Chapter
Computational Biology
pp 77-96
Date: 29 January 2013

Playing with Text and Data Files

Röbbe Wünschiers

10.1007/978-3-642-34749-8_7
Copyright information

Abstract

How to look at gigabyte large text files? How to extract relevant or unique lines? How to compare text files? And, how to edit them? This chapter introduces basic command line tools and introduces you to the powerful VIM text editor.

7.1

After learning some basics in the previous chapter, you will learn more advanced tools from now on. Since you are going to learn programming, you should be able to enter a program, that is, a text. Certainly your programs will not run immediately. This means you need to edit text files, too. In this chapter you will use very basic text editing tools. We will concentrate on the text editor Vim.

With Linux you have the choice from an endless list of text editors. From Microsoft Windows you might have heard about the *Notepad*. That is the standard Windows text editor. With Linux you can choose between `ed`, `vi`, `vim`, `elvis`, `emacs`, `nedit`, `kedit`, `gedit` and many more. In general, we can distinguish between three different types of text editors: (a) One group of text editors is line-oriented. `ed` belongs to this group. You can only work on one line of the text file and then need a command to get to the next line. This means that you cannot use the arrow keys to move the cursor (your current position) through the text. This is stone-age text editing and good only for learning purposes. (b) The next group of text editors is screen-oriented. Vim belongs to this group of editors. You see the text over the whole screen and can scroll up and down and move to every text position using the arrow keys. This is much more comfortable than using a line-oriented editor. We will predominantly work with Vim. It is powerful and comfortable enough for our purpose and usually available on all systems. You can even use it when you have no X-Server (see Sect. 3.3 on p. 26) running, which is often the case when you login to a remote computer. (c) The most comfortable text editors, of course, use the X-Server. However, you should not confuse comfortable with powerful! `nedit` or `kedit` are examples of editors which use the graphical user interface (GUI). Here, you can use the mouse to jump around in the text and copy and paste (drag and drop) marked text. We are not going to work with these nice editors.

7.2 Viewing and Analyzing Files

This section introduces some import tools that allow you to analyze text-based files in the command line.

7.2.1 A Quick Start: cat

Now, let us start with a very easy method to input text. For this, we use the command `cat` (concatenate). We have already used `cat` for displaying the content of a text file (see for example Terminal 7 on p. 46). We have also learned about redirecting the output of command into a file by using `>`. Now, when you use `cat` without specifying a filename you redirect the standard input (the keyboard) to the standard output (the screen). Let us try it out.

Terminal 24: Printing Text with cat

```
1 $ cat
2 this is text, now press Enter
3 this is text, now press Enter
4 now press Ctrl-Dnow press Ctrl-D now press Enter AND Ctrl-D
5 now press Enter AND Ctrl-D
6 $
```

In Terminal 24 we first run the command `cat` without anything. After hitting Enter we get into line 2 and enter the text "this is text, now press Enter". At the end of line 2 we press Enter. The text we just entered will be printed again and we end up in line 4. Here we enter "now press Ctrl-D" and then press Ctrl + D. Again, the text will be printed, but now we end up in the same line. Now we press first Enter and then Ctrl + D. We are back. What happens is that the input from the keyboard is redirected to the screen. When we press Enter a new line starts before the redirection; but this is not important now. It is important that you realize that we could also redirect the standard input into a file! How?

Terminal 25: Writing Text with cat

```
1 $ cat > text-file
2 bla bla. Now press Enter
3 ahh, a new line! Now press Ctrl-D$ cat text-file
4 bla bla. Now press Enter
5 ahh, a new line! Now press Ctrl-D$
6 $ rm text-file
7 $
```

In Terminal 25 we redirect the standard input into a file named *text-file*. In line 2 we enter some text and then hit Enter. We end up in line 3, enter some text again and press Ctrl + D. We get back the command line directly. No new line is generated. This is uncomfortable! If we now, as shown in lines 3–5, display the text with `cat text-file`, we will see the same phenomenon in line 5: the dollar character is on the right of this line, instead of being at a new line. However, if you now press Enter you get into a new line. Hint: You should always finish a text file with a new line. Finally, in line 6 we remove the file that we have just created with the command `rm`.

It is easy to create a text file, is it not? Now create a file named *genomes.txt* as shown in File 1.

File 1: genomes.txt

```
1 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
2 A. thaliana (plant) - 100,000,000 bp - 25,000 genes
3 S. cerevisiae (yeast) - 12,100,000 bp - 6034 genes
4 E. coli (bacteria) - 4,670,000 bp - 3237 genes
5 4 lines text?
6 4 lines text?
7 203 characters?
```

Now let us see what we can do with the text file. We have already learned something about sorting the content of a text file in Terminal 9 on p. 48. Let us take a look at it again.

7.2.2 Text Sorting

With `sort` you can, as the name implies, sort the content of a text file. This can be very helpful in order to make files more readable. The `sort` command comes with some useful options, which are, as usual, separated by a dash.

