Unix tutorial, session 1

H. Seitz (IGH)

December 5, 2013

Contents

1 How to get there

On a computer running MacOS X (and, presumably, also for the future versions of MacOS): you need to open a "terminal". This can be done by opening the "Applications" folder, then the "Utilities" sub-folder. You should see the program "terminal" (the icon is a kind of stylized black screen): click on it. It looks like, on some MacOS X installations, "terminal" is not available, but you should find another program called "X11" (also in the "Utilities" sub-folder of "Applications), which can be used the same way.

On a Windows computer: you can install a Unix emulator, called "Cygwin" (available at: `http://www.cygwin.com/`). When you run that program, it opens a window where you can type and use Unix commands; ask me if you have problems installing it.

For the long run, if you really like Unix, you can also install Unix on a PC (on a Mac, it's a little more complicated, but still possible). I installed "Ubuntu" (available at `http://www.ubuntu.com/`) on mine, and it is very easy to administer; you can opt for a double partition Windows+Ubuntu (which means that you keep Windows on your computer, and you will have the choice between the two OS's every time you start your computer).

When you start the "terminal" program on a Mac, or cygwin on a Windows PC, a window will appear, where you can type commands. That window is called a "terminal".

2 Files and directories

2.1 Navigating between directories

In the Unix vocabulary, a folder is more often called a "directory". Files are stored in directories, and directories can be stored within larger directories.

To change directory, the command to type is:

```
cd
```

(for "change directory"). For example, if you are in the directory "Large", which contains a sub-directory called "Small", type:

```
cd Small
```

(be careful: every Unix command is case-sensitive; if the directory name has a capital "s", then type it with a capital "s").

This command allows you to move downwards in the directory arborescence (from the large directory, to its sub-directories). To go up one step, type:

```
cd ..
```

The ".." symbol means: "the directory just above the current directory", so typing

```
cd ..
```

means: change to the directory immediately above the current one.

You can also take short cuts: if you want to move to the sub-directory "Tiny", which is in your directory "Small" (while you are currently in "Large", which contains "Small"), then you can type:

```
cd Small/Tiny
```

(separating the directory names by a slash, "/"). This is faster than moving one step at a time:

```
cd Small
```

(you hit "Enter", then you are in "Small"), then:

```
cd Tiny
```

(which brings you to "Tiny").

This was the first illustration of a concept that we shall see at many more occasions: there are often several ways to do one particular thing; they all have the same ultimate result, but some can be more efficient (*i.e.*: faster), or easier (hence, less prone to errors) than others.

Here, the slash was a delimiter between directory names; it can also be, generally, a part of the directory name: you can add it to the end of the directory name without any difference in the output of the command:
`cd Small/`
does the exact same job (*i.e.*: moving one step downwards, to the sub-directory called "Small") than
`cd Small`
This slash is here completely useless, but at least, it makes it absolutely clear that "Small" is a directory (and not a file), hence in most documentations, the directories are always named with a slash at the end, so that there is no ambiguity for the reader. I will also use that convention from now on: all the occurences of directory names in this document, will have a slash at the end (and I will stop putting quotes around them).

It is also possible to move up by several steps: in that case, the command is:
`cd ../..`
or, identically:
`cd ../../`
if you want to move, let's say, from `Tiny/` to `Large/` (skipping the intermediary stop in `Small/`). You can also move by more than two steps (things like `cd ../../../../../`, for example).

Exercise

If you are in `Small/`, what happens if you type:
`cd ../Small`
? And if you type
`cd Tiny/../..`
?

2.2 Creating and deleting directories

To create a directory (let's say we want to name it `Sequence_analyses/`), the command is:
`mkdir Sequence_analyses`
(with or without a slash at the end of the name). "Mkdir" stands for "make directory".

