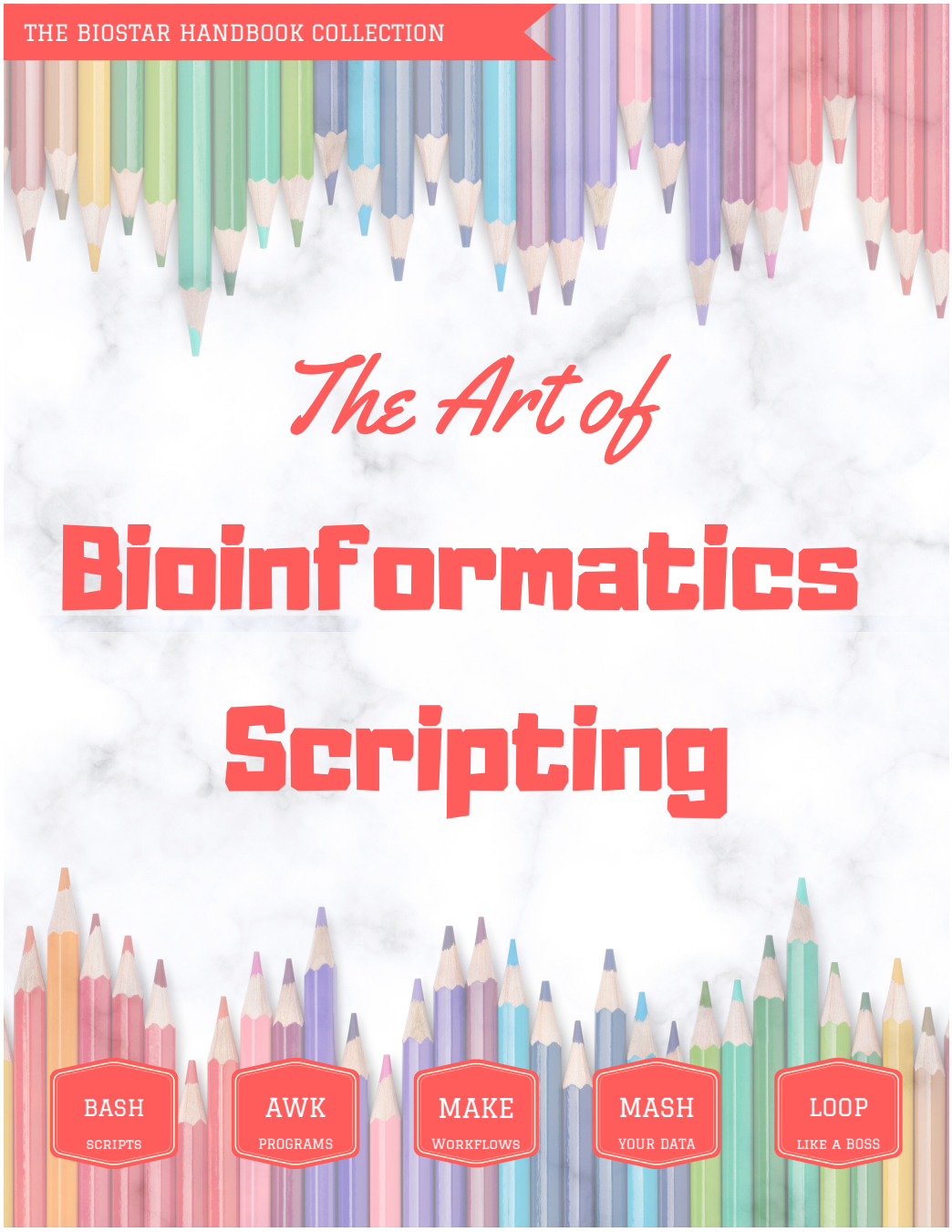


Book updated on February 7, 2020

Contents

Chapter 1

Welcome to scripting

Last updated on **February 07, 2020**

This book is the volume from the Biostar Handbook Collection[1] that introduces readers to the use of Unix shell in the context of performing bioinformatics data analyses. Prerequisites:

- Introduction to Unix[2]
- Data analysis with Unix[3]

All materials in the Biostar Handbook Collection[4] have been developed, improved, and refined over a decade in a research university setting as part of an accredited Ph.D. level bioinformatics training program. The contents of this book have provided the analytical foundation to thousands of students, many of whom have become full-time bioinformaticians and work at the most innovative companies in the world.

1.1 The Biostar Handbook Collection

- **The Biostar Handbook**[5] - The main introduction to Bioinformatics.

[1] https://www.biostarhandbook.com
[2] https://www.biostarhandbook.com/unixanalyis.html
[3] https://www.biostarhandbook.com/introduction-to-unix.html
[4] https://www.biostarhandbook.com
[5] https://www.biostarhandbook.com/

- **The Art of Bioinformatics Scripting**[6] - Learn Unix and Bash scripting.
- **RNA-Seq by Example**[7] - Learn RNA-Seq data analysis.

Access to all new books and materials is included with your subscription!

1.2 Is there an "art" to scripting?

I believe, there is.

For a beginner all automated processes look like magic. Yet, just like magic, when practiced haphazardly, these programs can blow into our faces - sometimes in the most devious manners:

- A Code Glitch May Have Caused Errors In More Than 100 Published Studies[8]

In the story above, the critical error was to rely on the sort order of files that the operating system produces when we ask it to list files in a directory. This flawed approach, of default sorting with pattern matching is widespread in bioinformatics as well; an unreliable choice of code might look like this:

```
# This is a counterexample! Do not use it!
for name in *.fq; do
    echo "runtool $name > $name.txt"
done
```

The code is suboptimal because it runs on whatever files match the pattern. This behaviour seems convenient at first, perhaps involves less typing, yet constructs designed like so frequently lead to ever increasing complications and unexpected problems.

A more robust approach discussed in the chapter How to write better loops explains how to do it better.

[6] https://www.biostarhandbook.com/books/scripting/index.html
[7] https://www.biostarhandbook.com/books/rnaseq/index.html
[8] https://www.vice.com/en_us/article/zmjwda/a-code-glitch-may-have-caused-errors-in-more-

1.3 How to download the book?

The book is available to registered users. The latest versions can be downloaded from:

- The Art of Bioinformatics Scripting, PDF[9]
- The Art of Bioinformatics Scripting, eBook[10]

The book is updated frequently, especially during the Spring and Fall semesters, when the book is used as a textbook. We recommend accessing the book via the website as the web version will always contain the most recent and up-to-date content. A few times a year we send out emails that describe the new additions.

Want to know when new content is published? Subscribe below:

1.4 Typesetting conventions

In the web version of the book long lines of code are wrapped. Here is an example of two long lines of code that are displayed as wrapped over four lines.

```
cat foo.fa | parallel --round-robin --pipe --recstart '>' 'blat -noHead genome.fa stdin >
cat foo.fa | parallel --round-robin --pipe --recstart '>' 'blat -noHead genome.fa stdin >
```

Long line wrapping only works in the HTML version. We recommend using the web version if you choose to copy paste code into a terminal. PDF and eBook formatters may insert various invisible characters into their formats that can cause problems.

1.5 How was the book developed?

We have been teaching bioinformatics and programming courses to life scientists for many years now. We are also the developers and maintainers of

[9] `art-of-scripting.pdf`
[10] `art-of-scripting.epub`

Biostars: Bioinformatics Question and Answer[11] website, the leading resource for helping bioinformatics scientists with their data analysis questions.

We wrote this book based on these multi-year experiences in training students and interacting with scientists that needed help to complete their analyses. We are uniquely in tune with the challenges and complexities of applying bioinformatics methods to real problems, and we've designed this book to help readers overcome these challenges and go further than they have ever imagined.

1.6 How do I access my account?

Logged in users may manage their accounts via the link below. You change your email or log out via this page.

Access Your Account

[11] `https://www.biostars.org`

Part I

WRITING SHELL SCRIPTS

Chapter 2

What are shell scripts

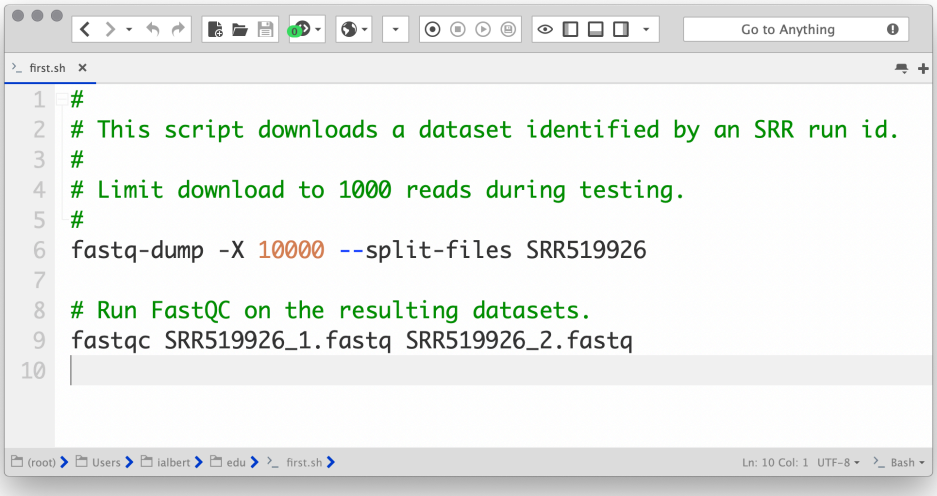
We will cite the Wikipedia definition[1] below:

> A shell script is a computer program designed to be run by the Unix shell, a command-line interpreter. The various dialects of shell scripts are considered to be scripting languages. Typical operations performed by shell scripts include file manipulation, program execution, and printing text. A script that sets up the environment runs the program, and does any necessary cleanup, logging, etc. is called **a wrapper**.

2.1 What is a bash script?

A bash script is a plain text file executed via the `bash` shell variant (dialect). Here is what a script might look like in your editor.

[1] `https://en.wikipedia.org/wiki/Shell_script`



Bash scripts are usually relatively short text files (10-100 lines) that contain a series of commands that you could also type into the terminal. Conventionally the shell file extension is `.sh` (naming it such will allow your editor to color it as seen above) and is run through the shell either with :

```
bash myscript.sh
```

or, even as:

```
myscript.sh
```

2.2 How do I write scripts?

Use a "plain text editor" to create scripts!

Writing in **Microsoft Word** and saving it as text will eventually cause problems even if initially it seems to work fine. **Microsoft Word** is a rich text editor that will eventually insert characters that are invisible and will mess up your programs.

Using **Microsoft Word** for scripting is asking for trouble! Don't say we did not warn you. For those that still choose to sin (you know who

> you are), your time of atonement is nigh.

Recommended editors:

- **Notepad++**[2] for Windows (recommended editor on Windows)
- **Komodo Edit** for Mac, Windows, Linux (used to be good but it is getting too complicated)
- **Sublime Text** for Mac, Windows, Linux

Programming editors:

- **PyCharm** for Mac, Windows Linux (recommended for Python programmers, I use this tool for both Python programming and text editing)
- **Visual Studio Code** for Mac, Windows Linux (excellent for coding as well)

2.3 Can I run a script directly?

If the first line of the script starts with a so-called shebang (hash pound)[3] line

```
#!/bin/bash
```

and the script has been marked as executable with

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

then the script may be executed by running it directly:

myscript.sh

Some scripts are named without an extension. For example:

```
myscript
```

in which case they look like an "executable" file. The practice of removing extensions is, in our opinion, a counterproductive and misleading practice as it does not correctly communicate that the program in question is a script.

[2] https://notepad-plus-plus.org/
[3] https://en.wikipedia.org/wiki/Shebang_(Unix)

In addition, your editor will not recognize the program as a script; it won't syntax highlight it. Keep the extension on your scripts!

In the book we use the explicit notation of executing bash scripts with:

```
bash myscript.sh
```

We recommend that you do that too.

2.4 Why should I write scripts?

The primary advantage of scripts is that these programs will allow you to collect and document related functionality in a single file.

Besides listing the commands, scripts allow you to use comments to describe the reasons, rationale, and decisions that you have made along the way.

Furthermore, scripts will let you organize and reuse solutions, thus massively improving your productivity.

2.5 How should I write my comments?

There is an art to writing comments. If the comments are long, they may interfere with understanding the command that you are explaining. So you want them short, but not too short as to be cryptic. With experience, you'll find the system that works for you.

Comments are lines starting with the `#` symbol. We recommend making comments short, complete sentences that end with a period.

```
#
# This script downloads a dataset identified by an SRR run id.
#
# Limit download to 1000 reads during testing.
#
fastq-dump -X 10000 --split-files SRR519926

# Run FastQC on the resulting datasets.
fastqc SRR519926_1.fastq SRR519926_2.fastq
```

Is the above code a good script? It is a good start; to make it better, we would need to work on it a bit more, as explained later in the section.

2.6 Who are we writing the documentation for?

The future us.

When we start a project, we are in tune with it, and it feels like we will never forget the small and straightforward decisions we made during the project. That belief is rooted in wishful thinking. We all have better things to do than remember minutiae. But then later few things feel more painful and frustrating than having to redo something just because we forgot a seemingly essential aspect of it.

Documentation helps us remember the reasons that we made individual decisions. It is a significant productivity boost and lets us get more done in less time.

2.7 Should every line have a comment?

Yes. I found that the best practice is to precede each line with a short comment. Leave spaces between lines of code so that that it is airy and looks fresh - helps with software rot[4].

2.8 Will bash work differently when typing into terminal versus running a script?

There might be a few differences.

When you type into the terminal bash operates in the so-called "interactive" mode designed for human interaction. For example, pressing a TAB will complete a filename. This is called tab-autocompletion[5]. (you do remember

[4] `https://en.wikipedia.org/wiki/Software_rot`

[5] `https://en.wikipedia.org/wiki/Command-line_completion`

to press tab, right?). Never write out fully any filename from the keyboard, always autocomplete! This action will ensure that you are using the correct names. If you are not tabbing regularly[6], get into the habit of doing so!

Within a script, TABs will not autocomplete.

One particularly annoying interactive behavior is the new meaning of the `!` character. When you type `!` into an interactive bash session, it is interpreted as an operator that can rerun a previous command as a match. It has many behaviors, for example, if you first typed

```
foo
bar
```

then later you could type:

```
!f
```

to rerun `foo`. This is called event designation[7].

It is an outdated "feature" that will allow you to run commands without showing you what it will do. Don't use it. It goes against the principles of being explicit and in full command. If you need a previous command, press the up arrow key to scroll through them.

