
NGS Course Nove Hradý Documentation

Release 1.0

Libor Morkovsky, Vaclav Janousek

April 27, 2015

1	Installation instructions	3
1.1	How to access the machine	5
1.2	Windows	5
1.3	Mac OS X and Linux	6
1.4	Time to log in!	6
2	Connecting to the virtual machine	7
2.1	Connect to control the machine	7
2.2	Testing the Internet connection	9
2.3	Connect to copy files	9
2.4	Connect to RStudio	10
3	Starter session	11
3.1	Basic orientation	11
3.2	Installing software	13
3.3	Show me the data!	15
4	Genomics session	19
5	Advanced UNIX session	21
5.1	List of Tasks:	21
6	Graphics session	23
6.1	Summarization	23
6.2	Tidy data	27
7	Genomic tools session	29
8	Quality session	33
8.1	Read quality	33
8.2	Variant quality	37
9	Links	39
9.1	Bash	39
9.2	R Studio	39
9.3	Visual design	39
9.4	Genomic tools	39
9.5	Genomic data formats	40

10 Best practice	41
10.1 Easiest ways to get UNIX	41
10.2 Essentials	41
10.3 Data organization	41
10.4 Building command lines	42
10.5 Parallelization	43
11 Reference manual for UNIX introduction	45
11.1 Basic orientation in UNIX	45
11.2 Exploring and basic manipulation with data	47
11.3 Building commands	49
11.4 Advanced text manipulation (sed)	51
11.5 More complex data manipulation (awk)	51
12 Important NGS formats	53
12.1 FASTQ - Short reads	53
12.2 SAM - Reads mapped to reference	53
12.3 BED and GFF - Annotations	53
12.4 VCF - Variants in individuals	53
13 Additional exercises	55
13.1 Counting heads	55
13.2 Dimensionality reduction	56
14 Course materials preparation	59
14.1 Online documentation	59
14.2 VirtualBox image	60
14.3 Slide deck	63
15 Slide decks	65

This course aims to introduce the participants to UNIX - an interface that is one of the most convenient options for working with big data. Special attention is put on working with Next Generation Sequencing (NGS) data. Most of the NGS data formats were designed to be textual, and textual data is where UNIX excels in its versatility. By combining basic UNIX tools one can achieve results that would require finding specialized software in other environments.

Not knowing ‘the right way’ to do things can be intimidating for the beginning users, so an additional section exploring the ‘best practice’ is available for self learning.

Initial instructions:

Installation instructions

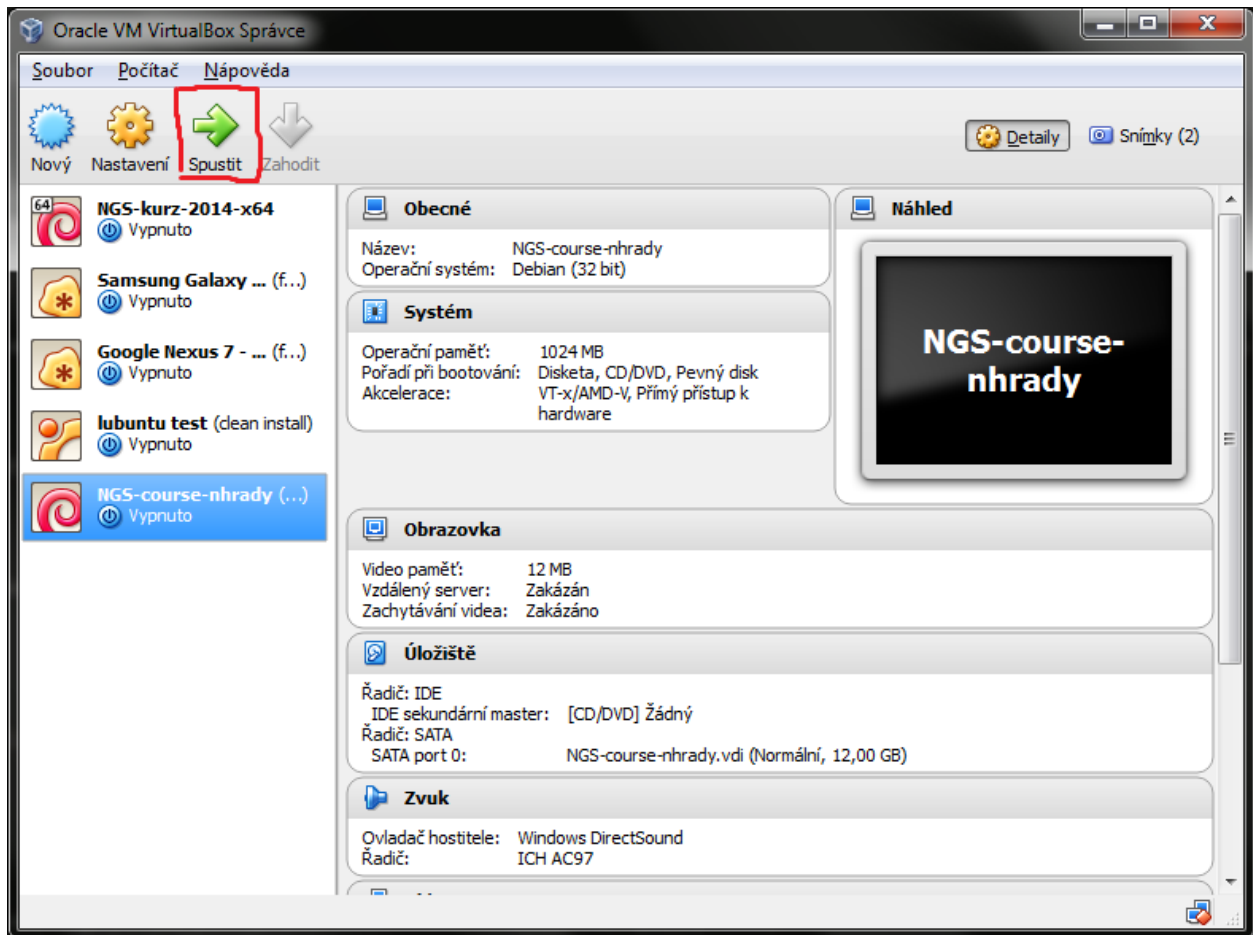
We will be using a virtual computer pre-installed with Debian Linux and sample data necessary for the exercises.

Note: You need to install the image even if your main system is Linux / Mac OS X!

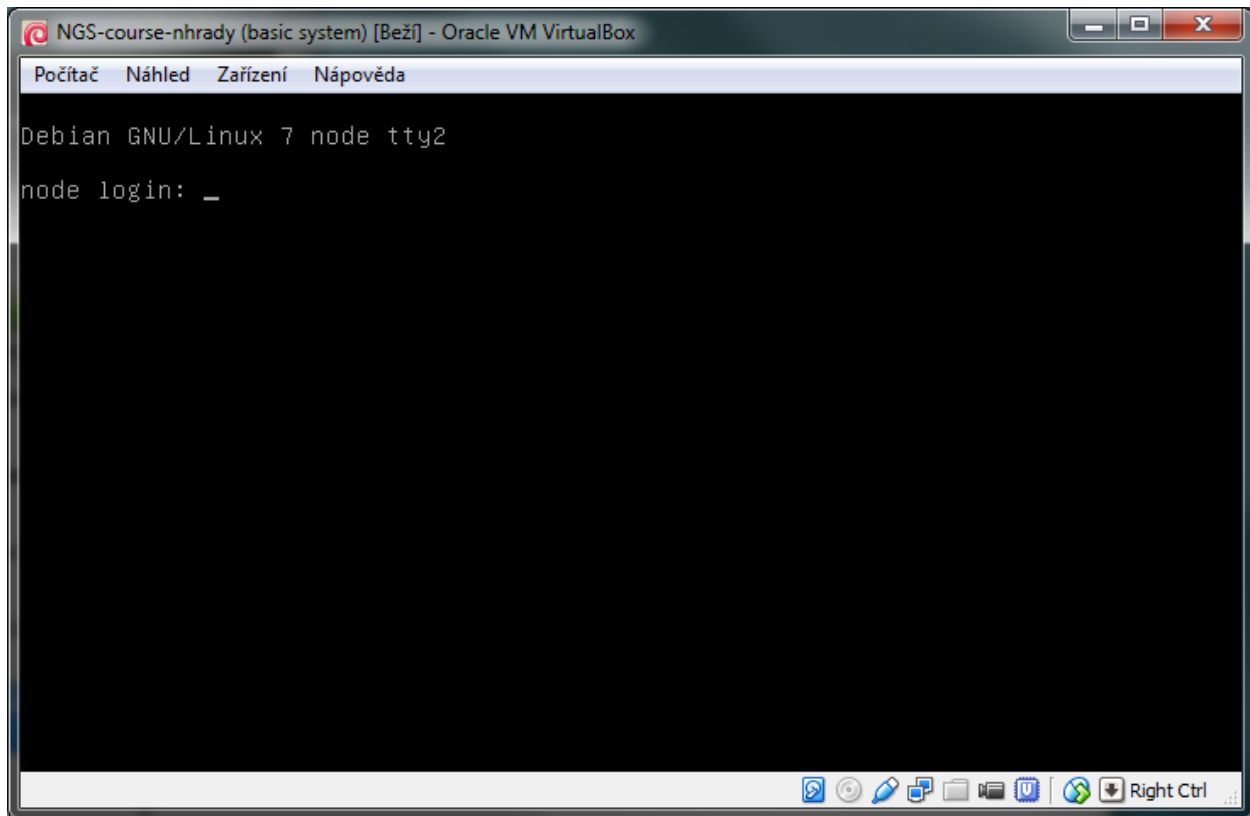
Installation steps (it should take about 10 minutes):

- Install VirtualBox (<https://www.virtualbox.org/wiki/Downloads>). It works on Linux and Mac too.
- Download the virtual machine image from this link: <http://goo.gl/FwFk8Z> You'll get a single file with .ova extension on your hard drive.
- You can either double click the .ova file, or run VirtualBox, and choose `File > Import Appliance`. Follow the instructions after the import is started.

After successful installation you should see something like this (only the machine list will contain just one machine). Check whether you can start the virtual machine: click `Start` in the main VirtualBox window:



After a while you should see something like this:



You don't need to type anything into that window, just checking that it looks like the screen shot is enough.

1.1 How to access the machine

Because it is much more comfortable to use a native terminal application than the small VM screen, you will connect to the machine depending on what system you are using.

Machine configuration details:

- Administrative user: *root*, password: *debian*
- Normal user: *user*, password: *user*
- ssh on port 2222
- RStudio on port 8787

In case of any problems try to find contact the tutors, we'll try to resolve all problems before the course.

1.2 Windows

Install PuTTY and WinSCP. PuTTY will be used to control the virtual computer. WinSCP will be used to transfer files between your computer and the virtual computer.

- PuTTY (<http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html> - look for putty.exe)
- WinSCP (<http://winscp.net/eng/download.php> - look for Installation package).

1.3 Mac OS X and Linux

Ssh is used to control the virtual computer. It should be installed in your computer.

Files can be transferred with `scp`, `rsync` or `lftp` (recommended) from the command line. *Scp* and *rsync* could be already installed in your system, if you want to use *lftp*, you'll probably have to install it yourself.

Mac users that prefer graphical clients can use something like *CyberDuck*. See <http://apple.stackexchange.com/questions/25661/whats-a-good-graphical-sftp-utility-for-os-x>.

1.4 Time to log in!

Try to log in following the instructions in *Connect to control the machine*.

Connecting to the virtual machine

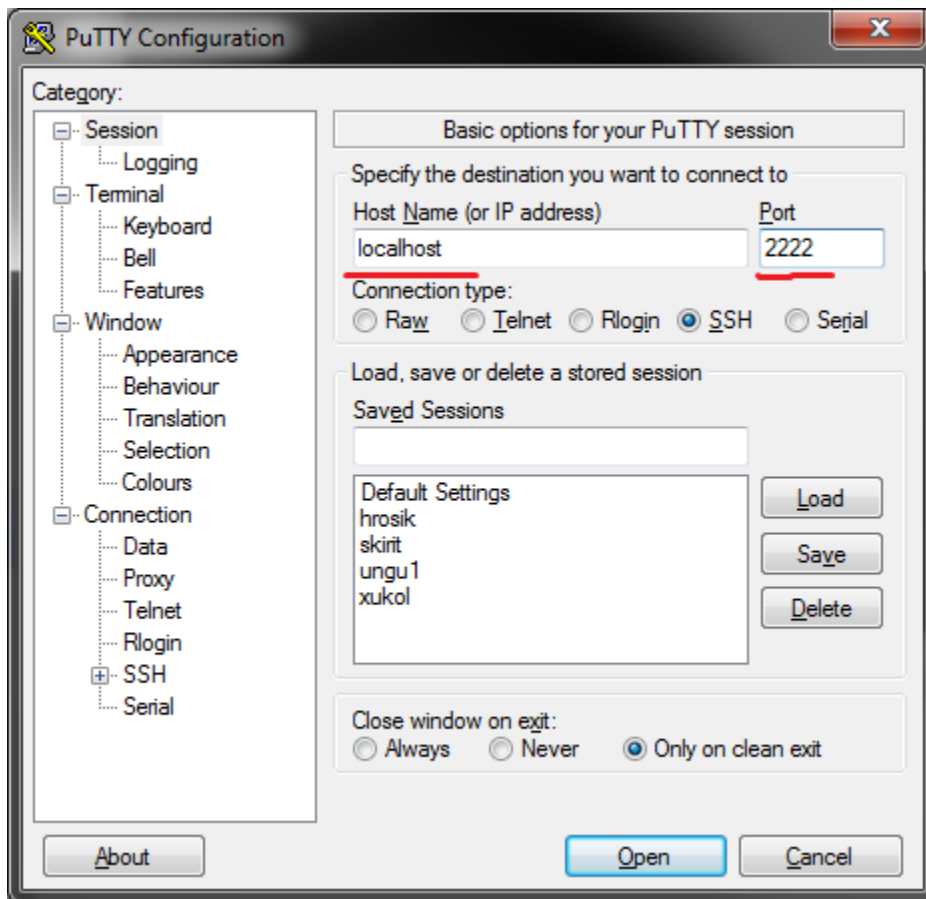
Note: You need to start the virtual machine first!