-n	Sorting takes into account the value of numbers. Otherwise, only their position in the alphabet would be considered and 50 would be printed before 8.
-r	The sorting result is displayed in reverse order.
-f	Lower and uppercase characters are treated equally.
-u	Identical lines are printed only once. Thus you get unique or non-redundant output (see Sect. 7.1.3 on p. 81).
-k x	Sorts according to the content of field *x*. For example, "-k 3" sorts according to the content of field 3.
-t s	Set *s* as a new field separator (instead of the space character). For example, "-t :" sets the colon and "-t \t" sets the tabulator as field separator, respectively.

How does this work in real life?

Terminal 26: Sorting Text with sort

```
1 $ sort -nu genomes.txt
2 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
3 4 lines text?
4 203 characters?
5 $ sort -n genomes.txt | sort -u
6 203 characters?
7 4 lines text?
8 A. thaliana (plant) - 100,000,000 bp - 25,000 genes
9 E. coli (bacteria) - 4,670,000 bp - 3237 genes
10 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
11 S. cerevisiae (yeast) - 12,100,000 bp - 6034 genes
12 $ sort genomes.txt | sort -n
13 4 lines text?
14 A. thaliana (plant) - 100,000,000 bp - 25,000 genes
15 E. coli (bacteria) - 4,670,000 bp - 3237 genes
16 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
17 S. cerevisiae (yeast) - 12,100,000 bp - 6034 genes
18 203 characters?
19 $ sort -u genomes.txt | sort -n > sorted-genomes.txt
```

In Terminal 26 we use the options `-n` and `-u` in order to sort the content of the *genomes.txt* file, take care of correct sorting of numbers and get rid of doublets; but what is this? The output of `sort -nu genomes.txt` in lines 2–4 is slightly short!! Three organisms are missing. Okay, here we encounter an error (bug) in the program `sort`. With such a well-established command as `sort`, this is a rather rare situation. What can we do? First of all you should report the bug to the Linux community and thereby help to improve the commands. In the manual pages (`man sort`) there is a statement where to report bugs: "Report bugs to bug-textutils@gnu.org". Then you have to think about alternatives to your original task. In our case the alternative is shown in line 5. Here, we learn a new way of redirecting the output of a program: *piping*. Pipes are powerful tools to connect commands (see Sect. 8.6 on p. 103). We have already seen how to redirect the output from a command into a file (`>`). Now we redirect the output from one command to another. Therefore, we use the "`|`" character. The "`|`" is a pipe, and this type of pipe sends the stream of data to another program. The use of pipes means that programs can be very modular, each one executing some narrow, but useful task. In Terminal 26 line 5 we pipe the output of the `sort` command with the option `-n` (sort numbers according to their value) to the `sort` command with the option `-u` (eliminate duplicates). The result will be displayed on the screen. We could also redirect the result into a file, as shown in line 19.

The next example uses the options "`-t`" and "`-k`" in order to define a new field delimiter and the field, which is to be considered for sorting.

Terminal 27: Field Directed Sorting with sort

```
1 $ sort -t "-" -k 3 -n genomes.txt
2 203 characters?
3 4 lines text?
4 4 lines text?
5 A. thaliana (plant) - 100,000,000 bp - 25,000 genes
6 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
7 E. coli (bacteria) - 4,670,000 bp - 3237 genes
8 S. cerevisiae (yeast) - 12,100,000 bp - 6034 genes
9 $
```

With the `sort` command in line 1 of Terminal 27 the file *genomes.txt* is sorted according to the contents of field 3 (`-k 3`). The field delimiter is set to the dash character (`-t -`). With these options, `sort` becomes very versatile and can be used to sort tables according to the values of a certain column.

7.2.3 Extract Unique Lines

Frequently, it is desired to extract unique lines from a text file. One example would be that you delete duplicate gene IDs from a file. The command `sort` can do so with the option `-u`. I still wish to show an alternative to this with the `uniq` command. Terminal 28 shows an example.

Terminal 28: Extracting Unique Lines with sort and uniq

```
1 $ sort pdbfilelist.txt
2 1ABF.pdb
3 1ABR.pdb
4 1ABF.pdb
5 1NTN.pdb
6 2TIR.pdb
7 $ sort pdbfilelist.txt | uniq
8 1ABF.pdb
9 1ABR.pdb
10 1NTN.pdb
11 2TIR.pdb
12 $
```

The `uniq` command requires that the data lines are sorted. It will replace successive lines containing the same content with just one copy of this line.

7.2.4 Viewing File Beginning or End

Imagine you have a number of text files but you cannot remember their content. You just want to get a quick idea. You could use `cat` or a text editor, but sometimes it is enough to see just a couple of lines to remember the file content. `head` and `tail` are the commands of choice. `head filename` displays the first 10 lines of the file *filename*. `tail filename` does the same for the last 10 lines. You might want to use the option `-n`. The command

head -n 15 structure.pdb

displays the first 15 lines of the file *structure.pdb*. Of course, the `-n` option applies to `tail`, too.