The most annoying side effect of history expansion is that some strings that work in a script will not work when typed into the shell. For example:

```
echo "Hello World!"
```

will work when you have it in a script but will fail when you type it into the terminal as in that case, it thinks the `!` is an event designator. Oh well…

[6] `https://en.wikipedia.org/wiki/Command-line_completion`

[7] `https://www.gnu.org/software/bash/manual/html_node/Event-Designators.html#Event-Designators`

Chapter 3

Writing better scripts

3.1 What is code "refactoring"?

Refactoring is an iterative process of improving the code to reduce its redundancy and make it more generic and simpler. The reason we need refactoring is that typically a process becomes better understood as we work and solve it. Periodically, we may need to revisit previous steps and rework them to match the later stages.

Refactoring may feel wasteful, as often we modify a section of the script that may seem to be already working just fine. But in any analysis, complexity can be a hindrance. Refactoring takes on complexity and attempts to simplify our work and will pay dividends in the future.

Refactoring takes some practice, and typically the returns are diminishing – every code can always be refactored once more, but after a few rounds of doing so the benefits are usually much smaller.

3.2 What is the most important improvement I can make to my code?

Move information that can change during running into so-called "variables". Variables help you separate the sections that you might want to change later from those that should always run the same way.

```
NAME=John
echo Hello ${NAME}!
```

then run it with:

```
bash sayhello.sh
```

Putting the variables first allows you to change them, making your code more adaptable quickly. Let's revisit the code we shown before:

```
# Limit download to 1000 reads.
fastq-dump -X 10000 --split-files SRR519926

# Run FastQC on the resulting datasets.
fastqc SRR519926_1.fastq SRR519926_2.fastq
```

Move the variable sections to the start, thus our script now looks like so:

```
# The selected SRR number.
RUN=SRR519926

# The number of reads to convert.
LIMIT=10000

# Get and convert the data.
fastq-dump -X ${LIMIT} --split-files ${RUN}

# Run FastQC on the resulting datasets.
fastqc ${RUN}_1.fastq ${RUN}_2.fastq
```

Note how nice and airy this code is. One comment per action. One empty line after each action. Imagine a fresh breeze bathing each line of code, keeping it from rotting[1]. You know scripting when you derive joy not just from getting the job done, but from writing simple, sturdy, and pretty lines of code.

[1] https://en.wikipedia.org/wiki/Software_rot

3.3 Why is it right to stop a script when an error occurs?

There is a saying coined in the book titled Pragmatic Programmer[2] that says:

- **Dead programs tell no lies.**

What it means is that the best course of action on any error is to stop right away, because that is the place where it will be the easiest to troubleshoot. In addition we don't want the lies and errors to accumulate, or perhaps silently become ignored.

By default, the bash shell will lie to us, it will merrily continue with the next command even if a previous command failed.

```
echo Step 1
lss
echo Step 2
ls -1
echo All Done! Good Job!
```

The typo `lss` will raise an error message but the script will continue on and congratulate on a job well done, when in fact not all commands completed. We can override this shell behavior via the `set` command like so:

```
set -u -e
```

that can be shortened to:

```
set -ue
```

Add this to every shell script you ever write to make troubleshooting easier.

3.4 How to make a script print the commands as it executes them?

Often it is quite useful to keep track of the commands as they get executed. To do so set the `-x` flag like so (adding the existing flags)

[2] `https://www.amazon.com/Pragmatic-Programmer-Journeyman-Master/dp/020161622X`

```
set -uex
```

3.5 How do I add runtime parameters?

The next level of refactoring comes from moving the variable sections of a script outside the script, and having them specified at the command line like so:

```
NAME=$1
echo Hello ${NAME}!
```

and you can run it with:

```
bash sayhello.sh John
```

or

```
bash sayhello.sh Jane
```

As you can see above, passing external variables are possible by using the positional variables `$1`, `$2` where `$1` stands for the first positional variable, `$2` for the second positional variable and so on.

Let's rewrite our code so that we can specify the run id and the number of reads to convert from each run id as positional parameters. Suppose we want to be able to run our script in the following manner:

```
getdata.sh SRR519926 1000
```

and when doing so, we get `1000` reads out of run `SRR519926`. The code that achieves that might look look like this:

```
# Downloads an SRR run.
# Converts a LIMIT number of spots.
# Assumes the data is in paired-end format.
# Runs FastQC on each resulting FastQ file.

# Stop on any error.
set -ue

# The first parameter is the SRR number.
RUN=$1
```

```
# The second parameter is the conversion limit.
LIMIT=$2

# Remind user what is going on.
echo "Rock on! Getting ${LIMIT} spots for ${RUN}"

# Get the data from SRA.
fastq-dump -X ${LIMIT} --split-files ${RUN}

# Run FastQC on the resulting datasets.
fastqc ${RUN}_1.fastq ${RUN}_2.fastq
```

We can now run the script like this:

```
bash getdata.sh SRR1553607 10000
```

We now have a reusable script that can get a lot done with just a simple command.

Chapter 4

How to write better loops

The use of inefficient looping code is pervasive in bioinformatics books and training materials. Time and again, you will see cringe-worthy looping examples that will cause more trouble down the road.

4.1 What should I not do?

When you build any script

1. Avoid identifying files by matching to patterns `*.fq`
2. Avoid using the `for` keywords in looping constructs.
3. Avoid using `if` constructs to decide what to do next.

4.2 How should I design my scripts?

1. Explicitly build the file names from a known root.
2. Use GNU parallel to "construct commands".

The most important rule is to build your commands from the **"root"** of the filenames. The "root" is the unique elements across all the files that you generate.

As always, there may be exceptions to the rules, but it starts with how you think about each problem.

4.3 What will bad code look like?

First, let's get sequencing data:

```
fastq-dump --split-files -X 10000 SRR1553607
fastq-dump --split-files -X 10000 SRR1972917
```

Suppose we need to shorten reads to a length of 20 bp. We chose to do it with `cutadapt -l 20`. Now let's write a lousy script first:

```
# NOTE! This is a counterexample! Do not use it!
for name in *.fastq; do
    echo "cutadapt -l 20 $name -o $name.trimmed.fq"
done
```

We print the commands instead of executing them, so that you better understand what is going on. Our script produces the following:

```
cutadapt -l 20 SRR1553607_1.fastq -o SRR1553607_1.fastq.trimmed.fq
cutadapt -l 20 SRR1553607_2.fastq -o SRR1553607_2.fastq.trimmed.fq
cutadapt -l 20 SRR1972917_1.fastq -o SRR1972917_1.fastq.trimmed.fq
cutadapt -l 20 SRR1972917_2.fastq -o SRR1972917_2.fastq.trimmed.fq
```

The problem with the code above is that we are pattern matching on the file names (the pattern may include files we did not intend to process) and that we have no control over the order. We can't safely run the code in any directory until we ensure that it does have other files that match the pattern. Sooner or later, we'll run into devious problems.

There are other inefficiencies as well, for example, this code keeps adding extensions to the filename, each trimmed file now ends with `.fastq.trimmed.fq`. This is a way to tell someone might have been using loops the wrong way … their file names seem to "grow" multiple file extensions.

To strip off the extension we would need to call upon a more complicated bash construct:

```
# NOTE! This is a counterexample! Do not use it!
for name in *.fastq; do
    echo "cutadapt -l 20 $name -o ${name%.*}.trimmed.fq"
done
```

Above we had to use a bash substitution operator `%.*`. Yuck! The code now produces:

```
cutadapt -l 20 SRR1553607_1.fastq -o SRR1553607_1.trimmed.fq
cutadapt -l 20 SRR1553607_2.fastq -o SRR1553607_2.trimmed.fq
cutadapt -l 20 SRR1972917_1.fastq -o SRR1972917_1.trimmed.fq
cutadapt -l 20 SRR1972917_2.fastq -o SRR1972917_2.trimmed.fq
```

The code is more complicated, less readable, and we still haven't fixed the problem of not knowing either the order nor the files that we operate on.

4.4 What is the correct solution to looping?

The core rule is to never execute tools on file "patterns" like `*.fastq` or `*.bam` etc. The order of the files may not be what you expect; there may be files other than what you want that match the pattern; the practice will eventually cause devious problems.

As a rule, compile lists unique "roots" for the files that you want to process. In our example, the roots are:

```
SRR1553607
SRR1972917
```

Put these into a file, call it `ids.txt` or something that makes sense to you. Using the root build each file name out explicitly like so:

```
cat ids.txt | parallel echo cutadapt -l 20 {}_1.fastq -o {}_1.trimmed.fq
cat ids.txt | parallel echo cutadapt -l 20 {}_2.fastq -o {}_2.trimmed.fq
```

this code now prints:

```
cutadapt -l 20 SRR1553607_1.fastq -o SRR1553607_1.trimmed.fq
cutadapt -l 20 SRR1972917_1.fastq -o SRR1972917_1.trimmed.fq
cutadapt -l 20 SRR1553607_2.fastq -o SRR1553607_2.trimmed.fq
cutadapt -l 20 SRR1972917_2.fastq -o SRR1972917_2.trimmed.fq
```

There are two lines of code instead of one as before (though we could optimize that further). Note how explicit we are about everything!

The code indicates that there are paired-end files, what the file names are relative to the root. Reading the command explains the data layout and

structure. Compare that to `*.fastq` pattern - we don't know anything about the data there!

As Billy Mays[1] *once may have said: Wait there is more! Start using GNU parallel today and you can double or triple what you are getting out of it, you can apply the same approach to downloading the data, the entire script now looks like this:*

```
cat ids.txt | parallel fastq-dump -X 10000 --split-files {}
cat ids.txt | parallel cutadapt -l 20 {}_1.fastq -o {}_1.trimmed.fq
cat ids.txt | parallel cutadapt -l 20 {}_2.fastq -o {}_2.trimmed.fq
```

Just replace the content of ***`ids.txt`*** *with different run numbers and the same script will analyze a different experiment. The solution is short, generic, self descriptive and logical. What more would you ever need from code? Stop using* ***`for`*** *and* ***`if`*** *and be happier with what you are getting! Call now!*

In a nutshell, explicitly build your code from a "root" of known information rather than crawling the directory for file names. Sticking to this simple tip can save you a lot of trouble.

4.5 Is there more to this approach?

Yes. It is not just a "cute trick". It is a philosophy, a different programming approach that were advocating for.

You see when we write a `for` or an `if` in a programming language we are building a so called imperative program[2]: we ask the computer to first do one thing, second at each step isolate another thing, then with the selected object do a third thing.

When we write our commands via GNU Parallel, we follow a so called descriptive, functional programming[3] paradigm. We attempt to describe what we want with a pattern, then have the computer fill in the pattern and for the full command.

Depending on the problem, descriptive approaches can be more efficient mode of solving a problem and, frankly it is also an foray into a programming

[1] `https://www.youtube.com/watch?v=ZTpXh33Mbeg`
[2] `https://en.wikipedia.org/wiki/Imperative_programming`
[3] `https://en.wikipedia.org/wiki/Functional_programming`

paradigm that will make you smarter. The challenge of descriptive programming is that it forces you to solve the whole problem at once, rather than isolating it into little pieces.

4.6 How to use GNU parallel

To use GNU parallell, you have to understand what it does. Here are the essential tips to get started.

Suppose you have a file called `ids.txt` that contains

```
A
B
C
```

Suppose you wanted to generate this output:

```
Hello A
Hello B
Hello C
```

4.6.1 1. There are multiple ways to specify the input to GNU parallel

You could get the input piped in from a file:

```
cat ids.txt | parallel echo Hello {}
```

You could also explicitly specify the parameters at the command line by separating them from the command with 3 colons (`:::`):

```
parallel echo Hello {} ::: A B C
```

Finally you could also pass the file at the end when separated with four colons (`::::`)

```
parallel echo Hello {} :::: ids.txt
```

Each command above produces the same output.

4.6.2 2. Processing multiple input sources

Often information has multiple facets, samples, replicates, genders etc. GNU parallel has elegant methods to combine them all.

Perhaps you want all possible combinations:

```
parallel echo Hello {} and  {} ::: A B  ::: 1 2
```

Note how both parameters get substituted in each location:

```
Hello A 1 and A 1
Hello A 2 and A 2
Hello B 1 and B 1
Hello B 2 and B 2
```

Use the numbered construct `{1}` if you want to separate the inputs:

```
parallel echo Hello {1} and {2} ::: A B  ::: 1 2
```

will print:

```
Hello A and 1
Hello A and 2
Hello B and 1
Hello B and 2
```

You can take inputs one from each with `-link`:

```
parallel --link echo Hello {1} and {2} ::: A B  ::: 1 2
```

will print:

```
Hello A and 1
Hello B and 2
```

4.6.3 Where to go next

- A good introductory material [Gnu Parallel tutorial] online. tutorial-parallel[4]
- Practical examples: GNU Parallel in bioinformatics (#parallel) chapter in this book

[4] `https://www.gnu.org/software/parallel/parallel_tutorial.html`

4.7 The Rubber Ducky self-test

There is a concept called Rubber duck debugging[5] stating that:

> Many programmers have had the experience of explaining a problem to someone else, possibly even to someone who knows nothing about programming and then hitting upon the solution in the process of explaining the problem.

What this means is that, when you have problem, explain the problem to the ducky - with real words, as if you were actually talking to the ducky. Have a ducky ask very simple followup questions like: *"What could cause this?"*, *"How do you know it does not work?"*, *"Why does it print that?"*

Chances are that act of verbalizing the problem will make you both understand the problem better and with that often helps you solve it.

If you feel embarassed about the ducky, replace the rubber duck with a friend, or significant other. It seems to works best when the other person knows nothing about the problem - it removes their "responsibility" of having to solve the problem.

4.8 Help Ducky!

Each command below will do something radically different! Can you explain to the rubber ducky what each of the commands does? If not, then run the commands, test them test, understand them. Now come back here tomorrow,

[5] `https://en.wikipedia.org/wiki/Rubber_duck_debugging`

or in a week, and see that you can explain the rubber ducky the same thing again. If not, rinse and repeat.