2.1 Connect to control the machine

To control the machine, you need to connect to the ssh service. This is also referred to as ‘logging in’.

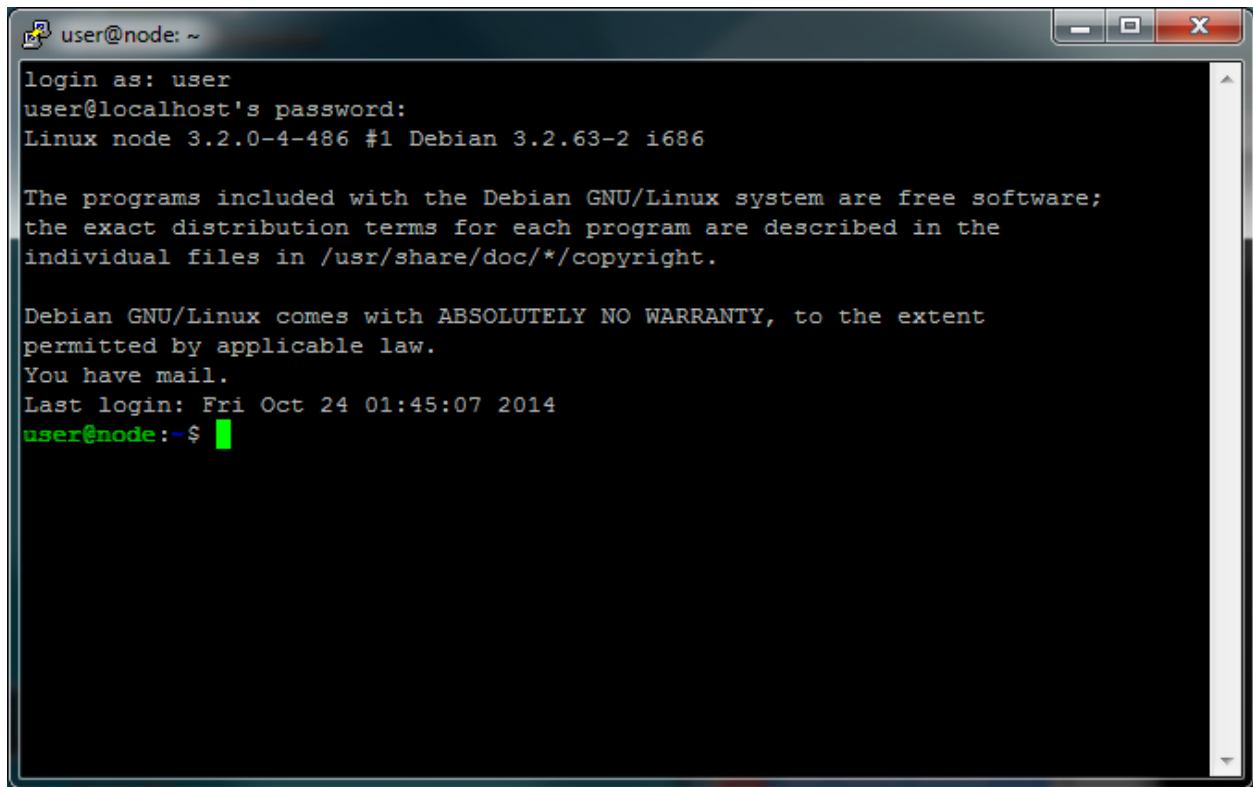
In Windows this is done with PuTTY.

- start PuTTY
- fill Host Name: `localhost`
- fill Port: `2222`
- click Open or press <Enter>



In the black window that appears, type your credentials:

- login as: `user`
- `user@localhost`'s password: `user`

A screenshot of a terminal window titled 'user@node: ~'. The window shows the output of an SSH login. It starts with 'login as: user', followed by 'user@localhost's password:', and then 'Linux node 3.2.0-4-486 #1 Debian 3.2.63-2 i686'. Below this, there is a message about Debian GNU/Linux being free software and the location of copyright files. Then, it states 'Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent permitted by applicable law.' and 'You have mail.'. The last line of the login sequence is 'Last login: Fri Oct 24 01:45:07 2014'. The prompt 'user@node:~\$' is shown at the bottom, with a green cursor.

In Mac OS X or Linux, you start your terminal program first ('Terminal', 'Konsole', 'xterm'). In the terminal window your shell is running (probably 'bash'). Here you use ssh to connect to the virtual machine:

```
ssh -p 2222 user@localhost
```

2.2 Testing the Internet connection

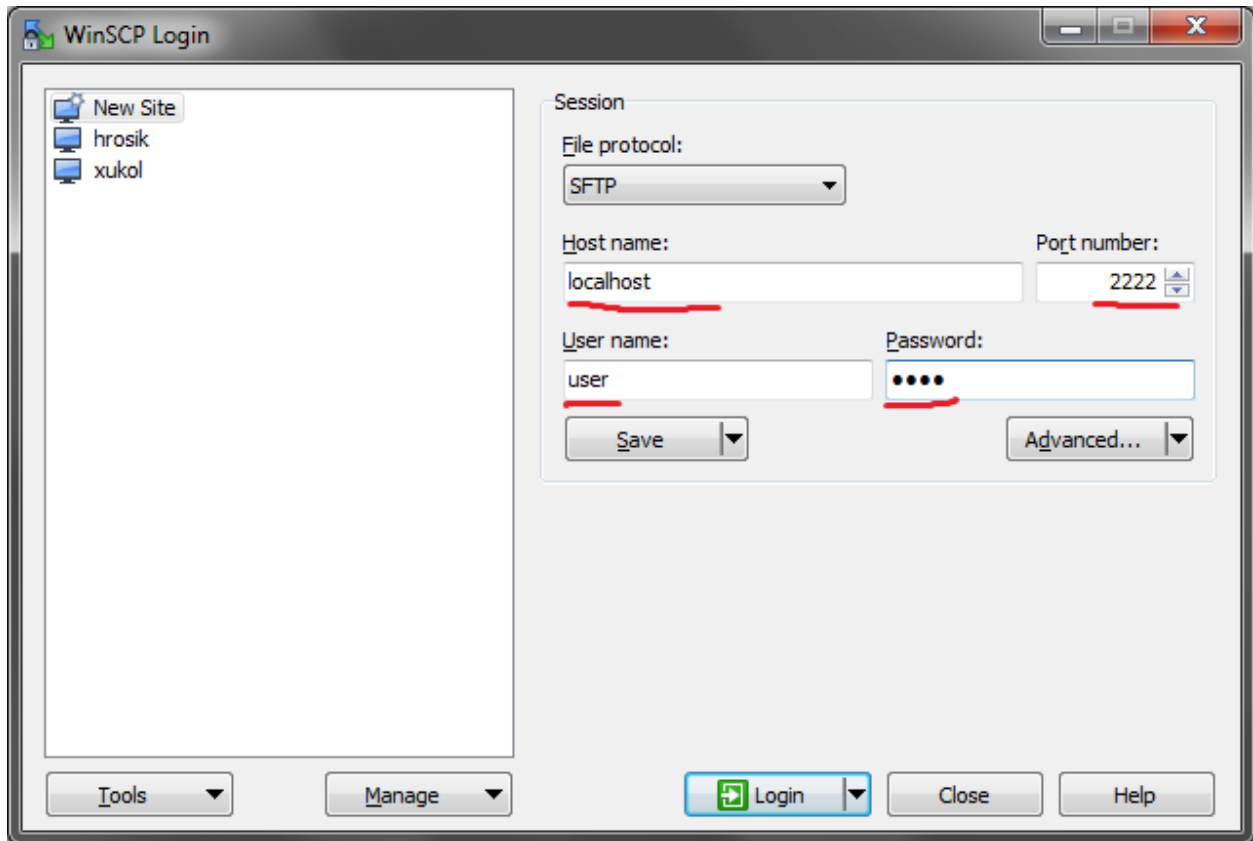
When you're logged in, check your internet connection from the virtual machine. Your main computer has to be connected to the internet, of course. Copy the following command, and paste it to the command prompt (click right mouse button in PuTTY window).

```
wget -q -O - http://goo.gl/n8XK2Y | head -1  
# <!DOCTYPE html>
```

If the `<!DOCTYPE html>` does not appear, something is probably wrong with the connection.

2.3 Connect to copy files

In Windows, WinSCP is used to copy files to Linux machines. You use the same information as for PuTTY to log in.



In Mac OS X or Linux, the most simple command to copy a file into a home directory of `user` on a running virtual machine is:

```
scp -P 2222 myfile user@localhost:~
```

2.4 Connect to RStudio

This is the easiest one, just click this link: [Open RStudio](#). Login with the same credentials (user, user).

Course contents:

Starter session

This session will give you all the basics that you need to smoothly move around when using a UNIX system (in the text mode!).

3.1 Basic orientation

3.1.1 Use multiple windows (and be safe when the network fails)

First, type `screen` in your terminal:

```
screen
```

Screen creates the first window for you. To create another one press `ctrl+a c`. To switch between the windows press `ctrl+a space`.

Note: Keyboard shortcuts notation: `ctrl+a space` means press `ctrl` key and `a` key simultaneously and `space` key after you release both of the previous keys.

3.1.2 Check what the computer is doing

Run `htop` in one of your screen windows:

```
htop
```

Htop displays CPU and memory utilization of the (virtual) computer. Continue your work in another window (`ctrl+a space`). You can switch back to the `htop` window to monitor progress of some lengthy operation.

3.1.3 Get help

Call Libor or Vaclav to get any help ;)

To find the name of the command that does what you need (`grep`), use google:

```
linux search for string
```

Once you know the name of the command that does what you need, all the details are easily accessible using `man`. To get all possible help about finding text patterns in files do:

```
man grep
```

3.1.4 Move around the directory structure

Unlike ‘drives’ in MS Windows, UNIX has a single directory tree that starts in `/` (called root directory). Everything can be reached from the root directory. The next important directory is `~` (called user’s home directory). It is a shortcut for `/home/user` here, `/home/..your login name..` in general.

Your bash session has a *working directory* that can be changed with `cd` (change directory) and printed with `pwd` (print working directory). All filenames and paths you type refer to your working directory (relative paths), unless you start them with `/` (absolute paths).

Try the following commands in the order they are provided, and figure out what they do. Then use your knowledge to explore the directory structure of the virtual machine.

```
pwd
ls
ls /
cd /
pwd
ls
cd
pwd
```

A neat trick to go back where you’ve been before the last `cd` command:

```
cd -
```

More in *Moving around & manipulation with files and directories*.

3.1.5 Prepare data in your home directory

All the data we are going to use are located in `/data`. However, we want to have it to be conveniently located in our own directory (`~`, `/home/user`). Without the need to copy all of it, we can use a symbolic link (`ln -s`). This keeps the data in their original location but creates a reference.

In case you do not know where your files are but you do know some part of the name, use the `locate` command. We know the names contained `fastq`, `vcf` and `gff3` suffices:

```
locate fastq
locate vcf
locate gff3
```

Note: To paste text into PuTTY just click right mouse button anywhere in the window. To copy text to clipboard, just select it. No keyboard shortcuts are necessary.

Once we know their actual position we can create symbolic links:

```
# create directory data
# <- this marks a comment - anything after first # is ignored
mkdir data

# go to your new data directory
cd data

# create a link to the nightingale reads
```



```
# and name it 'fastq'
ln -s /data/slavicci/00-reads fastq
```

You created a *symbolic link* named `fastq` with `/data/slavicci/00-reads` as a *target*. Check it by typing:

```
ls -l
```

Note: You should use bash *autocomplete* feature, when typing paths. It is easier, faster and less prone to error. Type a part of the path, like `/da` and press the `tab` key. When nothing appears, press `tab` once more. There is either no possible completion or more possibilities, that will be displayed on the second press.

It is possible to create a bad link. There is no validation on the target:

```
ln -s /nothing_here bad-link
```

```
# the bad link has a different color in the output
ls -l
```

```
# get rid of the bad link
rm bad-link
```

3.1.6 Check your keyboard

Before we do any serious typing, make sure you know where are the important keys. I'd suggest using English keyboard, if you don't want to constantly press right alt and five random letters before you find the one you need. You will definitely need those keys:

```
' - single quotes
" - double quotes
* - asterisk
~ - tilde
| - pipe
/ - slash
\ - backslash
[] - square brackets
```

While we're at it, we'll look into some keyboard shortcuts:

```
ctrl+c - kills current running program (except for bash, nano, vim, ...)
        - clears the command line in bash

ctrl+d - means end of input (if you run e.g. bc interactively)
        - end of input means logout in bash

ctrl+r - starts history search in bash, just type a part of a long command
        and it will come back (ctrl+c to the rescue;)

ctrl+k - clears the command line from cursor to the end,
        you will need this while exploring long pipelines...
```

3.2 Installing software

The easiest way to install software is via a package manager (eg. `apt-get` for all Debian variants). When the required software is not in the repositories, or one needs the latest version, it's necessary to take the more difficult path. The canonical UNIX way is:

```
wget -O - ..url.. | tar xvz      # download and unpack the 'tarball' from internet
cd ..unpacked directory..      # set working directory to the project directory
./configure                    # check your system and choose the way to build it
make && sudo make install       # convert source code to machine code and if successful, copy the result
```

3.2.1 Pipe viewer

First we'll get the latest pipe viewer. Pipe viewer can show you how much of the data was already processed in your *pipeline*. Google pipe viewer, choose the ivarch.com site. Check the current version number on the site. Now check the version in your image:

```
pv --version
```

Note: It is a good habit to include `--version` option for a command. You need to check the version of given tool in your system when you're trying to use some new features.