Very handy comes the option `-f` of `tail`. It initiates a kind of real-time mode causing `tail` to wait for new lines being appended to the input file. This way you can monitor a log file of a BLAST result file. Monitoring can be stopped typing Ctrl + C. Note that this option is ignored if `tail` reads for a pipe and instead of a file.

7.2.5 Scrolling Through Files

Assuming that you are an eager scientist collecting lots of data and having accordingly large files, you might look for something else than `cat`, `head` or `tail`. Of course, you want to see the whole file and scroll through it! If you want to read a long file on the screen, you should use `less`. (There is another program called `more`. It is older and thus not powerful. Just to let you know!). The syntax is easy:

less filename

With `less` you can scroll forward and backward through the text file named *filename*. After invoking `less`, the next page is displayed with Space and the next line is displayed with Enter. By pressing Q you quit `less`. After typing the slash character / you can enter a query term. With n and p you jump to the next and previous lines matching the query pattern, which can of course be a regular expression (Sect. 11 on p. 157). Another nice function is to jump to the end of the file with typing G or to the beginning of the file with 1 → G. Accordingly, you can jump to any line, e.g. line 34 by typing 3 → 5 → G.

On some systems you will find the command `zless`, which can be used in order to view compressed files.

7.2.6 Character, Word, and Line Counting

Another thing we might want to do is count the number of lines, words, and characters of a text file. This task can be performed with the command `wc` (word count). `wc` counts the number of bytes, whitespace-separated words and lines in each given file.

Terminal 29: Counting Lines and Words and Characters with wc

```
1 $ wc genomes.txt
2        7      44     247 genomes.txt
3 $
```

As we can see from the output in Terminal 29, the file *genomes.txt* consists of 7 lines, 44 words, and 247 characters.

7.2.7 Splitting Files into Pieces

The `split` command splits files into a number of smaller files. Of the list of option I wish to restrict myself to `-l` (lines).

Terminal 30: Splitting Files with split

```
1 $ split -l 2 genomes.txt genomes.
2 $ ls g*
3 genomes.aa  genomes.ab  genomes.ac  genomes.ad  genomes.txt
4 $ cat genomes.aa
5 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
6 A. thaliana (plant) - 100,000,000 bp - 25,000 genes
7 $
```

The `split` command in Terminal 30 on the preceding page split the file *genomes.txt* into small files. Each resulting file contains two lines `-l 2` and the prefix of the resulting files is *genomes.*. The appendices *aa*, *ab* and so forth are set automatically.

7.2.8 Cut and Paste Columns

You are most probably used to cut-and-paste when writing with office applications or the like. This one here is different. The `cut` cuts columns out of a file. Columns are defined as text strings that are separated by a delimiter defined with `-d`. The option `-f` (fields) states the columns that shall be extracted. Terminal 31 shows how it works.

Terminal 31: Cutting Columns with cut

```
1 $ cut -f 1,3 -d "-" genomes.txt
2 H. sapiens (human) - 30,000 genes
3 A. thaliana (plant) - 25,000 genes
4 S. cerevisiae (yeast) - 6034 genes
5 E. coli (bacteria) - 3237 genes
6 4 lines text?
7 4 lines text?
8 203 characters?
9 $
```

The command `paste` on the other hand fuses files to a table contained in one single file. Terminal 32 illustrates how.

Terminal 32: Pasting Files to Columns with paste

```
1 $ cat file-1.txt
2 (one)1
3 (one)2
4 $ cat file-2.txt
5 (two)1
6 (two)2
7 $ cat file-3.txt
8 (three)1
9 (three)2
10 $ paste file-1.txt file-2.txt file-3.txt
11 (one)1  (two)1  (three)1
12 (one)2  (two)2  (three)2
13 $ paste -s file-1.txt file-2.txt file-3.txt
14 (one)1  (one)2
15 (two)1  (two)2
16 (three)1  (three)2
17 $
```

Assuming that you have three files with two lines of content each, `paste` would fuse the file contents to one single tab-delimited file. The option `-s` makes `paste` reading the files in serial order.

7.2.9 Finding Text: grep

A very important command is `grep` (global regular expression printer, for a historical background see Sect. 11.1 on p. 157). `grep` searches the named input file(s) for lines containing a match to a given pattern. By default, `grep` prints the matching lines. Terminal 33 illustrates its function.

Terminal 33: Extracting Lines with grep

```
1 $ grep human genomes.txt
2 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
3 $ grep genes genomes.txt | wc -w
4       36
5 $ grep genes genomes.txt | wc -l
6       4
7 $ grep -c human genomes.txt
8 1
9 $ grep -c genes genomes.txt
10 4
11 $ grep 'H. sapiens' genomes.txt
12 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
13 $ grep 'bacteria\|human' genomes.txt
14 H. sapiens (human) - 3,400,000,000 bp - 30,000 genes
15 E. coli (bacteria) - 4,670,000 bp - 3237 genes
16 $
```

The name grep has nothing to do with the verb to grab. It is derived from a function of a very old Unix text editor called `ed`. Terminal 34 shows how the text file *text.txt* can be opened with `ed` (line 4). The command line now accepts `ed` commands (I added comments to the commands in the terminal—you should not type these).

