Chapter 1
The UNIX Command Line

Almost all commercial software published today comes with lush graphical user interfaces that allow users to work and play by touching and mousing. This is great for things like deleting a file by dragging it into a trash can, renaming a file by clicking on its name, editing text by mouse selection, and so on. However, in modern biology data may consist of dozens of files containing millions of sequencing reads, which makes it routinely necessary to do things like check the three billion nucleotides of the human genome for the occurrence of a particular motif, or compute averages from thousands of expression values distributed across dozens of files. Such operations are hard to perform using click-driven programs. This is because graphical user interfaces are excellent for carrying out the tasks their creators deem important, such as deleting a file by dragging and dropping it into a virtual trash bin, moving a file by dragging and dropping it between virtual folders, or opening a file by double-clicking on it. However, graphical user interfaces lack the universality that makes learning about computers so fascinating. Computers are universal machines in the sense that they can perform any precisely specified operation. All that is necessary is an interface that lets the user communicate every possible operation, not just a finite set, however large it may be.

To illustrate the importance of being able to communicate an infinite number of possible operations, think of the communication system we all know best, our language. Take any sentence that comes to mind and search the World Wide Web with it. Unless you were quoting from memory, chances are, your sentence is unique. This is because we do not parrot sentences we have heard, but use rules to construct new ones. The rules leave us free to think about what we want to say while saying it. Moreover, the words we use have a curiously vague relationship to what they mean. If someone says: "John is my friend.", the word "friend" neither looks nor sounds like a friend. Nevertheless, we know immediately what that word signifies. Taking our cue from language, we expect all powerful communication systems to be characterized by a set of rules and an arbitrary mapping between words and their meaning. Communicating effectively with a computer is no different.

B. Haubold and A. Börsch-Haubold, *Bioinformatics for Evolutionary Biologists*, https://doi.org/10.1007/978-3-319-67395-0_1

The UNIX command line, also known as the *shell*, is the de facto standard method for text-based, rather than graphics-based, computer communication. It has been around since the late 1960s and has proved flexible enough to adapt rather than go extinct like so many other programs over the years. In fact, it is available on all three major operating systems, and its behavior is governed by a standard, the POSIX standard. This means that once you have mastered the UNIX shell on one type of computer, you have mastered it on all. If you have never used it, now would be a good time to start by working through the chapters in this part. Even if you have used it before, we recommend you work through this material to make the most of the subsequent sections. For future reference, the Appendix contains a summary of commands and techniques for working on the command line.

1.1 Getting Started

This section is for everyone who has never used the UNIX command line, or shell, before. There are various versions of the shell to choose from, but on personal computers `bash` is the default. We explain how to create and destroy directories and files under the shell, list the contents of directories, access the history of past commands, and help with typing. Fluency in typing is particularly important in a text-based system like the shell, and we encourage readers to spend time on practicing the basic key combinations. The chapter closes with a description of the manual system. We assume you are sitting in front of a computer with an open terminal displaying a blinking cursor like this:

```
jdoe@unixbox:$ ▮
```

New Concepts

Name	Comment
*	wildcard to match any substring
autocompletion	makes typing easier
UNIX	operating system
command line	text-based interface to UNIX

New Programs

Name	Source	Help
`cd`	system	`man cd`
`ls`	system	`man ls`
`man`	system	`man man`
`mkdir`	system	`man mkdir`
`rmdir`	system	`man rmdir`
`rm`	system	`man rm`
`touch`	system	`man touch`

Problem 1 Create a directory for this course by typing

```
mkdir BiProblems
```

followed by the Enter key. List the contents of your current directory to make sure `BiProblems` has been created.

```
ls
```

Notice that we write the names of directories in upper case to distinguish them from file names, which we start in lower case. This is merely a convention, others prefer to use lower case throughout. However, UNIX is case sensitive, so `BiProblems`, `biProblems`, and `biproblems` would be three distinct names. Notice also that we visualize word boundaries by case changes. Again, this is only a convention, known as "camel case". Change into `BiProblems`

```
cd BiProblems
```

and list its contents

```
ls
```

It is empty. Create two more directories, `TestDir1` and `TestDir2`, and use `ls` to check they exist.