Suppose there is a file called `ids.txt` that contains

```
A
B
C
```

The poor little, confused rubber ducky asks you kindly: "*Your Majesty, can you please explain to me what the code below does?*". Let's help the ducky out!

```
cat ids.txt | parallel echo {} {} {}

cat ids.txt | parallel echo {} {} {} > {}.txt

cat ids.txt | parallel "echo {} {} {} > {}.txt"

cat ids.txt | parallel "echo {} {} {} \> {}.txt"

cat ids.txt | parallel "echo echo {} {} {} \> {}.txt" | bash
```

Once you understand the above, go ahead and explore the various other ways to automate:

```
parallel "echo {1} = {2}" :::: ids.txt ::: FIZZ BUZZ
```

Check out the marvelous examples in the chapter titled GNU Parallel in bioinformatics

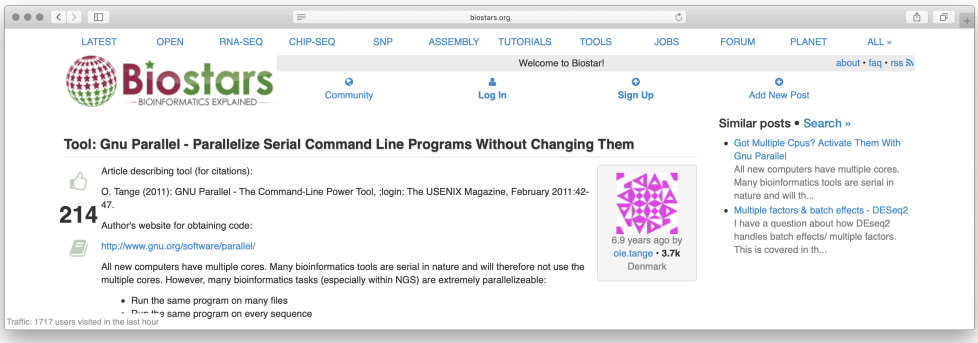
Chapter 5

GNU Parallel in bioinformatics

This section reproduces the Biostar post:

- Gnu Parallel - Parallelize Serial Command Line Programs Without Changing Them[1]

written by Ole Tange[2] who happens to also be the developer of GNU Parallel[3]. The article is reproduced here as it is perhaps the best demonstration of how parallel is used in the context of bioinformatcs analysis. The text has been edited in some location (for example the installation, instructions) and was annotated with notes.



[1] https://www.biostars.org/p/63816/
[2] https://github.com/ole-tange
[3] https://www.gnu.org/software/parallel/

5.1 Gnu Parallel - Parallelize Serial Command Line Programs Without Changing Them

Article describing tool (for citations):

- O. Tange (2011): GNU Parallel - The Command-Line Power Tool, ;login: The USENIX Magazine, February 2011:42-47.[4]

Author's website for obtaining code:

- `http://www.gnu.org/software/parallel/`

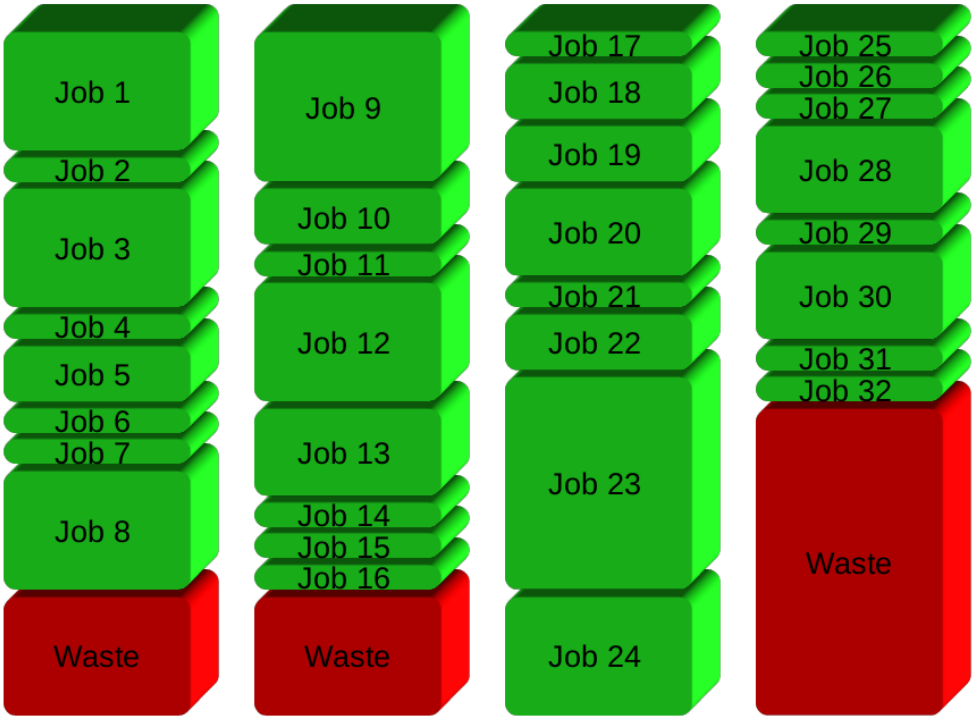
All new computers have multiple cores. Many bioinformatics tools are serial in nature and will therefore not use the multiple cores. However, many bioinformatics tasks (especially within NGS) are extremely parallelizeable:

- Run the same program on many files
- Run the same program on every sequence

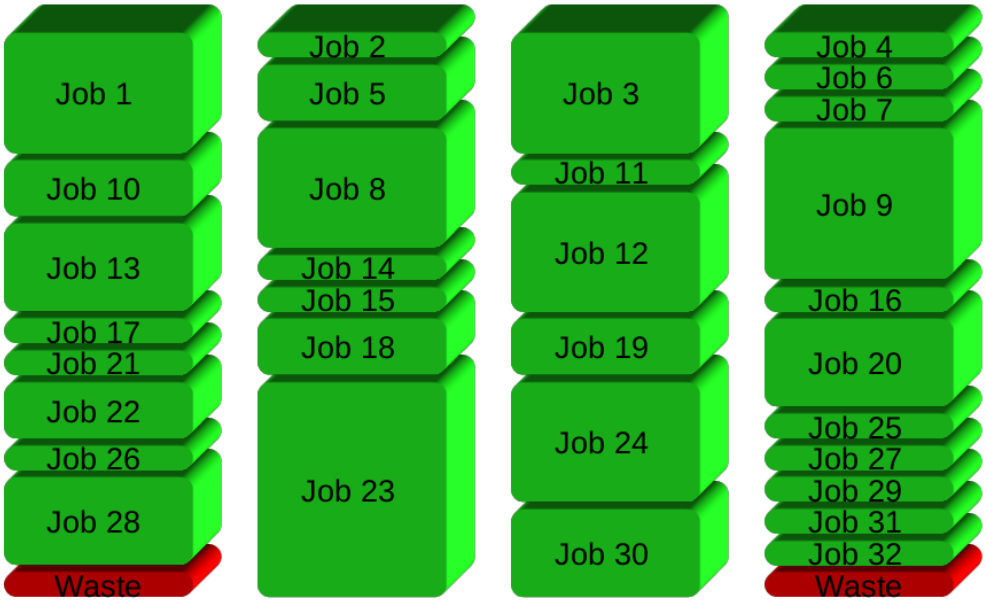
GNU Parallel is a general parallelizer and makes is easy to run jobs in parallel on the same machine or on multiple machines you have ssh access to.

If you have 32 different jobs you want to run on 4 CPUs, a straight forward way to parallelize is to run 8 jobs on each CPU:

[4] `https://www.usenix.org/publications/login/february-2011-volume-36-number-1/gnu-parallel-command-line-power-tool`



GNU Parallel instead spawns a new process when one finishes - keeping the CPUs active and thus saving time:



5.2 Installation

A personal installation does not require root access. It can be done in 10 seconds by doing this:

```
# Install GNU parallel.
conda install parallel

# Agree to terms (see later on the page)
parallel --citation
```

For other installation options see `http://git.savannah.gnu.org/cgit/parallel.git/tree/README`

Editor's note

to get help on the various feature of the tool see:

- `man parallel`
- GNU Parallel online documentation[5]

GNU parallel is **amazing** but there is a "nag-ware" aspect to it. Initially it nags you. The first time you run parallel it will ask you *to promise* that you will cite them. The requirement is annoying and overbearing. In my opinion, it also goes against the very principles of the GNU General Public license. You can bypass the "nag" by running the following with each run:

```
parallel --will-cite
```

or do it once with:

```
parallel --citation
```

This latter will stop the nagging - technically you have promised to cite the paper. You may want to hold two fingers crossed behind your back while pressing enter, so that you have an ironclad case of why the promise does not need to be kept.

Should you cite parallel in your work?

Think about it a bit in broader context. Unix was invented by *Dennis Ritchie* and *Ken Thompson*. Python was invented by *Guido van Rossum*. The sort method used within Python was invented by *Tim Peters* … and so on. What if every single person on this list started nagging us to cite them? Should we cite all the people that made a work "possible"? Where do we draw the line? I'll let you figure out the correct answer.

5.3 How to use GNU parallel in practice

5.3.1 EXAMPLE: Replace a for-loop

It is often faster to write a command using GNU Parallel than making a `for` loop:

```
for i in *gz; do
  zcat $i > $(basename $i .gz).unpacked
done
```

can be written as:

```
parallel 'zcat {} > {.}.unpacked' ::: *.gz
```

The added benefit is that the zcats are run in parallel - one per CPU core.

EXAMPLE: Parallelizing BLAT

This will start a blat process for each processor and distribute foo.fa to these in 1 MB blocks:

```
cat foo.fa | parallel --round-robin --pipe --recstart '>' 'blat -noHead genome.fa stdin >
```

EXAMPLE: Blast on multiple machines

Assume you have a 1 GB fasta file that you want blast, GNU Parallel can then split the fasta file into 100 KB chunks and run 1 jobs per CPU core:

```
cat 1gb.fasta | parallel --block 100k --recstart '>' --pipe blastp -evalue 0.01 -outfmt 6
```

If you have access to the local machine, server1 and server2, GNU Parallel can distribute the jobs to each of the servers. It will automatically detect how many CPU cores are on each of the servers:

```
cat 1gb.fasta | parallel -S :,server1,server2 --block 100k --recstart '>' --pipe blastp -
```

EXAMPLE: Run bigWigToWig for each chromosome

If you have one file per chomosome it is easy to parallelize processing each file. Here we do bigWigToWig for chromosome 1..19 + X Y M. These will run in parallel but only one job per CPU core. The {} will be substituted with arguments following the separator ':::'.

```
parallel bigWigToWig -chrom=chr{} wgEncodeCrgMapabilityAlign36mer_mm9.bigWig mm9_36mer_
```

EXAMPLE: Running composed commands

GNU Parallel is not limited to running a single command. It can run a composed command. Here is now you process multiple FASTA files using Biopieces (which uses pipes to communicate):

```
parallel 'read_fasta -i {} | extract_seq -l 5 | write_fasta -o {.}_trim.fna -x' ::: *.fna
```

See also: https://github.com/maasha/biopieces/wiki/HowTo#howto-use-biopieces-with-gnu-p

EXAMPLE: Running experiments

Experiments often have several parameters where every combination should be tested. Assume we have a program called experiment that takes 3 arguments: –age –sex –chr:

```
experiment --age 18 --sex M --chr 22
```

Now we want to run `experiment` for every combination of ages 1..80, sex M/F, chr 1..22+XY:

```
parallel experiment --age {1} --sex {2} --chr {3} ::: {1..80} ::: M F ::: {1..22} X
```

To save the output in different files you could do:

```
parallel experiment --age {1} --sex {2} --chr {3} '>' output.{1}.{2}.{3} ::: {1..8
```

But GNU Parallel can structure the output into directories so you avoid having thousands of output files in a single dir:

```
parallel --results outputdir experiment --age {1} --sex {2} --chr {3} ::: {1..80}
```

This will create files like outputdir/1/80/2/M/3/X/stdout containing the standard output of the job.

If you have many different parameters it may be handy to name them:

```
parallel --result outputdir --header : experiment --age {AGE} --sex {SEX} --chr {C
```

Then the output files will be named like outputdir/AGE/80/CHR/Y/SEX/F/stdout

If you want the output in a CSV/TSV-file that you can read into R or LibreOffice Calc, simply point –result to a file ending in .csv/.tsv:

```
parallel --result output.tsv --header : experiment --age {AGE} --sex {SEX} --chr
```

It will deal correctly with newlines in the output, so they will be read as newlines in R or LibreOffice Calc.

If one of your parameters take on many different values, these can be read from a file using ‘::::’

```
echo AGE > age_file
seq 1 80 >> age_file
parallel --results outputdir --header : experiment --age {AGE} --sex {SEX} --chr
```

If you have many experiments, it can be useful to see some experiments picked at random. Think of it as painting a picture by numbers: You can start from the top corner, or you can paint bits at random. If you paint bits at random, you will often see a pattern earlier, than if you painted in the structured way.

With `--shuf` GNU Parallel will shuffle the experiments and run them all, but in random order:

```
parallel --shuf --results outputdir --header : experiment --age {AGE} --sex {SEX}
```

EXAMPLE(advanced): Using GNU Parallel to parallelize you own scripts

Assume you have BASH/Perl/Python script called `launch`. It takes one arguments, ID:

```
launch ID
```

Using parallel you can run multiple IDs in parallel using:

```
parallel launch ::: ID1 ID2 ...
```

But you would like to hide this complexity from the user, so the user only has to do:

```
launch ID1 ID2 ...
```

You can do that using –shebang-wrap. Change the shebang line from:

```
#!/usr/bin/env bash
#!/usr/bin/env perl
#!/usr/bin/env python
```

to:

```
#!/usr/bin/parallel --shebang-wrap bash
#!/usr/bin/parallel --shebang-wrap perl
#!/usr/bin/parallel --shebang-wrap python
```

You further develop your script so it now takes an ID and a DIR:

```
launch ID DIR
```

You would like it to take multiple IDs but only one DIR, and run the IDs in parallel. Again just change the shebang line to:

```
#!/usr/bin/parallel --shebang-wrap bash
```

And now you can run:

```
launch ID1 ID2 ID3 ::: DIR
```

Learn more

See more examples: http://www.gnu.org/software/parallel/man.html

Watch the intro videos: https://www.youtube.com/playlist?list=PL284C9FF2488BC6D1

Walk through the tutorial once a year - your command line will love you for it: http://www.gnu.org/software/parallel/parallel_tutorial.html

Sign up for the email list to get support: https://lists.gnu.org/mailman/listinfo/parallel

#ilovefs

If you like GNU Parallel:

- Give a demo at your local user group/team/colleagues (remember to show them –bibtex)
- Post the intro videos on Reddit/Diaspora*/forums/blogs/ Identi.ca/Google+/Twitter/Fa lists
- Get the merchandise `https://www.gnu.org/s/parallel/merchandise.html`
- Request or write a review for your favourite blog or magazine
- Request or build a package for your favourite distribution (if it is not already there)
- Invite me for your next conference

When using programs that use GNU Parallel to process data for publication you should cite as per `parallel --citation`. If you prefer not to cite, contact me.

