Exercise 1:

use **man** to get more information about the head command; find and write down the head command with the correct parameter to display not only the first 10 lines of the poem but the entire poem (20 lines)

```
$ head -n 20 tmp/poem.txt
```

How many lines you see?

Two lines.
His **house** is in the village though;
To stop without a farm**house** near

Try to type **less p** in the command line and then press [tab] 4 times. Describe what has happened each time you press [tab].

1st tab the command line was autocompleted to "poems" directory.
2nd tab the command line was autocompleted to "poems/poem" file
3rd nothing because there is ambiguity (different UNIX systems may behave differently)
4th the command displays the two possibilities that fits "poems/poem", which are "poem2.txt" and "poem.txt"

Exercise 2

What is the standard **input** and standard **output** and of the command "*grep*" in the example above?

The standard input is the third column of the data.tsv file.
The standard output is are only the column that says sample2

Try it yourself, write a command that reads poems.txt file and prints only the lines that include the word "house" (use **cat** and **grep**)

```
$ cat tmp/poem.txt | grep house
His house is in the village though;
To stop without a farmhouse near
```

Exercise 3

What has been displayed?

The program generated the "Hello World" text on the screen.

Congratulations! So called "Hello world" programs which simply write "Hello world" on the screen are often the first programs you will write in any programming language. Looks simple, but it is often the hardest part of learning a new language. The fact that you have written it means that you have everything installed correctly, know the syntax, know where to write a code, how to compile, deploy, and finally, execute it. Think

how much work would it be to write an app for your mobile device that simply displays “Hello World” on the screen and how cool it would be?

Exercise 4

How did the name of the file change when you unzipped it?

The “.gz” extension was removed from the file name.

Count the number of proteins in the file using commands you have learned. In the fasta format file the first line of each protein record starts with “>” character.

```
cat human_proteome.fa | grep ">" | wc -l
19566
```

As you probably have noticed some of the commands can process either standard input or take a file as an input parameter, which means you can write certain commands in two ways. In the above solution, grep takes input from its standard input. But grep can also take its input from a file directly:

```
grep ">" human_proteome.fa | wc -l
```

It’s up to you which one you want to use. Both are equally correct.

Exercise 5

Explore the downloaded file using **`less`** command. Note which column hold abundance numbers.

The protein abundance is in the first column labled “parts per million”

Open manual of the `sort` command, look for option “k” and “n”, “r” what do they do?

k – specifies which column has to be sorted
n – specifies that sorting is numeric
r – reverses the sorting order

Remove the “n” parameter and run the sort command again. Explain what has happened.

The sorting is now alphabetic, which means that the numbers starting with higher digits are first.

Exercise 6

What is the output?

the output is numbers from 1 to 100

Exercise 7

Using the commands you have learned print out the number of files in the directory

```
$ ls -1 | wc -l
44
```

By utilizing the same concept of "**for**" loop from the previous exercise, can you try to iterate over all fasta files in the directory, print their name and the number of sequences in each file?

`for filename in *.fa`

You can put more than one command between **do** and **done**, one line would print the file name, the other will print the sequence count.

Write a shell script (count.sh):

```
#!/bin/bash
for filename in *.fa
do
      echo $filename
      cat $filename | grep ">" | wc -l
done
```

Change permission:

```
$ chmod +x count.sh
```

Execute:

```
$ ./count.sh | tail
```

```
886882.fa ← filename
5406 ← number of proteins in the proteome
911045.fa
5070
927666.fa
1907
927691.fa
1928
983920.fa
3801
```

Exercise 8:

Explore the file using **less**. Use **grep** to find out how many chromosomes are present in the file. Use **grep** (-v) to only print out the genomic sequence. How large is the genome?

```
$ cat dd.fasta | grep ">" | wc -l
11
```

There are 11 chromosomes

```
$ cat dd.fasta | grep -v ">" | wc -m
34910453
```

The genome is 34910453 nucleotides long.

How many reads are in the file?

```
$ cat rbaseq.fastq | grep "@" | wc -l
```

There are 5000 reads in the FASTQ file.

Exercise 9

Is the sequence "AAAAAGAGATACAT" present in the DD genome (`dd/dd.fasta`)?

```
$ grep AAAAAGAGATACAT dd.fasta
```

Yes, the sequence is in the genome.

Does **bowtie2** find more alignments compared to grep? Why could that be?

1. Bowtie allows for mismatches.
2. Bowtie matches sequences on the reverse strand
3. Grep won't be able to match a sequence divided on two lines

Exercise 10

The results are returned in SAM format and stored to the dd.sam file. How many reads align?

The output shown after the computation shown 2553 sequences aligned exactly one time and 2113 align more than one time, together 4666 aligned sequences

Note that the information show was actually displayed on screen as *standard error* as it was not redirected by ">" to the dd.sam

Alternatively you can write the following command to count the number of aligned sequences:

```
cat dd.sam | grep -v "@" | grep chr | wc -l
4666
```

where `-v @` omits the header and `grep chr` prints out only the lines which aligned to the chromosome utilizing the fact that all the chromosomes have “chr” prefix.

 Did they change the results?

Among others changing number of mismatches and seed length sequence changes number of aligned reads.

```
bowtie2 -N 1 -x dd -U rnaseq.fastq > dd.sam
```