Problem 2 To minimize typos, the command line supports autocompletion. Change again into `TestDir1`, but this time type only

```
cd T
```

followed by `Tab`. This completes the unambiguous part of the name, `TestDir`. To get the two possible completions, press `Tab` again. Type `1`, once more followed by `Tab`, to ensure correct typing. This technique of mixing typing and tabbing is very effective when using the shell. But it does take some getting used to. Practice it by changing out of the current directory

```
cd ..
```

and into it again. What happens if you enter

```
cd
```

without the trailing dots?

Problem 3 Use `rmdir` to remove the test directories. Then practice creating and removing these directories a few times. What happens if you enter

```
rmdir TestDir*
```

Problem 4 Recreate the directory `TestDir1` and change into it. Then create a file

```
touch testFile
```

and check it exists

```
ls
```

Remove the file

```
rm testFile
```

Recreate the file, then go to the parent directory. What happens if you now apply `rmdir` to `TestDir1`?

Problem 5 Recreate `TestDir1` and enter it. Create two test files, `testFile1` and `testFile2`. How would you remove both with one command?

Problem 6 File `a` is renamed `b` by

```
mv a b
```

Create file `a`,

```
touch a
```

then try renaming it. Can you guess what `mv` might stand for?

Problem 7 Commands are often repeated. To avoid repeated typing, the command line remembers a list of previous commands. You can walk this list up and down by using the arrow keys ↑ and ↓. Try this. What happens when you enter the command

```
history
```

Problem 8 By now you have probably noticed that the cursor cannot be positioned by clicking the mouse. This leaves the arrow keys as the navigation tools of first choice. However, the cursor is also responsive to more powerful key strokes; for example, when you press the `Ctrl` followed by `a`, while still keeping `Ctrl` pressed, the cursor jumps to the beginning of the line. We write this as

```
C-a
```

Similarly,

```
C-e
```

moves the cursor to the end of the line. Type

```
You cannot tune a mouse but you can tuna fish
```

and practice jumping to the beginning and the end of the line a few times. What happens if you enter this nonsense as a command?

Table 1.1 List of key combinations for navigating and editing the `bash` command line

Keystrokes	Explanation
`C-e`	Position cursor at end of line
`C-a`	Position cursor at beginning of line
`C-w`	Delete word to the left of cursor
`C-y`	Insert deleted text
`C-b`	Move cursor back to one position
`C-f`	Move cursor forward by one position
`C-d`	Delete character left of cursor
`M-b`	Jump back by one word
`M-f`	Jump forward by one word
`M-d`	Delete word to the right of cursor

Problem 9 Table 1.1 lists the most useful key combinations for navigating the `bash`. Apart from the combinations based on the `Ctrl` key, there are also combinations based on the so-called `Meta` key, `M`. By default this is mapped to `Esc`. It may also be mapped to `Alt`, which makes it to easier to reach.

Moving the cursor using key combinations is a bit awkward at first, but once you have mastered these shortcuts, using the command line becomes much easier. Experiment with each of the key combinations in Table 1.1. What happens if you keep pressing a combination, say `C-f`?

Problem 10 If you need help with a command, or would like to learn more about its options, access the corresponding section of the manual by typing, for example

```
man ls
```

Navigate the page with the arrow keys, and press `q` to quit. It is often useful to know which file in a directory was modified most recently. Read `man ls` to find out how files can be listed by modification time.

Problem 11 Find out more about how to navigate the man pages by again typing

```
man ls
```

and then activate the help function by pressing `h`. How would you look for the pattern `time` in a man page?

Problem 12 A very useful feature of the shell is that the output of a program can be used as input for another. For example in

```
ls | wc
```

the program `wc` reads as its input the output from `ls`. This construction is called a "pipe" or "pipeline". Can you interpret the result of your first pipeline? How would you count the number of files in your directory?

Problem 13 How would you find out more about pipelines under the `bash`?

Problem 14 The `bash` is a programming environment and can be used as a simple calculator. To add two numbers, type, for example,

```
((x=1+1)); echo $x
```

where `echo` prints the value of `x` to the screen. What happens if you leave out the double brackets?

Problem 15 If you like a more verbose output, enter

```
((x=1+1)); echo "The result is $x"
```

What happens if the double quotes are replaced by a single quotes?

Problem 16 Our simple calculation can also be expressed as

```
let x=1+1; echo $x
```

What happens if you leave out the `let`?

Problem 17 The `bash` can also multiply

```
let y=2*5; echo $y
```

and compute power of

```
let y=2**5; echo $y
```

What is 2^{10}?