The version found at the site should be higher then the one in your image. A good reason for update;) Copy the link for the `.tar.bz2` file on the site.

```
# go to the directory where software installations live
cd ~/sw

wget -O - ..paste the link here .. | tar xvj

# the complete command from above for those who are cheating
wget -O - http://www.ivarch.com/programs/sources/pv-1.6.0.tar.bz2 | tar xvj

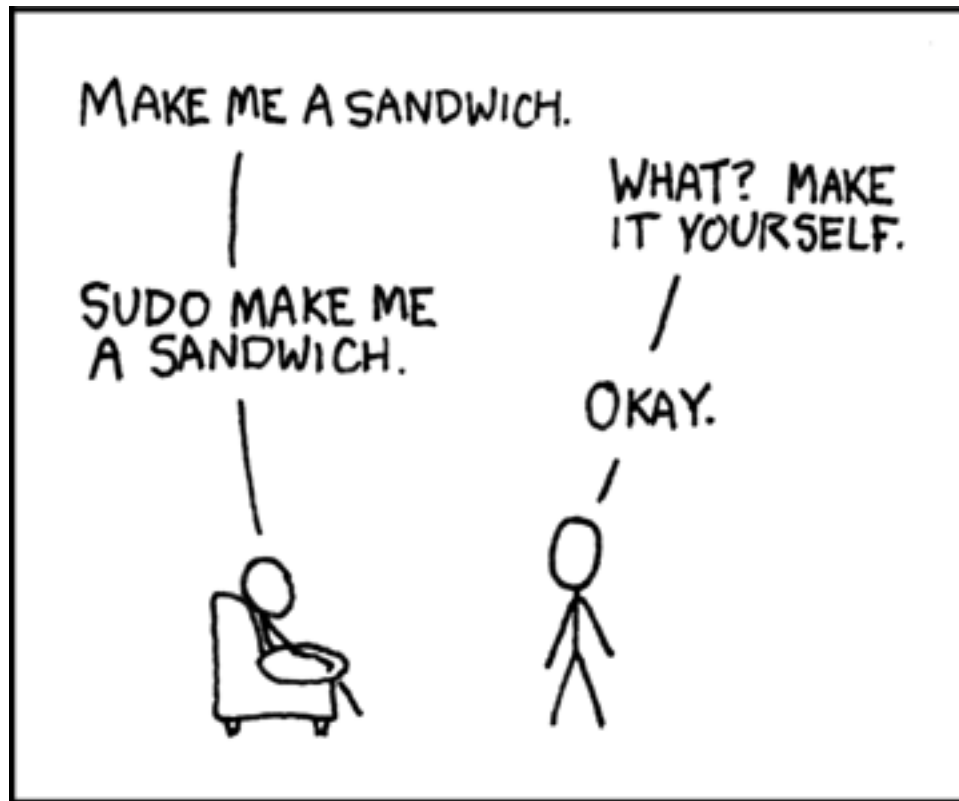
# do not copy this, try the autocompletion
# cd pv<tab> <tab> <6> <tab> <enter>

ls
# you can see green configure script in the listing

# to run something in current directory, the path has
# to be given
./configure
make

# to make changes system wide, super user 'powers' have to be used
sudo make install
```

Note: Normal users cannot change (and break) the (UNIX) system. There is one special user in each system called `root`, who has the rights to make system wide changes. You can either directly log in as `root`, or use `sudo` (super user do) to execute one command as `root`.



3.2.2 Bedtools

Another common place where you find a lot of software is *GitHub*. We'll install `bedtools` from a GitHub repository:

```
cd ~/sw

# get the latest bedtools
git clone https://github.com/arq5x/bedtools2
```

This creates a *clone* of the online repository in `bedtools2` directory.

```
cd bedtools2
make
```

The compilation should take a while, so you can flip to your *htop* window with `ctrl-a space` and watch the CPU spin;)

3.3 Show me the data!

Until now we were working with files and directories. But the real data is inside the files.

3.3.1 Explore FASTQ files

The `less` tool is used to list through contents of a text file. We will check some of the FASTQ files linked in our `~/data` directory.

```
# cd by itself means cd ~ (that is cd /home/user here)
# this will get you to your home directory, wherever you are
cd

# a file can be referenced in various ways
# option 1: absolute path (<q> to quit the viewer)
less /home/user/data/fastq/G59B7NP01.fastq

# option 2: relative path from working directory
less data/fastq/G59B7NP01.fastq

# option 3: move 'closer' to the file
cd data/fastq
less G59B7NP01.fastq
```

Note: Reminder: you don't have to type the whole file name. Try to use TAB auto-completion!

The data you see looks like mess. One of the reasons is there are long lines, that get wrapped so you see all the letters. But then you don't see the file structure. Add the `-S` option, and see the four different line types in the FASTQ file:

```
less -S G59B7NP01.fastq
```

The lines are:

1. sequence name
2. dna letters
3. + sign
4. encoded quality scores

The options can be given either one by one - which is more legible, or combined. Another interesting option is `-N`, showing the line numbers:

```
less -S -N G59B7NP01.fastq

# this is the same as above
less -SN G59B7NP01.fastq
```

Note: If you forgot to type `-S` at the prompt, you can type `-S` also while in `less`. Try it!

3.3.2 UNIX Pipes

For a quick glance over the contents of the file, you can also use the `head` command:

```
head G59B7NP01.fastq
```

The problem with the wrapped lines comes back again. `head` is not meant to be a file viewer, so it does not have any text wrapping options. Instead you can combine two tools. `cut` allows you to choose only a part of each line.

```
# show up to 50 characters from each
# of the first 10 lines in the file
head G59B7NP01.fastq | cut -c -50

# we can get only first four lines
head -4 G59B7NP01.fastq | cut -c -50
```

Using the `|` (pipe) character you instruct the shell to take the output of the first command and use it as an input for the second command. You can also use `less` as a part of the pipeline:

```
head -4 G59B7NP01.fastq | less -S
```

The complement to `head` is `tail`. It displays last lines of the input. It can be readily combined with `head` to show the second sequence in the file.

```
head -8 G59B7NP01.fastq | tail -4 | less -S

# or the third sequence data ;)
head -12 G59B7NP01.fastq | tail -4 | less -S
```

3.3.3 How many reads are there?

We found out that FASTQ files have a particular structure (four lines per read). To find the total number of reads in our data, we will use another tool, `wc` (stands for *word count*, not for a toilet at the end of the pipeline;). `wc` counts words, lines and characters.

Our data is in three separate files. To merge them on the fly we'll use another tool, `cat` (for conCATenate). `cat` takes a list of file names and outputs a continuous stream of the data that was in the files (there is no way to tell where one file ends from the stream).

```
ls

# now double click on each file name in the listing,
# and click right mouse button to paste (insert space in between)
cat G59B7NP01.fastq GS60IET02.RL1.fastq GS60IET02.RL2.fastq | wc -l
```

The number that appeared is four times the number of sequences (each sequence takes four lines). And there is even a built-in calculator in bash:

```
echo $(( 788640 / 4 ))
```

Imagine you've got 40 FASTQ files instead of 3. You don't want to copy and paste all the names! There is a feature that comes to rescue. It's called *globbing*. It allows you to specify more filenames at once by defining some common pattern. All your read files have `.fastq` extension:

```
echo *.fastq
```

`echo` is no magic, it outputs whatever you give it (try `echo ahoj`). The magic is done by bash - whenever it sees an asterisk (`*`), it tries to expand it by matching to the files and directories. `*.fastq` means *a file named by any number of characters followed by '.fastq'*.

Globbing works even across directories, try:

```
cd ..
echo fastq/*.fastq
```

Now we can use it in our read counting pipeline to make it shorter and more versatile:

```
cd fastq
cat *.fastq | wc -l
```

3.3.4 How many bases were sequenced?

`wc` can count characters (think bases) as well. But to get a reasonable number, we have to get rid of the other lines that are not bases.

One way to do it is to pick only lines comprising of letters A, C, G, T and N. There is a ubiquitous mini-language called *regular expressions* that can be used to define text patterns. *A line comprising only of few possible letters* is a text pattern. `grep` is the basic tool for using regular expressions:

```
cat *.fastq | grep '^[ACGTN]*$' | less -S
```

Check if the output looks as expected. This is a very common way to work - build a part of the pipeline, check the output with `less` or `head` and fix it or add more commands.

Now a short explanation of the `^[ACGTN]*$` pattern (`grep` works one line a time):

- `^` marks beginning of the line - otherwise `grep` would search anywhere in the line
- the square brackets (`[]`) are a *character class*, meaning one character of the list, `[Gg]rep` matches `Grep` and `grep`
- the `*` is a count suffix for the square brackets, saying there should be zero or more of such characters
- `$` marks end of the line - that means the whole line has to match the pattern

To count the bases read, we extend our pipeline:

```
cat *.fastq | grep '^[ACGTN]*$' | wc -c
```

The thing is that this count is not correct. `wc -c` counts every character, and the end of each line is marked by a special character written as `\n` (n for newline). To get rid of this character, we can use another tool, `tr` (transliterate). `tr` can substitute one letter with another (imagine you need to lowercase all your data, or mask lowercase bases in your Fasta file). Additionally `tr -d` (delete) can remove characters:

```
cat *.fastq | grep '^[ACGTN]*$' | tr -d "\n" | wc -c
```

Note: If you like regular expressions, you can hone your skills at <https://regex.alf.nu/>.

Genomics session

The commands for this session are in the talk's slide deck.

Advanced UNIX session

5.1 List of Tasks:

1. How many records in the GTF file
2. Explore the 'group' column (column 9) in the GTF file
3. Get list of chromosomes (column 1)
4. Get list of features (column 3)
5. Get the number of genes mapping onto chromosomes in total
6. Get the number of protein coding genes mapping onto chromosomes
7. Get the number of protein coding genes on chromosome X and Y
8. Get the number of transcripts of protein coding genes mapping onto chromosomes
9. Get the gene with the highest number of transcripts
10. Get the gene with the highest number of exons
11. What is the total size (in Mb) of coding sequences
12. Get the longest gene

Note: During the afternoon session we are going to use these commands:

```
cut
sort
uniq
grep
tr
sed
awk
```

1. How many records in the GTF file

```
cat Mus_musculus.NCBIM37.67.gtf | wc -l
```

2. Explore the 'group' column (column 9) in the GTF file

```
cut -f 9 Mus_musculus.NCBIM37.67.gtf | less -S
```

3. Get list of chromosomes (column 1)

```
cut -f 1 Mus_musculus.NCBIM37.67.gtf | sort | uniq
```

4. Get list of features (column 3)

```
cut -f 3 Mus_musculus.NCBIM37.67.gtf | sort | uniq
```

5. Get the number of genes mapping onto chromosomes in total

```
grep -v ^NT Mus_musculus.NCBIM37.67.gtf | cut -f 9 | cut -d ";" -f 1 | sort | uniq | wc -l
```

6. Get the number of protein coding genes mapping onto chromosomes

```
grep -v ^NT Mus_musculus.NCBIM37.67.gtf | grep protein_coding | cut -f 9 | cut -d ";" -f 1 | sort | u
```

7. Get the number of protein coding genes on chromosome X and Y

```
grep ^[XY] Mus_musculus.NCBIM37.67.gtf | grep protein_coding | cut -f 1,9 | cut -d ';' -f 1 | sort |
```

8. Get the number of transcripts of protein coding genes mapping onto chromosomes

```
grep -v ^NT Mus_musculus.NCBIM37.67.gtf | grep protein_coding | cut -f 9 | cut -d ";" -f 2 | sort | u
```

9. Get the gene with the highest number of transcripts

```
grep -v ^NT Mus_musculus.NCBIM37.67.gtf | grep protein_coding | cut -f 9 | cut -d " " -f 3,5,9 | tr -
```

10. Get the gene with the highest number of exons

```
grep -v ^NT Mus_musculus.NCBIM37.67.gtf | grep protein_coding | grep '$\texon\t' | cut -f 9 | cut -d
```

11. What is the total size (in Mb) of coding sequences

```
grep CDS Mus_musculus.NCBIM37.67.gtf | awk -F '$\t' 'BEGIN{OFS=FS;t=0}{s=$5-$4+1;t+=s}END{print t/100
```

12. Get the longest gene

```
grep protein_coding Mus_musculus.NCBIM37.67.gtf | grep '$\texon\t' | cut -f 1,4,5,9 | cut -d " " -f 1
```

```
< exons.bed awk -F '$\t' 'BEGIN{ OFS=FS }{if(NR==1){ gene=$4; chrom=$1; gene_start=$2; gene_end=$3 }e
```

```
## Detail structure:
```

```
awk -F '$\t' 'BEGIN{ OFS=FS }{
    if(NR==1){
        gene=$4; chrom=$1; gene_start=$2; gene_end=$3
    }else{
        if(gene==$4){
            if(gene_end<=$3){
                gene_end=$3
            }else{
                print gene,chrom,gene_start,gene_end,gene_end-gene_start;
                gene=$4;chrom=$1;gene_start=$2;gene_end=$3;
            }
        }
    }END{
        print gene,chrom,gene_start,gene_end,gene_end-gene_start
    }'
```

Graphics session

A picture is worth a thousand words.

Especially when your data is big. We'll try to show you one of the easiest ways to get nice pictures from your UNIX. We'll be using R, but we're not trying to teach you R. R Project is huge, and mostly a huge mess. We're cherry picking just the best bits;)

6.1 Summarization

R is best for working with 'tables'. That means data, where each line contains the same amount of 'fields', delimited by some special character like `;` or `<tab>`. The first row can contain column names. VCF is almost a nice tabular file. The delimiter is `<tab>`, but there is some mess in the beginning of the file:

```
</data/mus_mda/00-popdata/popdata_mda_euro.vcf less -S
```

6.1.1 Prepare the input file

We want to get rid of the comment lines starting with `##`, and keep the line starting with `#` as column names (getting rid of the `#` itself):

```
# create a new 'project' directory in data
mkdir ~/data/plotvcf

# we'll be reusing the same long file name, store it into a variable
IN=/data/mus_mda/00-popdata/popdata_mda_euro.vcf

# get rid of the '##' lines (quotes have to be there, otherwise
# '#' means a comment in bash)
<$IN grep -v '##' | less -S

# good, now trim the first #
<$IN grep -v '##' | tail -c +2 | less -S

# all looks ok, store it (tsv for tab separated values)
<$IN grep -v '##' | tail -c +2 > ~/data/plotvcf/popdata_mda_euro.tsv
```

Now we will switch to R Studio. You can just click here: [Open RStudio](#).