Terminal 34: Historical g/RE/p

```
1 $ cat text.txt
2 line 1
3 line 2
4 $ ed text.txt
5 14              # number of characters
6 ,p              # Print all lines
7 line 1
8 line 2
9 g/2/p           globally print lines matching RE
10 line 2
11 q              # quit
12 $
```

The command `g/2/p` searches globally (g) for matches to the regular expression `/2/` (see Chap. 11 on p. 157) and prints (p) matching lines. Since the shortcut for the regular expression is RE, the command becomes `g/RE/p`, which is grep.

`grep` really is much more powerful than you can imagine right now! It comes with a whole bunch of options, only one of which was applied in Terminal 33. In line 1 we search for the occurrences of the word human in the file *genomes.txt*. The matching line is printed out. In line 3 we search for the word genes and pipe the matching lines to the program `wc` to count the number of words (option `-w`) of these lines. The result is 36. Next, in line 5, we count the number of lines (`wc` option `-l`) in which the word genes occurs; we can obtain this result much more easily by using the `grep` option `-c` (count). With this option `grep` displays only the number of lines in which the query expression occurs. How can we search for two strings at once? This is shown in line 13. Note that we enclose the words we are searching for in single quotes (you could also use double quotes); and how about querying for two words that are not necessarily in one line? Use the combination "`\|`" between the words as shown in line 13 of Terminal 33 on the preceding page. This stands for the logical or. Be careful not to include spaces!

At this stage you should already have a feeling about the power of the Linux command line. We will get back to `grep` when we talk about regular expressions in Chap. 11 on p. 157. Just one last thing: with the option `--color` you can highlight the matching text in the terminal.

7.2.10 Text File Comparisons

Two interesting commands to compare the contents of text files are `diff` and `comm`. To see how these commands work, create two text files with a list of words, as shown in the following Terminal:

Terminal 35: Finding Differences with diff

```
1 $ cat>amino1
2 These amino acids are polar:
3 Serine
4 Tyrosine
5 Arginine
6 $ cat>amino2
7 These amino acids are polar and charged:
8 Lysine
9 Arginine
10 $ diff amino1 amino2
11 1,3c1,2
12 < These amino acids are polar:
13 < Serine
14 < Tyrosine
15 ---
16 > These amino acids are polar and charged:
17 > Lysine
18 $ diff -c amino1 amino2
19 *** amino1   2003-05-11 18:42:53.000000000 +0200
20 --- amino2   2003-05-11 18:43:20.000000000 +0200
21 ***************
22 *** 1,4 ****
23 ! These amino acids are polar:
24 ! Serine
25 ! Tyrosine
26   Arginine
27 --- 1,3 ----
28 ! These amino acids are polar and charged:
29 ! Lysine
30   Arginine
31 $
```

The result of the command `diff` (difference) indicates what you have to do with the file *amino1* to convert it into *amino2*. You have to delete the lines marked with "<" and add the lines marked with ">". With the option `-c` (line 18) the same context is shown in another way. The option `-u` (line 28) is used to display the differences only (indicated by a "!").

The command `comm` (compare) requires that the input files are sorted. We do this in line 1 in Terminal 36. Now we can compare both files in one line, separating them with the semicolon character (;).

Terminal 36: Finding Things in Common with comm

```
1 $ sort amino1>amino1s; sort amino2>amino2s
2 $ comm amino1s amino2s
3                 Arginine
4         Lysine
5 Serine
6 These amino acids are polar:
7         These amino acids are polar and charged:
8 Tyrosine
9 $ comm -12 amino1s amino2s
10 Arginine
11 $
```

The `comm` command prints out its result in three columns. Column one contains all lines that appear only in the first file (*amino1s*), column two shows all lines that are present only in the second file (*amino2s*), and the third column contains all files that are in both files. You can restrict the output with the options `-n`, *n* being one or more column numbers, which are not to be printed. Line 9 in Terminal 36 displays only the content of the third column ("minus 1 and minus 2"), which contains the common file content.

7.3 Editing Text Files

It is very helpful to know how to edit files in the command line. Suppose you are connected to a remote computer without graphical interface. The only chance then is to edit files with a command line editor. There are several such editors available. Here I present two of them: The Pico editor might help you when you forget how to use Vim. I highly recommend you to learn Vim—it is much much more powerful than Pico.