Problem 18 Subtraction also works as expected

```
let y=10-2; echo $y
```

What is $10 - 20$ according to the `bash`?

Problem 19 Division is denoted by

```
let y=10/2; echo $y
```

What is 10/3?

Problem 20 Floating point calculations on the command line can only be carried out using additional tools. One of these is the basic calculator, `bc`. Enter

```
bc -l
```

to start it, and to exit `quit`. In `bc` n^x is expressed as `n^x`. What is the number of distinct oligonucleotides of length 10? Can you guesstimate the result?

Problem 21 As usual for UNIX programs, the basic calculator can also be used as part of a pipeline:

```
echo 10/3 | bc -l
```

What happens if you drop the `-l` option?

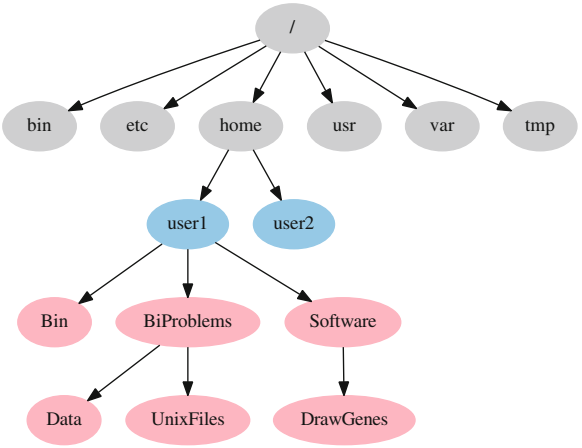


Fig. 1.1 The UNIX file system, slightly abridged. System directories are gray, home directories blue, and directories generated by users pink

1.2 Files

Files are kept inside directories, which may contain further directories. This hierarchy of directories forms a tree, and Fig. 1.1 shows an example containing the essential features of a typical UNIX file system. Its top node is the root, denoted `/`. The gray part of the tree is "given" and can only be changed by the system administrator, for example, when installing new software. The blue directories are called "home directories". Every user has one and can change it by adding new directories, which are depicted in pink. This separation between files accessible only to the administrator and files accessible to the user means that users need not worry about accidentally damaging the system—they cannot change the sensitive files. Our bioinformatics course forms a sub tree of the directory tree rooted on `BiProblems`, which you have already created. In the following section we learn to navigate the file system, and to manipulate individual files.

New Concepts

Name	Comment
PATH	directories searched for file name
directories	contain files and directories
file permissions	read, write, execute
file system	all directories
files	contain text (usually)

New Programs

Name	Source	Help
`apt`	system	`man apt`
`brew`	system	`man brew`
`cat`	system	`man cat`
`chmod`	system	`man chmod`
`cut`	system	`man cut`
`drawGenes`	book website	`drawGenes -h`
`emacs`	package manager	`man emacs`
`find`	system	`man find`
`gnuplot`	system	`man gnuplot`
`grep`	system	`man grep`
`head`	system	`man head`
`make`	system	`man make`
`tail`	system	`man tail`
`tar`	system	`man tar`
`which`	system	`man which`

Problem 22 Use `ls` to list the contents of the root directory. How many files and directories does it contain?

Problem 23 Change into the course directory `BiProblems` and list all files it contains. Use `man ls` to find out how to really list *all* files. Can you explain what you see?

Problem 24 Download the example data from the book website

```
http://guanine.evolbio.mpg.de/problemsBook/
```

copy it into your current directory, and unpack it

```
cp ~/Downloads/data.tgz . tar -xvzf data.tgz
```

This generates the directory `Data`. How many files does it contain?

Problem 25 It is often convenient to list all files that contain a certain substring in their name. For example, all files with the extension `fasta`:

```
ls *.fasta
```

How many FASTA files are contained in the `Data` directory?

Problem 26 Make a directory for this session and change into it:

```
mkdir UnixFiles
cd UnixFiles
```

The file `mgGenes.txt` contains a list of all genes in the bacterium *Mycoplasma genitalium*. Copy `mgGenes.txt` from `Data` into the current directory. How many genes does *M. genitalium* have?

Problem 27 The command

```
cat mgGenes.txt
```

prints the contents of `mgGenes.txt` to the screen. What does `cat -n` do? Use it to re-count the entries in `mgGenes.txt`.