In R Studio choose `File > New file > R Script`. R has a working directory as well. You can change it with `setwd`. Type this code into the newly created file:

```
setwd('~/.data/plotvcf')
```

With the cursor still in the `setwd` line, press `ctrl+enter`. This copies the command to the console and executes it. Now press `ctrl+s`, and save your script as `plots.R`. It is a better practice to write all your commands in the script window, and execute with `ctrl+enter`. You can comment them easily, you'll find them faster...

6.1.2 Load and check the input

Tabular data is loaded by `read.table` and its shorthands. On a new line, type `read.table` and press `F1`. Help should pop up. We'll be using the `read.delim` shorthand, that is preset for loading `<tab>` separated data with US decimal separator:

```
d <- read.delim('popdata_mda_euro.tsv')
```

A new entry should show up in the 'Environment' tab. Click the arrow and explore. Click the 'd' letter itself.

You can see that `CHROM` was encoded as a number only and it was loaded as `integer`. But in fact it is a factor, not a number (remember e.g. chromosome X). Fix this in the `read.delim` command, loading the data again and overwriting `d`. The plotting would not work otherwise:

```
d <- read.delim('popdata_mda_euro.tsv', colClasses=c("CHROM"="factor"))
```

6.1.3 First plot

We will use the `ggplot2` library. The 'grammatical' structure of the command says what to plot, and how to represent the values. Usually the `ggplot` command contains the reference to the data, and graphic elements are added with `+ geom_...()`. There are even some sensible defaults - e.g. `geom_bar` of a factor sums the observations for each level of the factor:

```
library(ggplot2)
ggplot(d, aes(CHROM)) + geom_bar()
```

This shows the number of variants in each chromosome. You can see here, that we've included only a subset of the data, comprising chromosomes 2 and 11.

6.1.4 Summarize the data

We're interested in variant density along the chromosomes. We can simply break the chromosome into equal sized chunks, and count variants in each of them as a measure of density.

There is a function `round_any` in the package `plyr`, which given precision rounds the numbers. We will use it to round the variant position to 1×10^6 (million base pairs), and then use this rounded position as the block identifier. Because the same positions repeat on each chromosome, we need to calculate it once per each chromosome. This is guaranteed by `group_by`. `mutate` just adds a column to the data.

You're already used to pipes from the previous exercises. While it's not common in R, it is possible to build your commands in a similar way thanks to the `magrittr` package. The name of the package is an homage to the Belgian surrealist René Magritte and his most popular painting.

Although the `magrittr %>%` operator is not a pipe, it behaves like one. You can chain your commands like when building a bash pipeline:

```
library(plyr)
library(dplyr)
```



Figure 6.1: Ceci n'est pas une pipe. This is not a pipe.

```
dc <- d %>% group_by(CHROM) %>% mutate(POS_block=round_any(POS, 1e6))
```

the above command is equivalent to

```
dc <- mutate(group_by(d, CHROM), POS_block=round_any(POS, 1e6))
```

Now you can check how the `round_any` processed the `POS` value. Click the `dc` in the **Environment** tab and look for `POS_block`. Looks good, we can go on. The next transformation is to count variants (table rows) in each block (per chromosome): You can use `View` in R Studio as `less` in bash.

```
dc %>%  
  group_by(CHROM, POS_block) %>%  
  summarise(nvars=n()) %>%  
  View
```

Note: To run the whole block at once with `ctrl+enter`, select it before you press the shortcut.

If the data look like you expected, you can go on to plotting:

```
dc %>%  
  group_by(CHROM, POS_block) %>%  
  summarise(nvars=n()) %>%  
  ggplot(aes(POS_block, nvars)) +  
    geom_line() +  
    facet_wrap(~CHROM, ncol = 1)
```

Now you can improve your plot by making the labels more comprehensible:

```
dc %>%  
  group_by(CHROM, POS_block) %>%  
  summarise(nvars=n()) %>%  
  ggplot(aes(POS_block, nvars)) +  
    geom_line() +  
    facet_wrap(~CHROM, ncol = 1) +  
    ggtitle("SNP denisty per chromosome") +  
    ylab("number of variants") +  
    xlab("chromosome position")
```

If you prefer bars instead of a connected line, it's an easy swap with `ggplot`.

```
dc %>%  
  group_by(CHROM, POS_block) %>%  
  summarise(nvars=n()) %>%  
  ggplot(aes(POS_block, nvars)) +  
    geom_bar(stat="identity") +  
    facet_wrap(~CHROM, ncol = 1) +  
    ggtitle("SNP denisty per chromosome") +  
    ylab("number of variants") +  
    xlab("chromosome position")
```

The `stat="identity"` is there, because `geom_bar` counts the rows otherwise. This could have saved us some more typing:

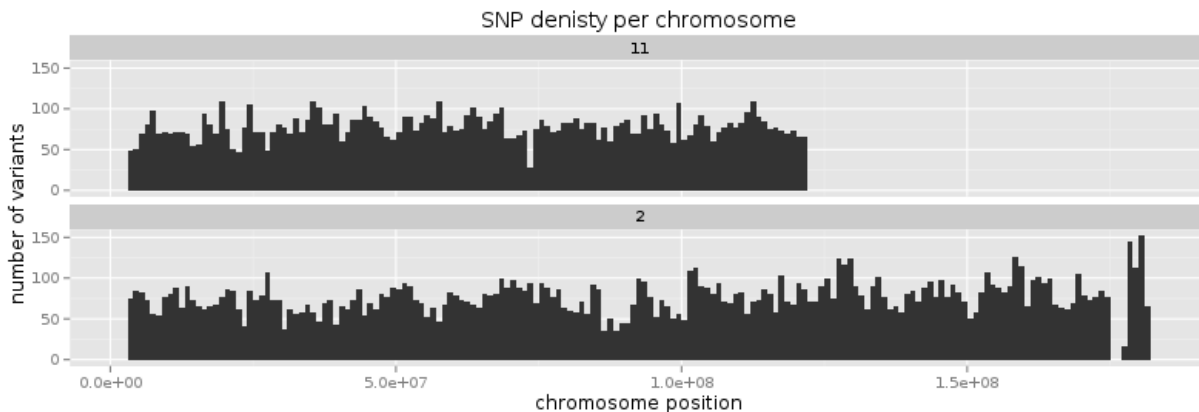
```
ggplot(d, aes(POS)) +  
  geom_bar() +  
  facet_wrap(~CHROM, ncol = 1) +  
  ggtitle("SNP denisty per chromosome") +  
  ylab("number of variants") +  
  xlab("chromosome position")
```

ggplot warned you in the **Console**:

```
stat_bin: binwidth defaulted to range/30. Use 'binwidth = x' to adjust this.
```

You can use `binwidth` to adjust the width of the bars, setting it to 1×10^6 again:

```
ggplot(d, aes(POS)) +
  geom_bar(binwidth=1e6) +
  facet_wrap(~CHROM, ncol = 1) +
  ggtitle("SNP denisty per chromosome") +
  ylab("number of variants") +
  xlab("chromosome position")
```



6.2 Tidy data

To create plots in such a smooth way like in the previous example the data has to loosely conform to some simple rules. In short - each column is a variable, each row is an observation. You can find more details in the [Tidy data](#) paper. There is an R package `tidyr` that helps you to get the data into the required shape.

The vcf is *tidy* when using the `CHROM` and `POS` variables. Each variant (SNP) is a row. The data is not tidy regarding variants in particular individuals. Individual identifier is a variable for this case, but it is stored as column name. This is not 'wrong', this format was chosen so the data is smaller. But it does not work well with ggplot.

Now if we want to look at genotypes per individual, we need the genotype as a single variable, not 18. `gather` takes the values from multiple columns and gathers them into one column. It creates another column where it stores the originating column name for each value.

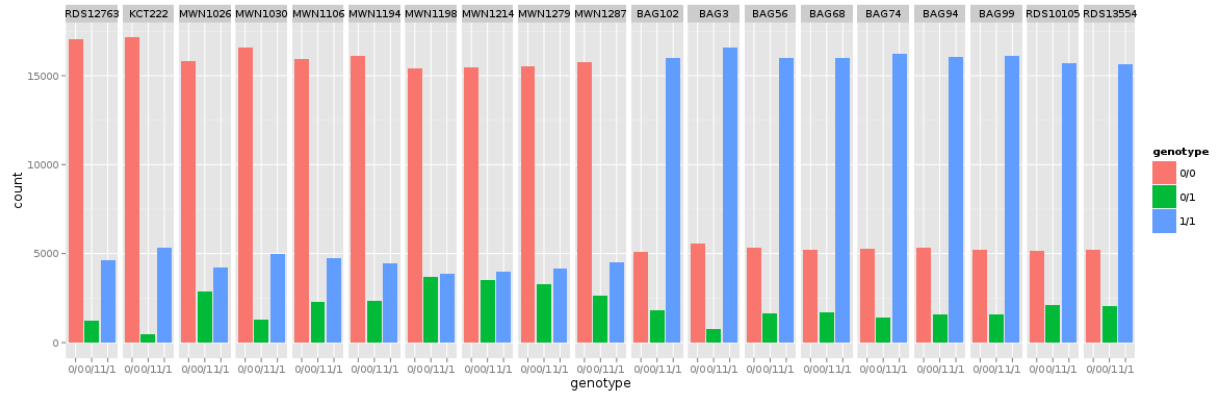
```
library(tidyr)
dm <- d %>% gather(individual, genotype, 10:28 )
```

Look at the data. Now we can plot the counts of reference/heterozygous/alternative alleles.

```
ggplot(dm, aes(genotype, fill=genotype)) + geom_bar()
```

And it is very easy to do it for each individual separately:

```
ggplot(dm, aes(genotype, fill=genotype)) +
  geom_bar() +
  facet_wrap(~individual, nrow=1)
```



Now try to change parts of the command to see the effect of various parts. Delete `, fill=genotype` (including the comma), execute. Then delete `, nrow=1`, execute.

Genomic tools session

Explore bedtools & bedops functionality

- <http://bedtools.readthedocs.org/en/>
- <http://bedops.readthedocs.org/en/>

```
## Prepare files (features.bed, genes.bed, my.genome)
```

```
cd
```

```
mkdir data/bed
cp /data/bed/* data/bed/.
cd data/bed/
```

```
## Get parts of features that overlap
```

```
bedops --intersect genes.bed features.bed
bedtools intersect -a genes.bed -b features.bed
```

```
## Merge entire features
```

```
bedops --merge genes.bed features.bed
```

```
cat *.bed | sortBed > features2.bed
bedtools merge -i features2.bed
```

```
## Get complement features
```

```
bedops --complement genes.bed features.bed

cat *.bed | sortBed > features2.bed
bedtools complement -i features2.bed -g my.genome
```

```
## Report A which overlaps B
```

```
bedops --element-of 1 genes.bed features.bed
bedtools intersect -u -a genes.bed -b features.bed
```

```
## Report B which overlaps A
```

```
bedops --element-of 1 features.bed genes.bed
bedtools intersect -u -a features.bed -b genes.bed
```

```
## Report A,B which overlap each other
```

```
bedtools intersect -wa -wb -a genes.bed -b features.bed
```

```
## What is the base coverage of features within genes?
```

```
bedmap --echo --count --bases-uniq genes.bed features.bed  
coverageBed -b genes.bed -a features.bed
```

Explore vcftools functionality

- <http://vcftools.sourceforge.net>

```
## Prepare data files
```

```
cd
```

```
mkdir data/diff
```

```
cp /data/mus_mda/00-popdata/*.txt data/diff/.
```

```
cp /data/mus_mda/00-popdata/popdata_mda.vcf.gz data/diff/.
```

```
cd data/diff/
```

```
## vcf file statistics - i.e. number of samples, number of SNPs
```

```
vcftools --gzvcf popdata_mda.vcf.gz
```

```
## Open compressed (.gz) vcf file and view it in less
```

```
vcftools --gzvcf popdata_mda.vcf.gz --recode --stdout | less -S
```

```
## Open compressed (.gz) vcf file and save it as a new file
```

```
vcftools --gzvcf popdata_mda.vcf.gz --recode --out new_vcf
```

```
## Select subset of samples
```

```
vcftools --gzvcf popdata_mda.vcf.gz --keep euro_samps.txt --recode --stdout | less -S
```

```
## Select subset of samples and SNPs based on physical position in genome
```

```
vcftools --gzvcf popdata_mda.vcf.gz --chr 11 --from-bp 22000000 --to-bp 23000000 --keep euro_samps.txt
```

```
## Select subset of samples and then select SNPs with no missing data and with minor allele frequency
```

```
vcftools --gzvcf popdata_mda.vcf.gz --keep euro_samps.txt --recode --stdout | vcftools --vcf - --max-
```

```
vcftools --gzvcf popdata_mda.vcf.gz --keep euro_samps.txt --recode --stdout | vcftools --vcf - --max-
```

```
## Calculate Fst
```

```
vcftools --vcf popdata_mda_euro.vcf --weir-fst-pop musculus_samps.txt --weir-fst-pop domesticus_samps.txt
```