7.3.1 The Sparse Editor Pico

Probably the easiest way to handle text editor is called `pico` (pine composer). It has the same look and feel as an old DOS editor (see Fig. 7.1). However, `pico` is not installed on all Linux systems. Thus, you should make the effort to learn `vi` (see Sect. 7.3.2), which is a bit harder but more universal and which offers many more possibilities (in fact, you need to memorize only six commands in order to work Vim). The `pico` editor was developed by the University of Washington. You start `pico` with the command `pico` and, optional, with a filename. If the file already exists, it will be opened. When you write a program it is highly recommended to use the option `-w` (wrap). It tells `pico` not to break (wrap) the lines at the right margin of the editing window, but only when you press the Enter key. Thus, the command to start `pico` would look like this `pico -w filename`. If you provide no filename, that is fine as well. You can save your text later and provide a filename at that time.

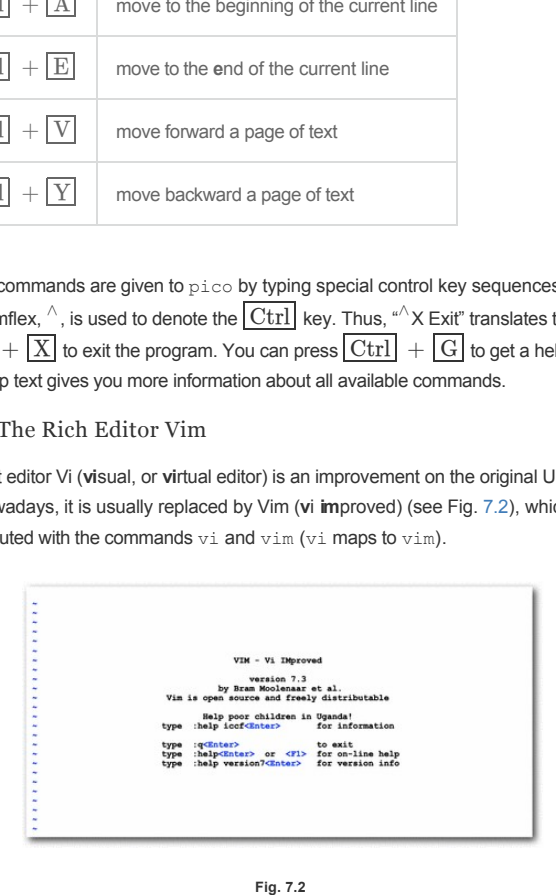

Fig. 7.1
Pico: The main window of the text editor Pico

When you have started `pico` you will get a kind of a graphical interface (see Fig. 7.1). The top of the display, called status line, shows the version number of `pico` that you use, the current file being edited and whether or not the file has been modified. The third line from the bottom is used to report informational messages and for additional command input. Now comes the comfortable part of `pico`: the bottom two lines list the available editing commands. Depending on your current action, the content of this window might change.

Each character you type is automatically inserted at the current cursor position. You can move the cursor with the arrow keys. If this does not work, you have to use the following commands:

Ctrl + F	move forward one character
Ctrl + B	move backward one character
Ctrl + P	move to the previous line
Ctrl + N	move to the next line
Ctrl + A	move to the beginning of the current line
Ctrl + E	move to the end of the current line
Ctrl + V	move forward a page of text
Ctrl + Y	move backward a page of text

Editing commands are given to `pico` by typing special control key sequences. A circumflex, ^, is used to denote the Ctrl key. Thus, "^X Exit" translates to: press Ctrl + X to exit the program. You can press Ctrl + G to get a help text. This help text gives you more information about all available commands.

7.3.2 The Rich Editor Vim

The text editor Vi (visual, or virtual editor) is an improvement on the original Unix editor `ed`. Nowadays, it is usually replaced by Vim (vi improved) (see Fig. 7.2), which can be executed with the commands `vi` and `vim` (`vi` maps to `vim`).

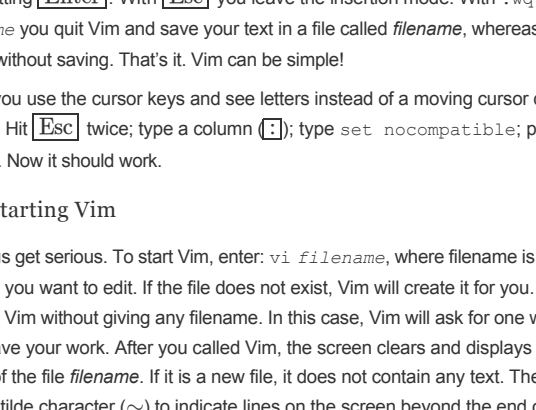

Fig. 7.2
Vim: The text editor Vim, which nowadays replaces vi

Vim (pronounced vee-eye improved, or just vim) provides text editing capabilities for beginners and experts alike. Three aspects of Vim make it appealing. First, Vim is supplied with all Linux systems. Thus, you can use Vim at other universities or any institutes with Unix or Linux systems. Second, Vim uses a small amount of memory, which allows efficient operation when the network is busy. Third, because Vim uses standard alphanumeric keys for commands, you can use it on virtually any terminal or workstation in existence without having to worry about unusual keyboard mappings. You see, there are really many advantages to learning Vim.