Problem 28 We often need to look at the beginning and the end of a file. This is done using the commands `head` and `tail`. Apply these to `mgGenes.txt`; can you make head or tail of what you see?

Problem 29 From our quick glance at the head and tail of `mgGenes.txt`, it looks as though genes at the beginning of the list are on the forward strand, genes at the end on the reverse. Since the list is ordered according to starting position rather than strand, this is intriguing. Do genes on the forward and reverse strand form separate blocks along the genome? To find out, use the program `grep`, which extracts lines from files matching a pattern, for example,

```
grep MG_12 mgGenes.txt
```

Use this and `wc` to count the genes on the forward strand and on the reverse strand. Do the counts add up to the total number of genes? If not, can you think of why?

Problem 30 To investigate whether one of the gene symbols contains the extra "`-`", we cut out the symbol column:

```
cut -f 5 mgGenes.txt
```

Which genes contain in their names "`-`", and which strand are they located on?

Problem 31 We are still trying to find out whether genes on the plus and minus strands form separate blocks along the *M. genitalium* genome. To exclude unexpected characters contained in the gene names, we cut out the first four columns of `mgGenes.txt` and extract the genes on the two strands. For subsequent analysis, we save the results by redirecting them to files using

```
grep pattern mgGenes.txt > pattern.txt
```

Save the genes on the forward strand in the file `plus.txt` and the genes on the reverse strand in the file `minus.txt`. Check again that the gene counts add up. Do the genes on the plus strand form a block along the genome (hint: use `head -n`)?

Problem 32 Redirection also works in reverse. Can you find out how to apply <?

Problem 33 Next, we need to use an editor. Our editor of choice is called `emacs`. If `emacs` is not installed on your system, please install it now. On many versions of Linux, including Ubuntu, this can be done using the cycle

```
sudo apt-get update         # update package database
apt-cache search emacs      # find suitable package
sudo apt-get install emacs # install package
```

On OSX you might use

```
brew install emacs
```

if the homebrew package manager is installed. What does the `sudo` in the `apt-` commands above stand for?

Problem 34 To avoid the problem with the gene name containing a "-", we can open `mgGenes.txt` in `emacs` and remove the offending hyphen. Open `mgGenes.txt`:

```
emacs mgGenes.txt &
```

This opens a new window running `emacs`. What happens if you leave out the ampersand (`&`)?

Problem 35 Save `mgGenes.txt` to `mgGenes2.txt` and replace the "-" in `rpmG-2` and `polC-2` by underscore, "_". Use `diff` to check the differences between `mgGenes.txt` and `mgGenes2.txt`.

Problem 36 Use `head` and `tail` to directly extract lines 56 and 288 from files `mgGenes.txt` and `mgGenes2.txt`.

Problem 37 Many of the commands for navigating the `bash` listed in Table 1.1 have the same function in `emacs`. What are the exception(s)?

Problem 38 `emacs` is a powerful editor with a rather weak GUI. We recommend you take some time to learn the most important keyboard shortcuts, which are summarized in Table 1 of the Appendix. In addition, we recommend you work through the `emacs` tutorial, which is invoked by `C-h t`. What is the command for exiting `emacs`?

Problem 39 Apart from programs like `emacs`, which are supplied through public software repositories, there are a number of programs written specifically for this course. These are supplied as source files accessible through the book website. As a first example, download the program `drawGenes`. It is a good idea to keep source packages in the same place, so make a directory `Software` in your home directory and copy the source package of `drawGenes` into it. Then unpack it (c.f. Problem 24), change into the new directory and compile the code by typing `make`. This generates the program `drawGenes`. Again, programs are best kept in one place, so make the directory `~/Bin` and copy `drawGenes` into it. Return to your current directory. What happens when you try executing `drawGenes`?

