1

Introduction to Linux and Command Line Tools for Bioinformatics
Shikai Liu and Zhanjiang Liu

Introduction

Dealing with huge omics datasets in the genomics era, bioinformatics is essential for the transformation of raw sequence data into meaningful biological information for all branches of life sciences, including aquaculture. Most tasks of bioinformatics are processed using the Linux operating system (OS). Linux is a stable, multi-user, and multi-tasking system for servers, desktops, and laptops. It is particularly suited to working with large text files. Many of the Linux commands can be combined in various ways to amplify the power of command lines. Moreover, Linux provides the greatest level of flexibility for development of bioinformatics applications. The majority of bioinformatics programs and packages are developed on the Linux OS. Although most programs can be compiled to run on Microsoft Windows systems, it is generally more convenient to install and use the programs on Linux systems. Therefore, familiarity with and understanding of basic Linux command lines is essential for bioinformatic analysis. In this chapter, we provide an introduction to the Linux OS and its basic command line tools.

An operating system (OS) is basically a suite of programs that make the computer work. It manages computer hardware and software resources and provides common services for computer programs. Examples of popular modern OSs include Microsoft Windows, Linux, macOS, iOS, BSD, Android, BlackBerry OS, and Chrome OS. All these examples share the root of a UNIX base, except for Microsoft Windows.

The UNIX OS was developed in the late 1960s and first released in 1971 by AT&T Bell Labs. It has been under continuous development ever since. UNIX is proprietary, however, which hindered its wide academic use. Researchers at University of California-Berkeley developed an alternative to AT&T Bell Labs’ UNIX OS, called the Berkeley Software Distribution (BSD). BSD is an influential operation system, from which several notable OSs such as Sun’s SunOS and Apple Inc’s macOS system are derived. In the 1990s, Linus Torvalls developed a non-commercial replacement for UNIX, which eventually became the Linux OS. Linux was released as free open source software, with its underlying source code publicly available, freely distributed, and freely modified. Linux is now used in numerous areas, from embedded systems to supercomputers. It is the most common OS powering web servers around the world. Many Linux distributions have been developed, such as Red Hat, Fedora, Debian, SUSE, and Ubuntu. Each distribution has the Linux kernel at its core, but builds on top
of that with its own selection of other components, depending on the target users of
the distribution. From the perspective of end users, there is no big difference between
Linux and UNIX. Both use the same shell (e.g., bash, ksh, csh) and other development
tools such as Perl, PHP, Python, and GNU C/C++ compilers. However, because of the
freeware nature of the Linux OS, it has the most active support community.

Linux is well known for its command line interface (CLI), while it also has a graphi-
cal user interface (GUI). Similar to Microsoft Windows, the GUI provides the user an
easy-to-use environment. Currently, the most common way to interact with a Linux OS
is via a GUI. In general, the GUI is powered by a derivative of the X11 Window System,
commonly referred to as “X11.” A desktop manager runs in the X11 Window System
and supplies the menus, icons, and windows to interact with the system. The KDE (the
default desktop for openSUSE) and GNOME (the default desktop for Ubuntu) are two of
the most popular desktop environments. On the modern Linux OS, although the GUI
provides the graphical “user-friendliness,” the “unhandy” text-based CLI is where the
true power resides. In the field of bioinformatics, almost all applications are executed
with CLI.

Linux is a stable, multi-user, and multi-tasking system for servers, desktops, and lap-
tops. It is particularly suited to working with large text files because it has a large number
of powerful commands that specialize in processing text files. Most of these commands
can be further combined in various ways to amplify the power of command lines. In
the genomics era, with sequencing data being explosively accumulated, bioinfor-
matics has become a scientific discipline of its own. Bioinformatics relies heavily on the
Linux OS because it mostly works with text files containing nucleotide and amino acid
sequences. Moreover, Linux provides the greatest level of flexibility for the development
of bioinformatics applications. The majority of bioinformatics programs and packages
are developed on Linux-based systems. Although most bioinformatics programs can be
compiled to run on Microsoft Windows systems, it is more convenient to install and use
the program on Linux-based systems.

In this chapter, we introduce the Linux OS and its basic command lines. All com-
mands introduced in Linux are valid for UNIX or any UNIX-like OSs. This chapter
functions as a boot camp of Linux command lines to assist bioinformatics beginners
in going through with the commands and packages discussed in the remaining chapters
of this book. Readers who are already familiar with Linux and its command lines can
skip this chapter.

Overview of Linux

The Linux OS is made up of three parts: the kernel, the shell, and the program
(Figure 1.1). The kernel is the hub of the OS, which allocates time and memory to
programs, and handles the file system and communications in response to system calls.
The shell and the kernel work together. As an illustration, let us suppose a user types in
a command line ls myDirectory. The ls command is used to list the contents of a
directory. In this process, the shell will search the file system for the file containing the
program ls, and then request the kernel, through system calls, to execute the program
(ls) to list the contents of the directory (myDirectory).
The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called shell. The shell is a command line interpreter, which interprets the commands that the user types in and passes them to the OS to perform. The shell can be customized by users, and different shells can be used on the same machine. The most influential shells include the Bourne shell (sh) and the C shell (csh). The Bourne shell was written by Stephen Bourne at AT&T as the original UNIX command line interpreter, which introduced the basic features common to all UNIX shells. Every UNIX-like system has at least one shell compatible with the Bourne shell. The C shell was developed by Bill Joy for Berkeley Software Distribution, which was originally derived from the UNIX shell with its syntax modeled after the C programming language. The C shell is primarily for interactive terminal use, and less frequently for scripting and OS control. Bourne-Again shell (bash) is a free software replacement for the Bourne shell, which is written as a part of the GNU Project. Bourne-Again shell is distributed widely as the shell for GNU OSs and as a default interactive shell for users on most GNU/Linux and macOS systems.

The users interact with the shell through terminals—that is, programs called terminal emulators. A bunch of different terminal emulators are available. Most Linux distributions supply several, such as gnome-terminal, konsole, xterm, rxvt, kvt, nxterm, and eterm. Although many different terminal emulators exist, they all do the same thing: open a window and give users access to a shell session. After opening a terminal, the shell will give a prompt (e.g., $) to request commands from the user. When the current command terminates, the shell gives another prompt.

A computer program is a list of instructions passed to a computer to perform a specific task or a series of tasks. Linux commands are themselves programs. A command can take options, which change the behavior of the command. Manual pages are available for each command, to provide detailed information on which options it can take, and how each option modifies the behavior of the command.