In celebration of its twentieth birthday in 2011 Vim has even been ported to Apple's iOS (http://applidium.com/en/applications/vim/) and other mobile devices, e.g. Android (http://play.google.com/store/apps/details?id=com.easyvim.android.vim). Astonishingly, it has many features of the modern editors such as auto-completion, a visual mode, language-aware syntax highlighting, integrated scripting language functionalities, macros recording and playback, markers management to quickly move around the edited file or multiple clipboards.

7.3.3 Installing Vim

In case that you are working with Ubuntu, e.g., as outlined in Sect. 4.1.2 on p. 40, you need to install Vim. By default, Ubuntu comes with vi installed. However Vim is much more comfortable to use. Therefore, execute the following command in the terminal to install Vim:

sudo apt-get install vim

Even if you now enter `vi`, Linux executes `vim`.

7.3.4 Immediate Takeoff

If you cannot wait to use Vim and make your first text file: here you go. Type `vi` to start the editor. Press `i` to start the insertion mode and enter your text. You start a new line by pressing Enter. With Esc you leave the insertion mode. With `:wq filename` you quit Vim and save your text in a file named *filename*, whereas with `:q!` Enter you quit without saving. Well—it works!

Note. If you use the cursor keys and see letters instead of a moving cursor do the following: Hit Esc twice; type a column (:); type `set nocompatible`; press Enter.

7.3.5 Starting Vim

Now let us get serious. To start Vim, enter `vi filename`, where filename is the name of the file you want to edit. If the file does not exist, Vim will create it for you. You can also start Vim without giving any filename. In this case, Vim will ask for one when you quit or save your work. After you call Vim, the screen clears and displays the content of the file *filename*. If it is a new file, it does not contain any text. Then Vim uses the tilde character (~) to indicate lines on the screen beyond the end of the file. Vim uses a cursor to indicate where your next command or text insertion will take effect. The cursor is the small rectangle, which is the size of one character, and the character inside the rectangle is called the current character. At the bottom of the window, Vim maintains an announcement line, called the mode line. The mode line lists the current line of the file, the filename, and its status. Let us now start Vim with the new file *text.txt*. The screen should then look like Terminal 37. Please note that I have deleted empty lines (marked by ~) to save space, i.e., paper. The cursor is represented by [].

Terminal 37: Starting Vim

```
1 []
2 ~
3 ~
4 ~
5 ~
6 ~
7 ~
8 ~
9 ~
10 "text.txt" [New File]                0,0-1        All
```

7.3.6 Modes

Line 10 in Terminal 37 shows the mode line. At this stage you cannot enter any text because Vim runs currently in the command mode. In order to enter text, the input mode must be activated. To switch from the command mode to the input mode, press the `i` key (you do not need to press Enter). Vim lets you insert text beginning at the current cursor location. To switch back to command mode, press Esc. You can also use Esc to cancel an unfinished command in command mode. If you are uncertain about the current mode, you can press Esc a few times. When Vim beeps, you have returned to the command mode. Okay, let us change to the input mode and enter some text.

Terminal 38: Entering Text in Vim

```
1 This is new text in line 1. Now I press ENTER
2 and end up in the second line. I could also write to the e
3 nd of the line. The text will be wrapped automatically[]
4 ~
5 ~
6 ~
7 ~
8 ~
9 ~
10 -- insert --                           3,54         All
```

You can see some changes in the mode line in Terminal 38. "`-- insert --`" indicates that you are in the input mode. Furthermore, the current cursor position (line 3, column 54) is indicated. Now press Esc to get back into the command mode, "`-- insert --`" will disappear. Now let us save the file: press `:w`, and then Enter. The mode line will display a message as shown in Terminal 39. If "`:w`" appears in your text you are still in the input mode!

Terminal 39: Saving Text in Vim

```
1 This is new text in line 1. Now I press ENTER
2 and end up in the second line. I could also write to the e
3 nd of the line. The text will be wrapped automatically
4 ~
5 ~
6 ~
7 ~
8 ~
9 ~
10 "text.txt" [new] 3L, 160C written     3,54         All
```

Commands are very often preceded with the colon "`:`" character. Let us try another command: type `:set number`. Now you see line numbers in front of each line. With `:set nonumber` you hide them again. Another command: type `:r !ls`. Woop. After hitting Enter you have a list of all files in your current working directory imported to your text file. That is magic, is it not? Vim executed the shell command `ls` and imported the result into the text file at the current cursor position.

7.3.7 Moving the Cursor

Probably your screen is quite full now. Let us move the cursor. Usually you can use the arrow keys (if you use the cursor keys and see letters instead of a moving cursor do the following: Hit Esc twice; type a column (:); type `set nocompatible`; press Enter.); but if they do not work you can use the following keys:

h	move one character to the left
l	move one character to the right
k	move up one line
j	move down one line

There are some more powerful commands for long-distance jumping.