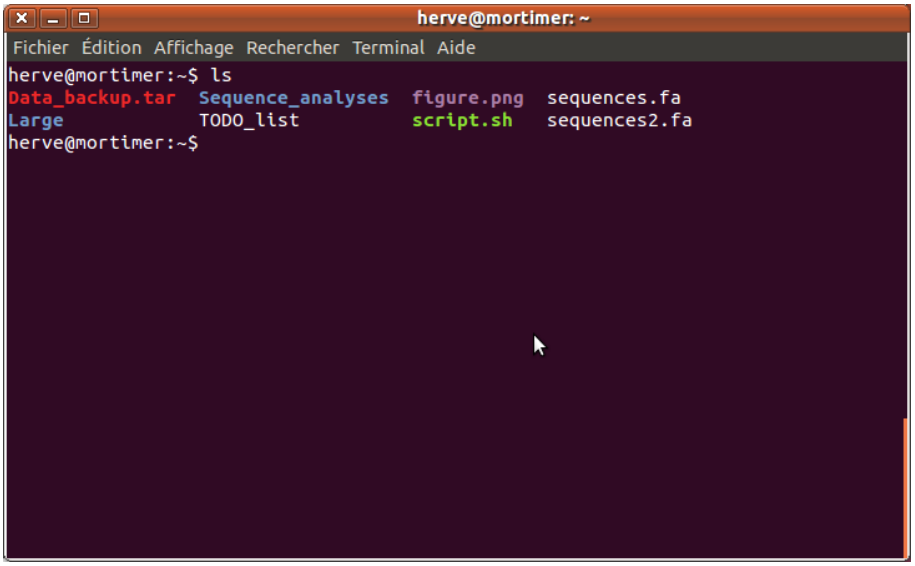
To delete a directory (let's say we want to delete the directory `Sequence_analyses/`, that we have just created), the command is:
`rmdir Sequence_analyses`
(for "remove directory"). This command works only if the directory to be deleted is empty – if it's not, then you will have to remove all the files and sub-directories it contains, before you can actually remove it (that's for safety, to make sure that you won't accidentally delete the directory which contains all the results of your 12-year-long PhD ...).

Note that the directory name I chose doesn't contain a space (I decided to put an underscore, "_", between "Sequence" and "analyses"). That's because, when you type commands, a space is interpreted as a delimiter between two fields (for example, there is a space between the name of the

command, and the command's argument: between "`cd`" and ".." in "`cd ..`", for example). It is actually possible to use directory names with spaces, but let's not go into that kind of details now.

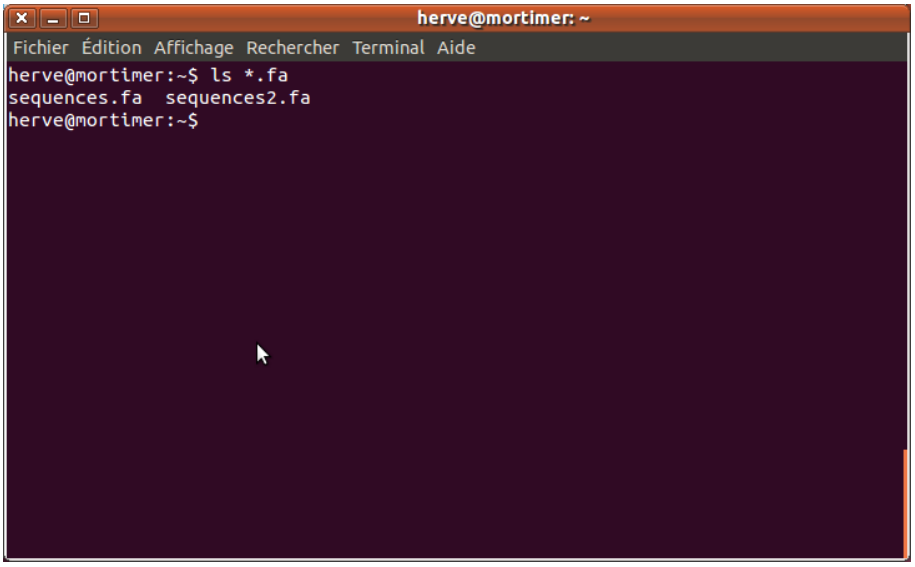
2.3 Listing what's in a directory

If you are in a directory, and wish to know what is inside, use the command
`ls`
(for: "list"). When I type it in the top directory of my account on mortimer, here is the output:



With most Unix installations, there is a color code, which helps reading this output (regular files are shown in white, directories in blue, archives and compressed files in red, executable files in green). On other installations, everything is in white.

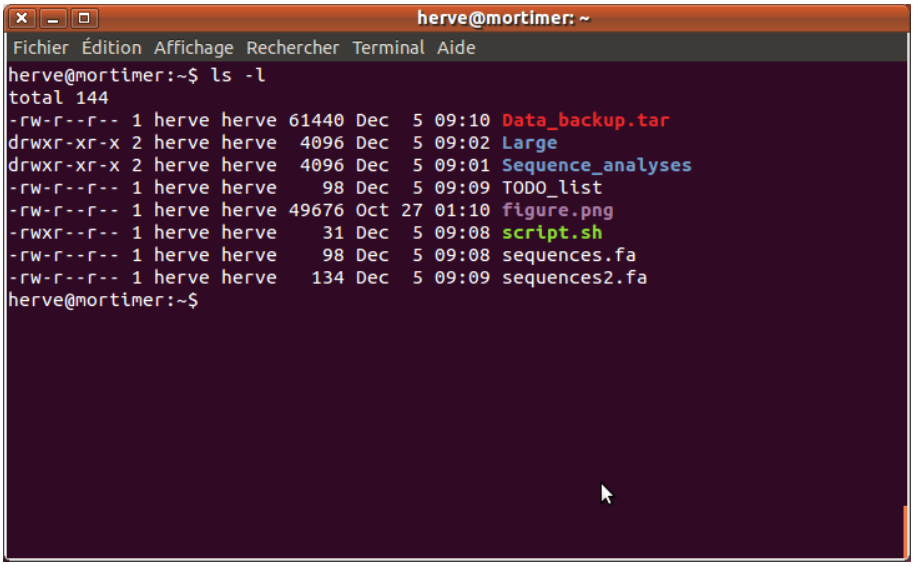
If your directory is full of things (if it contains hundreds of files or directories), you might want to restrict the list which is displayed (a list of hundreds of names is not always easy to parse, and sometimes you cannot even see it completely in your terminal). In that case, you can use a wildcard: the asterisk. Typing
`ls *.fa`
will list all the files and directories whose names are: (anything, represented by the asterisk), then ".fa". In the top directory of my account, this will give:



