University of Reading
Department of Computer Science

Julian Kunkel, Lucille L. Blumire, Oliver Barnwell

Tutorial Sheet 2
CS1PR / Autumn 2019
95 Minutes, 0 Marks Total
Date: 2019-10-07

Learning Objectives

The learning objectives in the tutorial are of practical nature and aim to provide tools for you.

Tools

- Comprehending the structure of the exercises
- Utilizing the basic Linux user interfaces to run applications
- Understanding key features of the Atom editor
- Applying GIT for effective management of your learning experience (portfolio)
- Utilizing basic shell commands to navigate and manipulate the directory tree
- Creating Bash scripts to execute a sequence of operations
- Creating simple flow graphs using LibreOffice

Theoretic

- Describing the outcome of a selected algorithm
- Converting algorithmic behaviour into flow graphs

Contents

Please now reboot the machine, during the boot process, a window pops up that allows booting into Linux which we will use for the tutorial and exercises. Note that this tutorial schedule is tight, you should be able to do most of it at the end of the session.

1. Tutorial (5 Minutes): Organization of the exercises and tutorial

As part of this tutorial, we will quickly walk over the organisational aspects of the exercises and the tutorials. The description below serves the purpose to document it.

Role of the tutorial/exercise Firstly, the tutorial equips you with additional knowledge and tools and teaches you how to use them effectively to complete the exercises. When teaching tools, the lecturer may give you an overview and then will walk through an interactive guide to overcome the typical obstacles faced by newcomers. Additionally, there is typically some time for you playing with the tools and performing some tasks – you may also use this time to discuss issues with your colleagues. By providing further details, the lecture will be augmented. Part of the tutorial sheet will be links to aid your independent study of specific topics.
As part of the tutorials, we will also employ *active learning* in the form of group work. Some group work is used to augment the exercises further, for instance, to obtain feedback from your peers and reflect on your skills. Such group work typically is awarded with additional marks as indicated in the submission. The assessment criteria for these marks are listed within the task. The number of marks that can be obtained in the tutorial is indicated on the sheet header of the tutorial.

Portfolio The portfolio is a version-controlled repository that contains all your generated artefacts from the tutorials and exercises together with comments from the marking. This tutorial sheet describes how the portfolio is organised – details were also given in the first tutorial session. You will also be able to provide feedback regarding the exercises (e.g., time used) that help us to optimise the activities. We utilize the popular Git[1] version control system and store the output on *http://csgitlab.reading.ac.uk/*. In the tutorial, you should have created your own private git repository there, e.g., calling it CS1PR, and made it writable to the user *di918039*.
Do not share the repository with anyone else!

In each week of the tutorial/exercises, create a directory *week<Number>* that contains in one subdirectory for each tutorial/exercise with all the necessary artefacts (code, report, diagrams) as defined in the submission description of the task.
Also, please include a feedback file `fb.txt` that contains the approximate time you have to spend (in minutes) for the individual exercises and additional (optional) comments. You can find an example here: *https://csgitlab.reading.ac.uk/di918039/cs1pr-portfolio/blob/master/fb.txt*

An **example portfolio** is available at: *https://csgitlab.reading.ac.uk/di918039/cs1pr-portfolio*

Submission You need to commit the files to your CS-Git repository and use *https://hps.vi4io.org/cs1pr-submit* to upload submission. The web portal will perform some checks, and it also allows you to verify your submission. You can see which exercises you submitted and which files will be generated (compiled) correctly. The deadline for the submission is Monday 09:00 just before the exercises. We recommend that you submit at latest on Sunday evening.

Marking Each exercise is worth about 2% of the overall CS1PR module. Most of the programming exercises are marked automatically by utilizing test cases. After the deadline, the system will use the latest submitted version and **generate** a feedback file (*fb.txt*) in each exercise directory indicating the marks you have achieved and giving you feedback in case your program did not work as expected. Then the **updated repository is pushed** to your CS-Gitlab portfolio.
It is therefore mandatory that your program follows the **expected input/output behaviour** as described in each exercise.

[1] *https://git-scm.com/*

Some exercises offer an **easy and hard version**. You can choose either of them to gain the same marks for the exercise. More experienced programmers are encouraged to attempt some of the hard versions in each exercise sheet – but keep the prescribed *time budget* in mind. You can also submit both, and you will receive the better mark: $mark = max(easy, hard)$.

Time management They should be designed to take about 5 hours per week of an independent study to complete. For your better planning, each mark indicated on an exercise should correspond to roughly 10 minutes of your time – this is considered to be the **time budget** for resolving the task. This time not only covers the coding and preparation of your portfolio for submission but also to resit the lecture and conduct further reading as needed to resolve the exercise.
We advise to watch the time budget – if you are struggling with it, reach out to your peers for help at *http://cs-reading.slack.com*. One strategy could be to invest a quarter of the time and see if you make sufficient progress to finish in time. If you haven't achieved anything, it is time to seek for help and attempt the next task.
Note that as we run entirely new exercises, the timing may need some adjustment. Reaching out in time is essential, and it should allow you to stick close to the time budget!