Problem 40 To make the system aware of the new directory for executables, `~/Bin`, we need to change the set of directories in which the system looks for executables when it receives a command like `ls`. This set of directories is defined in the `bash` variable `PATH`. To alter `PATH`, open `~/.bashrc` in `emacs` and add the line

```
export PATH=~/Bin:$PATH
```

at the end. Then return to your current working directory and load the new PATH information

```
source ~/.bashrc
```

The first thing to do now is to test the old `PATH` is still working by trying to execute `ls`. If this fails, `PATH` needs to be reset. On Linux this is done by entering

```
source /etc/environment
```

on OSX by entering

```
source /etc/profile
```

Then try again to change `PATH` in `.bashrc`. Once this has worked, test that `drawGenes` is executable from within your working directory

```
drawGenes -h
```

This might seem like a long-winded method for installing programs. The good news is that `.bashrc` is always loaded when a new terminal is opened, so `source` only needs to be executed if `.bashrc` is changed during a terminal session. The next program installed manually just needs to be copied into `~/Bin` to become available to you everywhere. The command `which` locates an executable file; try for example

```
which drawGenes
```

Where is `ls` located on your system?

Problem 41 Apart from programs, we can also search for files using, for example

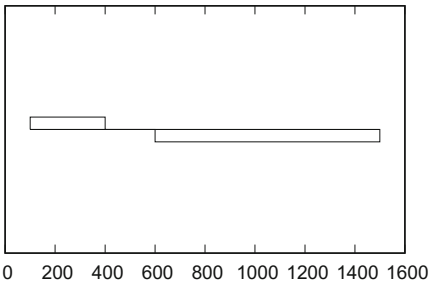
```
find ~/ -name "*.txt"
```

which looks for all files with the extension `txt`, starting in the home directory. How would you look up the location of `.bashrc`?

Problem 42 The program `drawGenes` converts gene coordinates like

```
100 400 +
600 1500 -
```

to figures like



Create a new file called `exampleGenes.txt` in `emacs` and copy the gene coordinates. Then reproduce the above figure using `drawGenes` together with `gnuplot`. Hint: Check the usage of `drawGenes` by typing

```
drawGenes -h
```

Problem 43 The commands of `gnuplot` can be abbreviated to the first few unique characters. What is the shortest version of your `gnuplot` command for plotting the *M. genitalium* genes?

Problem 44 `Gnuplot` is a powerful tool with many options, which are summarized in a reference card posted on our book website. For example, the command

```
set xlabel "Label"
```

labels the x-axis. Use it to label the x-axis of our example plot with "Position (bp)".

Problem 45 Draw the genes of *M. genitalium*. Is the bias in their distribution between the strands visible?

Problem 46 When dealing with longer commands like the one for drawing the genes in *M. genitalium* (Problem 45), it is often more convenient to edit them in a separate file, which can then be executed by the `bash`. Such files are called "scripts". Copy the solution to Problem 45 into the file `drawGenes.sh` and run it

```
bash drawGenes.sh
```

What happens if you try to execute `drawGenes.sh` directly by typing

```
./drawGenes.sh
```

Problem 47 There are three kinds of file permissions: read, write, and execute. To inspect them, execute the *long* version of `ls`:

```
ls -l
total52
-rw-rw-r-- 1 haubold haubold 83    Mar 3 15:15 drawGenes.sh
-rw-rw-r-- 1 haubold haubold 13284 Mar 3 15:15 mgGenes.txt
-rw-rw-r-- 1 haubold haubold 13284 Mar 3 15:15 mgGenes2.txt
-rw-rw-r-- 1 haubold haubold 5157  Mar 3 15:15 minus.txt
-rw-rw-r-- 1 haubold haubold 6762  Mar 3 15:15 plus.txt
```

This shows the total size of the files in kilobytes, followed by information about individual files in eight columns:

1. File type and permissions: The first character is the file type; the two most important file types are ordinary file (`-`), and directory (`d`). The next nine characters are divided into three blocks of the three permissions already mentioned: read (`r`), write (`w`), and execute (`x`). Permissions not granted are shown as hyphens. The first three permissions concern the user, that is you, the second the group, and the third the world, which is everybody.
2. Number of links; for files this is usually one, but directories may contain a greater number of links.
3. User name.
4. Group name.
5. File size in characters.
6. Date on which the file was last altered.
7. Time when the file was last altered.
8. File name.

We can make `drawGenes.sh` executable:

```
chmod +x drawGenes.sh
```

Check the result

```
ls -l drawGenes.sh
-rwxrwxr-x 1 haubold haubold 83 Mar 3 15:15 drawGenes.sh
```

Now you can run

```
./drawGenes.sh
```

What happens if you drop `./` from this command?