Now it's time to talk about "options". These are additional arguments, that you enter after the command name, and which alter the behaviour of the command. Most of these options are called with a dash, followed by a one-letter code. For example,

`ls -l`

is a variation of the `ls` that we just saw: when this option is invoked, `ls` does not only print the list of files and directories contained in the current directory, it also gives some info on each file or directory: the permission associated to that file or directory (can I open and read it? modify it? execute it? are these actions allowed to other users?,) their size (in bytes), the date and time when they were last modified, ...:



(the series of "r", "w" and dashes, at the beginning of each line, describes the permissions associated to these files; we'll talk about that later)

`ls` has many more options (`ls -S` sorts the files by size, `ls -R` lists the directories and their subdirectories recursively – instead of just showing only the subdirectories present in the current directory, it will also list their content, including their own subdirectories, including their own content,

and so on – , ...; to find out what options are available with a given command, see section 3.1, on page 7). It is also possible to combine several options (for example, `ls -l -S` will show many informations for each file, and it will sort them by size).

2.4 Handling files

We have not yet seen how files are created. We'll come to that later, for the moment, let's assume that you have some files, and you want to do things with them.

You can remove them, with the command:

`rm`

(for "remove"), followed by a space, then the name of the file you want to delete (for example, if I type

`rm sequences.fa`

in the top directory of my account on my laptop, I will remove the file 'sequences.fa'). Be careful! In command-line mode, there is no "recycle bin", or "trash" like with the graphical interfaces of MacOS or Windows: you cannot rescue a deleted file, once it's gone, it's gone. So be extremely cautious when you use that command ... There is a way to make things a little safer (`rm` will ask for a confirmation before it actually deletes the file), but it's not activated by default.

Like every Unix command, `rm` can also work with wildcards: typing

`rm *.fa`

in the top directory of my account on my laptop would delete all the files whose names end with ".fa".

You can move a file, from one directory to the other, with

`mv`

(for: "move"). The syntax is: `mv name_of_file destination`; so, if I have a file named 'file1.fa' in directory `Large/`, and I want to move it to the subdirectory `Small/`, I'll have to type:

`mv file1.fa Small`

That command, `mv`, can also be used to rename a file: in that case, the syntax is:

`mv old_name new_name`

(hence, typing `mv file1.fa file2.fa` will rename my 'file1.fa' into 'file2.fa').

Exercise

Let's assume that the current directory is `Small/`, and that there is a file named 'file1.fa' in the upstream directory (`Large/`). How can I delete it without changing directory (*i.e.*: without, first, typing `cd ..`)?

And if I want to remove the subdirectory `Tiny/` of my current directory `Small/`, and if `Tiny/` is full of files (let's say: hundreds of files; I'm a computer scientist, so I'm lazy, so I don't want to type these hundreds of names one after the other): how can I delete all these files (in order to be able to remove the subdirectory `Tiny/` itself: *cf* section 2.2, page 3)? I am really lazy, so I don't even want to change directory: how can I do that, from my current directory `Small/`?

3 Useful tips

3.1 The first (and only) commandment of the computer scientist

Each and every Unix program (including the small commands we are talking about: `mv`, `ls`, ...) comes with a manual. To display the manual related to a given command (let's say: `ls`), type:
`man ls`
("man" stands for "manual"). That will display a short description of the command, and an exhaustive list of the available options. You can navigate through that manual using the arrows of your keyboard (up and down), as well as the spacebar, which scrolls down by a full page at each time: faster than the "down" arrow if you want to scroll down a lot), then you exit that manual page by hitting `q` (for: "quit").

The *man* can appear very complicated, very technical, and (at least, the first times), you probably won't understand much. But most of the time, the help you're looking for is in there, so make an effort ...