Independent learning and the academic code of honour The exercises are designed to **maximize your learning experience**. **Independent learning** is crucial for your success in the university.
You must attempt to work on the exercises yourself (but keep the prescribed time budget in mind). It is also advised that you collaborate in our learning community, and, e.g., schedule meetings with peers to discuss and perform the exercises.

To conclude, the code of honour is:

- We promise to support your learning experience, for example with the **master class** and provide you with some feedback on Slack.
- You plan and invest around 5 hours per week to resolve the exercises.
- We promise that the exercises and tutorials will prepare you excellently for the exam.
- Any artefact in the portfolio generated by you and any contribution to it should be attributed and referenced properly (even mentioning if you have teamed-up with colleagues initially to grasp the concepts).
- You should be at anytime able to explain any artefact in your portfolio to us. If it is apparent that it is plagiarised, **the whole exercise sheet will be marked with 0 marks!**. We want to trust in you!

Further Reading

- Tips for time management: *https://www.topuniversities.com/blog/7-time-management-tips-students*
- UoR student charter: *https://www.reading.ac.uk/web/files/welcome/Student_Charter_2015.pdf*
- UoR guide to newcomers: *https://libguides.reading.ac.uk/freshers/studying*
- UoR policy of **Academic Misconduct** *https://www.reading.ac.uk/exa-misconduct.aspx*
- UoR wellbeing *https://student.reading.ac.uk/essentials/_support-and-wellbeing.aspx*

2. Tutorial (10 Minutes): Linux Environment

As part of this tutorial, we login to Linux and perform some basic operations. Particularly, starting some applications, like a browser, the `gnome-shell`, and the `atom` editor.

Steps

1. Login to Linux using your University credentials
2. Open the Atom editor
3. Follow the *Atom Basics* link referenced below in **Further Reading**
4. Search for how to use *Multiple Cursors and Selections* features
5. Use Atom to save/load a file/directory, edit a file, finding something in the project
6. Now use the remaining time to play a bit with the editor and save linux.txt with a short note.
7. Find `gnome-shell` in your applications and start it.
8. Pin the shell to your favourites.

Further Reading

- Ubuntu: *https://help.ubuntu.com/stable/ubuntu-help/getting-started.html.en*
- Atom Basics: *https://flight-manual.atom.io/getting-started/sections/atom-basics/*
- BCS Open Source Specialist Group: *https://ossg.bcs.org/about/*

Hints

Useful Atom commands to look up:

- Search in file (CTRL-F)
- Search in project (CTRL-Shift-F)
- Adding another cursor (Shift-Left click)
- Saving (CTRL-S)
- Converting the current selection to a search with multi-cursor (ALT-F3)
- Toggle line comments (CTRL-/)

3. Tutorial (10 Minutes): Shell Navigation

In this task, you will learn the basic Linux commands for navigating the Unix shell. At its core, the shell is simply a convenient tool which you can use to execute commands on a Linux computer. The shell provides a textual interface allowing to interact with the system, i.e., accessing and manipulating files and running programs. It also allows you to write programs (known as shell scripts) which combine these commands with more complex behaviour.
Despite initially seeming cumbersome and inefficient to many, the real power of shell scripting quickly becomes apparent to those who use it. An everyday use is to automate repetitive tasks which would otherwise be time-consuming to complete.

> *A Unix shell is both a command interpreter and a programming language. As a command interpreter, the shell provides the user interface to the rich set of GNU utilities. The programming language features allow these utilities to be combined.*[2]

To launch the shell simply start `gnome-shell` in the same way, you did in the previous tutorial step, refer to the *Hints* section for a useful keyboard shortcut.

[2] *https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html*

3.1 Shell Commands - using the shell as a command interpreter

When you first open a shell, you are presented with a blank command prompt which accepts any input entered via the keyboard. For example:

```
1 $ pwd
```

[3] This prints the **working directory**, i.e., in which directory you are right now! All file names are relative to this working directory.
Standard programs provide a manual and a short help for their usage. To show the help for pwd, use:

```
1 $ pwd --help
```

3.2 File system namespace – the directory organization

Linux follows the philosophy that everything is a file. Directories are indicated with a "/" separator. The root is the directory "/" and different storage devices are linked into this tree (underneath /mnt or the /media directories). Files or directories can be referred to either using absolute or relative file names.
There is an organizational structure in the directories according to the Filesystem Hierarchy Standard.
While you can use files with whitespaces (e.g., "this is a file.txt"), it is not advised as it makes execution in the shell more complicated and error-prone as names can be confused as arguments (in our example, "this" "is" "a" "file.txt") to a command.