If GNU Parallel saves you money:

- (Have your company) donate to FSF `https://my.fsf.org/donate/`

Part II

LEARN SCRIPTING

Chapter 6

How to learn scripting

When you start out, errors will abound. That's ok, we make errors all the time as well. This page lists strategies you can employ to move forward faster.

6.1 Learn the Defense Against Dark Arts!

When you start out you'll have the following two problems all the time!

1. The **curse of doom**: My program does not run!
2. The **curse of agony**: Where is my file?

Learn to deal with each, and you'll be well on your way to become productive.

6.1.1 1.The curse of doom: My program does not run!

Suppose you read about a new method that works like this:

```
brangelina --shift 300 --report 200
```

But when you type it and press enter it prints this:

```
-bash: brangelina: command not found
```

See the *curse of doom* in your terminal:

The most likely explanation for getting the error:

- The program is not installed.
- You have not activated the proper conda environment.
- The name is not what you typed (Use TAB completion!).

Make sure that you have followed all instructions as described in the [Computer setup][#computer-setup] chapter.

6.1.2 2. The curse of agony: Where is my file?

Suppose you type:

```
cat bennifer.txt
```

then it prints:

```
cat: bennifer.txt: No such file or directory
```

See the *curse of agony* in your terminal:

The most likely explanation for getting the error:

- The file does not exist.
- The file exists but is somewhere else on your computer.
- The name is not what you typed (Use TAB completion!).

Visit the chapter called Unix Bootcamp[1] and make sure that you understand file paths, relative and absolute paths, and the Unix tree.

The reasons listed above solve 90% of problems (perhaps 100% for beginners).

6.2 3. Both errors at the same time

You may also end up with both curses at the same time:

```
brangelina --input bennifer.txt
```

will stop with the curse of doom whereas invoking it as:

```
brangelina < bennifer.txt
```

will lash you with the curse of agony. In these cases you fix first problem only to be notified of the second. More confusingly, depending on how you use the commands either of the errors may show up first, like so:

[1] `https://www.biostarhandbook.com/the-unix-bootcamp.html`

Needless to say, super annoying.

6.3 What if I am sure that I typed the commands correctly?

Often you are **absolutely certain** that you did everything right, yet you are still getting the error. In those cases the answer is that you **only think** you are doing it the way you are supposed to.

Surprisingly often your eyes and brain will conspire against you, they might not let you see that tiny mistake that you made. It happens to me all the time. Sometimes I see only what I want to see. In those cases the best course of action is to clear your view:

```
clear
```

This clears the terminal and mind and with that all preconceptions you might have about what the command should look like. Now *retype everything* from the very beginning! Build your commands out one piece at a time, keep pressing TAB and make use of autocomplete.

6.4 Visualize commands in their final form

When you set the `-x` flag in a script it will trace the commands as they are executed and reports them with a `+` sign up front:

```
set -uex
NAME=John
ls -l | wc -l > foo.txt
echo "All done $NAME" | wc -c
```

it prints:

```
+ NAME=John
+ ls -l
+ wc -l
+ echo 'All done John'
```

```
+ wc -c
      14
```

Note the differences in output relative to the original script:

- The `$NAME` parameter was substituted
- Actions connected with the pipe `|` are listed as separate commands.
- The redirection into `> foo.txt` is not shown.
- You can see the final output of the program `14`.

Tracing scripts in your terminal:

6.5 Visualize commands run via `parallel`

Suppose a file called `ids.txt` has the following content:

```
A
B
C
```

to view multistage command run via `parallel` such as

```
cat ids.txt | parallel "foo {} | bar > {}.txt"
```

you will need to do two things:

1. Place an `echo` in front of the command.
2. Protect each bash operator `|`, `>` with a backslash `\` (or single quotes`''`).

Your command will then look like this:

```
cat ids.txt | parallel "echo foo {} \| bar  \> {}.txt"
```

now, when executed the command will produce:

```
foo A | bar > A.txt
foo B | bar > B.txt
foo C | bar > C.txt
```

Verify the the commands printed would be valid commands. Copy and paste one line and verify what it works:

```
foo A | bar > A.txt
```

An example of running GNU parallel. In the screen `^D` means pressing Ctrl+D (end of file character):

6.6 Make sure that you understand parameter substitution

Strings written with no quotes or double quotes will be substituted. Strings written in single quotes will be produced verbatim:

```
NAME=John

echo Hello $NAME
echo "Hello $NAME"
echo 'Hello $NAME'
```

The output will be:

```
Hello John
Hello John
Hello $NAME
```

The official documentation on `bash` has information (too much even) in the section titled parameter expansion[2]. You don't need to know all the rules, just a few and only if you need more advanced features.

when you trace your scripts with `set -uex` you can *see the commands* as these are executed.

6.7 Don't reset the `PATH`

There are a few essential variables used by the shell. Out of these `PATH` (and a few others) have convenient names that you may accidentally reuse in your scripts. Specifically never do anything like:

```
# Don't do this!
PATH=foo.txt
```

The contents if the `PATH` variable are used by bash to figure out where to look for programs. When you reset the `PATH` your bash shell will be unable to find programs and you will get `command not found` error (the **curse of doom**) even for programs that *ought* to work. If you ever get this:

[2] `https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.html#Shell-Parameter-Expansion`

```
ls
-bash: ls: command not found
```

You just broke the `PATH`. Wear it as a badge of honor! *Look Ma! I broke my system so thoroughly that not even `ls` works no more!*

I did this a few times in my life - once while writing this book - I was overly eager to use a meaningful name in an example script and forgot the rule.

Solution: rename the variable something else than `PATH` for example:

```
FILE=foo.txt
```

6.8 Make sure that you can see the whitespace!

When reading data from file spaces may look like tabs and viceversa.

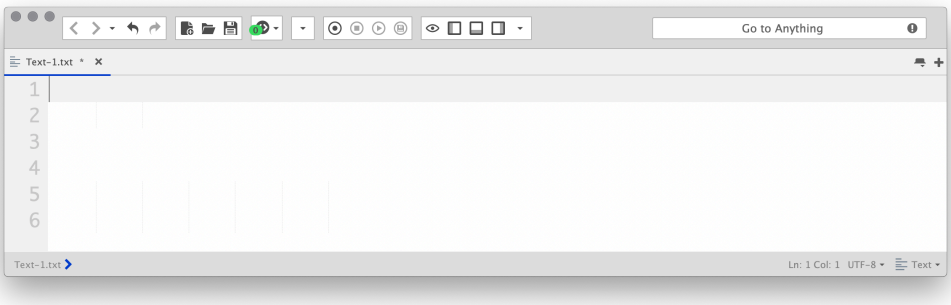
Chapter 7

Annoying problems

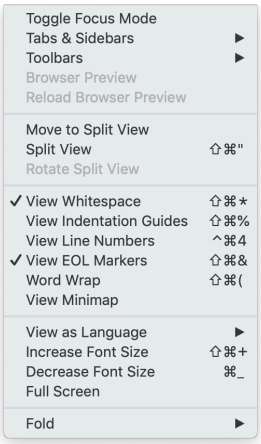
Some of these problems are more common for those using Unix via Windows, yet everyone will eventually need to troubleshoot similar situations.

7.1 Learn to visualize "whitespace"

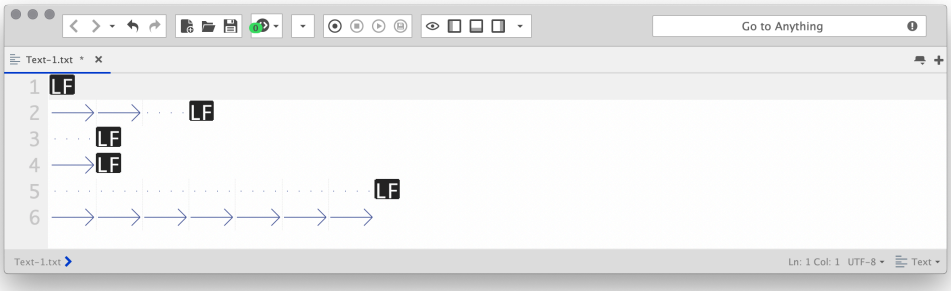
Whitespace is a "catch all" term that describes all "invisible" characters that would print out on paper as "nothing" (white paper). For example, what does this file contain? It looks like it is empty:



Let's turn on the visualization for both "whitespace" and "EOL" (end-of-line characters). The setting to view whitespaces depends on the Editor, below we're showing the choices in Komodo Edit here found under the top-level menu item called `View` :



We see that the file actually has quite a bit of content. Spaces are dots `.`, TABs are arrows `->`, and `LF` is the new line (line feed) character.



The "empty" file has `5` lines, `0` words and `51` characters:

```
cat Text-1.txt | wc
5       0      51
```

If this same file were in the so called **Windows text** format it would look like this:

In the Windows type of text each end-of-line is marked by two characters `CR` and ‘`LF` and (carriage return and line feed). The same “empty” file now has `5` lines, `0` words and `56` characters:

```
cat Text-1.txt | wc
5       0      56
```

7.2 Editor requirements

Your editor has (should have) the ability to:

1. Show you the difference between spaces and tabs.
2. Show you EOL (end of line) characters.
3. Switch line endings.
4. Show the line number for each line

7.3 Make sure the script is in Unix format!

The text files must have UNIX line endings to work properly in a UNIX environment.

Annoyingly enough the invisible characters that mark the end of a line endings are different on Windows than on Unix. For an easy to follow along summary read the

- The Great Newline Schism[1] post from the Coding Horror[2] blog.

[1] `https://blog.codinghorror.com/the-great-newline-schism/`
[2] `https://blog.codinghorror.com`

In a nutshell:

- Mac/ Unix use the `LF` (line feed) character to mark the end of the line.
- Windows uses two characters `LF` and `CR` (line feed and carriage return) to mark the end of the line. Thank you, Bill Gates, that is very helpful (sarcasm).

Make sure the file is in UNIX format, and if not convert it into UNIX line endings.

7.4 Use Notepad++ or EditPlus on Windows

Notepad++[3] is one of the simplest, easiest to use editors for Windows. Another excellent choice is EditPlus[4] We will use Notepad++[5] to demonstrate certain actions.

[3] `https://notepad-plus-plus.org/`
[4] `https://www.editplus.com/`
[5] `https://notepad-plus-plus.org/`

Part III

AWK PROGRAMMING

Chapter 8

Programming concepts

As soon as you start performing data analysis you will run into unanticipated problems:

- Existing software tools can rarely do all steps.
- You may need to bridge small formatting differences with simple transformations.
- You may need to compute values based on other values.

Often transformation themselves are simple yet impossible to do by hand due to the volume of the data. Not being able to get past these may bring your project to a halt; the ability to write programs becomes essential.

The good news is that the programming skills required to get past many of the obstacles you'll face are not particularly challenging to learn! Often writing just a few lines or applying a formula is enough. You can get a lot of mileage out of fundamental skills.

8.1 What kind of programming languages are in use?

Computer programs fall into two main categories:

- Compiled: the program is a standalone executable that can be run on any other computer of the same type.

- Interpreted: the program requires the presence of another, specific programming language the computer that executes the code.

8.2 What programming languages do Bioinformaticians use?

Modern bioinformatics uses mainly the following languages:

- `C` - Compiled. Efficient and powerful but with high barriers of entry.
- `Java` - Interpreted. Allows building more complex software with less effort.
- `Python` - Interpreted. Simple and straightforward to learn, it can be substantially slower than C and Java in some cases.
- `Awk` - Interpreted. Ancient and somewhat simplistic, considered obsolete, its usage patterns match the bioinformatics data formats.
- `Perl` - Interpreted. Formerly the most popular language for doing bioinformatics – by today it has fallen out of favor and has been largely supplanted by Python.

- `R` - Interpreted. The `R` language is used mainly for data analysis and visualization.

8.3 Which programming language should I learn?

If you are new to the field we would recommend starting with `Python`.

At the same time, we recommend familiarizing yourself with `awk` as it both extremely simple and it can be a convenient tool for everyday data analysis. In this book, we will demonstrate the use of `awk` to solve various problems.

8.4 How are programming languages used in bioinformatics?

We have noticed the following trends in our work, each successively more complex than the previous:

1. Apply an existing, documented data analysis protocol
2. Develop a novel computational method for a specific, peculiar facet of a problem.
3. Develop a better data analysis protocol.

Chapter 9

Programming with Awk

- Enlightened with Perl? You've ascended to a *PerlMonk*!
- Blazing trails with Python? You've become a *Pythoneer*!
- Following your dreams with Awk? You are *Awk-ward*!

To perform the examples download the features file that we used in the Data analysis with Unix chapter.

```
wget -nvc http://data.biostarhandbook.com/data/SGD_features.tab
```

9.1 Why is Awk popular with bioinformaticians?

Awk operates on one line at a time, and the typical awk program is so short that it can be listed at the command line. Thus the command become "explicit" we know everything that takes place. Compare:

```
cat SGD_features.tab | python analyze.py
```

with:

```
cat SGD_features.tab | awk -F '\t' ' $2=="ORF" { print $4, $2, $10, $11, $11-$10 }
```

In a terminal it would look like this:

In the first example we would need to obtain and investigate the program `analyze.py` to understand what it does. In the latter example, we see that

actions are listed in the command. Once you know a bit of `awk` you immediately see that a few columns are selected, some columns are rearranged, and a third column is created by subtracting column 11 from column 10.

If the program were more complicated putting it on one line might be unreadable, thus typically we only use `awk` when the programs we need to write are exceedingly simple.

Note how in the animated screencast I build the awk program from a smaller, simpler program, continously refining it, adding more into it. Press the up-arrow to save typing the command again. When I am happy with the command I will place it into a script.

As for the example above, while we can use `cut` to select columns we couldn't use it to rearrange them or to compute new or other values. Here is a comparison of `cut` and `awk`:

```
# Matching ORFs and printing the start/ends.
cat SGD_features.tab | cut -f 2,4,10,11 | grep ORF | head
```

produces:

```
ORF YAL069W      335    649
ORF YAL068W-A    538    792
ORF YAL068C     2169   1807
ORF YAL067W-A   2480   2707
ORF YAL067C     9016   7235
...
```

now with `awk` we can rearrange columns and compute even compute new ones (start-end coordinate)

```
cat SGD_features.tab | awk -F '\t' ' $2=="ORF" { print $4, $2, $10, $11, $11-$10 } ' | head
```

to print:

```
YAL069W     ORF   335    649    314
YAL068W-A   ORF   538    792    254
YAL068C     ORF  2169   1807   -362
YAL067W-A   ORF  2480   2707    227
YAL067C     ORF  9016   7235  -1781
...
```

The expression `$2=="ORF" { print $4, $2, $10, $11, $11-$10 }` is an awk program.