0	move to the beginning of the line
$	move to the end of the line
H	move to the top line of the screen
M	move to the middle line of the screen
L	move to the bottom line of the screen
G	move to the last line of the text
*n*G	move to the n-th line in the text
Ctrl + f	move one screen forward
Ctrl + b	move one screen backward

You have now learned some powerful tools for moving around in a file. You should memorize only the basic movements and know where to look up for the others (you can download a cheat sheet at http://www.kcomputing.com/vi.html).

7.3.8 Doing Corrections

Maybe the most relaxing thing to know is that you can always *undo* changes by typing `u`. With Vim you can even undo many commands, whereas Vi will recover only the last text change. If you are lucky, you can use the keys BkSp and Del in order to make *deletions* in the input mode. Otherwise you must make use of the commands in the command modus. To do so, first move the cursor so that it covers the first character of the group you want to delete, then type the desired command from the list below.

x	delete only the current character
D	delete to the end of the line
db	delete from the current character to the beginning of the current word
de	delete from the current character to the end of the current word
dd	delete the current line
dw	delete from the current character to the beginning of the next word

Note that the second letter of the command specifies the same abbreviations as the cursor movement commands do. In fact, you can use delete with all of the cursor movement specifiers listed above, e.g., `dH` would delete everything from the current line to the top line off the screen.

In other cases you will need only to *replace* a single character or word, rather than deleting it. Vim has change and replace functions, too. First, move to the position where the change should begin (the desired line or the beginning of the desired word). Next, type the proper command from the list below. Finally, enter the correct text, usually concluded with Esc (except for `r`).

cw	Change a word.
C	Overwrite to the end of the line.
r	Replace a single character with another one. No Esc necessary.
R	Overwrite characters starting from the current cursor position.
s	Substitute one or more characters for a single character.
S	Substitute the current line with a new one.
:r *file*	Insert an external file at the current cursor position.

The change command `c` works like the delete command; you can use the text portion specifiers listed in the cursor movement list.

7.3.9 Save and Quit

Vim provides several means of saving your changes. Besides saving your work before quitting, it is also a good idea to save your work periodically. Power failures or system crashes can cause you to lose work. From the command mode, you type `:w` (write) and hit Enter. In order to save the text in a new file, type `:w filename`. You quit Vim with `:q`. You can save and quit at once with `:x` or `:wq`. If you do not want to save your changes you must force quitting with `:q!`. Be cautious when abandoning Vim in this manner because any changes you have made will be permanently lost.

Up to this point you have learned more than enough commands to use Vim in a comfortable way. The next two sections explain some more advanced features which you might wish to use.

7.3.10 Copy and Paste

Frequently, you will need to cut or copy some text, and paste it elsewhere into your document. Things are easy if you can work with the mouse. When you mark some text with the mouse (holding the left mouse button) the marked text is in the memory (*buffer*). Pressing the right mouse button (or, on some systems, the left and right or the middle mouse buttons) pastes the text at the current cursor position. You can apply the same mechanism in the terminal window!

Things are slightly more complicated if you have only the keyboard. First you cut or copy the text into temporary storage, then you paste it into a new location. Cutting means removing text from the document and storing it, while copying means placing a duplicate of the text in storage. Finally, pasting just puts the stored text in the desired location. Vim uses a *buffer* to store the temporary text. There are nine numbered buffers in addition to an undo buffer. The undo buffer contains the most recent delete. Usually buffer 1 contains the most recent delete, buffer 2 the next most recent, and so forth. Deletions older than 9 disappear. However, Vim also has 26 named buffers (a–z). These buffers are useful for storing blocks of text for later retrieval. The content of a buffer does not change until you put different text into it. Unless you change the contents of a named buffer, it holds its last text until you quit. Vim does not save your buffers when you quit.

The simplest way to copy or move text is by entering the source *line numbers* and the destination line numbers. The `m` command moves (cuts and pastes) a range of text, and the `t` command transfers (copies and pastes) text. The commands have the syntax shown below:

:x m y	Move line number x below line number y.
:x, ymz	Move the lines between and including x and y below line z.
:x t y	Copy line x below line y.
:x, y t z	Copy lines between and including x and y below line z.

Another way is to use *markers*. You can mark lines with a letter from a to z. These markers behave like invisible bookmarks. To set a mark you use `mx`, with x being a letter from a to z. You can jump to a mark with `'x`. The following list shows you how to apply bookmarks to copy or move text. Note: bookmarks and line numbers can be mixed.

m x	Set a bookmark at the current line. x can be any letter from a–z.
' x	Jump to bookmark x.
:' x' y co' z	Copy lines between and including bookmarks x and y below bookmark z.
:' x' ym' z	Move lines between and including bookmarks x and y below bookmark z.
:' x,' y w *file*	Write lines between and including bookmarks x and y into a file named *file*.