Absolute filenames start with a "/" and need to contain the full path to the file.
The tilde character ~is used to refer to your home directory, e.g., you can use `$ ls ~`.

Relative filenames are relative in regards to the working directory. A dot "." indicates the current directory, e.g., `$ ls .` shows the files in the current working directory. ".." means the parent directory, thus, `$ ls ../` prints the files of the parent directory.

3.3 Useful commands

Next, we're going to look at a few commands that are essential when it comes to navigating in the Linux shell. These commands are all simple programs that come preinstalled on every Linux distribution[4], the shell allows you, the user, to execute these programs.

- `$ help <COMMAND>` - provides help for builtin commands such as `help`
  - `$ help help`
- `man` - Opens the manual page for a given command, you can use it for any subsequent command
  1. `$ man man`
     Inside a manpage you can use "/" to search for a phrase.
  2. `$ man pwd`
- `pwd` - Print working directory
  1. `$ pwd`
- `ls` - Lists files and subdirectories in the current/provided directory
  1. `$ ls`
  2. `$ ls /bin` – list all files inside /bin
  3. `$ ls -lah` – list details of the files

[3] Note: In our tutorials when giving an example of command execution we will begin each line with a dollar: $ character to represent that the command is to be executed.
[4] these core utility programs are usually located in the **/bin** directory. If you're interested, run the command `ls /bin` to see its contents

- `cd` - Changes location, allows you to switch directory.
    1. `$ cd /`
- `cat` - Reads a file and writes it to the standard output
    1. `$ cat filename`
- `mv` - Moves a file or directory to the specified location
    1. `$ mv src dest`
- `touch` - Creates a file with a given name
    1. `$ touch filename`
- `cp` - Copies a file or directory to the specified location.
    1. `$ cp src dest`
- `rm` - Removes a file or directory
    1. `$ rm filename`
    2. `$ rm -rf directory`
- `echo` - Displays a line of text/variable
    1. `$ echo ‘‘hello world’’`, writes the string `‘‘hello world’’` to stdout.
    2. `$ echo $myvar`, writes the value of the `$myvar` variable to stdout.
- `grep` - Returns strings that match a given pattern when compared to standard input (used to find if a file contains certain text for example)
    1. `$ grep ‘‘test’’ myfile`, searches file **myfile** for the string **“test”**
    2. `$ grep -r ‘‘test’’`, recursively searches all files below current directory for string **“test”**
- `read` - Reads a line from standard input
    1. `$ read input`, reads a line of user input into the variable `$input`
    2. `$ echo $input`, prints the variable input

3.4 Basic Syntax

The general structure of a shell command follows this format:

```
$ COMMAND <-OPTIONS> <ARGUMENTS>
```

1. **command** - the name of the command to execute
    a) Typically contains lower/uppercase letters and digits.
    b) Typically is 2-9 charaters in length
2. **option(s)** - one or more options to pass to the command
    a) Options are mostly one character in length (e.g., -h), long options maybe longer --help
    b) All options must begin with a hyphen
    c) All options must come before additional arguments
    d) Double hyphens (`--`) can be used to indicate the end of the option list
3. **argument(s)** - one or more arguments to pass to the command (e.g., filenames or input strings)

Steps

1. Open the gnome-shell
2. Familiarize yourself with the usage of man-pages: `$ man man`
3. Print your current working directory: `$ pwd`
4. Show the options of pwd: `$ pwd --help`
5. Open the manpage of pwd: `$ man pwd`
6. Show all files in the current working directory: `$ ls -lah`
7. Create a directory named `testdir`: `$ mkdir testdir`
8. Change your working directory to the newly created one: `$ cd testdir` or `$ cd ./testdir`
9. Output the current working directory once more. (see above)
10. Create an empty file named `testfile`: `$ touch testfile`
11. Open the file with atom: `$ atom testfile`
12. Rename the file to `renamed`: `$ mv testfile renamed`
13. Copy the renamed file to `copied`: `$ man cp` (familiarize yourself with it)
14. Delete the file `testfile2`: `$ man rm` (familiarize yourself with it)
15. Run `$ echo ~` and check the result!
16. Change back to your working directory: either use `$ cd ~` or `$ cd ..`

Hints

- Shortcut to start `gnome-terminal`: (CTRL-ALT-t)
- The up/down arrows navigate the history of commands
- CTRL-r allows searching in the history
- Use the left/right arrows (Pos1/End keys) to change the position of the text cursor
- The tab key autocompletes commands when youre halfway through typing them - it saves time!
- The middle mouse key copies previously marked text into the shell

Further Reading

- Commands: *https://maker.pro/linux/tutorial/basic-linux-commands-for-beginners*
- Command line structure: *https://www2.cs.duke.edu/csl/docs/unix_course/intro-14.html*
- *https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard*