9.2 How does Awk work?

An awk program works on a line by line basis and for each line of a file it attempts the following:

```
awk 'CONDITION { ACTIONS }'
```

For each line it `awk` tries to match the `CONDITION`, and if that condition matches it performs the `ACTIONS`. Do note the curly brackets and the quotes. Note that multiple conditions and actions may be present:

```
awk 'CONDITION1 { ACTIONS1 } CONDITION2 { ACTIONS2 } CONDITION3 { ACTIONS3 }'
```

The simplest condition is no condition at all. The simplest action is no action at all. This makes the simplest awk program `awk '{}'` that you can run:

```
cat SGD_features.tab | awk '{}'
```

Of course that does not do anything to your data. A simple action could be `print`. Awk automatically splits the input (see the caveats of default splitting later) into variables named `$1`, `$2`, `$3` etc.

```
cat SGD_features.tab | awk '{ print $5 }'  | head
```

The output is not what you might expect:

```
chromosome
335
chromosome
538
chromosome
1
62
```

It is a mix of values from different columns.

9.3 How is the output split into columns?

Superficially the default operation of awk appears to split lines into columns by any whitespace (spaces, tabs) – in reality, it does a lot more than that. It also collapses consecutive white spaces into a single one. This collapsing behavior leads to a subtle difference that often does not matter - but when it does, the most devious and subversive of errors may occur.

With the default splitting behavior when we split the lines containing:

```
A B
A  B
```

we end up with the same result column 1 is `A` and column 2 is `B` for each line. In some cases, this is what we might have wanted.

In general, and especially when processing tabular data, we don't want this behavior. Imagine a tab-delimited file where tabs indicate columns. Say in one row two empty columns follow each other. The default awk splitting behavior would collapse these two tabs into one and subsequently shift the rest of the columns by one, producing values from the wrong column. So reading column 10 would sometimes give you the value in column 10 but other times the value in column 11, 12, etc. Needless to say, this is highly undesirable behavior.

In general, you always want to specify the splitting character when using awk:

```
awk -F '\t'
```

The flag above will set the field separator to be the "tab" character. If you are confident that your file does not have consecutive tabs, for example, some file formats always require the presence of data, a zero or empty marker perhaps, then you would not need to set the field separator.

9.4 How can I write more complicated Awk programs?

While awk is extremely useful for simple processing or quick answers, we would advise you to avoid writing highly complex awk (or bash) programs altogether.

Awk is strongest when the whole program is simple enough to be listed (and understood) on the command line. When written that way we can visually inspect it and we can ideally tell precisely what the command does. If the commands were in a separate file, we'd have to investigate that separately, and that would take away most of the readability of the process.

That being said when there are more actions these can be placed into a file like so:

```
CONDITION1 {
    ACTIONS1
}

CONDITION2 {
    ACTIONS2
}
```

and we can run the program through `awk` via the `-f` flag:

```
cat mydata.txt | awk -f myprogram.awk  | head
```

9.5 What are some special awk variables I should know about?

When awk runs, many variables are set beyond the column variables `$1`, `$2` etc.

- `$0` is the original line.
- `NF` number of fields in the current line (number of columns that awk recognizes)
- `NR` number of records, the number of lines processed (line number)
- `OFS` output fields separator, the character placed between items when printed

As an example usage:

```
cat SGD_features.tab | awk '{ print NR, NF }'  | head
```

This prints:

```
1 30
2 9
3 40
4 9
5 14
6 40
...
```

The first number is the line number; the second number is how many columns does awk think that the file has. Look what happens if you split by tab characters:

```
cat SGD_features.tab | awk -F '\t' '{ print NR, NF }'  | head
```

you will get:

```
1 16
2 16
3 16
4 16
5 16
6 16
...
```

Note how splitting with different field separator makes `awk` think that the file has different number of columns. Read the previous entry if you are unsure what is happening!

9.6 Are there any unique Awk patterns I should know about?

Awk has special patterns called `BEGIN` and `END` to run specific tasks only once.

```
cat SGD_features.tab | awk ' BEGIN { print "Hello!" } END { print "Goodbye!" } '  | head
```

produces:

```
Hello!
Goodbye!
```

9.7 How can I build more complex conditions in Awk?

Make sure to use double quotes within patterns as to not conflict with the single quotes that the whole awk program is enclosed within.

You can use the following constructs.

- > or < comparisons: `'$1 < 60 { print 41 }'`
- ==, != for equal, not equal: `'{ $1 == "ORF" { print $1 }'`
- ~, !~ for pattern match, or pattern no match (regular expressions): `' $1 ~ "YALC|YALW" { print $1 }'`
- another way to match patterns is to enclose the pattern within /, this construct is called a regexp constant. These may be simpler to write, less putzing around with quotes: `' $1 ~ /YALC|YALW/ { print $1 }'`

9.8 How can I format the output of Awk?

Producing properly formatted output is always one of pressing concern. By default, printing with awk will concatenate outputs with the value of the `OFS` variable (Output Field Separator):

```
echo | awk ' { print 1,2,3 }'
```

by default will print:

```
1 2 3
```

changing `OFS` can put other characters between the elements, perhaps a `TAB`:

```
echo | awk '{ OFS="\t"; print 1,2,3 }'
```

this will now print:

```
1    2    3
```

At some point, you will need precise control of how values are formatted. Then you will need to use the so-called formatted print `printf` where the output is specified via different parameters, a formatting string, and a list of values. For example, a formatting string of `%d %d %d` will mean that the data should be formatted as three integers:

```
echo | awk '{ printf("%d %d %d",1,2,3) }'
```

Above we are echoing "nothing", to get the pipeline going. Typically you would have streamed opened on the left.

The `printf` approach to formatting values comes from the `C` language and has been implemented identically in numerous languages (such as Awk), so

when you learn it for one language, the same concepts will apply for all others. Here is a fancier printing example:

```
echo | awk '{ printf("A=%0.3f B=%d C=%03d",1,2,3) }'
```

will print:

```
A=1.000  B=2  C=003
```

See more printf examples[1]

9.9 How can I learn more about Awk?

- Awk One-Liners[2] one-liner awk scripts
- Awk One-Liners Explained[3] an explanation for the one-line awk scripts
- Awk CheatSheet[4] - a resource that lists all awk related variables
- Gnu Awk Manual[5] - a comprehensive, but reasonably complete resource

[1] https://www.gnu.org/software/gawk/manual/html_node/Printf-Examples.html#Printf-Examples

[2] http://www.catonmat.net/blog/wp-content/uploads/2008/09/awk1line.txt

[3] http://www.catonmat.net/blog/awk-one-liners-explained-part-one/

[4] http://www.catonmat.net/blog/awk-nawk-and-gawk-cheat-sheet/

[5] http://www.gnu.org/software/gawk/manual/gawk.html

Chapter 10

Programming with BioAwk

10.1 What is BioAwk?

BioAwk[1] is the brainchild of Heng Li, who after an online discussion where people were complaining about not having an 'awk' that is bioinformatics-aware, decided to take the source of awk and modify it to add the following features:

1. FASTA/FASTQ records are handled as if they were a single line.
2. Added new internal variables for known data formats. For example instead of having to remember that the 6th column is the `CIGAR` string in a sam file a variable `$cigar` will exist when reading SAM files.
3. Added a number of bioinformatics-relevant functions such as `revcomp` to produce the reverse complement (and others).

When bioawk is installed globally you may consult the help with:

```
man bioawk
```

To turn on the special parsing of formats pass the `-c FORMAT` flag where `FORMAT` can be one of the following:

- `sam`, `bed`, `gff`, `vcf`, `fastx`

You can get a quick reminder of the special variable names that bioawk knows about:

[1] `https://github.com/lh3/bioawk`

```
bioawk -c help
```

it will print:

```
bed:
   1:chrom 2:start 3:end 4:name 5:score 6:strand 7:thickstart 8:thickend 9:rgb 10:blockc
sam:
   1:qname 2:flag 3:rname 4:pos 5:mapq 6:cigar 7:rnext 8:pnext 9:tlen 10:seq 11:qual
vcf:
    1:chrom 2:pos 3:id 4:ref 5:alt 6:qual 7:filter 8:info
gff:
   1:seqname 2:source 3:feature 4:start 5:end 6:score 7:filter 8:strand 9:group 10:attri
fastx:
    1:name 2:seq 3:qual 4:comment
```

What this states is that when you are using the `sam` format you may use variable names such as `qname`, `flag` etc.

For example:

```
cat SRR1972739_1.fastq | bioawk -c fastx ' { print $name } ' | head
```

prints:

```
SRR1972739.1
SRR1972739.2
SRR1972739.3
...
```

to compute the GC content of each sequence:

```
cat SRR1972739_1.fastq | bioawk -c fastx ' { print $name, gc($seq) } ' | head
```

Above we are making use of bioawk specific variables `$name` and `$seq` as well as the bioawk specific function `gc` (GC content) to produce:

```
SRR1972739.1    0.306931
SRR1972739.2    0.49505
SRR1972739.3    0.415842
SRR1972739.4    0.514851
```

10.2 How to use Awk/BioAwk for processing data?

The applications are broad and useful for a wide variety of tasks.

10.2.1 Find alignments where the CIGAR string has deletions:

```
wget http://data.biostarhandbook.com/bam/demo.bam
samtools view -f 2 demo.bam | awk '$6 ~ /D/ { print $6 }' | head
samtools view -f 2 demo.bam | bioawk -c sam '$cigar ~ /D/ { print $cigar }' | head
```

10.2.2 Select alignments with an edit distance of 3

```
samtools view -f 2 demo.bam | awk '/NM:i:3/ { print $6, $12, $13, $14 }' | head
```

10.2.3 How many bases have coverage over 150x?

```
samtools depth demo.bam | awk '$3 > 150 { print $0 }' | wc -l
```

10.2.4 Mutate one part of the whole line

One really handy feature of awk is that allows mutating a part of the line and printing it out in otherwise original format:

```
samtools depth demo.bam | awk ' $3 <= 3 { $1="LO" } $3 > 3 { $1="HI" } { print $0 }'
```

Note how we assign to `$1` but then print `$0`. Here is what the code above does:

```
LO 46 3
HI 47 4
HI 48 4
...
```

This seemingly odd feature is * handy* when the line has many columns, and we need to overwrite just one while still keeping it in the same format. Imagine that you had a file with 100 columns and wanted to change only one column. It would be tedious and error-prone to piece back together the entire line. Using the technique shown above you can mutate just one column then print the line.

Chapter 11

Terminal multiplexing

For large datasets, the time required to complete the analyses will become longer. It becomes essential to ensure that the jobs do not get interrupted by closing a terminal, accidentally logging off etc.

A terminal multiplexer is a software that is designed to keep programs active even when you are not logged into the computer. You have several options to choose from we recommend tmux[1] or GNU Screen[2]

In our examples we will use the `tmux` terminal multiplexer.

11.1 How do I use terminal multiplexing?

First, you need to install `tmux` either globally or into your current conda environment:

```
conda install tmux
```

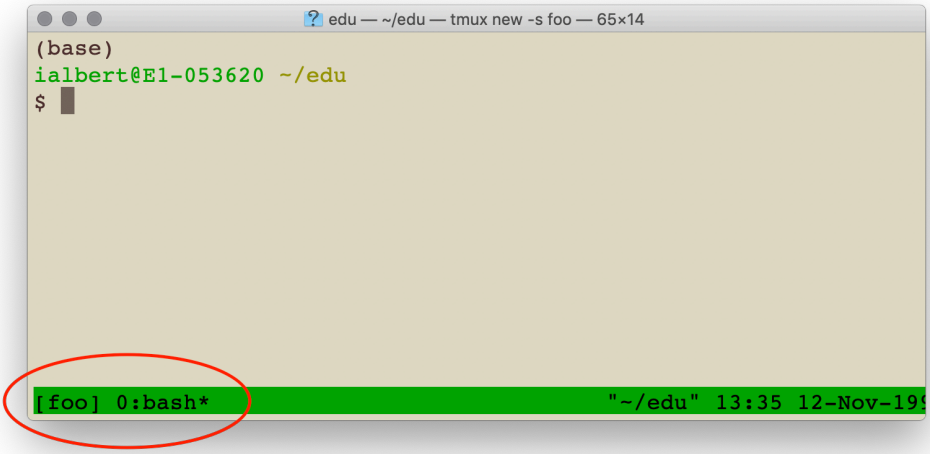
11.2 What does a terminal multiplexer do?