Problem 48 To include scripts located in the current directory in the PATH, open `~/.bashrc` and change the line

```
export PATH=~/Bin:$PATH
```

to

```
export PATH=.:~/Bin:$PATH
```

How is the `bash` made aware of this change? Can you now directly execute `drawGenes.sh`?

1.3 Scripts

In Problem 46 we wrote our first script, `drawGenes.sh`, to help draw the 525 genes of *Mycoplasma genitalium*. Scripts are used extensively in bioinformatics. Throughout this book, we use three kinds of scripts: `bash`, `sed`, and AWK. `Bash` scripts are

used to drive other programs. `Sed` scripts automate text editing, for example removing stray hyphens from gene names. Finally, AWK is a programming language for manipulating text files like `mgGenes.txt`. It is carefully described in a book by the authors of the language, Alfred Aho, Peter Weinberger, and Brian Kernighan, hence the name AWK [4].

New Concepts

Name	Comment
array	table in computer programs
hash	array indexed by strings
shell script	file containing commands
stream editor	in contrast to an interactive editor

New Programs

Name	Source	Help
`awk`	system	`man awk`
`sed`	system	`man sed`
`seq`	system	`man seq`
`uniq`	system	`man uniq`

1.3.1 `Bash`

Problem 49 Start this session by changing into the directory `BiProblems`. Then make a new directory, `UnixScripts`, and change into it. As we already saw in Problem 46, commands that work directly on the command line can usually be included in a `bash` script and then executed. The command we start off with is

```
echo Hello World!
```

Enter this on the command line. If we wanted to separate the two words by three blanks, we might try

```
echo Hello  World!
```

but this has the same effect as the original command. Try using single quotes to get the desired effect.

Problem 50 Scripts overcome the limitations of the command line as an editing environment. Write a script `hello.sh` containing

```
echo 'Hello World!'
```

A command can be repeated using a loop like

```
for((i=1; i<=10; i=i+1))   # i=1,2,...,10
do
    echo 'Hello World!'
done
```

where everything behind a hashtag is ignored and can be used for commenting. We can also write this script on a single line:

```
for((i=1; i<=10; i=i+1)); do echo 'Hello World!'; done
```

Run this code. What happens if you replace `echo` by `echo -n`?

Problem 51 An alternative way of looping in `bash` is

```
for i in $(seq 10)
do
    echo 'Hello World!'
done
```

Modify this loop such that it prints the numbers from 1 to 10 (hint: take a look at Problem 14).

Problem 52 Write the numbers from 1 to 10 on the same line.

Problem 53 We said that commands on the command line and in scripts are interchangeable. Execute

```
echo 5
```

on the command line. Find out by looking at the `man` page how to count in steps of two, or backwards.

Problem 54 We have already seen that the genes in *M. genitalium* are not distributed equally between the forward and the reverse strands along the genome. A simple way of visualizing this is to show the number of genes on one of the strands as a function of the number of genes surveyed. For this, copy first `mgGenes.txt` from `Data` to your current directory

```
cp ../Data/mgGenes.txt .
```

Then the command

```
cut -f 4 mgGenes.txt | head -n 100 | grep + | wc -l
```

counts the number of genes on the plus strand among the first 100 genes. Write a script that counts the number of genes on the plus strand among the first 1, 2, ..., 525 genes and save the script as `countGenes.sh`.

Problem 55 Edit your script such that it prints the number of genes on the plus strand as a function of the number of genes investigated. Then plot that function using `gnuplot`.

Problem 56 Loops in shell scripts can be nested. Edit `countGenes.sh` such that it prints the counts for the plus and the minus strands. Separate the two data sets by a blank line. Then plot the two functions in the same graph.

1.3.2 Sed

Problem 57 Instead of using an interactive editor like `emacs` to replace `-2` by `_2` in Problem 35, we could have used the stream editor `sed`:

```
sed 's/-2/_2/' mgGenes.txt
```

A construction like `s/a/b/` is a small program: Substitute (`s`) some expression `a` by some expression `b`. Carry out the replacement of `-2` by `_2`, and save the result in `mgGenes3.txt`. Use `diff` to check the new file is identical to your manual edit in `mgGenes2.txt`.

Problem 58 Next, we investigate how many genes have proper names. We start by cutting out the names in the fifth column, but still need to delete the blank lines:

```
cut -f 5 mgGenes.txt | sed '/^$/d'
```

where the `sed` command means, delete (`d`) a line whenever the start of a line (`^`) is followed directly by its end (`$`). How many of the 525 genes have a name rather than just an accession number?