4. Tutorial (10 Minutes): Shell Scripts and Redirection

The Linux shell also allows executing scripts, i.e., a small program that is designed to be executed by the command line interpreter (our shell). Hence, it provides a means for automatization. These scripts are useful when carrying out some simple (and sometimes intermediate) operations.
The shell provides a convenient way to write scripts as multi-line files. We simply create a new file using a text editor such as *Atom* and ensure that the first line of our script contains the line before the program code:

`myscript.sh`

```
#!/bin/bash
read input
message=''this is an example of a multiline bash program''
echo $message
echo ''User input: $input''
```

The first line tells the shell which program to run to execute this script. The `#!` is known as a `shebang or hashbang` in Linux terminology. It merely means that, when the file is executed, the `program in /bin/bash` is used for execution.
In order to make the script `myscript.sh` runnable we must set the **executable bit**. This is achieved using the `chmod` command with the `+x` option:

```
$ chmod +x myscript.sh
```

Then we can run the script using: `$ ./myscript.sh`

4.1 Standard Input and Output, Redirects and Chaining- making use of program output

We have seen how Bash can be used to execute individual programs which serve a variety of useful purposes, allowing us (as users) to navigate the file system, manipulate files, and read the contents of files.
There are many cases where we might wish to do more with the output of these programs than simply observe the result. Say for example we were tasked with making a list of files and retrieving only those that end in `.mp4` (video files).
A simple way of achieving this might be to write a custom program in a programming language such as C. While this is a valid (and tempting) solution, it very quickly becomes a time-consuming prospect when faced with more substantial tasks.
Luckily, the Bash shell provides an elegant solution to this problem: it introduces the concept of a standard "stream". These streams are channels in which every program writes to as output (known as **stdout**) and reads from as input (known as **stdin**). This powerful concept is also referred to as **chaining**. When all terminal programs are written using this convention, we can simply plug them together like pipes into a 'pipeline', connecting the output of one program to the input of another and so on.

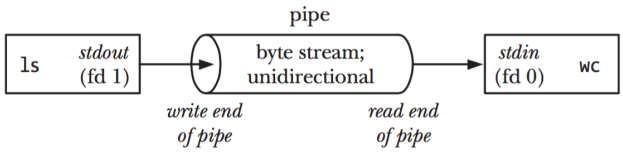

Figure 44-1: Using a pipe to connect two processes

Figure 1: Illustration of how the ouput of one process can be passed to the input of another.Source: The Linux Programming Interface: A Linux and UNIX System Programming Handbook

Pipes provide a simple one-way communication channel `(stdout -> stdin)` betwen processes and are created using the `''|''` (pipe) character between the commands to be chained.
Using the principle of chaining, we can simply write the following to achieve our goal of finding `.mp4` files in a given directory:

```
ls ./films | grep ''\.mp4'' | less
```

Breaking down the above chain, we can see that:

1. `ls ./films` - creates a process, with the stdout piped to the stdin of the next process

2. `grep ''\.mp4''` - reads input from stdin and searches it for filenames ending in `.mp4`.[5]. Output of process piped into the stdin of the next process.
3. `less` - reads input from `stdin` and allows the user to tab through results if a long list is returned. `stdout` isn't directed to any process so it written to the terminal output.

The standard behaviour of the Linux terminal is to print the output of **stdout**, **stderr** and **stdin** to the terminal. In some cases, however, we might want to change this default behaviour and redirect this output elsewhere.
The **redirect** operator: `>` allows us to redirect **stdout** to a file instead of the terminal output. If the file does not exist, it is created; otherwise, it is completely overwritten.
Taking the previous example further, if we were interested in saving the list of mp4 files for later use, we could use the **redirect (>)** operator to store it in a file. The listing below demonstrates this behaviour:

```
ls ./films | grep ''\.mp4'' > films.txt
```

The standard output of the chain is redirected into a file named films.txt.
The redirection operator can also be used to append to files when used twice: `>>`. It will append to the end of the file if it exists (and create it if it doesn't exist). Useful if you want to avoid overwriting existing content.
An example use might be if we wanted to add a new entry for *Star Wars* to the list of films without completely overwriting the existing file. The listing below will write our new movie to the end of the list of the movie.

Steps

1. Open the shell, run Atom on the file to create: myscript.sh
2. Use the `read` command to ask for the user's name.
3. Print a personalised greeting for the user once they have typed their name.
4. Append the user name to the file "users.txt"
5. Save the file (shell script)
6. Set the executable bit
7. Execute the shell script

Submission (directory: `2/tutorial/bash`)

`2/tutorial/bash/myscript.sh` Your shell script written following the above steps.
At the end of the session you will have learned how to work with your portfolio, do not worry for now.