Another, similar, documentation program is called "info" (type `info ls` to get some help on the `ls` command, for example), but once again, it can be very technical (and very often, it's the exact same than the *man* page).

Every user is expected to make the effort to try to solve his problems by himself before asking for some help – that's a convention among Unix users. That can look boring, that's certainly hard, but it's also a matter of respect to the people you're asking help to. If you *google* a little bit, or if you browse Unix discussion forums, you will certainly find many occurences of the acronym "RTFM" (litterally: "read the f*cking manual"). That's the typical answer people give to somebody who doesn't seem to make that effort, and who wants others to do the work for him. So, if you have a problem ... read the manual first, and try the ideas it could give you. In the long run, that's the best way to learn. That's why it's the first, and only commandment of the computer scientist: read the documentation ...

If really you couldn't solve your problem, you will certainly find some useful hints by *googling* a short description of your problem (or the error message that you got). If you had a problem, then, most likely, somebody else on the Internet had the same problem one day, and (s)he probably asked for some help on a web forum ... where you will find the answers that people gave him (her).

After all that, if you are still stuck, then you can ask somebody (on these web forums, for example). Describe the problem accurately, and show that you made the effort to search for a solution by yourself ("on that web forum (http://...), I found this, but I cannot apply the same solution, because ..."), and you will certainly get some help by a more experienced user.

3.2 Automatic completion

Some command names can be long; some file names can be long; so, typing a command (command name, plus the options, plus the file names on which you want that command to apply) can be tedious. There's a huge time saver: automatic completion. That's a feature of the Unix command-line: once you typed the first letters of a command (or a file) name, you press "Tab" (the tabulation key, on the left side of your keyboard), and it will complete automatically the name you started to type – unless there is an ambiguity (in that case, the completion will stop at the position of the ambiguity,

and if you press "Tab" a second time, it will display all the possible command or file names starting with these characters).

For example, I want to remove an empty directory with the command `rmdir`. If I just type "rm" and press "Tab", then the computer beeps, and does not complete the name of the command: that's because there are several possibilities (including `rm`, the command that removes files). I hit "Tab" a second time, and I can see the list of all possible commands whose names start with "rm". So I type one more letter ("d"), then press "Tab" again: now it works, the name is automatically completed ("ir" appears automatically after "rmd"). Of course, this was possible only because no other program had a name starting with "rmd". If one day you install such a program on your computer, then the automatic completion won't work on "rmd", you will have to add more letters, until there is no name ambiguity anymore.

It's the same for file names: if the current directory contains a file named 'file1.fa' and a 'file2.fa' (and nothing else starting with "fi"), then if you type "rm fi" then press "Tab", the file name will be automatically completed to the position of the ambiguity ("1" vs. "2", so the letters "le" will have been added automatically), so that I will just have to type "1" or "2", then press "Tab" once again for the file name to be completed till the end.

Yes, it sounds awfully complicated, but it's very easy to use, and it really becomes a reflex once you've used it a little bit – and it really saves a lot of typing.

3.3 Recalling a previously-typed command

Instead of typing a command, hit the "up" arrow: the last command you typed will appear, already typed (and you can modify it before executing it); if you press the "up" arrow multiple times, you will be able to recall older commands.

This is particularly convenient when you have to type many variations of a given command: you don't have to type it completely at each time, you just modify what needs to be modified.

3.4 Comparing two files

It is sometimes useful to know whether two files are identical or not (for example, you are working on a document, and for some reason, there are two copies of that document – for example, one on your laptop, and one on your desktop – and at some point, you don't remember whether the two versions are the same or not). If the file is short, and easy to parse, then it's not a big deal, and the verification is straightforward. But if the file is very long, and the difference is not obvious, then you could spend a lot of time checking, letter by letter, if the fasta file of your millions of Solexa reads is the same at the two locations ...

`md5sum` is a command that computes the MD5 sum of a file. The MD5 sum is a value (which typically contains letters and digits, mixed together), which depends on the content of the file. It is computed in such a way that even a very minor difference in the two files will yield completely different MD5 sums (if a single character has been changed in your million-line fasta file, the MD5 sums of the two versions won't have anything in common). There are a lot of possible MD5 sums, so that the probability that two distinct files have the same MD5 sums is ridiculously low – in fact, that was a well-known challenge in computer science till very recently: the goal was to find two different files with the same MD5 sum; it has finally been cracked a few years ago (as a consequence, it is also impossible to find out what was the original file if you are only given its MD5 sum; but this MD5 sum

is a proof of somebody's knowledge of the content of the file: so, that command makes it possible to release publicly the proof that you have an information, without releasing the information itself – and, once the information is known by everybody, everyone can compute the MD5 sum on the file, and check that, indeed, that was the MD5 sum you had released ...).