Exercise: Population differentiation

```
vcftools --gzvcf popdata_mda.vcf.gz --keep euro_samps.txt --recode --stdout | vcftools --vcf - --max-
```

```
vcftools --vcf popdata_mda_euro.vcf --weir-fst-pop musculus_samps.txt --weir-fst-pop domesticus_samps.txt
```

```
cp /data/mus_mda/02-windows/genome.fa.fai .

## Create windows of 1 Mb with 100 kb step
bedtools makewindows -g <(grep '^2\|^11' genome.fa.fai) -w 1000000 -s 100000 -i winnum | awk '{print $1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,$13,$14,$15,$16,$17,$18,$19,$20,$21,$22,$23,$24,$25,$26,$27,$28,$29,$30,$31,$32,$33,$34,$35,$36,$37,$38,$39,$40,$41,$42,$43,$44,$45,$46,$47,$48,$49,$50,$51,$52,$53,$54,$55,$56,$57,$58,$59,$60,$61,$62,$63,$64,$65,$66,$67,$68,$69,$70,$71,$72,$73,$74,$75,$76,$77,$78,$79,$80,$81,$82,$83,$84,$85,$86,$87,$88,$89,$90,$91,$92,$93,$94,$95,$96,$97,$98,$99,$100}'

## Create windows of 500 kb with 500 kb step
bedtools makewindows -g <(grep '^2\|^11' genome.fa.fai) -w 500000 -s 50000 -i winnum | awk '{print $1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,$13,$14,$15,$16,$17,$18,$19,$20,$21,$22,$23,$24,$25,$26,$27,$28,$29,$30,$31,$32,$33,$34,$35,$36,$37,$38,$39,$40,$41,$42,$43,$44,$45,$46,$47,$48,$49,$50,$51,$52,$53,$54,$55,$56,$57,$58,$59,$60,$61,$62,$63,$64,$65,$66,$67,$68,$69,$70,$71,$72,$73,$74,$75,$76,$77,$78,$79,$80,$81,$82,$83,$84,$85,$86,$87,$88,$89,$90,$91,$92,$93,$94,$95,$96,$97,$98,$99,$100}'

## Create windows of 100 kb with 10 kb step
bedtools makewindows -g <(grep '^2\|^11' genome.fa.fai) -w 100000 -s 10000 -i winnum | awk '{print $1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,$13,$14,$15,$16,$17,$18,$19,$20,$21,$22,$23,$24,$25,$26,$27,$28,$29,$30,$31,$32,$33,$34,$35,$36,$37,$38,$39,$40,$41,$42,$43,$44,$45,$46,$47,$48,$49,$50,$51,$52,$53,$54,$55,$56,$57,$58,$59,$60,$61,$62,$63,$64,$65,$66,$67,$68,$69,$70,$71,$72,$73,$74,$75,$76,$77,$78,$79,$80,$81,$82,$83,$84,$85,$86,$87,$88,$89,$90,$91,$92,$93,$94,$95,$96,$97,$98,$99,$100}'

## Concatenate windows of all sizes
cat windows_*.bed > windows.bed

## Input files for bedops need to be sorted
sort-bed windows.bed > windows_sorted.bed
sort-bed popdata_mda_euro_fst.bed > popdata_mda_euro_fst_sorted.bed

bedmap --echo --mean --count windows_sorted.bed popdata_mda_euro_fst_sorted.bed | grep -v NA | tr " " "\n"
```

Note: R ggplot2 commands to plot population differentiation

Get to the Rstudio by typing *localhost:8787* in your web browser.

```
library(ggplot2)

setwd("~/data/diff")

fst <- read.table("windows2snps_fst.bed", header=F, sep="\t")

names(fst) <- c("chrom", "start", "end", "win_id", "win_size", "fst", "cnt_snps")

fst$win_size <- factor(fst$win_size, levels=c("100kb", "500kb", "1000kb"))

qplot(fst, data=fst, geom="density", fill=I("blue")) + facet_wrap(~win_size)

ggplot(fst, aes(y=fst, x=start, colour=win_size)) +
  geom_line() +
  facet_wrap(~chrom, nrow=2) +
  scale_colour_manual(name="Window size", values=c("green", "blue", "red"))

q <- quantile(subset(fst, win_size=="500kb", select="fst")[,1], prob=0.99)[[1]]

ggplot(fst, aes(y=fst, x=start, colour=win_size)) +
  geom_line() +
  facet_wrap(~chrom, nrow=2) +
  geom_hline(yintercept=q, colour="black") +
  scale_colour_manual(name="Window size", values=c("green", "blue", "red"))

## Use of variables: var=value
## $( ) can be used to assign output of command as a variable
## do not use ` (backticks) please, they're deprecated and confusing..:)
q500=$( grep 500kb windows2snps_fst.bed | cut -f 6 | Rscript -e 'quantile(as.numeric(readLines("stdin")), 0.99)')

## Call variable
echo $q500
```

```
grep 500kb windows2snpsfst.bed | awk -v a=$q500 -F $'\t' 'BEGIN{OFS=FS}{if($6 >= a){print $1,$2,$3}}'

cp /data/mus_mda/05fst2genes/Mus_musculus.NCBIM37.67.gtf .

bedtools intersect -a signif_500kb.bed -b Mus_musculus.NCBIM37.67.gtf -wa -wb | grep protein_coding
```

Quality session

Many steps of genomic data processing have some associated quality value for their results. Here we will briefly check the first and last of those. But there is no simple way to set your quality thresholds. You have to recognize completely bad data. But after that there is a continuum. Sometimes you just need an idea of the underlying biology. Find some variants for further screening. Sometimes you're trying to pinpoint particular variant causing a disease.

8.1 Read quality

Each read that comes out of the (now common) sequencing machines like Illumina or Ion Torrent has a quality score assigned with each of the bases. This is not true for the upcoming NanoPore or almost forgotten SOLiD machines, that are reading more bases a time.

8.1.1 Phred encoding

The quality in Fastq files is encoded in **Phred quality score**, a number on a logarithmic scale, similar to decibels.

Phred quality	Probability of error
20	1 in 100
40	1 in 10,000
60	1 in 1,000,000

Each Phred number is in turn encoded as a single character, so there is straightforward mapping between the bases and the quality scores. The most common mapping is Phred+33:

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJ
| | | | |
0.2.....26...31.....41
```

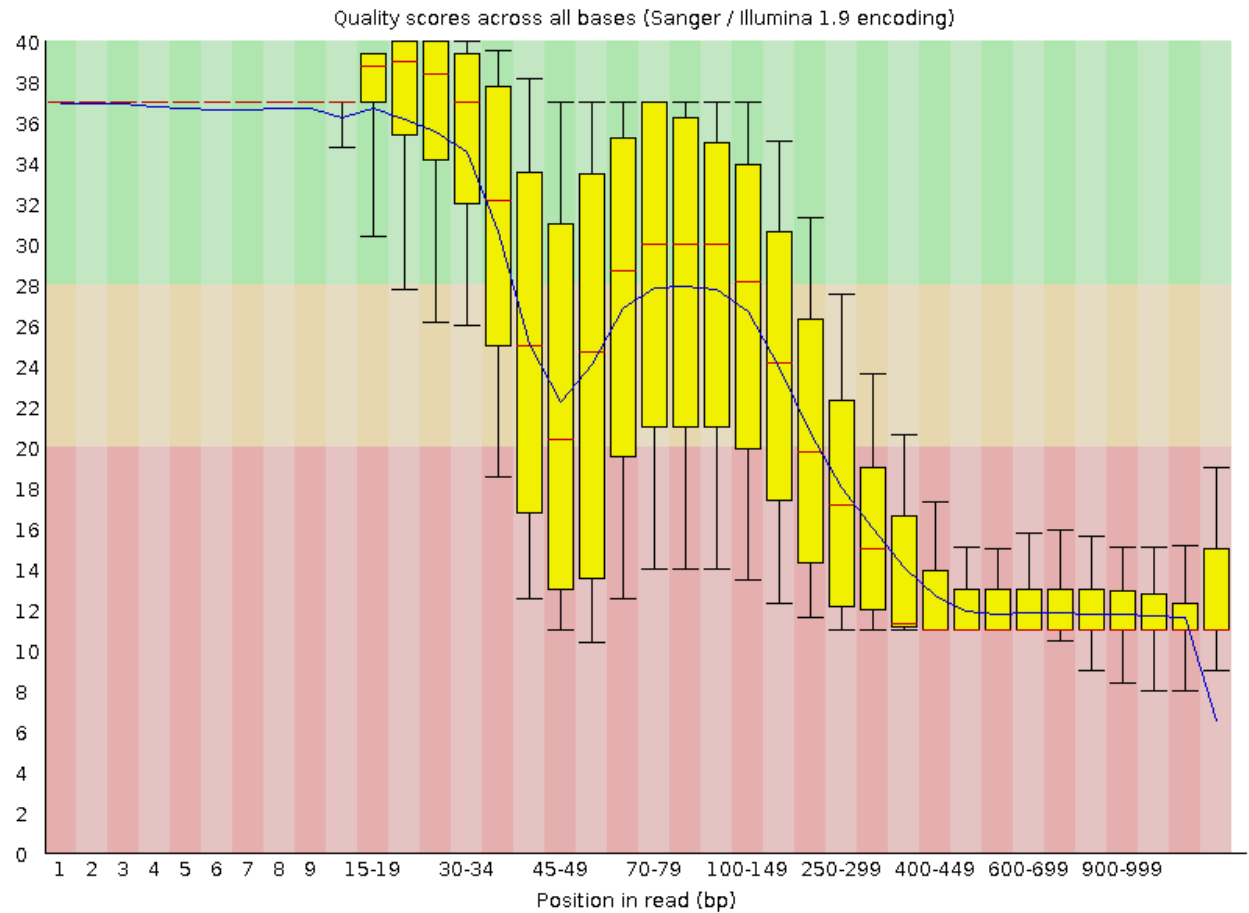
8.1.2 FastQC

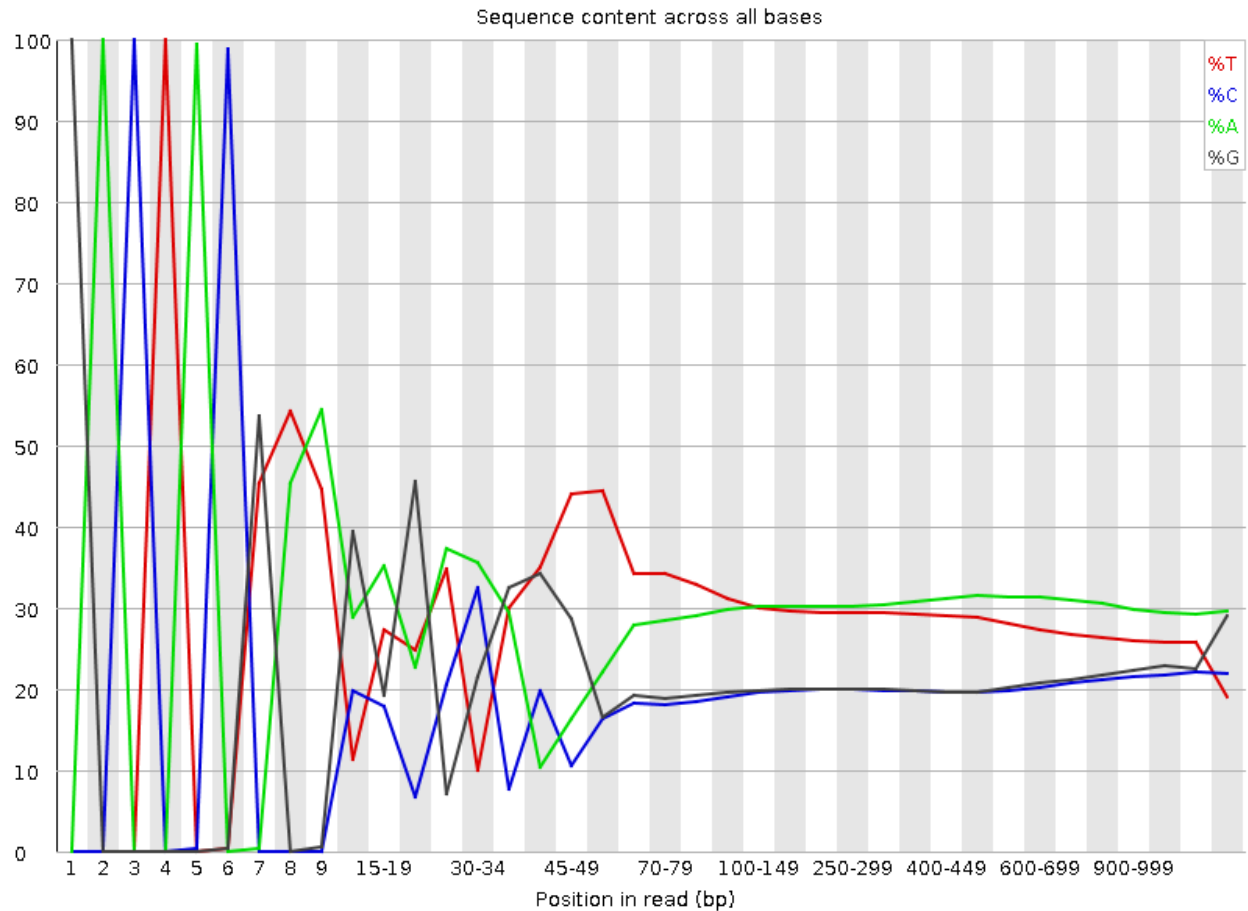
FastQC is a nice tool that you run with your data and get nice graphical reports on the quality. It is not completely installed in your images, so you can try to run a tool that was just unpacked (this is how this tool is distributed, there is no install script - a daily bread of a bioinformatician;). You have to use the full path to the tool to run it:

```
# make a project directory for the qualities
cd
mkdir data/quality
cd data
```

```
~/sw/FastQC/fastqc -o quality --noextract fastq/*.fastq
```

Now transfer the .html files from the virtual machine to yours. Open the files on your machine. You should see a report with plots like this:





8.1.3 Parsing Fastq and decoding Phred

To understand better what is in the FastQC plots, we will try to do the same using UNIX and ggplot. You should be able to understand the following pipeline, at least by taking it apart with the help of head or less. A brief description:

- `sed` replaces the leading '@' with an empty string in the first of every four lines and deletes the third of every four lines (the useless '+' line)
- `paste` merges every three lines
- `awk` selects only reads longer than 50 bases
- `head` takes first 1,000 sequences
- `awk` creates a Phred decoding table, then uses it to decode the values, outputs one row for each base (see 'tidy data')

```
IN=fastq/G59B7NP01.fastq
```

```
<$IN sed '1~4s/^@//;3~4d' |
paste - - - |
awk 'length($2) > 50' |
head -1000 |
awk 'BEGIN{OFS="\t";
      for(i=33;i<127;i++) quals[sprintf("%c", i)] = i - 33;
    }'
```

```
{
  l = length($2)
  for(i=1;i<=l;i++) {
    print $1, i, l - i, substr($2, i, 1), quals[substr($3, i, 1)];}
}'\
> quality/quals.tsv
```

8.1.4 Quality by position

The first of the FastQC plots shows a summary of base qualities according to position in the read. But it does not show quality scores for all possible positions, they are grouped into classes of similar importance. The further the base in the read, the bigger the group.

Fire up [R Studio](#) by clicking the link. Set your working directory to the directory with the quality data (in **Console**, don't forget to use tab completion):

```
setwd('~/.data/quality')
```

Now create a file where your plotting code will live, File > New file > R Script, then save it as `plots.R`. First we will read in the data.