Further Reading

- Shell scripting (also available as PDF) *https://www.shellscript.sh/*
- Shebang Wikipedia: *https://en.wikipedia.org/wiki/Shebang_(Unix)*

5. Group Work (15 Minutes): Understanding Algorithms

We'd like you to team up in groups of 3-4 people to analyze the given pseudo-code that was written as a shell script which is close to a programming language. Your goal is to describe its purpose and how it works in prose. Some commands have not been introduced, use the man pages or try these commands.

5.1 A more complex shell program

```
#!/bin/bash
for I in 1 2 3 4 5 6; do
  mkdir data$I
  cd data$I
  touch fb.txt
  cd ..
done
```

[5] We put a backslash (\) before the "." in order to escape it and prevent the `grep` tool from parsing it as regex

Steps

1. Team up with your colleagues, introduce yourself (1 minute)
2. Everyone should look at the shell-script and think about it. As you haven't learned all the commands, yet, you may want to run `$ man touch`. Run the script in the shell, check the output. Use the commands `$ ls` and `$ cat` to list directory content and the content of files. (3 minutes)
3. Discuss with your colleagues the purpose of the shell script. (5 minutes)
4. Use Atom to describe the algorithm (use between 30-60 words; 2 minutes)
5. We discuss together what the algorithm does and walk through its execution (2 minutes)
6. Modify your description following our discussion (use between 30-60 words; 2 minutes)
7. Save the file `algorithm.txt` in the directory according to the submission description below.

Submission (directory: `2/tutorial/algorithm`)

`2/tutorial/algorithm/algorithm.txt` A text file containing a brief algorithmic description in prose.

Marking Criteria

- one mark for a non-trivial description.

Hints

- Use the command `$ help` and `$ man` to find out what a command does

Further Reading

- PseudoCode: *https://en.wikipedia.org/wiki/Pseudocode*
- Guidelines for good PseudoCode:
  http://www.cs.cornell.edu/courses/cs482/2003su/handouts/pseudocode.pdf

6. Tutorial (15 Minutes): Git and our portfolio

Git is a distributed version control system that is widely used in the industry. A version control system allows you to maintain and track the progress, version history, and updates to documents you create. A distributed system is one which is spread out across several computers and allows you to collaborate more efficiently on shared documents such as code.
In this tutorial, we will start with the very basics that are necessary to manipulate your portfolio. We will expand this later. Git is a command line program, that can be invoked via `$ git <command> <options>`. It comes with help for any embedded command, e.g. `$ git help commit`. Don't be overwhelmed by Git and focus on the relevant commands. People can find Git to be confusing and complicated which is often caused by a lack of understanding of the core functionalities and philosophies of Git. This tutorial is based upon [1].

6.1 Basics

A version control system tracks the changes of a directory tree and its files over time. In Figure 2, we can see three files (A, B, C in green) on the y-axis and their evolution as versions overtime on the x-axis. In version 2, file A and file C are modified, in version 3, only file C, and so on. A version is created by the user who decides to store a **snapshot**, i.e., the current version in the history. Git keeps the whole **history** of these changes and allows us to inspect them.
Git has three main states that your files can reside in: committed, modified, and staged.

- **Committed** means that the data is safely stored in your local repository.
- **Modified** means that you have changed the file but have not committed it to your local repository yet.
- **Staged** means that you have marked a modified file in its current version to go into your next commit **snapshot**.

Figure 3

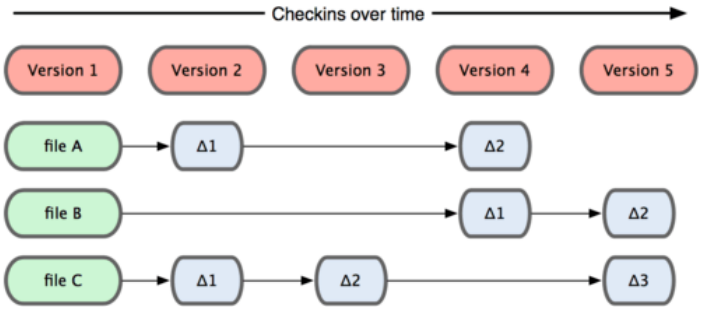

Figure 2: Git evolution of a repository over time (source: [1])