### Directories, Files, and Processes

Everything in Linux is either a file/directory or a process. A process is an executing program identified by a unique process identifier. A file is a collection of data such as a document (e.g., report and essay), a text of a program written in some high-level programming language (e.g., a shell script), a collection of binary digits (e.g., a binary executable file), or a directory. All the files are grouped together in the directory structure.
Linux files are arranged in a single-rooted, hierarchical structure, like an inverted tree (Figure 1.2). The top of the hierarchy is traditionally called the root (written as a slash—“/”). As shown in Figure 1.2, the home directory (“home”) contains a user home directory (“aubsxl”). The user home directory contains a subdirectory (“linuxDemo”) that has two files (“file1.txt” and “file2.txt”). The full path of the “file1.txt” is “/home/aubsxl/linuxDemo/file1.txt”.

Filename Conventions

In Linux, files are named conventionally, starting with a lower-case letter and ending with a dot, followed by a group of letters indicating the contents of the file. For instance, a file consisting of C code is named with the ending “.c”, such as “prog1.c”. A good way to name a file is to use only alphanumeric characters (i.e., letters and numbers) together with underscores (_) and dots (.). Characters with special meanings—such as /, *, &, %, and spaces—should be avoided. A directory is merely a special type of file (like “a container for files”); therefore, the rules and conventions for naming files apply to directories as well.

Wildcards

Wildcards are commonly used in Linux shell commands, and also in regular expressions and programming languages. Wildcards are characters that are used to substitute for other characters, increasing the flexibility and efficiency of running commands. Three types of wildcards are widely used: *, ?, and []. The star (*) is the most frequently used wildcard. It matches against one or more character(s) in the name of a file (or directory). For instance, in the “linuxDemo” directory, type

$ ls file*

This will list all files that have names starting with the string “file” in the current directory. Similarly, type

$ ls *.txt

This will list all files that have names ending with “.txt” in the current directory.
The question mark (?) is another wildcard, which matches exactly one character. For instance,

```
$ ls file?.txt
```

This will list both “file1.txt” and “file2.txt”, but will not list the file if it is named “file_1.txt.”

The third type of wildcard is a pair of square brackets ([ ]), which represents a range of characters (or numbers) enclosed in the brackets. For instance, the following command line will list files with names starting with any letter from a to z:

```
$ ls [a-z]*.txt
```

### File Permission

Each file (and directory) has associated access rights, which can be shown by typing “ls -l” in the terminal (Figure 1.3). Also, “ls -lg” gives additional information as to which group owns the file (e.g., “file1.txt” is owned by the group named “aubfish” in the figure).

The left-hand column in Figure 1.3 is a 10-symbol string that consists of symbols, including d, l, r, w, x, and -. If d is present, it will be at the left-hand end of the string, and will indicate a directory; otherwise - will be the starting symbol of the string indicating a file. The symbol of l is used to indicate the links of a file or directory.

The nine remaining symbols indicate the permissions, or access rights, and are taken as three groups of three (Figure 1.3).

1) The left group of three gives the file permissions for the user that owns the file (or directory) (i.e., “aubsxl” in the figure).
2) The middle group of three gives the permissions for the group of people who own the file (or directory) (i.e., “aubfish” in the figure).
3) The rightmost group of three gives the permissions for all other users.

The symbols have slightly different meanings, depending on whether they refer to a file or to a directory. For a file, the r (or -) indicates the presence or absence of permission to read and copy the file; w (or -) indicates the permission (or otherwise) to write (change) a file; and x (or -) indicates the permission (or otherwise) to execute a file. For a directory, the r allows users to list files in the directory; w allows users to delete files from the directory or move files into it; and x allows users to access files in the directory.

### Change File Permission

The owner of a file can change the file permissions using the `chmod` command. The options of `chmod` are listed in Table 1.1. For instance, to remove read, write, and execute permissions on the file “file1.txt” for the group and others, type

```
-rw-rw-r-- 1 aubsxl aubfish 0 Oct 16 10:30 file1.txt
```

![Figure 1.3](image-url) An illustration of file permission.
Table 1.1 The options of chmod command.

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>user</td>
</tr>
<tr>
<td>g</td>
<td>group</td>
</tr>
<tr>
<td>o</td>
<td>other</td>
</tr>
<tr>
<td>a</td>
<td>all</td>
</tr>
<tr>
<td>r</td>
<td>read</td>
</tr>
<tr>
<td>w</td>
<td>write</td>
</tr>
<tr>
<td>x</td>
<td>execute</td>
</tr>
<tr>
<td>+</td>
<td>add permission</td>
</tr>
<tr>
<td>-</td>
<td>take away permission</td>
</tr>
</tbody>
</table>

Table 1.2 List of octal numbers for file permissions.

<table>
<thead>
<tr>
<th>Number</th>
<th>Permission</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No permission</td>
</tr>
<tr>
<td>1</td>
<td>Execute only</td>
</tr>
<tr>
<td>2</td>
<td>Write only</td>
</tr>
<tr>
<td>3</td>
<td>Write and execute (1 + 2)</td>
</tr>
<tr>
<td>4</td>
<td>Read only</td>
</tr>
<tr>
<td>5</td>
<td>Read and execute (4 + 1)</td>
</tr>
<tr>
<td>6</td>
<td>Read and write (4 + 2)</td>
</tr>
<tr>
<td>7</td>
<td>Read, write, and execute (4 + 2 + 1)</td>
</tr>
</tbody>
</table>

$ chmod go-rwx file1.txt

To give read and write permissions on the file “file1.txt” to all, type

$ chmod a+rw file1.txt

The file permissions can also be encoded as octal numbers (Table 1.2), which can be used in the chmod command. For instance, to give all permissions on the file “file1.txt” to the owner, read and execute permission to the group, and no permission to others, type

$ chmod 750 file1.txt

Environment Variables

Each Linux process runs in a specific environment. An environment consists of a table of environment variables, each with an assigned value. When the user logs in, certain login files are executed, which initializes the table holding the environment variables for the process. The table becomes accessible to the shell once the login files pass the
Table 1.3 A list of examples of environment variables.