```
d <- read.delim("quals.tsv", col.names=c("seq", "pos", "end_pos", "base", "qual"), header=F)
```

We did not include column names in the data file, but it is easy to provide them during the load via `col.names` argument. Let's look at base quality values for first 10 sequences:

```
library(ggplot2)
library(dplyr)
sel <- levels(d$seq)[1:10]
ggplot(d %>% filter(seq %in% sel), aes(pos, qual, colour=seq, group=seq)) + geom_line()
```

The qualities on sequence level don't seem to be very informative. They're rather noisy. A good way to fight noise is aggregation. We will aggregate the quality values using boxplots and for different position regions. First set up the intervals:

```
# fastqc uses bins with varying size:
# 1-9 by one, up to 75 by 5, up to 300 by 50, rest by 100
# the real bin sizes are a bit weird, use some nice approximation

breaks <- c(0:9, seq(14, 50, by=5), seq(59, 100, by=10), seq(149, 300, by=50), seq(400, 1000, by=100))

# create nice labels for the intervals
labs <- data.frame(l=breaks[1:length(breaks)-1], r=breaks[2:length(breaks)]) %>%
  mutate(diff=r-l, lab=ifelse(diff > 1, paste0(l+1, "-", r), as.character(r)))
```

Check the `breaks` and `labs` variables. In the FastQC plot there are vertical quality zones, green, yellow and red. To replicate this, we need the values of the limits:

```
# data for quality zones
quals <- data.frame(ymin=c(0, 20, 28), ymax=c(20, 28, 40), colour=c("red", "orange", "green"))

# check if the quality zones look reasonably
ggplot(quals, aes(ymin=ymin, ymax=ymax, fill=colour)) +
  geom_rect(alpha=0.3, xmin=-Inf, xmax=Inf) +
  scale_fill_identity() +
  scale_x_discrete()
```

Now we can use the `breaks` to create position bins:


```
dm <- d %>% mutate(bin=cut(pos, breaks, labels=labs$lab))
```

```
# plot the qualities in the bins
ggplot(dm, aes(bin, qual)) +
  geom_boxplot(outlier.colour=NA) +
  ylim(c(0, 45))
```

Zones and boxplots look ok, we can easily combine those two into one plot. That's pretty easy with ggplot. We use `theme` to rotate the x labels, so they're all legible. In real world application the qualities are binned first, and then the statistics are calculated on the fly, so it is not necessary to load all the data at once.

```
ggplot(dm) +
  geom_rect(xmin=-Inf, xmax=Inf, data=quals, aes(ymin=ymin, ymax=ymax, fill=colour), alpha=0.3) +
  scale_fill_identity() +
  geom_boxplot(aes(bin, qual), outlier.colour=NA, fill="yellow") +
  geom_smooth(aes(bin, qual, group=1), colour="blue") +
  theme(axis.text.x=element_text(angle = 40, hjust = 1))
```

Now we can do the base frequency plot. We already have the position bins, so just throw ggplot at it:

```
ggplot(dm, aes(bin, fill=base)) + geom_bar()
```

We're almost there, just need to normalize the values in each column so they sum up to 1. Ggplot can do it for us:

```
ggplot(dm, aes(bin, fill=base)) + geom_bar(position="fill")
```

It's possible to rearrange the legend by reordering levels of the factor. As you can see, the visual fine-tuning added the most of the code:

```
levs <- rev(c("A", "C", "G", "T", "N"))
dm %>%
  mutate(baseo=factor(base, levels=rev(levs))) %>%
  ggplot(aes(bin, fill=baseo, order=factor(baseo, levs))) + geom_bar(position="fill")
```

If you still want to get the line chart, you need to calculate the relative frequencies yourself:

```
t <- dm %>%
  select(base, bin) %>%
  table %>%
  data.frame %>%
  group_by(bin) %>%
  mutate(Freqn=Freq / sum(Freq))

t %>%
  mutate(baseo=factor(base, levels=levs)) %>%
  ggplot(aes(bin, Freqn, colour=baseo, group=baseo)) + geom_line(size=1.3)
```

Now you can think for a while about what is better about the bar chart, and what is better about the line chart.

8.2 Variant quality

And now for something completely different. You're already familiar with the data and some reformatting and plotting tools. Two VCF files that are the actual output of `freebayes` variant caller are located in:

```
/data/slavici/02-variants
```

Your task now is to look at the files, then prepare the data for loading into R. In R you will use plotting to explore some relations in the data. The `INFO` column is full of various values. You're interested only in `DP` and `TYPE`.

- create a new project directory in your data
- concatenate the two files, so you get data for chr1 and chrZ in one file (`cat`)
- get rid of the comments (they start with `#`, that is `^#` regular expression)
- use the ‘pure, concatenated data’ for:
 - extract the first 6 columns (`cut`)
 - extract DP and TYPE columns, one by one (`egrep -o 'DP=[^;]*' | sed ..`)
- merge the data before loading to R (`paste`)
- add column names while loading the data with `read.delim`
- use `ggplot` to explore the relation between read depth and quality (scatter plot, log transformed axes)
- check if variant type affects the relation between read depth and quality (facets)
- you’ll find out that you need to filter out the ‘compound’ TYPEs, do it by filtering out anything with `,` in the TYPE column (choose your way, `grep` in bash, `filter` of `dplyr`, ...)

You can save each intermediate result. To me it makes sense to save the ‘pure data’ (concatenated, without comments). Then I would use `paste` to merge the data, using `<()` to get the extracted values:

```
paste <( cut .. ) <( egrep -o .. ) <( egrep -o )
```

I would set the column names while loading the data into R.

Good luck! (We will help you;)

varq_solution by Libor.

Additional reference materials:

Links

Here is few sources that you can go through, when you get bored during the course...

9.1 Bash

- <http://www.tldp.org/LDP/abs/html/>
- <http://www.catonmat.net/blog/bash-one-liners-explained-part-three/>
- http://wiki.bash-hackers.org/howto/redirection_tutorial
- <http://wiki.bash-hackers.org/scripting/bashchanges>

9.2 R Studio

- <http://www.rstudio.com/resources/cheatsheets/>

9.3 Visual design

- one secret link you got by email
- <http://www.smashingmagazine.com/2014/03/28/design-principles-visual-perception-and-the-principles-of-gestalt/>
- <http://www.vanseodesign.com/web-design/gestalt-principles-of-perception/>
- <http://colorbrewer2.org>

9.4 Genomic tools

- <http://bedtools.readthedocs.org/en/>
- <http://bedops.readthedocs.org/en/>
- <http://vcftools.sourceforge.net>

9.5 Genomic data formats

- <http://www.1000genomes.org/wiki/Analysis/Variant%20Call%20Format/vcf-variant-call-format-version-40>
- <http://www.ensembl.org/info/website/upload/gff.html>
- <https://genome.ucsc.edu/FAQ/FAQformat.html#format1>

Best practice

This is a collection of tips, that may help to overcome the initial barrier of working with a ‘foreign’ system. There is a lot of ways to achieve the solution, those presented here are not the only correct ones, but some that proved beneficial to the authors.

10.1 Easiest ways to get UNIX

An easy way of getting UNIX environment in Windows is to install a basic Linux into a virtual machine. It’s much more convenient than the dual boot configurations, and the risk of completely breaking your computer is lower. You can be using UNIX while having all your familiar stuff at hand. The only downside is that you have to transfer all the data as if the image was a remote machine. Unless you’re able to set up windows file sharing on the Linux machine. This is the way the author prefers (you can ask;).

It’s much more convenient to use a normal terminal like PuTTY to connect to the machine rather than typing the commands into the virtual screen of VirtualBox (It’s usually lacking clipboard support, you cannot change the size of the window, etc.)

Mac OS X and Linux are UNIX based, you just have to know how to start your terminal program.

10.2 Essentials

Always use `screen` for any serious work. Failing to use `screen` will cause your jobs being interrupted when the network link fails (given you’re working remotely), and it will make you keep your home computer running even if your calculation is running on a remote server.

Track system resources usage with `htop`. System that is running low on memory won’t perform fast. System with many cores where only one core (‘CPU’) is used should be used for more tasks - or can finish your task much faster, if used correctly.

10.3 Data organization

Make a new directory for each project. Put all your data into subdirectories. Use symbolic links to reference huge data that are reused by more projects in your current project directory. Prefix your directory names with 2 digit numbers, if your projects have more than few subdirectories. Increase the number as the data inside is more and more ‘processed’. Keep the code in the top directory. It is easy to distinguish data references just by having `[0–9]{2}` – prefix.

Example of genomic pipeline data directory follows:

```
00-raw --> /data/slavici/all-reads
01-fastqc
02-mm-cleaning
03-sff
10-mid-split
11-fastqc
12-cutadapt
13-fastqc
22-newbler
30-tg-gmap
31-tg-sim4db
32-liftover
33-scaffold
40-map-smalt
50-variants
51-variants
60-gff-primers
```

Take care to note all the code used to produce all the intermediate data files. This has two benefits: 1) your results will be really **reproducible** 2) it will **save you much work** when doing the same again, or trying different settings

If you feel geeky, use `git` to track your code files. It will save you from having 20 versions of one script - and you being completely lost a year later, when trying to figure out which one was the one that was actually working.

10.4 Building command lines

Build the pipelines command by command, keeping `| less -S` (or `| head` if you don't expect lines of the output to be longer than your terminal width) at the end. Every time you check if the output is what you expect, and only after that add the next command. If there is a `sort` in your pipeline, you have to put `head` in front of the `sort`, because otherwise `sort` has to process all the data before it gives out any output.

I (Libor) do prefer the 'input first' syntax (`<file command | comm2 | comm3 >out`) which improves legibility, resembles the real world pipeline (garden hose, input tap -> garden hose -> garden sprinkler) more, and when changing the input file names when reusing the pipeline, the names are easier to find.

Wrap your long pipelines on `|` - copy and paste to bash still works, because bash knows there has to be something after `|` at the end of the line. Only the last line has to be escaped with `\`, otherwise all your output would go to the screen instead of a file.

```
<infile sort -k3,3 |
    uniq -c -s64 |
    sort -k1rn,1 \
>out
```

You can get a nice progress bar if you use `pv` (pipe viewer) instead of `cat` at the beginning of the pipeline. But again, if there is a `sort` in your pipeline, it has to consume all the data before it starts to work.

Use variables instead of hard-coded file names / arguments, especially when the name is used more times in the process, or the argument is supposed to be tuned:

```
FILE=/data/00-reads/GS60IET02.RL1.fastq
THRESHOLD=300

# count sequences in file
<$FILE awk '(NR % 4 == 2)' | wc -l
# 42308
```

```
# count sequences longer than  
<$FILE awk '(NR % 4 == 2 && length($0) > $THRESHOLD)' | wc -l  
# 14190
```

10.5 Parallelization

Many tasks, especially in Big Data and NGS, are ‘data parallel’ - that means you can split the data in pieces, compute the results on each piece separately and then combine the results to get the complete result. This makes very easy to exploit the full power of modern multi core machines, speeding up your processing e.g. 10 times. GNU `parallel` is a nice tool that helps to parallelize bash pipelines, check the manual.

Reference manual for UNIX introduction

11.1 Basic orientation in UNIX

Multiple windows (screen)

You're all used to work with multiple windows (in MS Windows;). You can have them in UNIX as well. The main benefit, however, is that you can log off and your programs keep running.

To go into a screen mode type:

```
screen
```

Once in screen you can control screen itself after you press the master key (and then a command): `ctrl+a` key. To create a new window within the screen mode, press `ctrl+a c` (create). To flip among your windows press `ctrl+a space` (you flip windows often, it's the biggest key available). To detach screen (i.e. keep your programs running and go home), press `ctrl+a d` (detach).

To open a detached screen type:

```
screen -r # -r means restore
```

To list running screens, type:

```
screen -ls
```

Controlling processes (htop/top)

`htop` or `top` serve to see actual resource utilization for each running process. `Htop` is much nicer variant of standard `top`. You can sort the processes by memory usage, CPU usage and few other things.

Getting help (man)

Just any time you're not sure about program option while building a command line, just flip to next screen window (you're always using screen for serious work), and type `man` and name of the command you want to know more about:

```
man screen
```

11.1.1 Moving around & manipulation with files and directories

Basic commands to move around and manipulate files/directories.