Start a new session with `tmux`, we will name it `foo`:

```
tmux new -s foo
```

[1] `https://www.google.com/search?q=tmux`
[2] `https://www.google.com/search?q=%22gnu%20screen%22`

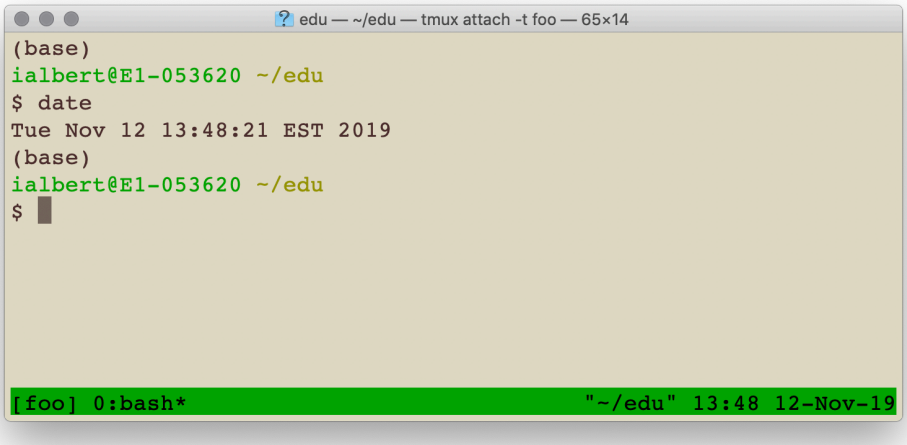
You may also ignore the naming and just run `tmux` as a command to get a number like `0`, `1` etc automatically assigned as a name). In general, it is a good habit to name your sessions, later you could have a hard time figuring out which session contains what. Our command above created a session named `foo` that on my system is displayed like so:



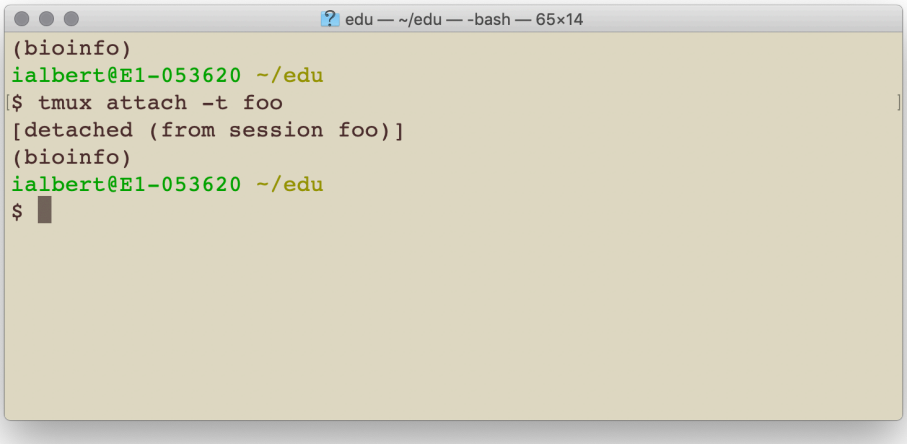
Now run any command in the terminal:

```
date
```

our window contains:



Let's demonstrate what `tmux` is used for. Press `ctrl-b` then the `d` key. The `ctrl-b` keypress is the so called "prefix" that triggers `tmux` commands. In this case the `d` triggers the "detach" command. We now get the following view:



The previous session is still running in the background. We can now exit the terminal, log off this computer, or do anything else (other than shutting the

computer down), and commands running in the session will remain active. Note: the detaching will also happen automatically if the connection with the terminal gets interrupted. To re-activate the previous session, we first list all available sessions:

```
tmux ls
```

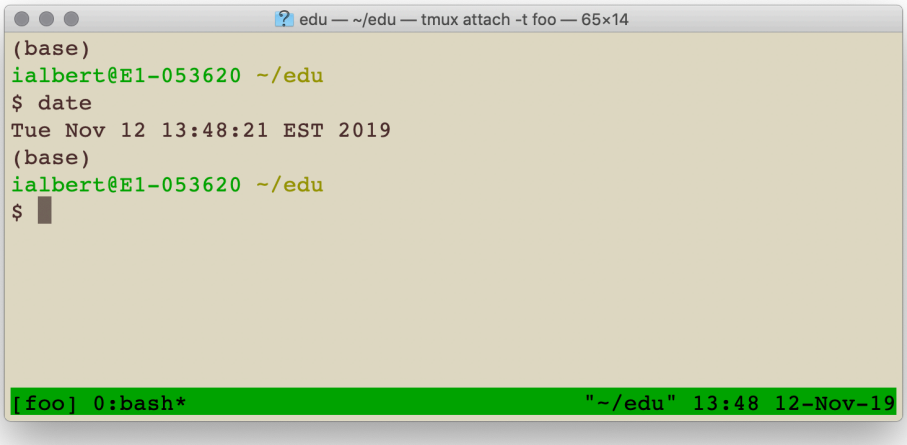
it prints:

```
foo: 1 windows (created Tue Nov 12 13:34:59 2019)
```

We can see the sessions named `foo` here. Note that you may have multiple sessions; moreover, each session may have multiple windows within it (see later). In our case, we have one session and one window. To reattach the session to the current terminal type:

```
tmux attach -t foo
```

and voila, the sessions is back to what it looked like before.

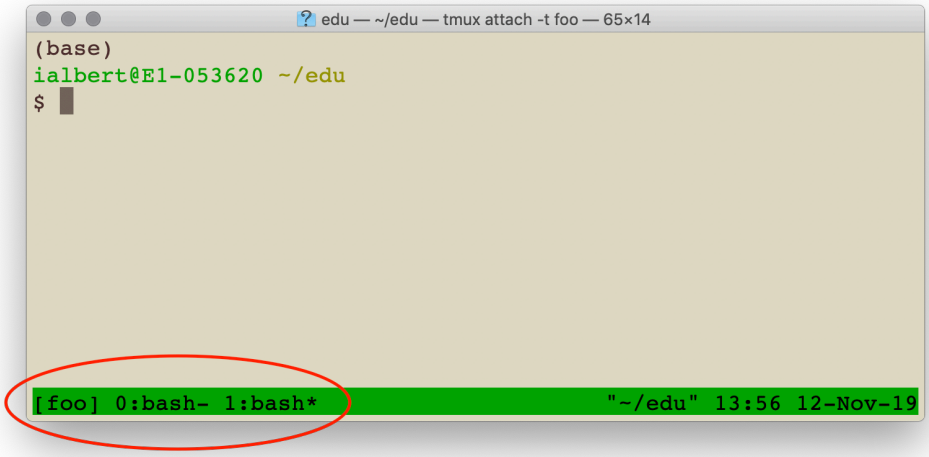


11.3 How to create multiple windows in a session?

Each session may contain several terminals that are all attached and detached together. The essential commands to manage these windows are:

- `ctrl-b, c` create window within a session
- `ctrl-b, n` go to next window in a session
- `ctrl-b, p` go to previous window in a session
- `ctrl-b, x` kills the window (it is not a detach!)

When you have multiple windows, the active window within a session is indicated with a star `*`. For example, here are two windows within a session:



11.4 Minimalist `tmux` survival guide

I found that I can get everything done with just three commands to launch `tmux`:

- `tmux ls` show what sessions I have
- `tmux new -s foo` create a named session
- `tmux attach -t foo` attach the named session

Within `tmux` I pretty much use only another three commands:

- `ctrl-b, c` create window within a session
- `ctrl-b, n` go to next window in a session
- `ctrl-b, d` detach tmux session

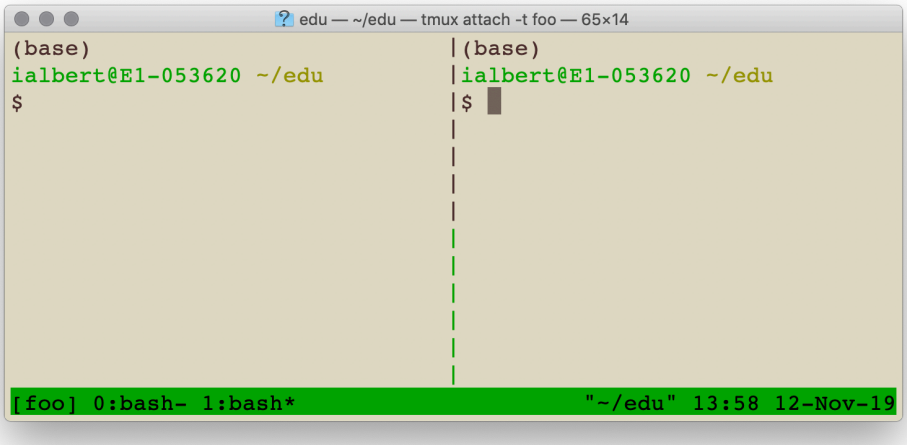
that's it.

11.4.1 What are panel commands?

While I prefer to use window commands some people like to see the commands in parallel in so-called "panels"

- `ctrl-b, %` split the screen in half from left to right
- `ctrl-b, "` split the screen in half from top to bottom
- `ctrl-b, x` kill the current pane
- `ctrl-b, <arrow key>` switch to the pane in whichever direction you press
- `ctrl-b, d` detach from tmux, leaving everything running in the background

Here we have window split into two panels:



11.5 Where can I learn more?

- Search Google for tmux[3] for detailed guides on how to get the most out of it.

[3] `https://www.google.com/search?q=tmux`

Part IV

ADVANCED CONCEPTS

Chapter 12

Advanced shell concepts

> Important notice: We ARE NOT advocating devoting substantial effort to learning bash as a primary programming language! If you want to learn to program start with Python. In this section we cover tips that can assist you with running programs in bash.

12.1 Self documenting bash programs

A well designed script is self documenting. For example if the program needs parameters to run and it did not get them, it should print a usage reminder.

What I like to do is to print usage instructions when the script gets no parameters. For example if I have a script and ran:

```
bash getdata.sh
```

it should print:

```
*** This script needs arguments to work! ***

Usage:
   getdata.sh  PRJNUM COUNT LIMT

Parameters:
  PRJNUM = SRA Bioproject number
  COUNT = how many sequencing runs to download
```

```
  LIMIT = how many reads to extract per sequencing run

Example:
   getdata.sh PRJN2234 1000 5
```

To achieve this effect add the following to the beginning of the script:

```
if [ $# -eq 0 ]; then
    echo
    echo "*** This script needs arguments to work! ***"
    echo
    echo "Usage:"
    echo "   getdata.sh  PRJNUM COUNT LIMT"
    echo
    echo "Parameters:"
    echo "  PRJNUM = SRA Bioproject number"
    echo "  COUNT = how many sequencing runs to download"
    echo "  LIMIT = how many reads to extract per sequencing run"
    echo
    echo "Example:"
    echo "   getdata.sh PRJN2234 1000 5"
    echo
    exit 1
fi
```

12.2 Better output redirection

Bash has one input stream (`stdin`) and two output streams (`stdout` and `stderr`). Typically

1. The output of the command will go to standard output (`stdout`)
2. The error messages will go to standard error (`stderr`)

The reason we say "typically" because the two streams are choices that the program developer has when printing output. There is nothing specifically "error" about them, it is just channel 1 and channel 2, where channel 2 is used for errors. It may happen that a program writes important messages to channel 2 (error channel).

By default both `stdout` and `stderr` are directed to the terminal. For example I might type:

```
ls * foo
```

it prints:

```
ls: foo: No such file or directory
A.txt
```

So we have a file called `A.txt` but nothing called `foo`. Next type:

```
ls * foo > B.txt
```

What will it print?

```
ls: foo: No such file or directory
```

Now, see how the line `A.txt` is not printed anymore. A new file called `B.txt` was created that contains the file name. But we still see the error message. We have redirected the so called standard output to a file, but the standard error is still printed to the terminal. To redirect the error stream use `2>`:

```
ls * foo > B.txt 2> err.txt
```

It is the same as writing

```
ls * foo 1> B.txt 2> err.txt
```

but we almost never use `1>` as long as `>` does the job.

12.3 Dealing with the copious output of tools

Most bioinformatics tools print huge amounts of information to the standard error stream.

```
bwa index somebigfasta.fa
```

will generate pages and pages of information. Find out out which stream the program prints to and redirect it to a file.

```
bwa index somebigfasta.fa 1> stdout.txt 2> stderr.txt
```

Later if you hit on an error you can investigate the files for messages.

12.4 What does matching pattern do?

What happens when we write:

```
# Create some text files.
touch A1.txt A2.txt A10.txt

# What happens here?
ls *.txt
```

to see that add `set -x` flag to your script:

```
+ ls A1.txt A10.txt A2.txt
A1.txt  A10.txt A2.txt
```

note how at the time of execution the command names are "filled" in by bash. Note also that the order is alphabetical, `A10.txt` comes before `A2.txt`

The important thing to remember here that it is that `*.txt` is not passed to `ls` at all. `ls` does not **know** that you wrote `*.txt`. Bash intercepts the pattern `*.txt` and expands it into a list of files, then passes these files to `ls`.

When you have lots of files it is possible for the shell to match so many names to make a command so long that it fails in various ways.

12.5 How can I manipulate file paths in bash?

Often we have to manipulate file names within bash to remove various parts of it. Perhaps we want to remove the directory name, or to keep only the file name, or to keep only the file name with no extension, or to remove the extension, etc. It never ceases to amaze me just how many times we need to modify file names.

Bash offers this functionality via a series of ludicrous, hard-to-remember pattern matching operators that I never remember and have to look up every time (I do it here or I Google them):

```
FILE=/A/B/C.txt.gz
echo $FILE
```

as expected it prints:

```
/A/B/C.txt.gz
```

Removes the directory from the name and keep the file name only use the `basename` shell command like so:

```
FILE=/A/B/C.txt.gz
NAME=$(basename ${FILE})
echo $NAME
```

prints:

```
C.txt.gz
```

To chop off the rightmost extension use the operator '%.*':

```
FILE=/A/B/C.txt.gz
CHOP=${FILE%.*}
echo $CHOP
```

it will print (note the `.gz` is missing):

```
/A/B/C.txt
```

Now to get only the extension `gz` use the operator `##*.` like so:

```
FILE=/A/B/C.txt.gz
CHOP=${FILE##*.}
echo $CHOP
```

it prints:

```
gz
```

To remove all path elements until the leftmost dot us the `#*.` operator:

```
FILE=/A/B/C.txt.gz
CHOP=${FILE#*.}
echo $CHOP
```

it prints:

```
txt.gz
```

If you need a transformation that is not offered as a single bash transformation, you can usually build it by successively applying separate instructions in order.

12.6 How can I get dynamic data into a variable?

This is a static declaration:

```
NAME=World
echo "Hello $NAME"
```

We can also execute a command and save its results in the variable – if we enclose it with backticks (‘ symbols):

```
VALUE=`ls -1 | wc -l`
echo "The number of files is $VALUE"
```

12.7 How can I assign default values to a variable?

To assign default values to variables use the construct:

```
FOO=${VARIABLE:-default}
```

For example, to set the `LIMIT` variable to the first parameter `$1` or `1000` if that was not specified:

```
LIMIT=${1:-1000}
```

12.8 How can I build more complicated scripts?

The best strategy to get started writing scripts is to write them such that instead of executing the commands you echo (print) them out on the screen. Put an `echo` command in front of each real command. So instead of:

```
fastqc $SOMETHING
```

write:

```
echo fastq $SOMETHING
```

Printing to the screen will allow you first to see what the commands will look like when executed. If it seems right, to execute your commands, pipe them to the `bash` command again.

```
bash myscript.sh | bash
```

Here is as an example another script that now takes a project is and a run count that can be run as:

```
bash getproject.sh PRJNA257197 10
```

We want our script to fetch 10 sequencing datasets from the project `PRJNA257197`, then run a trimmomatic quality trim on each, and finally, run a fastqc report on the resulting files.