<table>
<thead>
<tr>
<th>Environment variables</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>USER</td>
<td>The login name</td>
</tr>
<tr>
<td>HOME</td>
<td>The path name of the home directory</td>
</tr>
<tr>
<td>HOST</td>
<td>The name of the computer</td>
</tr>
<tr>
<td>ARCH</td>
<td>The architecture of the computer's processor</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>The name of the computer screen to display X11 windows</td>
</tr>
<tr>
<td>PRINTER</td>
<td>The default printer to send print jobs to</td>
</tr>
<tr>
<td>PATH</td>
<td>The directories the shell should search to find a command</td>
</tr>
<tr>
<td>PWD</td>
<td>The current working directory</td>
</tr>
<tr>
<td>BASH_VERSION</td>
<td>The version of bash being executed</td>
</tr>
<tr>
<td>BASHOPTS</td>
<td>The list of options that were used when bash was executed</td>
</tr>
<tr>
<td>HISTSIZE</td>
<td>Number of lines of command history allowed in memory</td>
</tr>
<tr>
<td>HISTFILESIZE</td>
<td>Number of lines of command history stored to a file</td>
</tr>
<tr>
<td>HOSTNAME</td>
<td>The hostname of the computer at this time</td>
</tr>
<tr>
<td>PS1</td>
<td>The primary command prompt definition</td>
</tr>
<tr>
<td>SHELOPTSD</td>
<td>Shell options that can be set with the set option</td>
</tr>
<tr>
<td>UID</td>
<td>The UID of the current user</td>
</tr>
</tbody>
</table>

Environment variables are used to pass information from the shell to programs that are being executed. Programs look “in the environment” for particular variables, and if they find the variables, they will use the stored values. Some frequently used environment variables are listed in Table 1.3. Standard Linux OS has two categories of environment variables: global environment variables and local environment variables.

Global Environment Variable

Global environment variables are visible from the shell session and from any subshells. An example of an environment variable is the HOME variable. The value of this variable is the path name of the home directory. To view global environment variables, the `env` or `printenv` command can be used. For instance, type

```
$ printenv
```

This command will display all the environment variables in the system. To display the value of an individual environment variable, only the `printenv` command can be used:

```
$ printenv HOME
```

This command line will display the path name of the home directory.

The `echo` command can also be used to display the value of a variable. However, when the environment variables are referred in this way, a dollar sign ($) needs to be placed before the variable name.

```
$ echo $HOME
```
**Local Environment Variable**

The shell also maintains a set of internal variables known as *local environment variables* that define the shell to work in a particular way. Local environment variables are available only in the shell where they are defined, and are not available to the parent or child shell. Even though they are local, they are as important as global environment variables. Linux systems define standard local environment variables by default. Users can also define their own local variables. There is no specific command to only display the local variables. To view local variables, the *set* command can be used, which displays all variables defined for a specific process, including local and global environment variables and user-defined local variables.

```
$ set
```

The output of the *set* command includes all global environment variables as displayed using the *env* or *printenv* command. The remaining variables are the local environment and user-defined variables.

**Setting Environment Variables**

A local variable can be set by assigning either a numeric or a string value to the variable using the equal sign.

```
$ myVariable=Hello
```

To view the new variable,

```
$ echo $myVariable
```

If the variable value contains spaces, a single or double quotation mark should be used to delineate the beginning and end of the string.

```
$ myVariable="Hello World"
```

The local variables set in the preceding example are available only for use with the current shell process, and are not available in any other child shell. To create a global environment variable that is visible from any child shell processes created by the parent shell process, a local variable needs to be created and then exported to the global environment. This can be done using the *export* command:

```
$ myVariable="Hello World"
$ export myVariable
```

After defining and exporting the local variable “myVariable”, the child shell is able to properly display the variable’s value.

When defining variables, spaces should be avoided among the variable name, the equal sign, and the assigned value. Moreover, in the standard bash shell, all environment variable names use uppercase letters by convention. It is advisable to use lowercase letters for the names of user-defined local variables to avoid the risk of redefining a system environment variable.

To remove an existing environment variable, the *unset* command can be used.

```
$ unset myVariable
```
Setting the PATH Environment Variable

When an external command is entered in the shell CLI, the shell will first search the system to locate the program. The PATH environment variable defines the directories in which the shell will look to find the command that the user entered. If the system returns a message saying “command: Command not found”, this indicates that either the command does not exist on the system or it is simply not in your path. To run a program, the user either needs to directly specify the absolute path of the program, or has to have the directory containing the program in the path.

The PATH environment variables can be displayed by typing:

```bash
$ echo $PATH
```

The individual directories listed in the PATH are separated by colons. The program path (e.g., “/home/aubsxl/linuxDemo”) can be added to the end of the existing path (the $PATH represents this) by issuing the command:

```bash
$ PATH=$PATH:/home/aubsxl/linuxDemo
```

To add this path permanently, add the preceding line to the .bashrc file after the list of other commands.

Basic Linux Commands

A typical Linux command line consists of a command name, followed by options and arguments. For instance,

```bash
$ wc -i FILE
```

The “$” is the prompt from the shell, requesting for the user’s command; “wc” is the name of a command that the shell will locate and execute; “-i” is one of the options that modify the behavior of the command; and “FILE” is an argument specifying the data file that the command wc should read and process. Manual pages can be accessed by using the man command to provide information on the options that a particular command can take, and how each option modifies the behavior of the command. To look up the manual page of the wc command, type

```bash
$ man wc
```

In Linux shell, the [Tab] key is a useful shortcut to complete the names of commands and files. By typing part of the name of a command, filename, or directory, and pressing the [Tab] key, the shell can automatically complete the rest of the name. If more than one command name begins with those typed letters, the shell will beep and prompt the user to type a few more letters before pressing the [Tab] key again.

Here, we introduce a set of the most frequently used Linux commands. For documentation on the full usage of these commands, the readers are referred to the manual pages of each command.

List Directory and File

The ls command is used to list the contents of a directory. By default, ls only lists files whose names do not begin with a dot (.). Files beginning with a dot (.) are known as
hidden files, and they usually contain important program configuration information. To list all files including hidden files, the -a option can be used.

```
$ ls -a
```

This command line will list all contents including hidden files in the current working directory.

```
$ ls -l
```

With the use of the -l option, this command line will list contents in the “long” format, providing additional information on the files.

```
$ ls -t
```

This command will show the files sorted based on the modification time.

