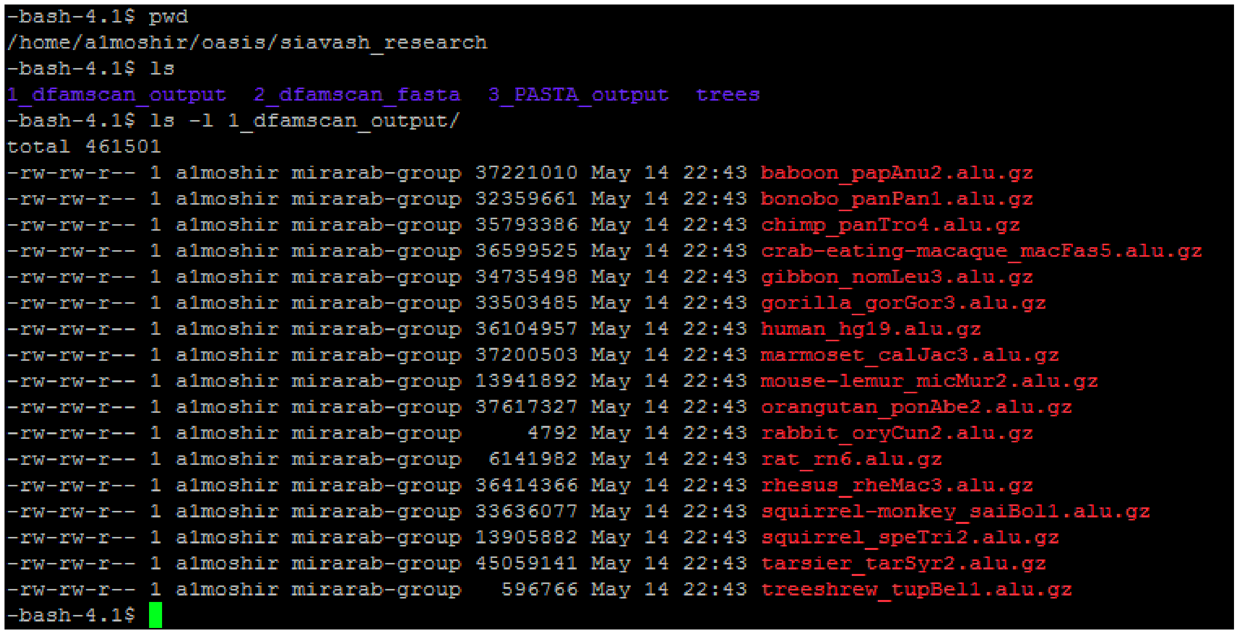
The Terminal-ator

Step 1

When non-Unix users see the Unix Command Line (i.e., the Terminal), they often immediately think of 80's hacker movies. Although we must admit that using your computer's terminal for regular computer science tasks isn't quite as intense as "hacking into the mainframe" of some evil corporation, the terminal is still quite powerful.

In this day and age, we have become used to the "point-and-click" (or even simply "touch") nature of modern operating systems, but most large-scale computational tasks are done remotely via some compute server. Consequently, knowing how to use the Unix command line properly is vital.

In this section, you will learn the basics of how to navigate the Unix command line, and in addition to basic navigation, you will be introduced to some of the more powerful Unix tools built into most Unix-based operating systems. Beware that we will only be touching the surface in this course, and as with most computer-related things, one can only become truly comfortable using the terminal with practice, practice, practice.



```
-bash-4.1$ pwd
/home/a1moshir/oasis/siavash_research
-bash-4.1$ ls
1_dfamscan_output  2_dfamscan_fasta  3_PASTA_output  trees
-bash-4.1$ ls -l 1_dfamscan_output/
total 461501
-rw-rw-r-- 1 a1moshir mirarab-group 37221010 May 14 22:43 baboon_papAnu2.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 32359661 May 14 22:43 bonobo_panPan1.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 35793386 May 14 22:43 chimp_panTro4.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 36599525 May 14 22:43 crab-eating-macaque_macFas5.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 34735498 May 14 22:43 gibbon_nomLeu3.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 33503485 May 14 22:43 gorilla_gorGor3.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 36104957 May 14 22:43 human_hg19.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 37200503 May 14 22:43 marmoset_calJac3.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 13941892 May 14 22:43 mouse-lemur_micMur2.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 37617327 May 14 22:43 orangutan_ponAbe2.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group     4792 May 14 22:43 rabbit_oryCun2.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group  6141982 May 14 22:43 rat_rn6.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 36414366 May 14 22:43 rhesus_rheMac3.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 33636077 May 14 22:43 squirrel-monkey_saiBol1.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 13905882 May 14 22:43 squirrel_speTri2.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group 45059141 May 14 22:43 tarsier_tarSyr2.alu.gz
-rw-rw-r-- 1 a1moshir mirarab-group   596766 May 14 22:43 treeshrew_tupBel1.alu.gz
-bash-4.1$
```

Figure: An example of the Unix Command Line

Step 2

Although you will find some Unix challenges in the next few steps, we will not actually teach you how to use the Unix command line on the Stepic platform because Codecademy has an excellent interactive self-paced free online course called Learn the Command Line.