Here is our script:

```
#
# Usage: getproject.sh PRJN NUM
#
# Example: bash getproject.sh PRJNA257197 3
#
# This program downloads a NUM number of experiments from an SRA Bioproject
# identified by the PRJN number then runs FastQC quality reports before
# and after trimming the data for quality.

# Immediately stop on errors.
set -uex

# The required parameter is the run id.
PRJN=$1

# How many experimental runs to get.
NUM=$2

# What is the conversion limit for each data.
LIMIT=10000

# Get the run information for the project.
# You may comment out this line if you already have an info.csv.
```

```
esearch -db sra -query $PRJN | efetch -format runinfo > runinfo.csv

# Keep only the ids that match SRR number.
cat runinfo.csv | cut -f 1 -d ',' | grep SRR | head -${NUM} > ids.csv

# We place an echo before each command to see what will get executed when it
cat ids.csv | parallel echo fastq-dump --split-files -X ${LIMIT} {}

# Generate the commands for fastqc.
cat ids.csv | parallel echo fastqc {}_1.fastq {}_2.fastq

# Generate the commands for trimmomatic.
# Here we run it with the -baseout flag to generatevfile extension of .fq.
cat ids.csv | parallel echo trimmomatic PE -baseout {}.fq {}_1.fastq {}_2.fast

# Run FastQC on the new data that matches the *.fq extension.
cat ids.csv |  parallel echo fastqc {}_1P.fq {}_2P.fq
```

Here is how you run the script:

```
bash getproject.sh PRJNA257197 1
```

Note – this script does not execute anything. Because we placed an `echo` before each command. Instead it generates the commands and prints them on the screen, like so:

```
fastq-dump --split-files -X 10000 SRR1972917
fastqc SRR1972917_1.fastq SRR1972917_2.fastq
trimmomatic PE -baseout SRR1972917.fq SRR1972917_1.fastq SRR1972917_2.fastq SLID
fastqc SRR1972917_1P.fq SRR1972917_2P.fq
...
```

You could now copy-paste each command separately and verify that they work step-by-step. Keep fixing up the script until everything checks out. You can now either remove the `echo` commands or pipe the output of the script through bash:

```
bash getproject.sh PRJNA257197 5 | bash
```

The key component to remember is that you want to *see* what gets executed.

Having a clear understanding of what the script will help you troubleshoot any errors that crop up.

For more ideas, see the article Unofficial Bash Strict Mode[1]

12.9 Everybody is wrong about the cat

Some scripting "gurus" can be smug about a construct they call the Useless Use of CaT[2] (UUOC). UUOC is defined as:

> Useless use of cat (UUOC) is common Unix jargon for command line constructs that only provide a function of convenience to the user. This is also referred to as "cat abuse". The activity of fixing instances of UUOC is sometimes called demoggification.

A typical example would be rewriting

```
cat filename | grep foo
```

as

```
grep foo < filename
```

Now some commands, for example `grep` can be written like:

```
grep foo filename
```

Occasionally (and quite rarely) there are legitimate uses of these latter constructs that affect command speed execution. But in general they only make your commands harder to read. When writing script consistency, clarity and simplicity should be primary objectives. For that reason, we strongly advocate constructs that may look like UUOC such as

```
cat filename | grep foo
```

These constructs makes it very clear that the data flows left to right, allow for easier chaining and substitution of different commands. We just wanted you to know about the useless cat[3] in case someone criticise you for it.

[1] http://redsymbol.net/articles/unofficial-bash-strict-mode/
[2] https://en.wikipedia.org/wiki/Cat_%28Unix%29
[3] https://en.wikipedia.org/wiki/Cat_%28Unix%29

Chapter 13

Mastery with Makefiles

Whereas writing scripts is useful and helps immensely, there may come a time when you want to keep all related functionality in one, single file, yet only run a sub-section of it. A software tool named `make` and its corresponding `Makefile` are designed to do that (and a lot more as well).

`make` will allow you to set up a veritable *command and control center* for your data analysis. With `Makefiles` you will be able to keep track of what you have done and how.

In our opinion the path to reproducible research starts with `Makefiles`. Perhaps at some point in the future when scientists submit their data analysis results, they will also be required to also attach their `Makefile`s that describe the steps they took in a clear and explicit way.

> **Note**
>
> In this section we will only cover a tiny featureset of `make`. There is a lot more to the tool than what we mention. The great thing is that even using a sliver of make's functionality can help you immensely.
>
> As you become more confident you may want to read more about the tool and what it can do.

13.1 What is a Makefile?

You can think of makefiles as scripts with "targets".

By default `make` will print both the command that gets executed and the result of the command itself. You can turn of this behavior if you want to, though usually you do want to see what gets executed. Initially it can be a bit confusing to see both the command and the output, but your eyes will get used to it soon.

A `Makefile` is a text file (by default a file called `Makefile` will be used) that lists commands grouped by so called "rules", which you can think of as groups of commands that should be executed together. We indicate the commands that belong to a rule by indenting them via a TAB character:

```
foo:
    echo Hello John!

bar:
    echo Hello Jane!
    echo Hello Everyone!
```

Note that in the above code you *MUST* have tabs in front of the commands for the makefile to work with `make`. If we viewed tab characters as '–> it would look like so:

```
foo:
--> echo Hello John!

bar:
--> echo Hello Jane!
--> echo Hello Everyone!
```

This `Makefile` has two target rules: `foo` and `bar`. Once you create this file you may type:

```
make foo
```

and it will print

```
Hello John!
```

or you can type:

```
make bar
```

and it will print:

```
Hello Jane!
Hello Everyone!
```

Note how `make` automatically executes the `Makefile` as long as it was called with that name. If we wanted to execute a different makefile we would have to specify it like so `make -f somefile`.

And that's it. That's all you need to know to be productive with makefiles!

13.2 Why do I get the rule error?

If you get the error:

```
make: *** No rule to make target 'foo:'. Stop.
```

it means that your target is not in the Makefile. Make sure the target is in the file and the file is saved.

13.3 Why do I get the separator error?

If you get the error:

```
Makefile:3: *** missing separator.  Stop.
```

it means that your tasks are not tab delimited. Unfortunately this is a common and endemic problem, many text editors will add spaces when you press a TAB. You need to find the setting in your editor that allows you to enter a TAB character. In Komodo Edit for example you check a checkbox in: `Preferences -> Editor -> Indentation -> Prefer TAB character`.

13.4 How can I tell if my Makefile has tabs or spaces?

The `-t` flag to cat prints tab characters:

```
cat -t Makefile
```

will print a ^I character for tabs. Like this:

```
foo:
^Iecho Hello John!

bar:
^Iecho Hello Jane!
^Iecho Hello Everyone!
```

it will look like this:

13.5 Is there more to make?

Yes there is. A lot more.

For many stopping here when covering `make` sounds like heresy - but what about dependency management, automatic execution over many targets etc. Fear not. Using `Makefiles` as shown above already puts you a long way ahead. This one feature of Makefile is sufficient to greatly simplify your life and allows you package your entire workflow into a single file.

For every analysis you should create a single `Makefile` and perhaps other scripts with it. Then you basically have a "central command script" from which you can orchestrate your analysis.

```
make fastqc
make align
make plots
...
```

13.6 Why doesn't my Makefile work?

One common error to check for is neglecting to put tabs in front of the commands. Many text editors insist on inserting a number of space characters even if you press TAB and not SPACE on your keyboard.

View the whitespaces in your editor!

13.7 Can I use bash shell constructs in a Makefile?

The syntax used inside makefiles may superficially look like a bash shell, but only the commands themselves are passed and executed as a bash shell.

There is a fine distinction there - one that you may not ever need to bridge - if you do, remember that a `Makefile` is not a `bash` script. For example, variables in Makefiles are specified the same way as in bash but will be filled in before it gets passed to bash:

```
NAME=Jane
hello:
    echo Hello ${NAME}
```

13.8 Why does my Makefile print every command?

By default `make` echoes every command. This helps you see what it is trying to do. To turn that off add a `@` symbol to the line:

```
NAME=Jane
hello:
    @echo Hello ${NAME}
```

You can just as well leave it on though, it is good to know what is going on.

13.9 Why does my Makefile stop on an error?

Remember, that is a good thing. You really want to stop on any error so you can examine and fix it.

In the odd case when you really don't care whether a command succeeded or not and you don't want the make to stop add the `-` sign in front of the command:

```
hello:
    -cp this that
```

```
    echo "Done"
```

Will always print "Done" even if the file `this` is not present.

13.10 Why is the first action executed?

If you run `make` with no task label it will execute the first label it sees. It is a good idea to place usage information there so that you can remind yourself of it just by running `make`.

```
NAME=Jane
usage:
    @echo "Usage: make hello, goodbye, ciao"

hello:
    @echo Hello ${NAME}
```

13.11 What was `make` developed for?

We only covered the basic features of `make`, the ones that we believe will give you the most benefits with the least amount of overhead.

Rest assured that `make` has several other very handy features. One important feature is that it can track the so-called dependencies between files. It can be told to automatically detect which sections need to be re-run when some of the inputs change.

In our personal observation these features of `make` are only useful when data analysis is performed at very large scale. In our typical work we almost never need to specify these dependencies. That keeps the use of `make` a lot simpler. But we encourage you to explore and learn more about `make`.

A good start would be the document titled

- Automation and Make[1] from Software Carpentry[2].

[1] `http://swcarpentry.github.io/make-novice/`
[2] `http://swcarpentry.github.io/make-novice/`

But remember - the simplest feature that you learned above already saves you a lot of effort.

We have beautiful reproducible workflows in make, without using any of the advanced features. Unless you use these advanced features a lot you'll tend to have to re-learn them. Often not worth the time.

13.12 Are there are alternatives to Makefiles?

There are but most people don't need them.

The concept of `make` goes back a long way and in time, many alternatives have been proposed. One that seems to have taken off with bioinformaticians is `snakemake`

- snakemake[3]

If you find yourself having to develop overly complex Makefiles it might be worth exploring the alternatives. On our end we get a lot done with simple and uncomplicated Makefiles.

[3] `https://bitbucket.org/snakemake/snakemake/wiki/Home`

Part V

APPENDIX

Chapter 14

Appendix Notes

Content in the appendix has been collected from external sources.

Links to the original content are provided, though not all content may be available anymore.

- Useful one-line scripts for sed
- Useful one-line scripts for awk

Chapter 15

Useful one-line scripts for awk

Compiled by Eric Pement pemente@northpark.edu[1] version 0.22

Latest version of this file is usually at:

- http://www.student.northpark.edu/pemente/awk/awk1line.txt

15.1 EXPLANATIONS

For detailed explanations of each pattern see:

- https://catonmat.net/awk-one-liners-explained-part-one

15.2 USAGE

```
awk '/pattern/ {print "$1"}'    # standard Unix shells
```

15.3 FILE SPACING

```
# double space a file
awk '1;{print ""}'
```

[1] mailto:pemente@northpark.edu

```
awk 'BEGIN{ORS="\n\n"};1'

# double space a file which already has blank lines in it. Output file
# should contain no more than one blank line between lines of text.
# NOTE: On Unix systems, DOS lines which have only CRLF (\r\n) are
# often treated as non-blank, and thus 'NF' alone will return TRUE.
awk 'NF{print $0 "\n"}'

# triple space a file
awk '1;{print "\n"}'
```

15.4 NUMBERING AND CALCULATIONS

```
# precede each line by its line number FOR THAT FILE (left alignment).
# Using a tab (\t) instead of space will preserve margins.
awk '{print FNR "\t" $0}' files*

# precede each line by its line number FOR ALL FILES TOGETHER, with tab.
awk '{print NR "\t" $0}' files*

# number each line of a file (number on left, right-aligned)
# Double the percent signs if typing from the DOS command prompt.
awk '{printf("%5d : %s\n", NR,$0)}'

# number each line of file, but only print numbers if line is not blank
# Remember caveats about Unix treatment of \r (mentioned above)
awk 'NF{$0=++a " :" $0};{print}'
awk '{print (NF? ++a " :" :"") $0}'

# count lines (emulates "wc -l")
awk 'END{print NR}'

# print the sums of the fields of every line
awk '{s=0; for (i=1; i<=NF; i++) s=s+$i; print s}'

# add all fields in all lines and print the sum
```

```
awk '{for (i=1; i<=NF; i++) s=s+$i}; END{print s}'

# print every line after replacing each field with its absolute value
awk '{for (i=1; i<=NF; i++) if ($i < 0) $i = -$i; print }'
awk '{for (i=1; i<=NF; i++) $i = ($i < 0) ? -$i : $i; print }'

# print the total number of fields ("words") in all lines
awk '{ total = total + NF }; END {print total}' file

# print the total number of lines that contain "Beth"
awk '/Beth/{n++}; END {print n+0}' file

# print the largest first field and the line that contains it
# Intended for finding the longest string in field #1
awk '$1 > max {max=$1; maxline=$0}; END{ print max, maxline}'

# print the number of fields in each line, followed by the line
awk '{ print NF ":" $0 } '

# print the last field of each line
awk '{ print $NF }'

# print the last field of the last line
awk '{ field = $NF }; END{ print field }'

# print every line with more than 4 fields
awk 'NF > 4'

# print every line where the value of the last field is > 4
awk '$NF > 4'
```

15.5 TEXT CONVERSION AND SUBSTITUTION

```
# IN UNIX ENVIRONMENT: convert DOS newlines (CR/LF) to Unix format
awk '{sub(/\r$/,"");print}'  # assumes EACH line ends with Ctrl-M
```

```
# IN UNIX ENVIRONMENT: convert Unix newlines (LF) to DOS format
awk '{sub(/$/,"\r");print}

gawk -v BINMODE="w" '1' infile >outfile

# Use "tr" instead.
tr -d \r <infile >outfile          # GNU tr version 1.22 or higher

# delete leading whitespace (spaces, tabs) from front of each line
# aligns all text flush left
awk '{sub(/^[ \t]+/, ""); print}'

# delete trailing whitespace (spaces, tabs) from end of each line
awk '{sub(/[ \t]+$/, "");print}'

# delete BOTH leading and trailing whitespace from each line
awk '{gsub(/^[ \t]+|[ \t]+$/,"");print}'
awk '{$1=$1;print}'        # also removes extra space between fields

# insert 5 blank spaces at beginning of each line (make page offset)
awk '{sub(/^/, "     ");print}'

# align all text flush right on a 79-column width
awk '{printf "%79s\n", $0}' file*

# center all text on a 79-character width
awk '{l=length();s=int((79-l)/2); printf "%"(s+l)"s\n",$0}' file*

# substitute (find and replace) "foo" with "bar" on each line
awk '{sub(/foo/,"bar");print}'        # replaces only 1st instance
gawk '{$0=gensub(/foo/,"bar",4);print}'  # replaces only 4th instance
awk '{gsub(/foo/,"bar");print}'       # replaces ALL instances in a line

# substitute "foo" with "bar" ONLY for lines which contain "baz"
awk '/baz/{gsub(/foo/, "bar")};{print}'

# substitute "foo" with "bar" EXCEPT for lines which contain "baz"
```

```
awk '!/baz/{gsub(/foo/, "bar")};{print}'

# change "scarlet" or "ruby" or "puce" to "red"
awk '{gsub(/scarlet|ruby|puce/, "red"); print}'

# reverse order of lines (emulates "tac")
awk '{a[i++]=$0} END {for (j=i-1; j>=0;) print a[j--] }' file*

# if a line ends with a backslash, append the next line to it
# (fails if there are multiple lines ending with backslash...)
awk '/\\$/ {sub(/\\$/,""); getline t; print $0 t; next}; 1' file*

# print and sort the login names of all users
awk -F ":" '{ print $1 | "sort" }' /etc/passwd

# print the first 2 fields, in opposite order, of every line
awk '{print $2, $1}' file

# switch the first 2 fields of every line
awk '{temp = $1; $1 = $2; $2 = temp}' file

# print every line, deleting the second field of that line
awk '{ $2 = ""; print }'

# print in reverse order the fields of every line
awk '{for (i=NF; i>0; i--) printf("%s ",i);printf ("\n")}' file

# remove duplicate, consecutive lines (emulates "uniq")
awk 'a !~ $0; {a=$0}'

# remove duplicate, nonconsecutive lines
awk '! a[$0]++'                     # most concise script
awk '!($0 in a) {a[$0];print}'      # most efficient script

# concatenate every 5 lines of input, using a comma separator
# between fields
awk 'ORS=%NR%5?",":"\n"' file
```