For example, here is the MD5 sum of the source code for the present document, at this very stage of the redaction: 9ccde94c5ae2ce1233f1c928f5736738

Now (I added a that md5sum value, then a line break, these these few words), it is: 75ad5a9740da56f63f7

Once `md5sum` told you that two files are different, you can find where the difference lies with the program `diff` (syntax: `diff name_of_first_file name_of_second_file`). The output of `diff` displays the insertions and deletions of one file relatively to the other (a substitution is considered like a combined insertion and deletion).

3.5 What is plain text?

Plain text (also called "raw text") is just made of regular characters (letters, digits, punctuation signs, symbols like %, $ and the rest). There is no meta-information (no notion of "underline", colors, paragraph justification, *etc.*): it's just the text. So, for example, a ".doc" Word file is not plain text: it contains many additional informations on the page setup, the presentation of the text, things like that. These additional informations, of course, are also coded in the document.

That's why a ".doc" document cannot be opened by anything else than Word (or its emulators, like the one provided in OpenOffice): other programs might be able to display correctly the text, but they won't be able to interpret all the stuff which comes with it, and that Word uses to code additional features of the document (colors, underlines and so on) – so in the end, even the text parts of the document won't be recognized (because they are mixed with these un-interpreted informations), and displayed, by these other programs.

The fasta format, which is widely used to store and use sequence data, is a plain text format: a fasta file really contains only letters, digits and symbols. Word can open plain text files (including fasta files), but then, by default, when you save it with Word, it will be automatically converted into a ".doc" file (for example, Word will have decided to use a particular font, of a particular size, to display that text, and that font information will now be included in the file); you need to specifically ask to save the file as "raw text" (I'm not even sure it is possible with every version of Word) to keep it raw.

Of course, that kind of problem can cause some incompatibilities; that is why many programs on the Internet (accessible with a web interface) require the data to be uploaded as a raw text file; or they ask you to copy-paste your data in a window of the web browser (the program can then handle that text the way it wants, and most probably it will keep it as raw text). There is also a more philosophical issue about all that: raw text is readable by any program, and it will always be; on the other hand, proprietary formats are linked to a particular program (for a very long time, only Microsoft Word could read ".doc" files, even MacOS didn't have Word), and in the end, the readability of your files relies on the good will of the managers of a company whose interest is mostly to maximize their sales; in their point of view, it is naturally tempting to keep customers prisoners of their programs. This is not only leftist paranoia :-), there is also a clear implication on the future availability of your data: if you store your sequence files under these proprietary formats only (I'm not talking only about Word, but also MacVector, Sequencher, Vector NTI, ...), then you have to

pray that your favorite program (Word, MacVector, ...) will still be available in 10, 15 or 20 years. Don't expect too much from these: 20 years ago, molecular biologists were certainly confident that their HPLC programming languages or their cardboard Current Contents databases would still be in use in 2013 – just imagine what their archives are worth now ...

So it is obviously very useful, for display issues, to use these fancy formats, but do not rely only on them, and always keep fasta versions of your sequences. Moreover, as fasta is an open format (whose specifications are public, and which can be opened by any program), we will be able to do many sequence manipulations with home-made scripts based on Unix commands.

4 Redirection to a file, content of a text file

4.1 Redirecting the output of a command to a file

When you type a command, it is ofter useful to keep its output in a file (for a later use). This can be achieved with the ">" symbol:

```
ls > file1
```

will not display the result of the `ls` command (as you would get if you just typed `ls`), but it will create (or: overwrite) a text file called 'file1', whose content will be: the output of `ls`.

4.2 Displaying the content of a text file

The two most convenient ways of displaying the content of a file are the programs `less` and `cat`.

Typing:

```
less file1
```

will allow you to read the content of 'file1' (you can navigate in the file using the "arrow up" and "arrow down" keys of your keyboard; you can scroll down faster by hitting the spacebar; and you can reach the end of the file by typing a capital `F`). When you are done, exit `less` by typing: `q` (like "quit"). The name of that program, `less`, comes from the name of another program, called `more`, which does similar things, except it is less convenient to use (you cannot navigate with the arrows, for example), so, even if it's called `more`, it is *less* convenient than `less`.

Typing:

```
cat file1
```

will display the content of 'file1' in your terminal; you don't have to quit that program: your prompt will re-appear automatically after the content of the file has been displayed, ready to take the next command (whereas you have to quit `less` before you can type another command). As `cat` displays the content of the file, then exits, you cannot navigate in the file: if the file is too long, it won't appear completely in your terminal (you will only see the end); it is possible to recall what had been displayed in the terminal, by moving vertically the cursor on the side of the terminal, but that will recall only the last few hundreds of lines – if your file was very long, you won't see everything.