Work through the Codecademy lessons (the resources they make available for free will suffice for our purposes, but you have the option of buying a subscription to have access to their paid resources if you want extra practice), and when you are finished (or if you are already comfortable using the Unix command line), continue to the next step!

Step 3

UNIX CHALLENGE: Basic Navigation: Creating a File

To warm up, we'll start with something simple: create a file called `message.txt` in the working directory containing the text "Hello, world!" (no quotes)

HINT: You may want to use `vim`, a command line text editor, to write your code. Use the Interactive `vim` Tutorial to learn how to use it if you are unfamiliar with it.

You have an unlimited number of attempts.
Time limit: `60 mins`

To solve this problem please visit https://stepik.org/lesson/27734/step/3

Step 4

UNIX CHALLENGE: Basic Navigation: Creating Directories

Now something a bit more challenging: create a directory called `files` in the working directory, and within that directory, create 3 files named `file1.txt`, `file2.txt`, and `file3.txt` (the content of these files is irrelevant).

You have an unlimited number of attempts.
Time limit: `60 mins`

To solve this problem please visit https://stepik.org/lesson/27734/step/4

Step 5

UNIX CHALLENGE: Real-World Tasks

Note: This unix challenge is an actual *challenge*!

In the Unix terminal, you will be given the file `example.fa` in the working directory. Perform the following tasks using this file:

1. Count the number of sequences in `example.fa`, and save the result into a file in the working directory called **`1_num_seqs.txt`** (Hint: you can `grep` for the '>' symbol and then use `wc` to count the number of lines)
2. In each sequence (i.e., in each DNA string), the "seed" is defined as characters 2-8 of the sequence (inclusive, 1-based counting). Find the unique "seed" sequences in `example.fa`, and save the result into a file in the working directory called **`2_unique_seeds.txt`** (Hint: you can do an inverse `grep` for the '>' symbol to isolate the sequences, then use `cut` to extract the "seed" sequences, and then use `sort` and `uniq` to remove duplicate lines)

You have an unlimited number of attempts.
Time limit: `60 mins`

To solve this problem please visit https://stepik.org/lesson/27734/step/5

Step 6

In general, the vast majority of your time with the Unix command line will be basic navigation, file manipulation, and program execution. If you don't remember all of the fancier Unix commands off the top of your head, that's perfectly fine. You can always search Google if you have a specific problem you need to solve, and StackOverflow is always your friend on that front.

If you are still not comfortable with the Unix command line, as we mentioned before, the best way to learn is to practice. For more practice, we suggest installing a Linux distribution onto a virtual machine via VirtualBox (a great tutorial on how to install Ubuntu, one of the more user-friendly Linux distributions, using VirtualBox can be found in this post in the Ubuntu support forums). If you end up really enjoying Linux and want to use it as one of your main Operating Systems on your computer (as opposed to through a virtual machine on top of your main Operating System), you might even want to think about dual-booting your main Operating System with Ubuntu (or whatever Linux distribution you prefer).

Step 7

Earlier in this text, we reviewed/introduced some basic C++ syntax and usage. Further, throughout this text, we will have C++ coding challenges embedded within the lessons in order to test your understanding of the data structures and algorithms we will be discussing. For these coding challenges, you will be performing all of your programming directly within Stepic in order to avoid issues regarding setting up your environment and whatnot. However, as a Computer Scientist, you will likely be developing and running your own code from the command line, either locally on your own computer or remotely on some compute cluster. As such, it is important for you to know how to compile and run code from the Unix command line.

In the second half of this lesson, we will be learning how to write, compile, and execute C++ code directly from the command line.

Step 8

In an introductory programming course, or as a beginner programmer, you are likely to write code that is quite simple and that can be written cleanly in a single file. For example, say we have a file called `HelloWorld.cpp` that consists of the following:

```
#include <iostream>
int main() {
    std::cout << "Hello, world!" << std::endl;
}
```

We could compile our code using **`g++`**, a C++ compiler, and execute it, all from the command line:

```
$ g++ HelloWorld.cpp # compile our program
$ ./a.out            # run our program ("./a.out" means "in this directory, a file called "a.out")
Hello, world!        # our code's output
```

The default filename for the executable file was `a.out`, which isn't very descriptive. To specify a filename, we can use the **`-o`** flag:

```
$ g++ -o hello_world HelloWorld.cpp # compile our program and name the output hello_world
$ ./hello_world                     # run our program (hello_world)
Hello, world!                       # our code's output
```

Step 9

UNIX CHALLENGE: Writing, Compiling, and Executing C++ Code from the Unix Command Line using `g++`

In the Unix terminal, perform the following tasks in the current working directory:

1. Write a program **`PrintNums.cpp`** that prints the numbers 1-10 to standard output (`std::cout`), one number per line
2. Use **`g++`** to compile **`PrintNums.cpp`** and create an output executable file called **`print_nums`**

The expected output to standard output (`std::cout`) from running `print_nums` should look like the following:

```

```

HINT: You may want to use `vim`, a command line text editor, to write your code. Use the Interactive `vim` Tutorial to learn how to use it if you are unfamiliar with it.

