Exercises bash scripting

Download the files:

https://www.dropbox.com/s/4x34u51vn1za17f/bashScripting.zip

Work locally on your laptop

The following instructions have been tested on Ubuntu. Mac and Windows users can use Ubuntu in the Virtual Machine. All programs and packages are already installed within the Virtual Machine.

1. Copy the zipped data to your computer:
   ```
   wget https://www.dropbox.com/s/4x34u51vn1za17f/bashScripting.zip
   ```

2. Unzip the data:
   ```
   unzip bashScripting.zip
   ```

Exercise 1:

a) Write a script to unzip all files in the "Data" folder.
b) Write a script to go through all fastq files in the folder and count the number of reads in the files (hint: line number divided by 4 gives the number of reads) and write those to a file called "readNumber.txt"

Exercise 2:

a) Write a script which renames all unpaired files (samples with no R2 files) to "*_unpair.fastq".
b) Merge all unpaired files into one file called "Unpair.fastq"

Exercise 3:

Write function to compress a file (keep the original file) and moves it to a specified folder. Make sure that you check if the folder exists and if not create it. Apply it for all fastq files in the folder and move them to a folder called "compressed"

Exercise 4:

a) Write a script to convert the "ERR000061.1_R1.fastq" file to a FASTA file
b) Turn the above script into a function and apply it to all fastq files in your folder

Exercise 5: Setup a pipeline for read quality check, trimming and mapping using bash scripting

We will work with paired-end data sets of different insertion sizes (200, 500). The reads come from two human samples from the 1000 genomes project. We will setup a bash pipeline, which will include read quality check, read trimming and read mapping against the human reference genome (chr20). With this exercise you will learn step by step how pipelines are setup and generalized using bash scripting.

The very basic bash script looks as follows (**script.sh**, included in the download zip; data files can be found within the data_script folder):

```
#!/bin/bash

cd /home/student/Documents/bashScripting


#1. Quality check: FastQC
##########################
mkdir fastQCResults
fastqc -t 4 -o fastQCResults
/home/student/Documents/bashScripting/data_script/ERR000064_200_1.fastq
fastqc -t 4 -o fastQCResults
/home/student/Documents/bashScripting/data_script/ERR000064_200_2.fastq
fastqc -t 4 -o fastQCResults
/home/student/Documents/bashScripting/data_script/ERR000061_500_1.fastq
fastqc -t 4 -o fastQCResults
/home/student/Documents/bashScripting/data_script/ERR000061_500_2.fastq


#2. quality trimming: Trimmomatic
##################################
# - remove leading and trailing low quality basses (<3) or N
# - 4 base sliding window -> remove when average quality is < 15
# - remove reads which are shorter than 20 bp
java -jar /usr/share/java/trimmomatic.jar PE -threads 4
/home/student/Documents/bashScripting/data_script/ERR000064_200_1.fastq
/home/student/Documents/bashScripting/data_script/ERR000064_200_2.fastq
ERR000064_200_1_trimPair.fastq ERR000064_200_1_trimUnpair.fastq
ERR000064_200_2_trimPair.fastq ERR000064_200_2_trimUnpair.fastq LEADING:3
TRAILING:3 SLIDINGWINDOW:4:15 MINLEN:20
java -jar /usr/share/java/trimmomatic.jar PE -threads 4
/home/student/Documents/bashScripting/data_script/ERR000061_500_1.fastq
/home/student/Documents/bashScripting/data_script/ERR000061_500_2.fastq
ERR000061_500_1_trimPair.fastq ERR000061_500_1_trimUnpair.fastq
ERR000061_500_2_trimPair.fastq ERR000061_500_2_trimUnpair.fastq LEADING:3
TRAILING:3 SLIDINGWINDOW:4:15 MINLEN:20
```

```
#2. Quality check: FastQC
##########################
#merge unpaired files
cat ERR000064_200_1_trimUnpair.fastq ERR000064_200_2_trimUnpair.fastq >
ERR000064_200_trimUnpair.fastq
cat ERR000061_500_1_trimUnpair.fastq ERR000061_500_2_trimUnpair.fastq >
ERR000061_500_trimUnpair.fastq

fastqc -t 4 -o fastQCResults ERR000064_200_1_trimPair.fastq
fastqc -t 4 -o fastQCResults ERR000064_200_2_trimPair.fastq
fastqc -t 4 -o fastQCResults ERR000061_500_1_trimPair.fastq
fastqc -t 4 -o fastQCResults ERR000061_500_2_trimPair.fastq
fastqc -t 4 -o fastQCResults ERR000064_200_trimUnpair.fastq
fastqc -t 4 -o fastQCResults ERR000061_500_trimUnpair.fastq


#5. map reads to reference: Bowtie2
###################################
# index reference file
bowtie2-build /home/student/Documents/bashScripting/data_script/chr20.fa
/home/student/Documents/bashScripting/data_script/chr20

bowtie2 --fast-local -p 4 -q --phred33 -I 50 -X 350 --no-unal -x
/home/student/Documents/bashScripting/data_script/chr20 -1
ERR000064_200_1_trimPair.fastq -2 ERR000064_200_2_trimPair.fastq -U
ERR000064_200_trimUnpair.fastq -S ERR000064.sam
bowtie2 --fast-local -p 4 -q --phred33 -I 350 -X 650 --no-unal -x
refPath/chr20 -1 ERR000061_500_1_trimPair.fastq -2
ERR000061_500_2_trimPair.fastq -U ERR000061_500_trimUnpair.fastq -S
ERR000061.sam
```