These commands can display the content of any file (as long as you have the permission to read that file); but that content will be readable only if it's a raw text file (try to do `less` or `cat` on another kind of file, and you will see ...). These two programs allow you to view the content of a file, but not to modify it.

Exercise

The name "cat" doesn't refer to the pet – it refers to "catenate" (a synonym for "concatenate"). Pick two text files (let's call them `file1` and `file2`), and apply `cat` to both, in the same command:
`cat file1 file2`
Why has this program been called `cat`?

5 Selecting pattern-matching lines with grep

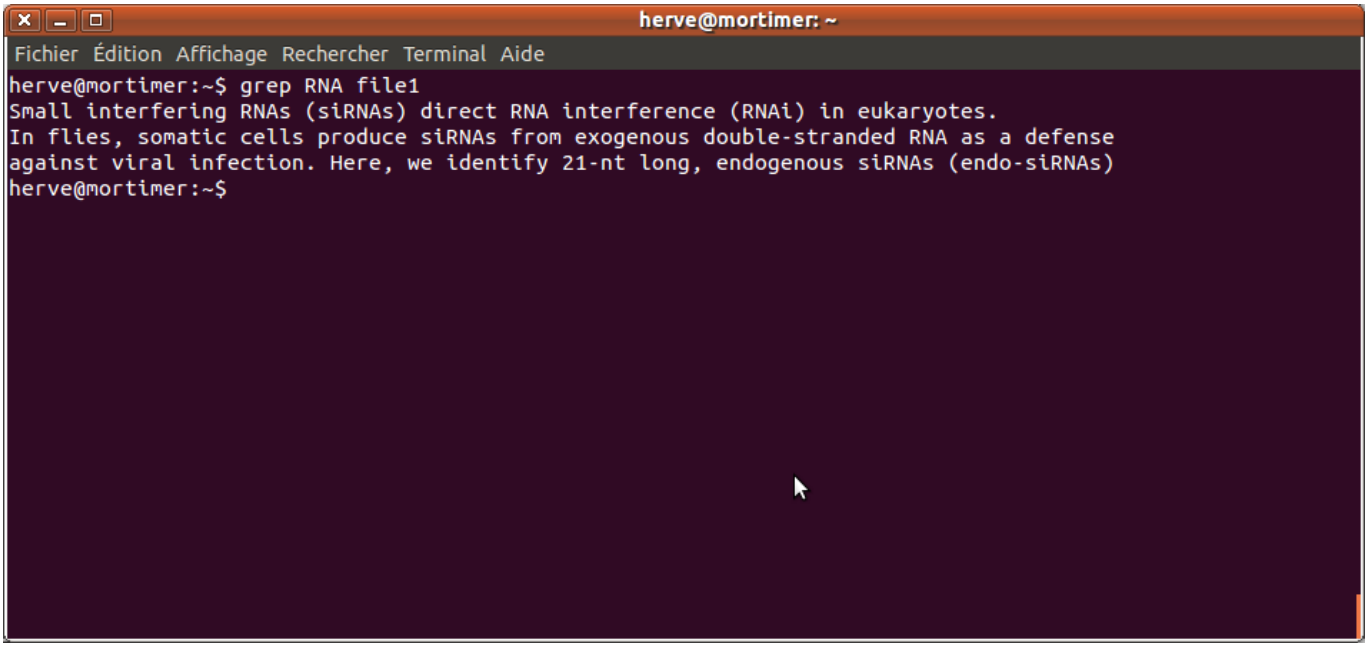
5.1 Basic use of grep

The command `grep` extracts, from a file, all the lines matching a pattern (which has to be given as an argument of grep). The syntax is:
`grep pattern file`

For example, let's consider a text file, called 'file1', which contains these four lines:

