TCANGCAATGTT
```

```
+SRR30833097.1 NB500947:1144:HM5GMBGXH:1:11101:4455:1091 length=86
AAAAAEEEEEEEEEEEEEEEEEEEEE<E//EEE#EEEEEE<#EEEE#E/AEEE#/#EEEE##EEE/EEAEEEEAE
6EE#/EEEAAEA
```

Each entry starts with `@` and is followed by a sequence identifier and some other information about the sequence. The line directly below it shows the raw sequence. The `+` line again can contain information about the sequence and below are the quality values for each nucleotide.

b) How can I count the number of reads in a fastq file? Describe different ways to perform that.

BASH

Using `grep` and `wc`. Grep is used to search for patterns using regular expression in a file. With `'^@'` we search for every line that starts (`^`) with `@`. It outputs every line that matches our pattern. We pipe this in `wc -l`, as we used before, to count the number of lines.

```
grep -e '^@' SRR30833097.fastq | wc -l
```

AWK

The AWK language is useful for pattern scanning and text proccessing. It proccesses files line by line just like `grep`. In this case we don't look for a certain identifier, but we use the fact that every sequence occupies 4 lines to our advantage.

`NR` (number of records) is a builtin awk-variable. It records lines processed so far. When reading line by line from file, this is essentially the current line number.

With `NR % 4 == 1` we calculate if the current line is the first line of 4. This is true for lines 1, 5, 9... which are lines that contain `@` and represent one sequence each.

The filtered lines are returned, which we can again pipe into `wc -l`.

```
awk 'NR % 4 == 1' SRR30833097.fastq | wc -l
```

Python

I made two scripts, one that uses pattern matching like the BASH command and one that uses calculating the mod of line numbers. They are a little more verbose than the other two options, but still pretty straightforward.

I also measured time needed to process the file to determine which approach is faster.

Pattern matching

```
# lines.py

import sys
from time import time

if len(sys.argv) != 2:
    raise Exception("Usage: python3 lines.py <fastq-file>")

fname = sys.argv[1]

print("Processing file...")
start_time = time()

with open(fname, "r") as file:
    n = 0
    for line in file:
        if line[0] == "@":
            n += 1

end_time = time()
final_time = end_time - start_time
print(f"number of sequences in file: {n}")
print(f"time: {final_time}")
```

The program outputs:

```
$ python3 lines.py SRR30833097.fastq
Processing file...
number of sequences in file: 4537365
time: 6.220864295959473 sec
```

Line count calculation

These are the only altered lines.

```
# nlines.py
# snip

with open(fname, "r") as file:
    data = file.readlines()
    n = len(data) / 4

# snip
```

```
$ python3 nlines.py SRR30833097.fastq
Processing file...
```

```
number of sequences in file: 4537365.0
time: 4.813834190368652 sec
```

The second approach was faster by about 1.4 seconds, which is a difference, but it's not huge, since we are already using a file with 18 million lines.
The first one may be more robust since it doesn't make assumptions about line content, while the second one *assumes* we have lines that can be evenly divided by 4.

C instead of Python?

Since Python is slow by its nature, maybe a simple C program will do this faster.

Pattern matching

```c
// clines.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define MAX_LINE_LEN 100

int main(int argc, char *argv[]) {
  if (argc != 2) {
    printf("Usage: ./clines <fastq-file>");
    exit(1);
  }

  FILE *file = fopen(argv[1], "r");

  if (file == NULL) {
    perror("Error opening file.");
    exit(1);
  }

  int n = 0;              // line counter
  char line[MAX_LINE_LEN]; // line buffer

  // for measuring time more precisely than time_t
  struct timeval start_time;
  struct timeval end_time;
  double final_time = 0;

  printf("Processing file...\n");
  gettimeofday(&start_time, NULL);

  while (fgets(line, sizeof(line), file) != NULL) {
    if (line[0] == '@') {
      n++;
    }
  }

  gettimeofday(&end_time, NULL);
  // first term is difference in seconds
  // second term is diference in microseconds converted to seconds
  final_time = (double)(end_time.tv_sec - start_time.tv_sec) +
               (double)(end_time.tv_usec - start_time.tv_usec) /
1000000.0;

  printf("sequences in file: %d\n", n);
  printf("time: %f sec\n", final_time);

  return 0;
}
```

When we compile and run this program, we get this:

```
$ gcc clines.c -o clines && ./clines SRR30833097.fastq
Processing file...
sequences in file: 4537365
time: 1.321521 sec
```

Now this is a great improvement from the Python scripts. Will the second approach with calculating the mod of lines be faster?

Line count calculation

These are the only altered lines.

```
  // nclines.c
  // snip

  while (fgets(line, sizeof(line), file) != NULL) {
    n++;
  }
  n = n / 4;

  // snip
```

```
$ gcc nclines.c -o nclines && ./nclines SRR30833097.fastq
Processing file...
sequences in file: 4537365
time: 1.296595 sec
```

It's faster by a fraction. Again, not a big difference, but faster nevertheless, since we skip the comparisons for every line and do only one operation more after we read the whole file.

c) What about the quality of your reads?

d) Describe your fastqc and/or multiqc and interpret the results (https://mugenomicscore.missouri.edu/PDF/FastQC_Manual.pdf)