 Try to understand what the script is doing.

To run this script we first need to make the file executable:

```
chmod +x script.sh
./script.sh  #you don't need to run it at that stage
```

a) Set up variables for every program path

This will allow you to quickly change the path to the programs if a new version is out (you just need to change it once in the script instead of multiple times).

- A variable can be defined as follows: name=hello
- And can be called by typing ${name}, eg: echo ${name}

So the path to the program can be defined in a variable:

```
#!/bin/bash
pathFastqc=fastqc
...
```

And the variable can be called at the point the program should be executed:

```
${pathFastqc} -t 4 -o fastQCResults ERR000064_200_1.fastq
```

 Do that for every program in the file

b) Set up variable for working directory

Most of the times you want to store all your results into one directory. Thus it makes sense to define a variable pointing to this working directory. This will also allow you to quickly change the working directory.

```
#!/bin/bash
pathFastqc=fastqc
...
~~cd /home/student/Documents/bashScripting~~
workPath=/home/student/Documents/bashScripting
mkdir ${workPath}  #creates working directory if it doesn't exist
cd ${workPath}     #change to working directory

#1. Quality check: FastQC
##########################
mkdir ${workPath}/fastQCResults
${pathFastqc} -t 4 -o ${workPath}/fastQCResults
/home/student/Documents/bashScripting/data_script/ERR000064_200_1.fastq
...
```

 Make sure that you write all the output of your pipeline into the working directory

c) Set up a variable for data directory

Define a variable giving the path to the directory where the original data is stored (in our case the fastq files)

```
...
dataPath=/home/student/Documents/bashScripting/data_script

#1. Quality check: FastQC
##########################
mkdir ${workPath}/fastQCResults
${pathFastqc} -t 4 -o ${workPath}/fastQCResults
${dataPath}/ERR000064_200_1.fastq
...
```

 Apply it to the whole pipeline

d) Integrate a "for-loop" in the 1. Step (quality checking).

"for-loops" can be used to iterate over all files in a folder. For example:

```
for i in ${dataPath}/*.fastq
do
  echo $i
done
```

This code will print the filenames of all .fastq files within your data folder (the * means "anything", so `${dataPath}/*.fastq` gives a list of all files in the data folder that ends with .fastq):

```
ERR000064_200_1.fastq
ERR000064_200_2.fastq
ERR000061_500_1.fastq
ERR000061_500_2.fastq
```

 Try to integrate a "for-loop" into the 1. quality check step.

e) Create an array with file names

If you have a look at the script file, you will realize that we use the names of the input files all over again. Thus we could define them in variables. However, maybe we have more input files in another data set, thus we would like to make the pipeline independent on the number of input files. This can be reached by putting the file names in an array (somehow a list of variables). Afterwards we can iterate over the array by using a "for-loop" to execute different commands (see step 6). Thus the commands are executed as many times as the size of the array.