```
pwd      # prints current directory path
cd       # changes current directory path
ls       # lists current directory contents
ll       # lists detailed contents of current directory
```

```
mkdir  # creates a directory
rm     # removes a file
rm -r  # removes a directory
cp     # copies a file/directory
mv     # moves a file/directory
locate # tries to find a file by name
```

Usage:

cd

To change into a specific subdirectory, and make it our current working directory:

```
cd go/into/specific/subdirectory
```

To change to parent directory:

```
cd ..
```

To change to home directory:

```
cd
```

To go up one level to the parent directory then down into the directory2:

```
cd ../directory2
```

To go up two levels:

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

ls

To list also the hidden files and directories (-a) in current in given folder along with human readable (-h) size of files (-s), type:

```
ls -ash
```

mv

To move a file data.fastq from current working directory to directory /home/directory/fastq_files, type:

```
mv data.fastq /home/directory/fastq_files/data.fastq
```

cp

To copy a file data.fastq from current working directory to directory /home/directory/fastq_files, type:

```
cp data.fastq /home/directory/fastq_files/data.fastq
```

locate

This quickly finds a file by a part of its name or path. To locate a file named data.fastq type:

```
locate data.fastq
```

The `locate` command uses a database of paths which is automatically updated only once a day. When you look for some recent files you may not find them. You can manually request the update:

```
sudo updatedb
```

Symbolic links

Symbolic links refer to some other files or directories in a different location. It is useful when one wants to work with some files accessible to more users but wants to have them in a convenient location at the same time. Also, it is useful when one works with the same big data in multiple projects. Instead of copying them into each project directory one can simply use symbolic links.

A symbolic link can be created by:

```
ln -s /data/genomes/luscinia/genome.fa genome/genome.fasta
```

11.2 Exploring and basic manipulation with data

less

Program to view the contents of text files. As it loads only the part of a file that fits the screen (i.e. does not have to read entire file before starting), it has fast load times even for large files.

To view text file while disabling line wrap and add line numbers add options `-S` and `-N`, respectively:

```
less -SN data.fasta
```

To navigate within the text file while viewing use:

Key	Command
Space bar	Next page
b	Previous page
Enter key	Next line
/<string>	Look for string
<n>G	Go to line <n>
G	Go to end of file
h	Help
q	Quit

cat

Utility which outputs the contents of a specific file and can be used to concatenate and list files. Sometimes used in Czech as translated to 'kočka' and then made into a verb - 'vykočkovat';)

```
cat seq1_a.fasta seq1_b.fasta > seq1.fasta
```

head

By default, this utility prints first 10 lines. The number of first n lines can be specified by `-n` option (or by `-. .number..`).

To print first 50 lines type:

```
.. code-block:: bash
```

```
head -n 50 data.txt
```

```
# is the same as head -50 data.txt
```

```
# special syntax prints all but last 50 lines head -n -50 data.txt
```

tail

By default, this utility prints last 10 lines. The number of last n lines can be specified by `-n` option as in case of head.

To print last 20 lines type:

```
tail -n 20 data.txt
```

To skip the first line in the file (e.g. to remove header line of the file):

```
tail -n +2 data.txt
```

grep

This utility searches a text file(s) for lines matching a text pattern and prints the matching lines. To match given pattern it uses either specific string or regular expressions. Regular expressions enable for a more generic pattern rather than a fixed string (e. g. search for a followed by 4 numbers followed by any capital letter - `a [0-9] {4} [A-Z]`).

To obtain one file with list of sequence IDs in multiple fasta files type:

```
grep '>' *.fasta > seq_ids.txt
```

To print all but #-starting lines from the vcf file use option `-v` (print non-matching lines):

```
grep -v ^# snps.vcf > snps.tab
```

The `^#` mark means beginning of line followed directly by `#`.

wc

This utility generates set of statistics on either standard input or list of text files. It provides these statistics:

- line count (`-l`)
- word count (`-w`)
- character count (`-m`)
- byte count (`-c`)
- length of the longest line (`-L`)

If specific word provided it returns count of this word in a given file.

To obtain number of files in a given directory type:

```
ls | wc -l
```

The `|` symbol is explained in further section.

cut

Cut out specific columns (fields/bytes) out of a file. By default, fields are separated by TAB. Otherwise, change delimiter using `-d` option. To select specific fields out of a file use `-f` option (position of selected fields/columns separated by commas). If needed to complement selected fields (i.e. keep all but selected fields) use `--complement` option.

Out of large matrix select all but first column and row representing IDs of rows and columns, respectively:

```
< matrix1.txt tail -n +2 | cut --complement -f 1 > matrix2.txt
```

sort

This utility sorts a file based on whole lines or selected columns. To sort numerically use `-n` option. Range of columns used as sorting criterion is specified by `-k` option.

Extract list of SNPs with their IDs and coordinates in genome from vcf file and sort them based on chromosome and physical position:

```
< snps.vcf grep ^# | cut -f 1-4 | sort -n -k2,2 -k3,3 > snps.tab
```

uniq

This utility takes sorted lists and provides unique records and also counts of non-unique records (`-c`). To have more numerous records on top of output use `-r` option for `sort` command.

Find out count of SNPs on each chromosome:

```
< snps.vcf grep ^# | cut -f 2 | sort | uniq -c > chromosomes.tab
```

tr

Replaces or removes specific sets of characters within files.

To replace characters `a` and `b` in the entire file for characters `c` and `d`, respectively, type:

```
tr 'ab' 'cd' < file1.txt > file2.txt
```

Multiple consecutive occurrences of specific character can be replaced by single character using `-s` option. To remove empty lines type:

```
tr -s '\n' < file1.txt > file2.txt
```

To replace lower case to upper case in fasta sequence type:

```
tr "[:lower:]" "[:upper:]" < file1.txt > file2.txt
```

11.3 Building commands

Globbering

Refers to manipulating (searching/listing/etc.) files based on pattern matching using specific characters.

Example:

```
ls
# a.bed b.bed seq1_a.fasta seq1_b.fasta seq2_a.fasta seq2_b.fasta
ls *.fasta
# seq1_a.fasta seq1_b.fasta seq2_a.fasta seq2_b.fasta
```

Character `*` in previous example replaces any number of any characters and it indicates to `ls` command to list any file ending with `".fasta"`.

However, if we look for `fastq` instead, we get no result:

```
ls *.fastq
#
```

Character `?` in following example replaces just right the one character (`a/b`) and it indicates to `ls` functions to list files containing `seq2_` at the beginning, any single character in the middle (`a/b`) and ending with `".fasta"`

```
ls
# a.bed b.bed seq1_a.fasta seq1_b.fasta seq2_a.fasta seq2_b.fasta
ls seq2_?.fasta
# seq2_a.fasta seq2_b.fasta

ls
# a.bed b.bed seq1_a.fasta seq1_b.fasta seq2_a.fasta seq2_b.fasta
ls seq2_[ab].fasta
# seq2_a.fasta seq2_b.fasta
```

One can specifically list altering characters (a,b) using brackets []. One may also be more general and list all files having any alphabetical character [a-z] or any numerical character [0-9]:

```
ls
# a.bed b.bed seq1_a.fasta seq1_b.fasta seq2_a.fasta seq2_b.fasta
ls seq[0-9]_[a-z].fasta
# seq1_a.fasta seq1_b.fasta seq2_a.fasta seq2_b.fasta
```

TAB completion

Using key TAB one can finish unique file names or paths without having to fully type them. (try and see)

From this perspective it is important to think about names for directories in advance as it can spare you a lot time in future. For instance, when processing data with multiple steps one can use numbers at beginnings of names:

- 00-beginning
- 01-first-processing
- 02-second-processing
- ...

Variables

UNIX environment enables to use shell variables. To set primer sequence 'GATACGCTACGTGC' to variable PRIMER1 in a command line and print it on screen using echo, type:

```
PRIMER1=GATACGCTACGTGC
echo $PRIMER1
# GATACGCTACGTGC
```

Note: It is good habit in UNIX to use capitalized names for variables: PRIMER1 not primer1.

Pipes

UNIX environment enables to chain commands using pipe symbol |. Standard output of the first command serves as standard input of the second one, and so on.

```
ls | head -n 5
```

Subshell

Subshell enables to run two commands and capture the output into single file. It can be helpful in dealing with data files headers. Use of subshell enables to remove header, run the set of operations on the data, and later insert the header back to file. The basic syntax is:

```
(command1 file1.txt && command2 file1.txt) > file2.txt
```

To sort data file based on two columns without including header type:

```
(head -n 1 file1.txt && tail -n +2 file1.txt | sort -n -k1,1 -k2,2) > file2.txt
```

Subshell can be used also to preprocess multiple inputs on the fly (saving useless intermediate files):

```
paste <(< file1.txt tr ' ' '\t') <(<file2.txt tr ' ' '\t') > file3.txt
```

11.4 Advanced text manipulation (sed)

sed “stream editor” allows you to change file line by line. You can substitute text, you can drop lines, you can transform text... but the syntax can be quite opaque if you’re doing anything more than substituting *foo* with *bar* in every line (`sed 's/foo/bar/g'`).

11.5 More complex data manipulation (awk)

awk enables to manipulate text data in a very complex way. In fact, it is a simple programming language with functionality similar to regular programming languages. As such it enables enormous variability in ways of how to process text data.

It can be used to write a short script and which can be chained along with UNIX commands in one pipeline. The biggest power of *awk* is that it’s line oriented and saves you lot of boilerplate code that you would have to write in other languages, if you need moderately complex processing of text files. The basic structure of the script is divided into three parts and any of these three parts may or may not be included in the script (according to the intention of user). The first part ‘BEGIN{ }’ conducts operation before going through the input file, the middle part ‘{ }’ goes throughout the input file and conducts operations on each line separately. The last part ‘END{ }’ conducts operation after going through the input file.

The basic syntax:

```
< data.txt awk 'BEGIN{<before data processing>} {<process each line>} END{<after all lines are p
```

Built-in variables

awk has several built-in variables which can be used to track and process data without having to program specific feature.

The basic four built-in variables:

- FS - input field separator
- OFS - output field separator
- NR - record (line) number
- NF - number of fields in record (in line)

There is even more built-in variables that we won’t discuss here: RS, ORS, FILENAME, FNR

Use of built-in variables:

awk splits each line into columns based on white space. When a different delimiter (e.g. TAB) is to be used, it can be specified using `-F` option. If you want to keep this custom Field Separator in the output, you have to set the Output Field Separator as well (there’s no command line option for OFS):

```
< data.txt awk -F '$\t' 'BEGIN{OFS=FS}{print $1,$2}' > output.txt
```

This command takes file `data.txt`, extract first two TAB delimited columns of the input file and print them TAB delimited into the output file `output.txt`. When we look more closely on the syntax we see that the TAB delimiter was set using `-F` option. This option corresponds to the FS built-in variable. As we want TAB delimited columns in the output file we pass FS to OFS (i.e. output field separator) in the BEGIN section. Further, in the middle section we print out first two columns which can be extracted by numbers with \$ symbol (\$1, \$2). The numbers correspond to position of the column in the input file. We could, of course, use for this operation the `tr` command which is even simpler. However, the *awk* enables to conduct any other operation on given data.

Note: The complete input line is stored in \$0.

The NR built-in variable can be used to capture each second line in a file type:

```
< data.txt awk '{ if(NR % 2 == 0){ print $0 } }' > output.txt
```

The % symbol represents modulo operator which returns the remainder of division. The if() condition is used to decide on whether the modulo is 0 or not.

Here is a bit more complex example of how to use awk. We write a command which retrieves coordinates of introns from coordinates of exons.

Example of input file:

GeneID	Chromosome	Exon_Start	Exon_End
ENSG00000139618	chr13	32315474	32315667
ENSG00000139618	chr13	32316422	32316527
ENSG00000139618	chr13	32319077	32319325
ENSG00000139618	chr13	32325076	32325184
...

The command is going to be as follows:

When we look at the command step by step we first remove header and sort data based on GeneID and Exon_Start columns:

```
< exons.txt tail -n +2 | sort -k1,1 -k3,3n | ...
```

Further, we write a short script using awk to obtain coordinates of introns:

```
... | awk -F $'\t' 'BEGIN{OFS=FS}{
    if(NR==1){
        x=$1; end1=$4+1;
    }else{
        if(x==$1) {
            print $1,$2,end1,$3-1; end1=$4+1;
        }else{
            x=$1; end1=$4+1;
        }
    }
}' > introns.txt
```

In the BEGIN{} part we set TAB as output field separator. Further, using NR==1 test we set GeneID for first line into x variable and intron start into end1 variable. Otherwise we do nothing. For others records NR > 1 condition x==\$1 test if we are still within the same gene. If so we print exon end from previous line (end1) as intron start and exon start of current line we use as intron end. Next, we set new intron start (i.e. exon end from current line) into end1. If we have already moved into new one x<>\$1) we repeat procedure for the first line and print nothing waiting for next line.

Important NGS formats

A selection of the most commonly used formats in NSG data processing pipelines.

12.1 FASTQ - Short reads

Sequencing instruments produce not only base calls, but usually can assign some quality score to each called base. Fastq contains multiple sequences, and each sequence is paired with quality scores for each base. The quality scores are encoded in text form.

- <http://maq.sourceforge.net/fastq.shtml>

12.2 SAM - Reads mapped to reference

SAM stands for Sequence Alignment/Mapping format. It includes parts of the original reads, that were mapped to a reference genome, together with the position where they belong to. There is an effective binary encoded counterpart called **BAM**.

- <http://samtools.github.io/hts-specs/SAMv1.pdf>

12.3 BED and GFF - Annotations

Annotations are regions in given reference genome with some optional additional information. BED is very simple and thus easy to work with for small tasks, GFF (General Feature Format) is a comprehensive format allowing feature nesting, arbitrary data fields for each feature and so on.

- <http://genome.ucsc.edu/FAQ/FAQformat.html#format1>
- <http://www.ensembl.org/info/website/upload/gff.html>

12.4 VCF - Variants in individuals

VCF stands for Variant Call Format. Given a reference and a set of sequenced individuals, VCF is a format to store the differences in these individuals, compared to the reference, efficiently. There is also a binary counterpart **BCF**.

- <http://samtools.github.io/hts-specs/VCFv4.2.pdf>

Additional exercises

These are tasks that do not fit any particular session, but we still consider them interesting enough to share them with you.

13.1 Counting heads

This is a nice example where bash can be used to solve a combinatorial problem by enumerating all the possibilities. And it is a long pipeline, so you have a lot of code to examine;)

Eight people are sitting around a circular table. Each has a coin. They all flip their coins. What is the probability that no two adjacent people will get heads?

The basic idea is that there is not much possibilities (only 2 to the power of 8, that is 256). We can just enumerate all the combinations and check if there is two adjacent heads.

This is the final solution, take your time to take it apart to see what each piece does.