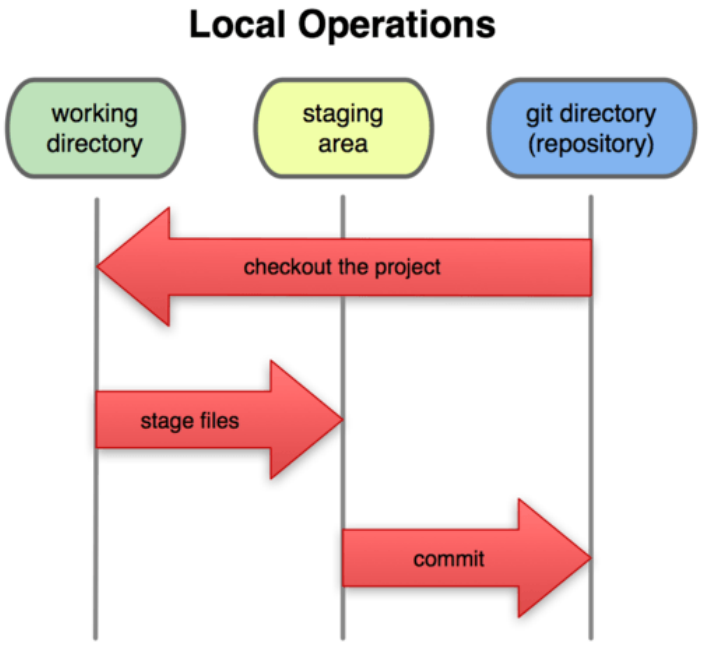

Figure 3: The basic operation and workflow when using Git (source: [1])

Components of Git

- The **Git directory** is a repository where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.
- The **working directory** is a single checkout of one version of the project, i.e., the files and directories you can see in Linux/Windows and modify. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.
- The **staging area** is a simple file, generally contained in your Git directory, that stores information about what will go into your next commit. Its sometimes referred to as the index, but its becoming standard to refer to it as the staging area.

Basic Git workflow The basic local workflow is illustrated in Figure 3.

3. You modify files in your working directory using whatever tool you like.
4. You stage the files, adding snapshots of them to your staging area (using `$ git add <file|directory>`).
5. You do a commit (using `$ git commit`), which takes the files as they are in the staging area and stores them as a snapshot permanently to your Git directory with a message for the changes you made.

If a particular version of a file is in the Git directory, its considered committed. If its modified but has been added to the staging area, it is staged. And if it was changed since it was checked out but has not been staged, it is modified.

Creating a repository Firstly, we have to create a Git **repository**. There are two ways to create a git repository, either creating one in an existing directory (using `$ git init`) or cloning an existing repository from the internet to work on (`$ git clone <URL>`). A repository lives in the directory *.git* in the working directory.

Remote workflow With the basic workflow, we can work locally with our repository. To collaborate with others, we must interact with a **remote repository**. A repository contains the complete history of changes, but they are loosely connected, and synchronization must be requested explicitly. Hence, we extend our steps from above with three additional steps:

1. Cloning the repository from a remote location (using `$ git clone`); this must be done only once.
2. Receiving the changes made on the remote repository (using `$ git pull`); this merges changes into your local repository and ensures that both repositories have the same history.
3. *You modify files in your working directory using whatever tool you like.*
4. *You stage the files, adding snapshots of them to your staging area (using `$ git add <file|directory>`).*
5. *You do a commit (using `$ git commit`), which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory together with a message of the changes you made.*
6. You push the changes you made to the remote repository (using `$ git push`), such that anyone that has access to the repository can pull them. That also is a great way of backing up files.

Configuration of your identiy As Git records with every commit, who created it, you must set your user name and e-mail address. This can be done in the shell via:

```
$ git config --global user.name ‘‘John Doe’’
$ git config --global user.email ‘‘johndoe@student.reading.ac.uk’’
```

Helpful commands to interact with the repository and the history

- Information about a repository can be seen using `$ git status`. This command will show any unstaged files, i.e., files unknown to Git or modified in your local version, or staged files ready for commit. It also provides help by showing useful commands to interact with the repository (in its current status). It is possible to explicitly ignore files to show up, for more on this: research `.gitignore` files
- To visualise the history, you can run the command `$ git log` which shows all the commits. Detailed information is provided using the commands: `$ git log --all --decorate --oneline --graph` You can use a mnemonic like “git log a dog” to help remember all those parameters.
- The differences of files can be shown via `$ git diff`. A useful command is: `$ git diff --color-words <FILENAME>`

6.2 Your Git portfolio

The portfolio is a Git repository that contains all artefacts generated by you from the tutorials and exercises, together with comments from the marking. You will also be able to provide feedback regarding the exercises (e.g., time used) that help us to optimise the activities. In this tutorial, you will create your own private git repository there, e.g., calling it CS1PR, and made it writable to the user *di918039*.
The remote Git repository of the portfolio is stored on the CS department’s GiT server *https://csgitlab.reading.ac.uk/* that provides a simple interface to Git.
Your repository can be uploaded to the automatic marking system via *https://hps.vi4io.org/cs1pr-submit*.