An array is defined as follows:

```
names=(ERR000064_200 ERR000061_500)
```

The values of an array can be accessed by its index (attention: indexes start at 0!):

```
echo ${names[0]} #this will return ERR000064_200
```

If we now apply this to our pipeline it looks as follows:

```
names=(ERR000064_200 ERR000061_500)
...
#2. quality trimming: Trimmomatic
#################################
# - remove leading and trailing low quality basses (<3) or N
# - 4 base sliding window -> remove when average quality is < 15
# - remove reads which are shorter than 20 bp
java -jar ${pathTrimmomatic} PE -threads 4 ${dataPath}/${names[0]}_1.fastq
${dataPath}/${names[0]}_2.fastq ${names[0]}_1_trimPair.fastq
${names[0]}_1_trimUnpair.fastq ${names[0]}_2_trimPair.fastq
${names[0]}_2_trimUnpair.fastq LEADING:3 TRAILING:3 SLIDINGWINDOW:4:15
MINLEN:20
java -jar ${pathTrimmomatic} PE -threads 4 ${dataPath}/${names[1]}_1.fastq
${dataPath}/${names[1]}_2.fastq ${names[1]}_1_trimPair.fastq
${names[1]}_1_trimUnpair.fastq ${names[1]}_2_trimPair.fastq
${names[1]}_2_trimUnpair.fastq LEADING:3 TRAILING:3 SLIDINGWINDOW:4:15
MINLEN:20
...
```

 Apply it to the whole pipeline

f) Integrate a "for-loop" in the rest of the steps.

As mentioned in step 5 we can now use the "for-loop" whenever the same function is applied to the files. The easiest way to iterate over an array is as follows:

```
for name in ${names[*]}
do
  echo $name
done
```

the command `${names[*]}` returns all items in the list, which can be accessed by $name within the for-loop.
However, if you need to call items in parallel in another array, you can return the indexes of an array instead of the items itself:

```
for i in ${!names[*]}
do
  echo ${names[i]}
  echo ${address[i]}
done
```

the command `${!names[*]}` returns all index in the array, instead the values. This can then be used within the for loop to call the values of an array.

 Try to integrate "for-loops" into the rest of the steps.

In the 5. Step (reference mapping with Bowtie2) we need to specify the size range of the paired-end libraries. To automatize this we need to define an array with the insertion sizes:

```
lib=(200 500)  #same order as in names array
```

Now we can define a lower and an upper bound of the insertion size. Let's say we define this as +- 150bp of the library insertion length. This can be done as follows:

`$((${lib[0]}-150))` and `$((${lib[0]}+150))`

 Try to integrate this in the "for-loop" of the bowtie2 mapping

g) Set up a variable for the number of threads to use

A lot of the programs can use multiple cores. To define how many threads should be used by the programs we can define a variable giving the number of usable threads. This makes it easy to adapt the pipeline to the computational resources available.

```
Nthreads=4
```

 Apply it to the whole pipeline

h) Set up a variable for reference genome

Finally, we replace the reference path with a variable:

```
ref=/home/student/Documents/bashScripting/data_script/chr20.fa
```

 Apply it to the whole pipeline

In the 5. Step (reference mapping) we need the path to the reference without the file ending (.fa). This can easily be reached by following command, where `%.fa` removes the .fa from the string:

```
${ref%.fa}
```

➔ **Now the pipeline is set up in a way that it is applicable to any number of data files and different species. You just need to adapt the variables and arrays in the header (first part of the file).**

i) directly create a bam instead of a sam file

To further improve the pipeline we could directly pipe the output of Bowtie2 to samtools and convert the sam file to a sorted bam file without storing the sam file. This will reduce the memory required to store the results.

```
${pathBowtie} --fast-local -p 4 -q --phred33 -I 50 -X 350 --no-unal -x
refPath/chr20 -1 ERR000064_200_1_trimPair.fastq -2
ERR000064_200_2_trimPair.fastq -U ERR000064_200_Unpair.fastq ~~-S ERR000064.sam~~|
${pathSamtools} view -bS - | ${pathSamtools} sort - ERR000064.sorted
```

`samtools view -bS` converts the sam to a bam file and `samtools sort` sorts the bam file

 Integrate this in the pipeline

Solutions:

- Exercise 1-4: BashScripting_solutions.sh
- Exercise 5: script_mod.sh