One last method uses the commands `d` (delete) or `y` (yank). With this method you can make use of different *buffers*. Go to the line you wish to copy or cut and press `yy` (yank) or `dd` (delete), respectively. Then move the cursor to the line behind which you want to insert the text and type `p` (paste). In order to copy a line into a buffer type "*x* `yy`, with x being a letter from a–z. You insert the buffer with "*x* `p`. To copy more than one line precede the command `yy` with the number of lines. `2yy` copies three lines and `3yw` copies three words. You see, Vim is very flexible and you can combine many commands. If you are going to work a lot with it you should find out for yourself which commands you prefer.

7.3.11 Search and Replace

Finally, let us talk about another common issue: searching and replacing text. As files become longer, you may need assistance in locating a particular instance of text. Vim has several search and search and replace features. Vim can search the entire file for a given string of text. A string is a sequence of characters. Vim searches forward with the slash (/) or backward with the question mark key (?). You execute the search by typing the command, then the string followed by Enter. To cancel the search, press Esc instead of Enter. You can search again by typing `n` (forward) or `N` (backward). Also, when Vim reaches the end of the text, it continues searching from the beginning. This feature is called *wrapscan*. Of course, you can use wildcards or regular expressions in your search. We will learn more about this later in Sect. 11.6.1 on p. 172. Let us take a look at the search and replace commands:

/xyz	Search forward for xyz.
?xyz	Search backward for xyz.
n	Go to the next occurrence.
N	Go to the previous occurrence.
:s/abc/xyz/	Replace the first instance of `abc` with `xyz` in the current line.
:s/abc/xyz/g	Replace all instances of `abc` with `xyz` in the current line.
:s/abc/xyz/gc	Ask before replacing each instance of `abc` with `xyz` in the current line.
:%s/abc/xyz/gc	Replace all instances of `abc` with `xyz` in the whole file.
:%s/abc/xyz/gc	Ask before replacing each instance of `abc` with `xyz` in the whole file.
:x, ys/abc/xyz/g	Replace all instances of `abc` with `xyz` between lines *x* and *y*.

Remember, this is only a small selection of commands that I present here.

7.4 Text File Conversion (Unix ↔ DOS)

It is common that text files generated either on Linux or DOS/Windows cause problems on the other system. The reason lies in different syntax for the newline command, i.e., the file ending. An easy way out is provided by the commands `unix2dos filename(s)` and `dos2unix filename(s)`, respectively.

If this command is not available on your system you would have to install it. Ubuntu users can run `sudo apt-get install dos2unix`, which installs both `dos2unix` and `unix2dos`.

7.4.1 Batch Editing

Finally, let me show you that Vim commands can be handled like a programming language, i.e., you can store them in a file and execute the commands as a batch job. In the following example we work on File 1 on p. 79 again. First, create a text file containing the Vim commands as shown in File 2. Note that new commands are allowed in this file.

File 2: *edit.vi*

```
1 :g/^[^A-Z]/d
2 :%s/,/./g
3 :%s/ - /\t/g
4 :w %<.new
5 :q
```

The first command globally deletes all lines matching the pattern, i.e., all lines not starting with an uppercase letter (see Chap. 11 on p. 157). Then all commas are substituted with dots throughout the file. Command three replaces ␣-␣ with tabulators. Line 4 makes use of the special variable % that holds the filename. % < folds the filename without the filename extension. Thus, **%<.new** converts the filename *genomes.txt* into *genomes.new*. **:w** writes the edited content to that file. Finally the Vim session is quitted with **:q**.

Terminal 40 shows the execution.

Terminal 40: Batch Jobs with Vim

```
1 $ vi -e genomes.txt < edit.vi
2 $ cat genomes.new
3 H. sapiens (human) 3.400.000.000 bp     30.000 genes
4 A. thaliana (plant) 100.000.000 bp 25.000 genes
5 S. cerevisiae (yeast)   12.100.000 bp    6034 genes
6 E. coli (bacteria)  4.670.000 bp     3237 genes
7 $
```

Take a look at Sect. 8.10 on p. 106, how you would run this batch job on several files.

If you have followed this section about Vim up to this stage, you should have obtained a very good overview of its capabilities. It can do more and it offers a whole range of options that one could set to personal preferences. However, since I do not believe that anyone is really going deeper into this, I stop at this point. You are welcome to read some more lines about Vim in focussed books (Qualline 2001; Robbins 1999; Lamb and Robbins 1998).

7.5 Exercises

Now sit down and play around with some text. This is elementary!

7.1. Create a text file named *fruits.txt* using `cat`. Enter some fruits, one in each line. Append some fruits to this file.

7.2. Create a second text file named *vegetable* containing a list of vegetables, again, one item per line. Now concatenate fruits and vegetables onto the screen and into a file named *dinner*.

7.3. Sort the content of *dinner*.

7.4. Take some time and exercise with Vim. Open a text document with Vim or type some text and go through the description in this section. You must know the basic commands in order to write and edit text files!

Copyright information