**Create Directory and File**

The `mkdir` command is used to create new directories. For instance, to create a directory called “linuxDemo” in the current working directory, type

```
$ mkdir linuxDemo
```

A file can be created using the `touch` command. To create a text file named “linuxDemo.txt” in the current working directory, type

```
$ touch linuxDemo.txt
```

Files can also be created and modified using text file editors such as nano, vi, and vim. To create a file in nano, a simple text editor, type

```
$ nano filename.txt
```

In nano, text can be entered or edited. To write the file out, press the keys [Ctrl] and [O]. To exit the application, press the keys [Ctrl] and [X]. vi and vim are advanced text editors. To create a file using vim, type

```
$ vim linuxDemo.txt
```

vim has two different editing modes: *insert* mode and *command* mode. Insert mode can be initiated by pressing the key [I] to insert text. To return to command mode, press [ESC]. In command mode, press [Shift] and [:] to enter the command. To exit and write out the file, press [Shift] and [:], then type in `wq` and press [Enter] to save. To quit without saving changes, type in: `q!` and press [Enter].

**Change to a Directory**

The `cd` command is used to change from the current working directory to other directories. For instance, to change to the “linuxDemo” directory, type

```
$ cd linuxDemo
```

To find the absolute pathname of current working directory, the `pwd` command can be used, type

```
$ pwd
```
1 Introduction to Linux and Command Line Tools for Bioinformatics

This will print out the absolute pathname of the working directory, for example, 
“/home/aubsxl/linuxDemo”

In Linux, there are several shortcuts for working with directories. For instance, the 
dot (.) represents the current directory, and the double-dot (..) represents the parent 
of the current directory. Home directory can be represented by the tilde character (∼), 
which is often used to specify paths starting at the home directory. For instance, the path 
“/home/aubsxl/linuxDemo” is equivalent to “∼/linuxDemo”:

$ cd .

This will stay in the current directory.

$ cd ..

This will change to one directory level above the current directory.

$ cd ∼

This will go to the home directory. Moreover, typing cd with no argument will also 
lead to the home directory.

$ cd

Manipulate Directory and File

The cp command is used to copy a file/directory.

$ cp file1 file2

This command will make a copy of “file1” in the current working directory and call it 
“file2”.

$ cp file1 file2 myDirectory

This command line will copy “file1” and “file2” to the directory called “myDirectory”.

The mv command can be used to move a file from one place to another. For instance,

$ mv file1 file2 myDirectory

This command line will move, rather than copy (no longer existing in the original 
directory), “file1” and “file2” to the directory called “myDirectory”.

The mv command can also be used to rename a file when used without indications of 
a directory.

$ mv file1 file2

This command line will rename “file1” as “file2”.

The rm command can be used to delete (remove) a file.

$ rm file1

This command will remove the file named “file1”.

To delete (remove) a directory, the rmdir command should be used.

$ rmdir old.dir
Only an empty directory can be removed or deleted by the `rmdir` command. If a directory is not empty, the files within the directory should first be removed.

The `ln` command is used to create links between files.

```bash
$ ln file1 linkName
```

This command line will create a link to "file1" with the name "linkName". If "linkName" is not provided, a link to "file1" is created in the current directory using the name of "file1" as the "linkName". The `ln` command creates hard links by default, and creates symbolic links if the `-s` option is specified.

**Access File Content**

The command `cat` is used to concatenate the files. It can also be used to display the contents of a file on screen. If the file is longer than the size of the window, it will scroll past, making it unreadable. To display long files, the `less` command can be used. The `less` command writes the contents of a file onto the screen, one page at a time. Press the [Space bar] to see the next page, and type [Q] to quit reading. Using `less`, one can search through a text file for a keyword (pattern), by typing forward slash (/) followed by the keyword. For instance, to search through "linuxDemo.txt" for the word "linux", type

```bash
$ less linuxDemo.txt
```

Then, still in `less`, type a forward slash (/) followed by the word to be searched: "/linux". The `less` command will find and highlight the keyword. Type [N] to search for the next occurrence of the word.

The `head` command is used to display the first $N$ lines of the file. By default, it writes the first 10 lines of a file to the screen. With more than one file, it displays contents of each file and precedes each output with a header giving the file name. When using the `-n` option, it prints the first $N$ lines instead of the first 10. With the leading -, it prints all but the last $N$ lines of each file. For instance,

```bash
$ head file1
```

This will print the first 10 lines of "file1".

```bash
$ head -n 50 file1
```

This will print the first 50 lines of "file1".

```bash
$ head -n -50 file1
```

This will print all but the last 50 lines of "file1".

Similarly, the `tail` command is used to write the last $N$ lines of a file. Similar options can be used as those in `head` command.

**Query File Content**

The `sort` command is used to sort the contents of a text file line by line. By default, lines starting with a number will appear before lines starting with a letter; and lines starting with a lowercase letter will appear before lines starting with the same letter in uppercase. The sorting rules can be changed by providing the `-r` option. For instance,

```bash
$ sort months.txt
```
This will sort the file “months.txt” by default sorting rules, based on the first column.

$ sort -r months.txt

This will sort the file in the reverse order, based on the first column.

$ sort -k 2 months.txt

This will sort the file “months.txt” based on the second column.

$ sort -k 2n months.txt

This will sort the file based on the second column by numerical value. By default, the file will be sorted in ascending order; to sort in reverse order, use the \( -r \) option:

$ sort -k 2nr months.txt

The sort can be performed based on multiple lines. To sort the file first based on the third column, and then sort based on the second column in numerical value, type

$ sort -k 3 -k 2n months.txt

The `cut` command is used to select sections of text from each line of files. It can be used to select fields or columns from a line by specifying a delimiter. This command looks for the “tab” delimiter by default; otherwise, the \( -d \) option should be used to define the delimiter. For instance,

$ cut -f1 months.txt

This will cut the first column of the file.

$ cut -f1,2 months.txt

This will cut the first and second columns.

$ cut -f1-3 months.txt

This will cut the first to the third columns.

$ cut -d ' ' -f3 months.txt > seasons

This will cut the third column based on “spaces” as delimiters.

The `uniq` command is used to report and filter out repeated lines in a file. It only detects adjacent repeated lines, and therefore the file usually needs to be sorted before using `uniq`.

$ uniq months.txt

This will print lines with duplicated lines merged to the first occurrence.

$ uniq -c months.txt

This will print out lines prefixed with a number representing how many times they occur, with duplicated lines merged to the first occurrence.

$ uniq -d months.txt

This will only print duplicated lines.