```
(echo "obase=2;"; printf "%d\n" {0..255}) | bc | # generate numbers 0-255 in binary
  sed 's/^/0000000/' | egrep -o '{8}$' | # pad the output to 8 characters
  sed 'h;G;s/\n// ' | # double the pattern on each line to simulate ring
grep -v 11 | # ignore all patterns where two heads (1) are next
wc -l # count the rest
```

To get a more understandable code, we can split it to functional parts. Then we can just play and try different implementations of the parts:

```
generate () { (echo "obase=2;"; printf "%d\n" {0..255}) | bc ;}
pad () { sed 's/^/0000000/' | egrep -o '{8}$' ;}
ring () { sed p | paste - - | tr -d "\t" ;}
```

```
generate | pad | ring | grep -v 11 | wc -l
```

These are alternative solutions - you can paste them one by one, and check if the pipe is still working.

```
generate () { (echo "obase=2;"; echo {0..255} | tr " " "\n") | bc ;}
generate () { (echo "obase=2;"; echo {0..255} | xargs -n1) | bc ;}
generate () { (echo "obase=2;"; printf "%d\n" {0..255}) | bc ;}

pad () { awk '{print substr(sprintf("0000000%s", $0), length);}' ;}
pad () { sed 's/^/0000000/' | rev | cut -c-8 | rev ;}
pad () { sed 's/^/0000000/' | egrep -o '{8}$' ;}

ring () { sed p | paste - - | tr -d "\t" ;}
```

```
ring () { sed 'h;G;s/\n//' ;}

generate | pad | ring | grep -v 11 | wc -l
```

The question was asking for the probability, that's one more division:

```
echo "scale=3;$( generate | pad | ring | grep -v 11 | wc -l ) / 256" | bc
```

13.1.1 Solutions by participants

One way to get a shorter (but much slower) solution is to ignore the binary conversion altogether, just use a huge list of decimal numbers and filter out anything that does not look like binary. Few variants follow:

```
seq -w 0 11111111 | grep ^[01]*$ | awk '!/11/ && !/^1.*1$/' | wc -l
seq -w 0 11111111 | grep ^[01]*$ | grep -v -e 11 -e ^1.*1$ | wc -l
seq -w 0 11111111 | awk '/^[01]*$/ && !/11/ && !/^1.*1$/' | wc -l
seq -w 0 11111111 | awk '!/[2-9]/ && !/11/ && !/^1.*1$/' | wc -l
seq -w 0 11111111 | grep -v -e [2-9] -e 11 -e ^1.*1$ | wc -l
```

I believe there are still a few ways to make it shorter;)

13.2 Dimensionality reduction

Methods like PCA and MDS (sometimes called PCoA to increase the confusion) are usually regarded as black box by many. Here we try to present a simple example that should help with getting a better idea on what are these magic boxes actually doing.

13.2.1 Load and visualize your data set

Let's link the project directory to your own data directory:

```
ln -s /data/banana/ ~/data
```

Now you can go to R and load the data:

```
setwd('~/data/banana')
d <- read.csv("webapp/data/rotated.csv")
```

Plot the data to look what we've got:

```
library(ggplot2)
ggplot(d, aes(x, y)) + geom_point() + coord_equal()
```

13.2.2 Correct the distortion

Maybe you can already recognize what's in your data. But it appears to be a bit .. rotated. Here is a code for 3d rotation of points, copy, paste and run it in your R session:

```
# create a 3d rotation matrix
# https://www.math.duke.edu/education/ccp/materials/linalg/rotation/rotm3.html
rotX <- function(t) matrix(c(cos(t), sin(t), 0, -sin(t), cos(t), 0, 0, 0, 1), nrow=3)
rotY <- function(t) matrix(c(1, 0, 0, 0, cos(t), sin(t), 0, -sin(t), cos(t)), nrow=3)
```

```

rotZ <- function(t) matrix(c(cos(t), 0, -sin(t), 0, 1, 0, sin(t), 0, cos(t)), nrow=3)
rot3d <- function(tx, ty, tz) rotX(tx) %*% rotY(ty) %*% rotZ(tz)

# rotate a data frame with points in rows
rot3d_df <- function(df, tx, ty, tz) {
  rmx <- rot3d(tx, ty, tz)
  res <- data.frame(t(apply(df, 1, function(x) rmx %*% as.numeric(x))))
  colnames(res) <- colnames(df)
  res
}

```

Now try to rotate the object a bit, so we can see it better. Try to find good values for the rotation yourself (numbers are in radians, $0..2\pi$ makes sense):

```

dr <- rot3d_df(d, .9, .1, 2)
ggplot(dr, aes(x, y)) + geom_point() + coord_equal()

```

Enter PCA. It actually finds the best rotation for you. Even in a way that the first axis has the most variability (longest side of the object), the second axis has the maximum of the remaining variability etc.

```

pc <- prcomp(as.matrix(dr))
ggplot(data.frame(pc$x), aes(PC1, PC2)) + geom_point() + coord_equal()
ggplot(data.frame(pc$x), aes(PC1, PC3)) + geom_point() + coord_equal()
ggplot(data.frame(pc$x), aes(PC2, PC3)) + geom_point() + coord_equal()

```

13.2.3 MDS

Metric MDS (multidimensional scaling) with *euclidean* distance equals to PCA. We will use the non-metric variant here, which tries to keep only the order of pairwise distances, not the distances themselves. You prefer MDS when you want to use a different distance than *euclidean* - we're using *manhattan* (*taxicab*) distance here:

```

library(MASS)
dmx <- dist(dr, "manhattan")
mds <- isoMDS(dmx)
ggplot(data.frame(mds$points), aes(X1, X2)) + geom_point() + coord_equal()

```

13.2.4 Shiny

And now there is something you definitely wanted, while you were trying to find the good values for rotation of your object:

```
setwd('/data/banana/webapp/')

```

Now File > Open, and open `server.R`. There should be a green Run App button at the top right of the editor window. Click that button!

Supplemental information:

Course materials preparation

This section contains the steps that we did to produce the materials that course participants got ready-made. That is the **linux machine image**, **online documentation** and the **slide deck**.

14.1 Online documentation

Login to <https://github.com>. Create a new project called *ngs-course*, with a default readme file.

Clone the project to local machine and initialize *sphinx* docs. Choose SSH clone link in GitHub.

```
git clone git@github.com:libor-m/ngs-course.git
```

```
cd ngs-course
```

```
# use default answers to all the questions
# enter project name and version 1.0
sphinx-quickstart
```

Now track all files created by *sphinx-quickstart* in current directory with *git* and publish to GitHub.

```
git add .
git commit -m 'empty sphinx project'

# ignore _build directory in git
echo _build >> .gitignore
git add .gitignore
git commit -m 'ignore _build directory'

# publish the first docs
# setting up argument less git pull with '-u'
git push -u origin master
```

To get live view of the documents, login to <https://readthedocs.org>. Your *GitHub* account can be paired with *Read the Docs* account in *Edit Profile/Social Accounts*, then you can simply ‘import’ new projects from your GitHub with one click. Import the new project and wait for it to build. After the build the docs can be found at <http://ngs-course.readthedocs.org> (or click the View button).

Now write the docs, commit and push. Rinse and repeat. Try to keep the commits small, just one change a time.

```
git add _whatever_new_files_
git commit -m '_your meaningful description of what you did here_'
git push
```

References that may come handy:

- [Thomas Cokelaer's cheat sheet](#)

Use <http://goo.gl> to shorten a link to www.seznam.cz, to get a tracking counter url for the network connectivity test.

14.2 VirtualBox image

14.2.1 Create new VirtualBox machine

- Linux/Debian (32 bit)
- 1 GB RAM - this can be changed at the users machine, if enough RAM is available
- 12 GB HDD as system drive (need space for basic system, gcc, rstudio and some data)
- setup port forwarding
 - 2222 to 22 (ssh, avoiding possible collisions on linux machines with sshd running)
 - 8787 to 8787 (rstudio server)
 - 5690 to 5690 (rstudio + shiny)

14.2.2 Install Debian

Download Debian net install image - use i386 so there is as few problems with virtualization as possible. Not all machines can virtualize x64.

<https://www.debian.org/CD/netinst/>

Connect the iso to IDE in the virtual machine. Start the machine. Choose `Install`.

Mostly the default settings will do.

- English language (it will cause less problems)
- Pacific time zone (it is connected with language, no easy free choice;)
- hostname `node`, domain `vbox`
- users: `root:debian`, `user:user`
- simple partitioning (all in one partition, no LVM)
- Czech mirror to get fast installer file downloads
- pick only SSH server and Standard system utilities

Log in as root:

```
apt-get install sudo
usermod -a -G sudo user
```

Login as user (can be done by `su user` in root shell):

```
# colrize prompt - uncomment force_color_prompt=yes
# add ll alias - uncomment alias ll='ls -l'
# fast sort and uniq
# export LC_ALL=C
# maximal width of man
# export MANWIDTH=120
```



```
# # wget impersonating normal browser
# # good for being tracked with goo.gl for example
# alias wget='H="--header"; wget $H="Accept-Language: en-us,en;q=0.5" $H="Accept: text/html,application/javascript;q=0.9,*/*;q=0.8"'
nano ~/.bashrc
. ~/.bashrc

# set timezone so the time is displayed correctly
echo "TZ='Europe/Prague'; export TZ" >> ~/.profile

# some screen settings
cat > ~/.screenrc <<EOF
hardstatus alwayslastline
hardstatus string '%{= kG}%{G}%H%? %1`%?%{g}][%= %{= kw}%-w%{+b yk} %n*%t%?(%u)%? %{-%}%+w %=%{g}][%
defscrollback 20000

startup_message off
EOF

# everyone likes git and screen
sudo apt-get install git screen pv curl wget

# add important stuff to python
sudo apt-get install python-dev python-pip python-virtualenv

# java because of fastqc
sudo apt-get install openjdk-7-jre-headless
```

This is what it takes to create a basic usable system in VirtualBox. We can shut it down now with `sudo shutdown -h now` and take a snapshot of the machine. If any installation goes haywire from now on, it's easy to revert to this basic system.

14.2.3 Install additional software

R is best used in RStudio - server version can be used in web browser.

```
mkdir sw
cd sw

# install latest R
# http://cran.r-project.org/bin/linux/debian/README.html
sudo bash -c "echo 'deb http://mirrors.nic.cz/R/bin/linux/debian wheezy-cran3/' >> /etc/apt/sources.list"
sudo apt-key adv --keyserver keys.gnupg.net --recv-key 381BA480
sudo apt-get update
sudo apt-get install r-base
sudo R
> update.packages(libPaths(), checkBuilt=TRUE, ask=F)
> install.packages(c("ggplot2", "dplyr", "reshape2", "GGally", "stringr", "vegan", "svd", "tsne", "tidyr"))

# RStudio with prerequisites
wget http://ftp.us.debian.org/debian/pool/main/o/openssl/libssl0.9.8_0.9.8o-4squeeze14_i386.deb
sudo dpkg -i libssl0.9.8_0.9.8o-4squeeze14_i386.deb
sudo apt-get install gdebi-core
wget http://download2.rstudio.org/rstudio-server-0.98.1081-i386.deb
sudo gdebi rstudio-server-0.98.1081-i386.deb
```

There are packages that are not in the standard repos, or the versions in the repos is very obsolete. It's worth it to

install such packages by hand, when there is not much dependencies.

```
# pipe viewer
wget -O - http://www.ivarch.com/programs/sources/pv-1.5.7.tar.bz2 | tar xvj
cd pv-1.5.7/
./configure
make
sudo make install
cd ..

# parallel
wget -O - http://ftp.gnu.org/gnu/parallel/parallel-latest.tar.bz2|tar xvj
cd parallel-20141022/
./configure
make
sudo make install

# tabtk
git clone https://github.com/lh3/tabtk.git
cd tabtk/
# no configure in the directory
make
# no installation procedure defined in makefile
# just copy the executable to a suitable location
sudo cp tabtk /usr/local/bin

# fastqc
cd ~/sw
wget http://www.bioinformatics.babraham.ac.uk/projects/fastqc/fastqc_v0.11.3.zip
unzip fastqc_v0.11.3.zip
rm fastqc_v0.11.3.zip
chmod +x FastQC/fastqc

# and some more:
# bcftools, samtools, vcftools, htslib
```

14.2.4 Sample datasets

Use data from my nightingale project, subset the data for two selected chromosomes.

```
# see read counts for chromosomes
samtools view 41-map-smalt/alldup.bam | mawk '{cnt[$3]++;} END{for(c in cnt) print c, cnt[c];}' | sort
# extract readnames that mapped to chromosome 1 or chromosome Z
mkdir -p kurz/00-reads
samtools view 41-map-smalt/alldup.bam | mawk '($3 == "chr1" || $3 == "chrZ"){print $1;}' | sort > kurz/00-reads
parallel "fgrep -A 3 -f kurz/readnames {} | grep -v '^--$' > kurz/00-reads/{/}" ::: 10-mid-split/*.fa

# reduce the genome as well
# http://edwards.sdsu.edu/labsite/index.php/robert/381-perl-one-liner-to-extract-sequences-by-their-
perl -ne 'if(/^>(\S+)/){$c=grep{/^$1$/}qw(chr1 chrZ)}print if $c' 51-liftover-all/lp2.fasta > kurz/20

# subset the vcf file with grep
# [the command got lost;]
```

Prepare the /data folder.

```
sudo mkdir /data
sudo chmod user:user /data
```

Transfer the files to the VirtualBox image, /data directory using WinSCP.

Do the quality checks:

```
cd /data/slavici
~/sw/FastQC/fastqc -o 04-read-qc --noextract 00-reads/*

# update the file database
sudo updatedb
```

14.2.5 Packing the image

Now shut down the VM and click in VirtualBox main window `File > Export appliance`. Upload the file to a file sharing service, and use the *goo.gl* url shortener to track the downloads.

14.3 Slide deck

Libor's slide deck was created using Adobe InDesign (you can get the CS2 version almost legally for free). Vasek's slide deck was created with Microsoft Powerpoint. Images are shamelessly taken from the internet, with the 'fair use for teaching' policy ;)

Slide decks

Starter session (Libor)

Genomics session (Vasek)

Advanced UNIX session (Vasek)

Graphics session (Libor)

Genomic tools (Vasek)