Problem 59 Apart from substituting (`s`) and deleting (`d`), `sed` can print (`p`) particular lines, for example,

```
sed -n '56p' mgGenes3.txt
```

The option `-n` causes `sed` to *not* print non-matching lines. What happens if you leave out the `-n`? Find out by comparing the `sed` result to `mgGenes3.txt` using `diff`.

Problem 60 `Sed` can also output a range of lines:

```
sed -n 'x,yp'
```

where `x` is the starting line, `y` the end. Write a `sed` script that replaces `head` and check your result with `diff`.

Problem 61 In Problem 30 we used `grep` to find the gene names containing a hyphen ("-"). Use `sed` to carry out the same search.

Problem 62 What is the range of gene positions in *M. genitalium*? The entries in `mgGenes3.txt` happen to be sorted, and we could rely on that; but let us make sure and sort all start and end positions ourselves. First, we write all positions in a single column by replacing TAB by newline:

```
cut -f 2,3 mgGenes3.txt | sed 's/\t/\n/'
```

In case your version of `sed` does not allow this substitution, try the equivalent `tr` command

```
cut -f 2,3 mgGenes3.txt | tr '\t' '\n'
```

Next, sort the positions using `sort`. The default mode of `sort` is alphabetical. Find out how to sort the positions numerically to discover the smallest and the largest position.

Problem 63 What would happen if by accident you sorted the gene positions alphabetically?

Problem 64 Check that the genes in `mgGenes3.txt` are sorted by start position.

Problem 65 Next we ask, whether any of the genes in *M. genitalium* overlap. Here is a hypothetical pair of overlapping genes:

```
G1 1000 2000 +
G2 1990 3000 +
```

Does the genome of *M. genitalium* contain such configurations?

Problem 66 Several `sed` commands can be applied to the same input. For example, we might want to remove empty lines from the gene symbols *and* remove all underscores:

```
cut -f 5 mgGenes3.txt | sed '/^$/d;s/_//';
```

Instead of writing the `sed` commands on the command line, they can be written in a file, say `filter.sed`, and executed as

```
sed -f filter.sed
```

where `filter.sed` is

```
/^$/d  # delete empty lines
s/_//  # remove underscores
```

Gyrases are an important family of genes involved in the maintenance of DNA topology. How many gyrases does the genome of *M. genitalium* contain? User `filter.sed` in your answer.

1.3.3 AWK

Problem 67 A typical AWK program looks like this:

```
awk '{print $2}' mgGenes3.txt
```

Try out the code above; which column does it print? Print the last column.

Problem 68 It is not necessary to provide an input file. For example, enter

```
awk '{print "You entered: " $0}'
```

The program now waits for input and prints whatever is entered, which is referred to as `$0`. In AWK—like on the shell—two strings are concatenated simply by writing them next to each other. To exit, press `C-c C-c`. Write an AWK program that prints the sum of two numbers entered interactively by the user.

Problem 69 The general structure of an AWK program is

```
pattern {action}
pattern {action}
...
```

Without a pattern, all input lines are matched. A pattern might be

```
$4~/[+]/
```

to match lines where the fourth column contains a plus. Write an AWK program that prints only the genes on the plus strand; then write a variant that prints only the genes on the minus strand.

Problem 70 What happens if you leave out the action block in your previous command?

Problem 71 Recall that `drawGenes` converts input like

```
100 400 +
```

to a box above the zero line

```
100 0
100 1
400 1
400 0
```

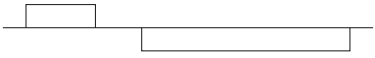
and input like

```
600 1500 -
```

to a box below the zero line

```
600 0
600 -1
1500 -1
1500 0
```

which we can then plot as



Check this by comparing the output from

```
cut -f 2-4 mgGenes3.txt | head
```

to the output from

```
cut -f 2-4 mgGenes3.txt | drawGenes | head
```

Write an AWK program to carry out this transformation. Save it in `drawGenes.awk` and run it

```
awk -f drawGenes.awk mgGenes3.txt |
gnuplot -p pipe.gp
```

where `pipe.gp` contains

```
unset ytics
set xlabel "Position (bp)"
plot[][-10:10] "< cat " title "" with lines
```

Problem 72 Use AWK and `sort` to find the lengths and accession numbers of the longest and shortest genes in *M. genitalium*.