$ uniq -u months.txt

This will only print unique lines.
The `split` command is used to split a file into several. It outputs fixed-sized pieces of input files to files named “PREFIXaa”, “PREFIXab”, etc.

```
$ split myfile.txt
```

This will, by default, split “myfile.txt” into several files, each containing 1000 lines, and prefixed with “x”.

```
$ split -l 2000 myfile.txt myfile
```

This will split “myfile.txt” into several files, each containing 2000 lines, and prefixed with “myfile”.

```
$ split -b 100 myfile.txt new
```

This will split the file “myfile.txt” into separate files called “newaa”, “newab”, “newac”, etc., with each file containing 100 bytes of data.

The `grep` command is one of many standard UNIX utilities that can be used to search files for specified words or patterns. To print out each line containing the word “linux”, type

```
$ grep linux linuxDemo.txt
```

The `grep` command is case sensitive, meaning that it distinguishes between “Linux” and “linux”. To ignore upper/lower case distinctions, use the `-i` option.

```
$ grep -i linux linuxDemo.txt
```

To search for a phrase or pattern, the phrase or pattern should be enclosed in a pair of single quotes. For instance, to search for “Linux system”, type

```
$ grep -i ‘Linux system’ linuxDemo.txt
```

Some of the other frequently used options of `grep` are:

- `-v` to display those lines that do NOT match
- `-n` to precede each matching line with the line number
- `-c` to print only the total count of matched lines

More than one option can be used at a time. To print out the number of lines without the words “linux” and “Linux”, type

```
$ grep -ivc linux linuxDemo.txt
```

The `wc` command can be used to query the file content for word count. To do a word count on “linuxDemo.txt”, type

```
$ wc -w linuxDemo.txt
```

To find out how many lines the file has, type

```
$ wc -l linuxDemo.txt
```

**Edit File Content**

Files can be manually edited using text editors such as nano, vi, and vim. To automatically edit files, `sed`, a stream editor, can be used. `sed` is mostly used to replace text, but
can also be used for many other things. Here, a few examples are provided to illustrate the use of `sed`:

1) **Common usage:** To replace or substitute a string in a file, type

   ```
   $ sed 's/unix/linux/' linuxDemo.txt
   ```

   This command will replace the word “unix” with “linux” in the file. Here, the “s” specifies the substitution operation, and “/” is a delimiter. The word “unix” is the searching pattern, and the word “linux” is the replacement string. By default, `sed` command only replaces the first occurrence of the pattern in each line.

   To replace the \( n \)th occurrence of a pattern in a line, the `/1, /2, ... , /n` flags can be used. For instance, the following command replaces the second occurrence of the word “unix” with “linux” in a line.

   ```
   $ sed 's/unix/linux/2' linuxDemo.txt
   ```

   To replace all the occurrence of the pattern in a line, the substitute flag `/g` (global replacement) can be used. For instance,

   ```
   $ sed 's/unix/linux/g' linuxDemo.txt
   ```

   To replace the text from the \( n \)th occurrence to all the occurrences in a line, the combination of `/1, /2, etc., and /g` can be used. For instance,

   ```
   $ sed 's/unix/linux/3g' linuxDemo.txt
   ```

   This `sed` command will replace the word “unix” with “linux” starting from the third occurrence to all the occurrences.

2) **Replacing on specific lines:** The `sed` command can be restricted to replace the string on a specific line number. An example is

   ```
   $ sed ‘3 s/unix/linux/’ linuxDemo.txt
   ```

   This `sed` command replaces the string only on the third line. To replace the string on several lines, a range of line numbers can be specified. For instance,

   ```
   $ sed ‘1,3 s/unix/linux/’ linuxDemo.txt
   ```

   This `sed` command replaces the lines in the range of 1–3. Another example is

   ```
   $ sed ‘2,$ s/unix/linux/’ linuxDemo.txt
   ```

   This `sed` command replaces the text from the second line to the last line in the file. The “$” indicates the last line in the file.

   To replace only on lines that match a pattern, the pattern can be specified to the `sed` command. If a pattern match occurs, the `sed` command looks for the string to be replaced, and then replaces the string.

   ```
   $ sed ‘/linux/ s/unix/centos/’ linuxDemo.txt
   ```

   This `sed` command will first look for the lines that have the word “linux”, and then replace the word “unix” with “centos” on those lines.

3) **Delete, add, and change lines:** The `sed` command can be used to delete the lines in a file by specifying the line number, or a range of line numbers. For instance,
$ sed '2 d' linuxDemo.txt
This command will delete the second line.

$ sed '5,$ d' linuxDemo.txt
This command will delete lines starting from the fifth line to the end of the file.

To add a line after line(s) in which a pattern match is found, the “a” command can be used. For instance,

$ sed '/unix/ a "Add a new line"' linuxDemo.txt
This command will add the string “Add a new line” after each line containing the word “unix”.

Similarly, using the “i” command, the sed command can add a new line before a pattern match is found.

$ sed '/unix/ i "Add a new line"' linuxDemo.txt
This command will add the string “Add a new line” before each line containing the word “unix”.

The sed command can be used to replace an entire line with a new line using the “c” command.

$ sed '/unix/ c "Change line"' linuxDemo.txt
This sed command will replace each line containing the word “unix” with the string “Change line”.

4) Run multiple sed commands: To run multiple sed commands, the output of one sed command can be piped as input to another sed command.

$ sed 's/unix/linux/' linuxDemo.txt | sed 's/os/system/'
This command line will first replace the word “unix” with “linux”, and then replace the word “os” with “system”. Alternatively, sed provides the -e option to run multiple sed commands. The preceding output can be achieved in a single sed command, as shown in the following:

$ sed -e 's/unix/linux/' -e 's/os/system/' linuxDemo.txt

Redirect Content

Most processes initiated by Linux commands take their input from the standard input (the keyboard) and write to the standard output (the terminal screen). By default, the processes write their error messages to the terminal screen. In Linux, both the input and output of commands can be redirected, using > to redirect the standard output into a file, and using < to redirect the input file. For instance, to create a file named “fish.names” that contains a list of fish names, type

$ cat > fish.names

Then type in the names of some fish. Press [Enter] after each one.

catfish
zebrafish
1 Introduction to Linux and Command Line Tools for Bioinformatics

In this process, the `cat` command reads the standard input (the keyboard) and redirects (`>`) the output into a file called “fish.names”. To read the contents of the file, type

```
$ cat fish.names
```

The form `>>` appends standard output to a file. To add more items to the file “fish.names”, type

```
$ cat >> fish.names
```

Then type in the names of more fish

**seabass**
**croaker**

`^D ([Ctrl] and [D] to stop)`

The redirect `>` is often used with the cat command to join (concatenate) files. For instance, to join “file1” and “file2” into a new file called “file3”, type

```
$ cat list1 list2 > file3
```

This command line will read the contents of “file1” and “file2” sequentially, and then output the text to the file “file3”.