You have an unlimited number of attempts.
Time limit: `60 mins`

To solve this problem please visit https://stepik.org/lesson/27734/step/9

Step 10

The simple `g++` command we just learned is great if your code is simple and fits in just a handful of files, but as you become a more seasoned programmer, you will begin embarking on more and more complex projects. When working on a large and robust project, good practice dictates that the program should be modularized into smaller individual components, each typically in its own file, for the sake of organization and cleanliness. Although this makes the actual programming and organization easier, it makes compiling a bit more complex: the more components a program has, the more things we need to specify to `g++` for compilation.

As a result, developers of complex projects will include a **`Makefile`**, which is a file that contains all of the information needed to compile the code. Then, as a user, we simply download the project and call the **`make`** command, which parses the `Makefile` for us and performs the compilation on its own. The basic syntax for a `Makefile` is as follows (by `[tab]`, we mean the tab character):

```
target: dependencies
[tab]system command
```

For example, we could compile the previous example using the following `Makefile`:

```
all:
	g++ -o hello_world HelloWorld.cpp
```

Note that the whitespace above is a **tab character**, *not* multiple spaces. In a `Makefile`, you **must** use tab characters for indentation. To compile our code, assuming we have `HelloWorld.cpp` and `Makefile` in the same directory, we can go into the directory and do this:

```
$ make          # run the make command to compile our code
$ ./hello_world # run our program
Hello, world!   # our code's output
```

In the example above, our target is called **`all`**, which is the default target for `Makefiles`. The `make` program will use this target if no other one is specified. We also see that there are no dependencies for target `all`, so `make` safely executes the system commands specified.

Depending on the project, if it has been modularized enough, it is useful to use different targets because, if you modify a single file in the project, make won't recompile *everything*: it will *only* recompile the portions that were modified. Also, you can add custom targets for doing common tasks, such as removing all files created during the compilation process. For example, say our `HelloWorld.cpp` code depended on another file, `PrintWords.cpp`, we might create the following `Makefile`:

```
all: hello_world                        # default all target, which has 1 dependency: hello_world

hello_world: HelloWorld.o PrintWords.o # hello_world target, which has 2 dependencies
        g++ -o hello_world HelloWorld.o PrintWords.o

HelloWorld.o: HelloWorld.cpp           # HelloWorld.o target, which has 1 dependency
        g++ -c HelloWorld.cpp

PrintWords.o: PrintWords.cpp           # PrintWords.o target, which has 1 dependency
        g++ -c PrintWords.cpp

clean:                                 # clean target, to remove all compiled files
        rm *o hello_world
```

We can then compile and run our code as follows:

```
$ make          # compile our code
$ ./hello_world # run our program
Hello, world!   # our code's output
$ make clean    # remove all files resulting from compilation using the Makefile's clean target
```

You can get pretty fancy when you write a `Makefile`! So as your projects become more and more complex, be sure to write a `Makefile` that is robust enough to handle your project with ease. There is an excellent resource on about writing a `Makefile`, written by Hector Urtubia, that can be found here.

Step 11

UNIX CHALLENGE: Writing, Compiling, and Executing C++ Code from the Unix Command Line using a `Makefile`

In the Unix terminal, perform the following tasks in the current working directory:

1. Write a program **`PrintNums.cpp`** that prints the numbers 1-10 to standard output (`std::cout`), one number per line
2. Write a **`Makefile`** such that your code can be compiled with just the `make` command. It should create an output executable file called **`print_nums`**
3. Compile your code using the **`make`** command

The expected output to standard output (`std::cout`) from running `print_nums` should look like the following:

```
1
2
3
4
5
6
7
8
9
10
```

HINT: You may want to use `vim`, a command line text editor, to write your code. Use the Interactive `vim` Tutorial to learn how to use it if you are unfamiliar with it.

You have an unlimited number of attempts.

Time limit: `60 mins`

To solve this problem please visit https://stepik.org/lesson/27734/step/11

Step 12

As we mentioned before, being fluent in using the Unix command line is an essential quality to have, especially in this day and age. Some examples of tasks that require usage of the Unix command line include the following:

- Running large computational tasks on a remote compute cluster
- Developing, compiling, and executing code
- Automating repetitive tasks for convenience
- So much more!

Just like with most things, the only way to become comfortable with the Unix command line is practice, practice, practice. In time, you'll get the hang of things!