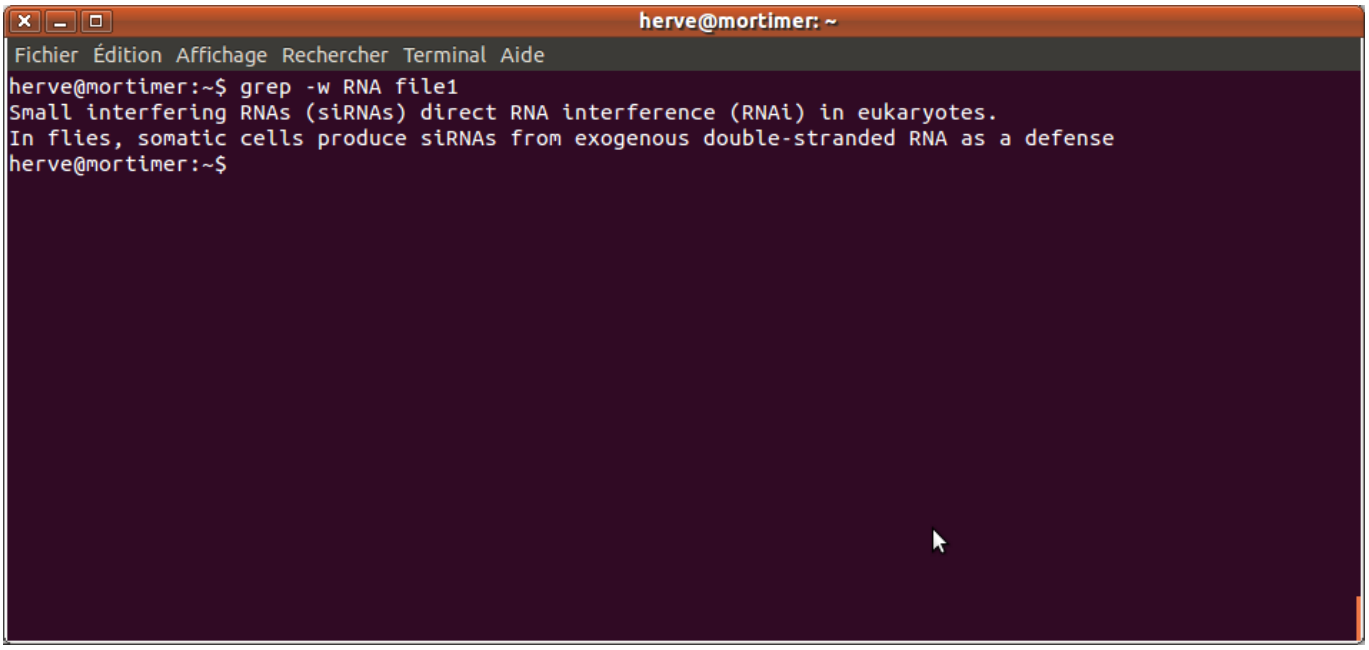
```
Small interfering RNAs (siRNAs) direct RNA interference (RNAi) in eukaryotes.
In flies, somatic cells produce siRNAs from exogenous double-stranded RNA as a defense
against viral infection.  Here, we identify 21-nt long, endogenous siRNAs (endo-siRNAs)
corresponding to transposons and heterochromatic sequences in the somatic cells of
```

Then typing `grep RNA file1` will extract all the lines containing the character string "RNA":



(the last line does not contain "RNA", so it is not selected). Note that `grep` will select all the lines matching the pattern, even if that pattern is embedded in a longer word (for example, "RNA" did not appear as a full word in the, selected, third line: it appeared as a sub-string of "siRNAs"); but the search is case-specific (`grep RNA file1` and `grep rna file1` won't give the same output).

As always with Unix commands, you can alter the behaviour of `grep` with its options: for example, `grep -w` ("w" for "word") will select only the lines where the pattern is a full word (*i.e.*, it is flanked by non-word constituent characters – like commas, dashes, ... – or the beginning or the end of a line). Here:



(the third line is no longer selected, as "RNA" is not a full word in this line). Similarly, option "-i" tells `grep` to be case-insensitive, option "-v" reverts the search (*i.e.* the selected lines are the ones which *do not* match the pattern), and so on. Type `man grep` to learn about the other options.

Exercise

You can use several options simultaneously (you can type them like that:
`grep -w -i -r pattern file`, or in a more synthetic form: `grep -wir pattern file`).
What should be the output of:
`grep -vi in file1`
? What should be the output of:
`grep -c endo file1`
?

5.2 Using grep with regular expressions

The pattern doesn't have to be a fixed string: it can contain wildcards, or even variables.

Square brackets delimit a list of characters, any of which can be used to match the pattern; for example; `grep [air]uble file1` will search for every line containing (either an "a", or an "i", or an "r"), followed by "uble" (in our 'file1', that command won't retrieve any line – but `grep [airo]uble file1` would retrieve the line which contains the word "double"). In a pair of square brackets, a dash separates the limits of an interval of characters (`[a-f]` means "any lower-case letter between 'a' and 'f', in the alphabetical order"; `[a-z]` means "any lower-case letter",

and `[a-zA-Z]` means "any letter"; `[0-9]` means "any digit", and so on). A dot (".") means: any character.

Here the asterisk has a special meaning: it means "any number of times (potentially zero)". So, the command `grep or*e file1` will select the line containing the word "corresponding", in our 'file1': "corresponding" indeed contains an "o", followed by a number of "r" (here: two), followed by an "e". Be careful, as the numbers covered by the asterisk can be zero, that command would also select the lines containing the string "oe" (with zero "r" between the "o" and the "e").