Similarly, the redirects apply to other commands. For instance,

```
$ sed -e ‘s/unix/linux/’ -e ‘s/os/system/’ linuxDemo.txt > linuxDemo_edit.txt
```

This command line will perform substitutions, and output to the new file “linux Demo_edit.txt” instead of the terminal screen.

The `pipe (|)` is used to redirect the output of one command as the input of another command. For instance, to find out how many users are logged on, type

```
$ who | wc -l
```

The output of the `who` command is redirected as the input of the `wc` command. Similarly, to find out how many files are present in the directory, type

```
$ ls | wc -l
```

The output of the `ls` command is redirected as the input of the `wc` command.

**Compare File Content**

The `diff` command compares the contents of two files and displays the differences. Suppose we have a file called “file1”, and its updated version named “file2”. To find the differences between the two files, type

```
$ diff file1 file2
```
In the output, the lines beginning with < denotes “file1”, while lines beginning with > denotes “file2”.

The \texttt{comm} command is used to compare two sorted files line-by-line. To compare sorted files “file1” and “file2”, type

\begin{verbatim}
$ comm file1 file2
\end{verbatim}

With no options, \texttt{comm} produces a three-column output. The first column contains lines unique to “file1”, the second column contains lines unique to “file2”, and the third column contains lines common to both files. Each of these columns can be suppressed individually with options.

\begin{verbatim}
$ comm -3 file1 file2
\end{verbatim}

This command line will show the lines in both files.

\begin{verbatim}
$ comm -1 file1 file2
\end{verbatim}

This command line will show the lines only in “file1”.

\begin{verbatim}
$ comm -2 file1 file2
\end{verbatim}

This command line will show the lines only in “file2”.

\textbf{Compress and Archive Files and Directories}

1) \texttt{zip} is a compression tool that is available on most OSs such as Linux/UNIX, macOS, and Microsoft Windows. To zip individual files (e.g., “file1” and “file2”) into a zip archive, type

\begin{verbatim}
$ zip abc.zip file1 file2
\end{verbatim}

To extract files from a zip folder, use \texttt{unzip}

\begin{verbatim}
$ unzip abc.zip
\end{verbatim}

To extract to a specific directory, use the -d option.

\begin{verbatim}
$ unzip abc.zip -d /tmp
\end{verbatim}

2) The \texttt{gzip} command can be used to archive and compress files. For example, to compress “linuxDemo.txt”, type

\begin{verbatim}
$ gzip linuxDemo.txt
\end{verbatim}

This will compress the file and place it in a file called “linuxDemo.txt.gz”. To decompress files created by \texttt{gzip}, use the \texttt{gunzip} command.

\begin{verbatim}
$ gunzip linuxDemo.txt.gz
\end{verbatim}

3) \texttt{bzip2} compresses and decompresses files with a high rate of compression together with reasonably fast speed. Most files can be compressed to a smaller file size with \texttt{bzip2} than with the more traditional \texttt{gzip} and \texttt{zip} programs. \texttt{bzip2} can be used without any options. Any number of files can be compressed simultaneously by merely listing their names as arguments. For instance, to compress the three files named “file1”, “file2”, and “file3”, type

\begin{verbatim}
$ bzip2 file1 file2 file3
\end{verbatim}
Table 1.4  A list of frequently used tar options.

<table>
<thead>
<tr>
<th>Options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-A</td>
<td>Append tar files to an archive</td>
</tr>
<tr>
<td>-c</td>
<td>Create a new archive</td>
</tr>
<tr>
<td>-d</td>
<td>Find differences between archive and file system</td>
</tr>
<tr>
<td>-r</td>
<td>Append files to the end of an archive</td>
</tr>
<tr>
<td>-t</td>
<td>List the contents of an archive</td>
</tr>
<tr>
<td>-u</td>
<td>Only append files that are newer than those existing in archive</td>
</tr>
<tr>
<td>-x</td>
<td>Extract files from an archive</td>
</tr>
<tr>
<td>-f</td>
<td>Use archive file or device F (default &quot;-&quot;, meaning stdin/stdout)</td>
</tr>
<tr>
<td>-j</td>
<td>Use to decompress .bz2 files</td>
</tr>
<tr>
<td>-v</td>
<td>Verbosely list files processed</td>
</tr>
<tr>
<td>-z</td>
<td>Use to decompress .gz files</td>
</tr>
</tbody>
</table>

$ bzip2 file1 file2 file3

bunzip2 (or bzip2 -d) decompresses all specified files. Files that are not created by bzip2 will be detected and ignored, and a warning will be issued.

$ bunzip2 abc.tar.bz2

4) tar is an archiving program designed to store and extract files from an archive file known as a tarfile. The first argument to tar must be one of the options A, c, d, r, t, u, x (Table 1.4), followed by any optional functions. The final arguments to tar are the names of the files or directories that should be archived.

To create a tar archive named “abc.tar” by compressing three files, type

$ tar -cvf abc.tar file1 file2 file3

To create a gzipped tar archive named “abc.tar.gz” by compressing three files, type

$ tar -czvf abc.tar.gz file1 file2 file3

To extract files from the tar archive “abc.tar”, type

$ tar -xvf abc.tar

To extract files from the tar archive “abc.tar.gz”, type

$ tar -xvzf abc.tar.gz

Access Remote Files

Two programs (wget and curl) are widely used to retrieve files from websites via the command-line interface. For instance, to download the BLAST program “ncbi-blast-2.2.31 -x64-linux.tar.gz” from NCBI ftp site using curl, type the following:

$ curl ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/LATEST/ncbi-blast-2.2.31+-x64-linux.tar.gz > ncbi-blast-2.2.31+-x64-linux.tar.gz
Alternatively, this can be done using `wget` as following:

```
```

In addition, the program `scp` (e.g., secure copy) can be used to copy files in a secure fashion between UNIX/Linux computers, as following:

To send a file to a remote computer,

```
$ scp file1 aubsxl@dmcs.asc.edu:/home/aubsxl/linuxDemo
```