Steps

1. Start a Gnome-Shell, setup your Git identify
   `$ git config --global user.name ‘‘<NAME>’’`
   `$ git config --global user.email ‘‘<EMAIL>@student.reading.ac.uk’’`
2. Login to CS-Gitlab *https://csgitlab.reading.ac.uk/*
3. Create the repository
   - Click on “New Project”, use “cs1pr” as the project name.
   - Ensure that the visibility level is set to “Private”.
   - Confirm.
   - It will now redirect you to a page that contains the URL for the repository, choose HTTPS and copy the URL to the clipboard.
4. Clone your repository. In the shell, type `$ git clone URL`, where URL is the URL as shown on your GitLab page. This command will create a new directory called as the repository, in your case should be cs1pr. You have to run the clone command only once to fetch and initialize your repository using your current repository on cs-gitlab.

5. Grant me permissions to the repository in CS-Gitlab:
   - Click on the left on "Settings", "Members".
   - Insert "di918039" into the "Select members to invite" form, select me.
   - Select developer in "Choose a role permission"
   - Click "Add to project"
6. In the following, we will create the basic structure in the repository. An example structure is provided in a repository that you may clone:
   `$ git clone https://csgitlab.reading.ac.uk/di918039/cs1pr-portfolio`
7. We can copy the repository to your directory `$ cp -r cs1pr-portfolio/* cs1pr/`
8. Change the working directory into your repository: `$ cd cs1pr/`.
9. Check what possibly changes we made to the working copy: `$ git status`. You may check the above listed suggested Git commands.
10. Add README.txt to the staging area of your repository: `$ git add README.txt`
11. Check what possible changes we made to the working copy: `$ git status`, check what is different!
12. Add run-test.sh to the staging area: `$ git add run-test.sh`. Remember every time you made changes to your working copy, you have to add them.
13. Modify README.txt by using Atom to include your name.
14. Check what possible changes we made to the working copy: `$ git status`
15. Check the details of the changes: `$ git diff --color-words`
16. Add README.txt to the staging area: `$ git add README.txt`
17. Now we commit the snapshot of changes from the changing area using the command: `$ git commit`. This command will open a text editor, and it also gives you the opportunity to revisit the changes made to the *index* you intend to commit. This command will open an editor where you can type in a message about your changes (this could be something like "Created README file"). **It is advised to check the changes thoroughly before committing!**. Typically, it will open the editor `nano`, to close the editor, press CTRL-X (as shown on the bottom of the editor), press Y to confirm you want to safe and then press enter to confirm the filename. Now you made your first commit. This is now stored in Git forever and cannot be lost.
18. Push your local changes with its most recent commits to the remote CS-Gitlab server: `$ git push`
19. Check on the CS-Gitlab web page that your changes are there.
20. Attempt to submit your repository to the submission tool: *https://hps.vi4io.org/cs1pr-submit*. It will return an error, as not all necessary files exist, yet!
21. Copy the file `algorithm.txt` from the previous exercise to the correct location, first we create the directory: `$ mkdir algorithm`, then we copy the file: `$ cp algorithm.txt algorithm/` (make sure to use the correct filename, as we do not know where you saved algorithm.txt).
22. Add it to Git (see above). Commit and push.
23. Check on the CS-Gitlab web page that your changes are there.
24. Finally, to create a feedback file: **fb.txt**. Such a file can be created in any exercise directory by you. It follows a simple format such that the first line is the time spend on the exercise (in minutes), then followed by additional lines with comments. Create such a file inside the directory `2/git/` and store the time needed to complete this tutorial.

It is advised to check the changes thoroughly before committing! Also, pull/push at least on a daily basis to prevent the loss of data..

Hints

- Note that it is important that you run any Git command from a subdirectory of the cloned working directory. Git will try to find the repository by looking for the `.git` directory in a parent directory.

Further Reading

1 *https://git-scm.com/book/en/v1/Getting-Started-Git-Basics*

- Git cheatsheet with important commands
  https://github.github.com/training-kit/downloads/github-git-cheat-sheet.pdf

7. Tutorial (15 Minutes): FlowCharts with LibreOffice

According to Wikipedia[6]: *Flowcharts are used in designing and documenting simple processes or programs. Like other types of diagrams, they help visualize what is going on and thereby help understand a process, and perhaps also find less-obvious features within the process, like flaws and bottlenecks. There are different types of flowcharts: each type has its own set of boxes and notations. The two most common types of boxes in a flowchart are:*

- a processing step, usually called activity, and denoted as a rectangular box.
- a decision, usually denoted as a diamond.

In this tutorial, we will use LibreOffice to draw simple flow charts.

Steps

1. Look at the Description of the Wikipedia page: *https://en.wikipedia.org/wiki/Flowchart*
2. Start LibreOffice - Select the Draw application
3. Start the help (Menu: help)
4. In the help, search for flowchart
5. Add some Flowchart elements (bottom icons)
6. Add some text boxes for the decisions (F2)
7. Use connectors to connect them (bottom icons)
8. Color the boxes nicely (top icon bar)
9. Changing the page size to fit the outcome (Menu: Format - Page Properties)
10. Save your Flowchart
11. Exporting the generated artefact image (Menu: File - Export PDF)

You may use the file `1/flow-chart-example.odg` in our example portfolio.