Another special character is the caret ("^"): it means "beginning of a line"; similarly, the dollar sign ("$") means "end of a line". So the command:

```
grep ^c file1
```

will select the line starting with a "c", and the command:

```
grep cells$ file1
```

will select the line ending with "cells".

Exercise

What should be the output of the command:

```
grep ins*t file1
```

? And what should I type if I want to select the lines containing an "a", followed by at least one "l" (so that my command would select the lines containing the words "Small" and "viral", but not the other lines – even if they contain the letter "a")?

You can also use variables in the definition of a pattern to be matched. Let's say you define a variable, called `x`, and you want to match the pattern: letter "a", repeated x times. Then you should type:

```
grep a'\{'$x'\}' file
```

(the dollar sign always designates variables – useful to distinguish the variable x from the simple letter "x" – and it has to be flanked by curly braces, each of which is preceded by a backslash, and quoted; you may use double quotes " instead of the simple quotes ').

5.3 Special characters

As with every other Unix command, `grep` arguments (the options, the pattern to be matched, and the input file name) are separated by spaces. So what if the pattern you search contains a space? A similar problem can arise with other special characters (for example, you want to extract the title lines from a fasta file, and type: `grep > file.fa`; that command won't be intrepreted the way you would like: it will be interpreted as "redirect the output of command `grep` to the file "`file.fa`", so it will overwrite `file.fa` – and as `grep` wasn't given any argument, its output will be empty, so basically, you will be replacing your precious fasta file by an empty file ...).

There are two ways to solve that problem: the first one is to protect the special characters by backslashes (a backslash in front of a special character tells your computer that you are talking about that character, and you are not trying to use its fancy features):

```
grep a\ thing file.fa
grep \> file.fa
```

will select the lines containing the string "a thing" (with a space between "a" and "thing"), and the lines containing a ">", respectively.

The second way is to protect the whole pattern by quotes (either simple quotes or doube quotes):

```
grep 'a thing' file.fa
grep '>' file.fa
```

will also extract the lines containing "a thing" and ">", respectively, without any interference with the usual meaning of the space, and of the ">" character. Be careful: this second solution doesn't work for every special character: characters with a special meaning in regular expressions (things like `[a-z]`, or like `.*` – which means: any character, any number of times; see sub-section 5.2, page 12) won't be protected by the quotes, and will still be parsed as regular expressions. In those cases, if you really want to mean "dot", or "opening square bracket", you have to protect them both with a backslash and with quotes:

```
grep '\.'
grep '\['
```

for example (the closing square bracket is less demanding than the opening square bracket: you can type `grep ] file` and it will be parsed as: "find the lines containing a closing square bracket in `file`").

As a general rule, it is often good (and never bad) to protect the pattern by quotes: I suggest you always do it, unless you explicitly want to use the special meaning of some characters. If the pattern you want to match contains characters that have special meanings in regular expressions (dot, opening square bracket, dollar sign, ...), you'll also have to protect them with backslashes.

6 Piping several commands

If you want to apply sequentially several commands to a file (for example, you want to extract the lines matching pattern "pattern1", but not pattern "pattern2"), you could apply the first command and redirect its output in a file (`grep pattern1 file > output_file`), then apply the second command to that output file (`grep -v pattern2 output_file`).

There is a more direct way to do that: you can send automatically the output of the first command to the second one, with a "pipe" (symbolized by the vertical bar: "`|`"):

```
grep pattern1 file | grep -v pattern2
```

That composite command will only display the final output (the output of the second command, applied to the output of the first one); of course, as the second command is fed by the first one, there is no file name in the invocation of the second command (it is "`grep -v pattern2`", not "`grep -v pattern2 name_of_file`").

You can pipe as many commands as you want, and of course, they don't have to be the same command (`grep`, in the example above). For example, you can pipe the output of `ls` with a `grep` search:

```
ls | grep -v '\.jpg$'
```

will list all the files in the current directory, but, before printing the result, it will counter-select all the lines ending with ".jpg" (note that you have to protect the dot with a backslash, otherwise it will mean "any character", so the lines ending with "jpg" preceded by a character which is not a dot would also be excluded; also note that, in order to select only the lines which *end* with that pattern,

and not all the ones which *contain* that pattern, you need to add a dollar sign at the end of the pattern).

Exercise

How can I count the sequences in a fasta file? How can I count the sequences starting with a "U" in a fasta file (assuming that every sequence fits on a single line)?

How can I extract the titles of the sequences containing the pattern "UACGCGU" (assuming that every sequence fits on a single line)? I want to extract all the patterns: "NUACGCGU" from that fasta file (where N can be any nucleotide), and get rid of the rest of the sequences: how can I do?