To retrieve a file from a remote computer,

```
$ scp aubsxl@dmcs.asc.edu:/home/aubsxl/linuxDemo/file1
```

### Check Process and Job

A process is an executing program identified by a unique PID (process identifier). The `ps` command provides a report of the current processes. To see information about the processes with their associated PIDs and status, type

```
$ ps
```

The `top` command provides an ongoing look at processor activity in real time. It displays a list of the most CPU-intensive processes on the system, and can provide an interactive interface for manipulating processes. It can sort the tasks by CPU usage, memory usage, and runtime. To display top CPU processes, type

```
$ top
```

A process may be in the foreground, in the background, or suspended. In general, the shell does not return the Linux prompt until the current process has finished executing. Some processes take a long time to run and hold up the terminal. Backgrounding a long process allows for the immediate return of the Linux prompt, enabling other tasks to be carried out while the original process continues executing. To background a process, type an `&` at the end of the command line. The `&` runs the job in the background and returns the prompt straight away, allowing the user to run other programs while waiting for that process to finish. Backgrounding is useful for jobs that will take a long time to complete.

When a process is running, backgrounded, or suspended, it will be entered into a list along with a job number. To examine this list, type

```
$ jobs
```

To restart (foreground) a suspended processes, type

```
$ fg jobnumber
```

For instance, to restart the first job, type

```
$ fg 1
```

Typing `fg` with no job number will foreground the last suspended process.
To kill a job running in the foreground, type \(^{\text{C}}\) (\([\text{Ctrl}]\) and \([\text{C}]\)). To kill a suspended or background process, type

\[
$ \text{kill jobnumber}
\]

**Other Useful Command Lines**

**quota**
The *quota* command is used to check current quota and how much of it has been used.

\[
$ \text{quota -v}
\]

**df**
The *df* command reports on the space left on the file system. To find out how much space is left on the current file system, type

\[
$ \text{df}
\]

**du**
The *du* command outputs the number of kilobytes used by each subdirectory. It is useful to find out which directory takes up the most space. In the directory, type

\[
$ \text{du -s *}
\]

The \(-s\) flag will display only a summary (total size), and the \(*\) indicates all files and directories.

**free**
The *free* command displays information on the available random-access memory (RAM) in a Linux machine. To display the RAM details, type

\[
$ \text{free}
\]

**zcat**
The *zcat* command can read gzipped files without decompression. For instance, to read the gzipped file “abc.txt.gz”, type

\[
$ \text{zcat abc.txt.gz}
\]

For text with large size, the *zcat* output can be piped through the *less* command.

\[
$ \text{zcat abc.txt.gz | less}
\]

**file**
The *file* command classifies the named files according to the type of data, such as text, pictures, and compressed data. To report on all files in the home directory, type

\[
$ \text{file *}
\]

**find**
The *find* command searches through the directories for files and directories with a given name, date, size, or any other specified attribute. This is different from *grep*, which finds contents within files. To use *find* to search for all files with the extension
of “.txt”, starting at the current directory (.) and working through all sub-directories, and then to print the name of the file to the screen, type

$ find . -name "*.txt" -print

To find files over 1 MB in size, and to display the result as a long listing, type

$ find . -size +1M -ls

history

The history command can display a list of commands the user has typed in. Each command is given a number according to the order it is entered. To repeat a command, the user can either use the cursor keys to scroll up and down the list or type history for a list of previous commands.

$ history

Getting Help

There are manuals within the Linux system that give detailed information about most commands. The manual pages tell which options a particular command can take, and how each option modifies the behavior of the command. The man command can be used to read the manual page for a particular command. For instance, to find more information about the wc (word count) command, type

$ man wc

Alternatively, the whatis command can be used; type

$ whatis wc

This command line will give a one-line description of the command, but omit any information about options, etc.

When the user is not sure of the exact name of a command, the apropos command can be used.

$ apropos copy

This command line will list all the commands with the keyword “copy” in their manual page headers.

Installing Software Packages

Although each OS is preinstalled with a large number of commonly used commands, specific programs often need to be installed to perform specific tasks. On Microsoft Windows systems, every program has a simple “Setup.exe” or “program.zip” file. The installation can be done by simply clicking the “Setup.exe” file, which is followed by on-screen instructions. In the world of bioinformatics, software packages are commonly distributed in the form of either precompiled executables or source codes. To install precompiled executables, they can be simply put to the environment path, while software most often needs to be installed from the source code in which the compiling process is required.
Installing Packages from a Configured Repository

The standard Linux package format is RPM. The RPM packaging system was originally developed by Red Hat and is widely used in the Linux community. Distributions using it include Fedora, Mandriva, Red Hat (naturally), and SUSE. An RPM package file is normally named in the pattern “program-version-other.rpm”. Another popular package format is DEB, the Debian software package. Debian packages and the Advanced Packaging Tool (APT) provide several advanced features that are now commonly used, such as automatic dependency resolution and signed packages. Debian packages are used by Debian GNU/Linux and distributions based on it, including Ubuntu, Knoppix, and Mepis. A Debian package file is normally named in the pattern “program-version-other.deb”.

To install software from the configured repository, the user must become a SuperUser. A broad array of tools is available to work with DEB packages, and `apt-get` is commonly used. The use of `apt-get` is straightforward because it not only keeps track of what packages are installed, but also what other packages are available. To install packages using `apt-get`, type

$ sudo apt-get install packagename

To remove the software, type

$ sudo apt-get remove packagename

`yum` does for RPM packages roughly what `apt-get` does for Debian packages. As with `apt-get`, `yum` can download and install packages from a configured repository.

$ sudo yum install packagename

To remove the software, type

$ sudo yum remove packagename

Installing Software from Source Code

Software packages delivered in tarballs are mostly in source code, which must be compiled before installation. A number of steps are required to install the software package from source code: (1) locate and download the source code, which is usually compressed; (2) unpack the source code; (3) compile the code; (4) install the resulting executable; and (5) set paths to the installation directory. Of these steps, the most difficult is the compilation part.

Compiling source code is the process that converts high-level human-readable language code into a form that the computer can understand. For instance, C language source code is converted into a lower-level language called assembly language. The assembly language code is then further converted into object code, which the computer can directly understand. The final stage in compiling a program involves linking the object code to code libraries that contain certain built-in functions. The final stage produces an executable program.