Hints

- You may want to use copy/paste to duplicate coloured elements.
- The colour picker shows recently used colours.

8. Group Work (15 Minutes): Describing Algorithms as a Flow Chart

In this task, we will use Libreoffice to create a flowchart for the Git "workflow". Document the steps needed to work with CSGitLab. There should be an iteration in the workflow.

Steps

1. Team up in groups of two students[7], work on one computer first
2. Use Libreoffice to draw the flow chart of the algorithm
3. Discuss your flow chart with another group
4. We discuss the flow charts together (5 minutes)
5. Save the file as `algorithm.odg` in the directory called `algorithm` (the same as before for the prose text!) and export it to PDF – all team members can submit the same file!
6. Use Atom to save the file `algorithm-team.txt` where you add the names of your peers

[6] *https://en.wikipedia.org/wiki/Flowchart*
[7] A group of three is an exception, only possible when the number of attendees in odd.

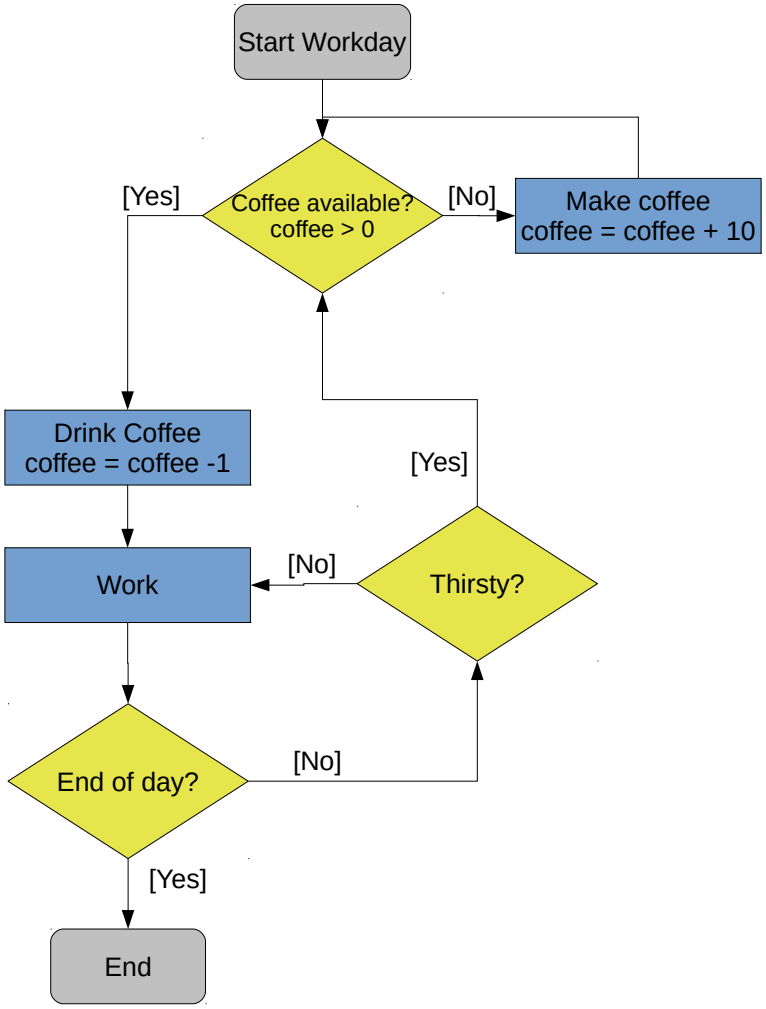

Figure 4: Flow-Chart produced using LibreOffice

Submission (directory: `2/tutorial/algorithm`)

`2/tutorial/algorithm/algorithm.odg`	The Libreoffice representation of the flow chart
`2/tutorial/algorithm/algorithm.pdf`	The PDF of the flow chart

Marking Criteria

- one mark for each file

9. Tutorial (0 Minutes): Optional: Setup of your own Linux Environment

Instead of relying on the Linux environment on the machines, you can setup a Linux image on your as a VirtualMachine, i.e., you can run it under macOS or Windows. We provide a VirtualMachine that comes with the necessary software here: *https://hps.vi4io.org/shared/programming.ova*. The image uses VirtualBox[8]. VirtualBox is a free virtualization environment allowing you to run virtual machines on your system (the host system).
You can import the image either to your laptop or to the systems in the lab[9].
The username and password is ubuntu.

Steps

1. Download the file from *https://hps.vi4io.org/shared/programming.ova*; it should be available in `C:\programming_image` as well.

[8] *https://www.virtualbox.org/*
[9] In the labs store the data locally, though, as the network drive might have a performance issue

2. Start VirtualBox
3. Click import