Problem 73 This program counts the lines in `mgGenes3.txt`:

```
awk '{c = c + 1}END{print "Lines: " c}' mgGenes3.txt
```

Notice the `END` pattern, which precedes a block executed once after all the lines in a file have been dealt with. A shorter way of expressing the line count is

```
awk '{c += 1}END{print "Lines: " c}' mgGenes3.txt
```

And since we are just adding 1 at every step, we can write even more succinctly

```
awk '{c++}END{print "Lines: " c}' mgGenes3.txt
```

Compute the average length of genes in *M. genitalium*.

Problem 74 We can save the lengths of all genes in the array `len` and then print them

```
{
      l = $3-$2+1
      len[n++] = l
}END {
      for(i=0; i<n; i++)
          print len[i]
}
```

An array can be thought of as a table with indexed entries; in the case of `len` the table looks like this:

index	value
0	1143
1	933
2	1953
...	
524	810

The variance of values x_i is defined as

$$s^2 = \frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1},$$

where $\overline{x}$ is their average. Modify the array-code above to determine the variance of gene lengths. Check your result using the program `var`, which is available from the book site. If there is a discrepancy between your result and `var`, try using `printf` instead of `print`:

```
printf "Var: %e\n", v
```

where `%e` is the "engineering" format used by `var`.

Problem 75 We have already seen that genes are not distributed uniformly between the forward and reverse strand along the *M. genitalium* genome and that the variance of their lengths is huge. Our next question is, are gene lengths also distributed nonuniformly along the *M. genitalium* genome? To investigate, again save the lengths in an array and then plot the cumulative length as a function of gene rank. Normalize the cumulative length such that it lies between 0 and 1 and plot it together with the value expected if gene lengths are distributed uniformly along the genome.

Problem 76 The program `uniq` finds unique entries in an alphabetically sorted list. Use `sort` and `uniq` to determine the number of unique gene names in `mgGenes3.txt`. Is any name repeated? Hint: Recall from Problem 58 how to remove empty lines.

Problem 77 Use `sort` and `uniq` to find the number of distinct gene lengths in *M. genitalium*.

Problem 78 The option `-c` switches `uniq` into counting mode. To find the most frequent gene lengths, numerically sort the output of `uniq -c`. What are the five most frequent gene lengths? Hint: Reverse-sort the output using the `-r` option of `sort`.

Problem 79 Here is an AWK version of `uniq -c`:

```
{
      count[$1]++
}END {
      for(a in count)
          print count[a], a
}
```

In contrast to `uniq`, this program works on unsorted and sorted input. Consider the construct `count[$1]++`. Since `$1` can be any string, not just a number, it is called a *key* rather than an index. And since consecutive index numbers are characteristic of arrays, `count` is called a *hash* instead of an array. The `for in` construct goes through all the keys of a hash. Notice also the comma in the `print` command for delineating two strings. Copy the AWK version of `uniq -c` into `uniqC.awk` and make sure it generates the same output as `uniq -c` (Problem 78). This is best done by removing the leading blanks in the output from `uniq -c` with

```
sed 's/ *//'
```

which means, substitute (`s`) one or more (`*`) blanks by nothing.

Problem 80 Use `uniqC.awk` to plot the count of gene lengths as a function of length.

Problem 81 To complement the `END` pattern, there is also a `BEGIN` pattern, which opens a block executed before any input lines are dealt with. This makes it possible to write "ordinary" programs, which are executed once. If, for example, we would like to generate a random DNA sequence in the program `ranSeq.awk`, we could write:

```
BEGIN{
     print ">RandomSequence"
     srand(seed)
     s[0] = "A"
     s[1] = "T"
     s[2] = "C"
     s[3] = "G"
     for(i=0; i<10; i++){
         j = int(rand() * 4)
         printf("%s", s[j])
     }
     printf("\n")
}
```

and execute

```
awk -v seed=$RANDOM -f ranSeq.awk
```

The output is in FASTA format, which means that each sequence gets a header line, which starts with >, followed by the sequence data on one or more lines. Notice the `-v` option, which allows variables in the program to be set on the command line. Write a version of `ranSeq.awk` which takes as input not only the seed for the random number generator, but also the sequence length. Note that the `-v` option needs to be repeated for every variable set on the command line.