As the number of UNIX-based OSs increases, it becomes difficult to write programs that could run on all derivative systems. The characteristics of some systems change from version to version, and developers frequently do not have access to every system.
A number of utilities and tools have been developed for programmers and end users to conduct compiling steps. The GNU configure and build system simplifies the building of programs distributed as source code. All programs are built using a simple, standardized, two-step process. The program builder does not need to install any special tools in order to build the program. Therefore, building a program is normally as simple as running configure followed by make.

The configure command is a shell script that is used to check the details of the system in which a software is going to be installed. This script checks for dependences required by the particular software to work properly in the system. If any of the major requirements is missing, the configure script exits, and installation is not able to proceed until those required dependences are installed. The configure script supports a wide variety of options. The --help option can be used to get a list of options for a particular configure script. Two frequently used generic options are the --prefix and --exec-prefix, which are used to specify the installation directories. The directory defined by the --prefix option holds machine-independent files such as documentation, data, and configuration files. The directory defined by the --exec-prefix option is normally a subdirectory of the --prefix directory, which holds machine-dependent files such as executables. The main job of the configure script is to create a “Makefile”, which contains various steps that need to be taken when compiling the software, depending on the results of checking performed by the configure script.

The make program is a utility available on almost all UNIX systems. It depends on the Makefile, which instructs on how to compile the software and where to install the finished compiled binaries (executables), manual pages, data files, dependent library files, configuration files, etc.

**Compiling a Package**

To install a package, one needs to carefully read the README and INSTALL text files, which contain important information on how to compile and run the software. Generally, the shell commands ./configure; make; and make install should configure, build, and install the package, respectively.

The simplest way to compile a package is:

1) cd to the directory containing the package’s source code. Type ./configure to configure the package for values of system-dependent variables. If configure has run correctly, it will create a Makefile with all the necessary options.

2) Type make to compile the package. After this, the executables will be created. To check if everything is compiled successfully, type make check to run any self-tests that come with the package, generally using the just-built uninstalled binaries.

3) Type make install to install the programs and any data files and documentation. By default, make install installs the package’s commands under “/usr/local/bin”, and includes files under “/usr/local/include”. The installation prefix can be specified other than “/usr/local” by giving configure the option --prefix = PREFIX, where PREFIX must be an absolute file name. Separate installation prefixes can be specified for architecture-specific files and architecture-independent files. If the option --exec-prefix = PREFIX is passed to configure, the package
uses PREFIX as the prefix for installing programs and libraries. Documentation and other data files still use the regular prefix.

4) Optionally, to remove the program binaries and object files from the source code directory, the `make clean` command can be executed.

### Accessing a Remote Linux Supercomputer System

Many ways are available for users to access a Linux system, such as installing Linux on a personal computer, running a Linux virtual machine, and using a live CD to run a Linux system. However, to run most bioinformatics applications, users need to access remote Linux machines, such as supercomputer clusters that provide much larger computing resources.

To gain access to a remote Linux-based system, a user name, password, and hostname (or IP address) are required. Once the account information is available, remote access can be done from Linux, macOS, and Microsoft Windows systems.

#### Access Remote Linux from Local Linux System

To connect to a remote UNIX/Linux computer securely, the program `ssh` (e.g., `secure shell`) can be used as below:

```
$ ssh user@hostname
```

The “user name” and “host name” need to be replaced with your user name and machine name before running this command. After running this command, a prompt will show up to request for the password of the account that is being connected to. After typing in the password and pressing [Enter], the remote computer is accessed via the CLI. Once done, all operations can be performed as if you were sitting in front of the supercomputer. The disconnection can be done by typing “exit” and pressing [Enter]. It should be noted that `ssh` will only work if “Remote Login and File Sharing” is enabled on the computer that is being connected to.

#### Access Remote Linux from macOS

macOS includes an application called “Terminal”, which is located in the “Applications Utilities” folder. As in the Linux system, the remote Linux computer can be connected by using Terminal in macOS. To access a remote Linux system, launch Terminal and type

```
$ ssh user@hostname
```

Replacing the “user” and “hostname” with your user name and machine name, press [Enter], then type in the password to establish the connection.

#### Access Remote Linux from Microsoft Windows

On Microsoft Windows systems, a variety of third-party tools can be used to connect to a remote Linux system. One of the popularly used tool is PuTTY, which is a free program and can be installed by downloading the executable from the PuTTY website (http://www.chiark.greenend.org.uk/~sgtatham/putty/). Launching PuTTY will open a
configuration window. Click “Session” in the left pane, and then enter user@hostname in the text box “Host Name (or IP address)”, replacing the “user” and “hostname” with your user name and machine name. Click “Open” to establish a connection with the remote Linux system.

**Demonstration of Command Lines**

Here, we discuss a number of frequently used Linux commands to illustrate how to use Linux command lines and how to install Linux programs from scratch using source code. The introduction and usage of BLAST programs are not detailed here, and are provided in Chapter 2.

The step-by-step demonstration of command lines:

1) Login the remote Linux account using ssh

   ```
   $ ssh aubsxl@dmc.asc.edu
   ```

2) Create a directory named “linuxDemo”

   ```
   $ mkdir linuxDemo
   ```

3) Go to the directory “linuxDemo”

   ```
   $ cd linuxDemo
   ```

4) Create a subdirectory named “blast”

   ```
   $ mkdir blast
   ```

5) Download the BLAST source code from the NCBI FTP site

   ```
   ```

6) Decompress the tarball package

   ```
   $ tar -xvzf ncbi-blast-2.2.31+-src.tar.gz
   ```

7) Go to the new folder (“ncbi-blast-2.2.31+-src”) just created after decompression

   ```
   $ cd ncbi-blast-2.2.31+-src
   ```

8) Go to subdirectory c++

   ```
   $ cd c++
   ```

9) Configure the software package

   ```
   $ ./configure --prefix=/home/aubsxl/blast
   ```

10) Compile the software package

    ```
    $ make
    ```

11) Install the binary executables to the defined directory

    ```
    $ make install
    ```
12) Attach the path of compiled BLAST executables to the environment path

```bash
$ echo ‘export PATH=$PATH:/home/aubsxl/linuxDemo/blast/bin’ >> ~/.bashrc
```

Further Reading

A large number of useful books and web tutorials are available for beginners to learn UNIX/Linux. Some of the excellent ones are listed here:


5) UNIX Tutorial for Beginners: [http://www.ee.surrey.ac.uk/Teaching/Unix/](http://www.ee.surrey.ac.uk/Teaching/Unix/).