15.6 SELECTIVE PRINTING OF CERTAIN LINES

```
# print first 10 lines of file (emulates behavior of "head")
awk 'NR < 11'

# print first line of file (emulates "head -1")
awk 'NR>1{exit};1'

# print the last 2 lines of a file (emulates "tail -2")
awk '{y=x "\n" $0; x=$0};END{print y}'

# print the last line of a file (emulates "tail -1")
awk 'END{print}'

# print only lines which match regular expression (emulates "grep")
awk '/regex/'

# print only lines which do NOT match regex (emulates "grep -v")
awk '!/regex/'

# print the line immediately before a regex, but not the line
# containing the regex
awk '/regex/{print x};{x=$0}'
awk '/regex/{print (x=="" ? "match on line 1" : x)};{x=$0}'

# print the line immediately after a regex, but not the line
# containing the regex
awk '/regex/{getline;print}'

# grep for AAA and BBB and CCC (in any order)
awk '/AAA/; /BBB/; /CCC/'

# grep for AAA and BBB and CCC (in that order)
awk '/AAA.*BBB.*CCC/'

# print only lines of 65 characters or longer
```

```
awk 'length > 64'

# print only lines of less than 65 characters
awk 'length < 64'

# print section of file from regular expression to end of file
awk '/regex/,0'
awk '/regex/,EOF'

# print section of file based on line numbers (lines 8-12, inclusive)
awk 'NR==8,NR==12'

# print line number 52
awk 'NR==52'
awk 'NR==52 {print;exit}'        # more efficient on large files

# print section of file between two regular expressions (inclusive)
awk '/Iowa/,/Montana/'             # case sensitive
```

15.7 SELECTIVE DELETION OF CERTAIN LINES

```
# delete ALL blank lines from a file (same as "grep '.' ")
awk NF
awk '/./'
```

CREDITS AND THANKS:

Special thanks to Peter S. Tillier for helping me with the first release of this FAQ file.

For additional syntax instructions, including the way to apply editing commands from a disk file instead of the command line, consult:

"sed & awk, 2nd Edition," by Dale Dougherty and Arnold Robbins O'Reilly, 1997 "UNIX Text Processing," by Dale Dougherty and Tim O'Reilly Hayden Books, 1987 "Effective awk Programming, 3rd Edition." by Arnold Robbins O'Reilly, 2001

To fully exploit the power of awk, one must understand "regular expressions." For detailed discussion of regular expressions, see "Mastering Regular Expressions, 2d edition" by Jeffrey Friedl (O'Reilly, 2002).

The manual ("man") pages on Unix systems may be helpful (try "man awk", "man nawk", "man regexp", or the section on regular expressions in "man ed"), but man pages are notoriously difficult. They are not written to teach awk use or regexps to first-time users, but as a reference text for those already acquainted with these tools.

USE OF '' IN awk SCRIPTS: For clarity in documentation, we have used the expression '' to indicate a tab character (0x09) in the scripts. All versions of awk, even the UNIX System 7 version should recognize the '' abbreviation.

Chapter 16

Useful one-line scripts for sed

Compiled by Eric Pement - pemente[at]northpark[dot]edu version 5.5

Latest version of this file (in English) is usually at:

- http://sed.sourceforge.net/sed1line.txt
- http://www.pement.org/sed/sed1line.txt

16.1 EXPLANATIONS

For detailed explanations of each pattern see:

- https://catonmat.net/sed-one-liners-explained-part-one

16.2 FILE SPACING

```
# double space a file
sed G

# double space a file which already has blank lines in it. Output file
# should contain no more than one blank line between lines of text.
sed '/^$/d;G'

# triple space a file
```

```
sed 'G;G'

# undo double-spacing (assumes even-numbered lines are always blank)
sed 'n;d'

# insert a blank line above every line which matches "regex"
sed '/regex/{x;p;x;}'

# insert a blank line below every line which matches "regex"
sed '/regex/G'

# insert a blank line above and below every line which matches "regex"
sed '/regex/{x;p;x;G;}'
```

16.3 NUMBERING

```
# number each line of a file (simple left alignment). Using a tab (see
# note on '\t' at end of file) instead of space will preserve margins.
sed = filename | sed 'N;s/\n/\t/'

# number each line of a file (number on left, right-aligned)
sed = filename | sed 'N; s/^/     /; s/ *\(.\{6,\}\)\n/\1  /'

# number each line of file, but only print numbers if line is not blank
sed '/./=' filename | sed '/./N; s/\n/ /'

# count lines (emulates "wc -l")
sed -n '$='
```

16.4 TEXT CONVERSION AND SUBSTITUTION

```
# IN UNIX ENVIRONMENT: convert DOS newlines (CR/LF) to Unix format.
sed 's/.$//'               # assumes that all lines end with CR/LF
sed 's/^M$//'              # in bash/tcsh, press Ctrl-V then Ctrl-M
```

```
sed 's/\x0D$//'          # works on ssed, gsed 3.02.80 or higher

# IN UNIX ENVIRONMENT: convert Unix newlines (LF) to DOS format.
sed "s/$/`echo -e \\\r`/"            # command line under ksh
sed 's/$'"/`echo \\\r`/"             # command line under bash
sed "s/$/`echo \\\r`/"               # command line under zsh
sed 's/$/\r/'                        # gsed 3.02.80 or higher

# IN DOS ENVIRONMENT: convert Unix newlines (LF) to DOS format.
sed "s/$//"                          # method 1
sed -n p                             # method 2

# IN DOS ENVIRONMENT: convert DOS newlines (CR/LF) to Unix format.
# Can only be done with UnxUtils sed, version 4.0.7 or higher. The
# UnxUtils version can be identified by the custom "--text" switch
# which appears when you use the "--help" switch. Otherwise, changing
# DOS newlines to Unix newlines cannot be done with sed in a DOS
# environment. Use "tr" instead.
sed "s/\r//" infile >outfile       # UnxUtils sed v4.0.7 or higher
tr -d \r <infile >outfile          # GNU tr version 1.22 or higher

# delete leading whitespace (spaces, tabs) from front of each line
# aligns all text flush left
sed 's/^[ \t]*//'                # see note on '\t' at end of file

# delete trailing whitespace (spaces, tabs) from end of each line
sed 's/[ \t]*$//'                # see note on '\t' at end of file

# delete BOTH leading and trailing whitespace from each line
sed 's/^[ \t]*//;s/[ \t]*$//'

# insert 5 blank spaces at beginning of each line (make page offset)
sed 's/^/     /'

# align all text flush right on a 79-column width
sed -e :a -e 's/^.\{1,78\}$/ &/;ta'  # set at 78 plus 1 space

# center all text in the middle of 79-column width. In method 1,
```

```
# spaces at the beginning of the line are significant, and trailing
# spaces are appended at the end of the line. In method 2, spaces at
# the beginning of the line are discarded in centering the line, and
# no trailing spaces appear at the end of lines.
sed  -e :a -e 's/^.\{1,77\}$/ & /;ta'                  # method 1
sed  -e :a -e 's/^.\{1,77\}$/ &/;ta' -e 's/\( *\)\1/\1/'  # method 2

# substitute (find and replace) "foo" with "bar" on each line
sed 's/foo/bar/'           # replaces only 1st instance in a line
sed 's/foo/bar/4'          # replaces only 4th instance in a line
sed 's/foo/bar/g'             # replaces ALL instances in a line
sed 's/\(.*\)foo\(.*foo\)/\1bar\2/' # replace the next-to-last case
sed 's/\(.*\)foo/\1bar/'            # replace only the last case

# substitute "foo" with "bar" ONLY for lines which contain "baz"
sed '/baz/s/foo/bar/g'

# substitute "foo" with "bar" EXCEPT for lines which contain "baz"
sed '/baz/!s/foo/bar/g'

# change "scarlet" or "ruby" or "puce" to "red"
sed 's/scarlet/red/g;s/ruby/red/g;s/puce/red/g'   # most seds
gsed 's/scarlet\|ruby\|puce/red/g'                # GNU sed only

# reverse order of lines (emulates "tac")
# bug/feature in HHsed v1.5 causes blank lines to be deleted
sed '1!G;h;$!d'               # method 1
sed -n '1!G;h;$p'             # method 2

# reverse each character on the line (emulates "rev")
sed '/\n/!G;s/\(.\)\(.*\n\)/&\2\1/;//D;s/.//'

# join pairs of lines side-by-side (like "paste")
sed '$!N;s/\n/ /'

# if a line ends with a backslash, append the next line to it
sed -e :a -e '/\\$/N; s/\\\n//; ta'
```

```
# if a line begins with an equal sign, append it to the previous line
# and replace the "=" with a single space
sed -e :a -e '$!N;s/\n=/ /;ta' -e 'P;D'

# add commas to numeric strings, changing "1234567" to "1,234,567"
gsed ':a;s/\B[0-9]\{3\}\>/,&/;ta'                     # GNU sed
sed -e :a -e 's/\(.*[0-9]\)\([0-9]\{3\}\)/\1,\2/;ta'  # other seds

# add commas to numbers with decimal points and minus signs (GNU sed)
gsed -r ':a;s/(^|[^0-9.])([0-9]+)([0-9]{3})/\1\2,\3/g;ta'

# add a blank line every 5 lines (after lines 5, 10, 15, 20, etc.)
gsed '0~5G'                   # GNU sed only
sed 'n;n;n;n;G;'              # other seds
```

16.5 SELECTIVE PRINTING OF CERTAIN LINES

```
# print first 10 lines of file (emulates behavior of "head")
sed 10q

# print first line of file (emulates "head -1")
sed q

# print the last 10 lines of a file (emulates "tail")
sed -e :a -e '$q;N;11,$D;ba'

# print the last 2 lines of a file (emulates "tail -2")
sed '$!N;$!D'

# print the last line of a file (emulates "tail -1")
sed '$!d'                     # method 1
sed -n '$p'                   # method 2

# print the next-to-the-last line of a file
sed -e '$!{h;d;}' -e x         # for 1-line files, print blank line
```

```
sed -e '1{$q;}' -e '$!{h;d;}' -e x  # for 1-line files, print the line
sed -e '1{$d;}' -e '$!{h;d;}' -e x  # for 1-line files, print nothing

# print only lines which match regular expression (emulates "grep")
sed -n '/regexp/p'           # method 1
sed '/regexp/!d'             # method 2

# print only lines which do NOT match regexp (emulates "grep -v")
sed -n '/regexp/!p'          # method 1, corresponds to above
sed '/regexp/d'              # method 2, simpler syntax

# print the line immediately before a regexp, but not the line
# containing the regexp
sed -n '/regexp/{g;1!p;};h'

# print the line immediately after a regexp, but not the line
# containing the regexp
sed -n '/regexp/{n;p;}'

# print 1 line of context before and after regexp, with line number
# indicating where the regexp occurred (similar to "grep -A1 -B1")
sed -n -e '/regexp/{=;x;1!p;g;$!N;p;D;}' -e h

# grep for AAA and BBB and CCC (in any order)
sed '/AAA/!d; /BBB/!d; /CCC/!d'

# grep for AAA and BBB and CCC (in that order)
sed '/AAA.*BBB.*CCC/!d'

# grep for AAA or BBB or CCC (emulates "egrep")
sed -e '/AAA/b' -e '/BBB/b' -e '/CCC/b' -e d    # most seds
gsed '/AAA\|BBB\|CCC/!d'                        # GNU sed only

# print paragraph if it contains AAA (blank lines separate paragraphs)
# HHsed v1.5 must insert a 'G;' after 'x;' in the next 3 scripts below
sed -e '/./{H;$!d;}' -e 'x;/AAA/!d;'

# print paragraph if it contains AAA and BBB and CCC (in any order)
```

```
sed -e '/./{H;$!d;}' -e 'x;/AAA/!d;/BBB/!d;/CCC/!d'

# print paragraph if it contains AAA or BBB or CCC
sed -e '/./{H;$!d;}' -e 'x;/AAA/b' -e '/BBB/b' -e '/CCC/b' -e d
gsed '/./{H;$!d;};x;/AAA\|BBB\|CCC/b;d'         # GNU sed only

# print only lines of 65 characters or longer
sed -n '/^.\{65\}/p'

# print only lines of less than 65 characters
sed -n '/^.\{65\}/!p'        # method 1, corresponds to above
sed '/^.\{65\}/d'            # method 2, simpler syntax

# print section of file from regular expression to end of file
sed -n '/regexp/,$p'

# print section of file based on line numbers (lines 8-12, inclusive)
sed -n '8,12p'               # method 1
sed '8,12!d'                 # method 2

# print line number 52
sed -n '52p'                 # method 1
sed '52!d'                   # method 2
sed '52q;d'                 # method 3, efficient on large files

# beginning at line 3, print every 7th line
gsed -n '3~7p'               # GNU sed only
sed -n '3,${p;n;n;n;n;n;n;}' # other seds

# print section of file between two regular expressions (inclusive)
sed -n '/Iowa/,/Montana/p'             # case sensitive
```

16.6 SELECTIVE DELETION OF CERTAIN LINES

```
# print all of file EXCEPT section between 2 regular expressions
```

```
sed '/Iowa/,/Montana/d'

# delete duplicate, consecutive lines from a file (emulates "uniq").
# First line in a set of duplicate lines is kept, rest are deleted.
sed '$!N; /^\(.*\)\n\1$/!P; D'

# delete duplicate, nonconsecutive lines from a file. Beware not to
# overflow the buffer size of the hold space, or else use GNU sed.
sed -n 'G; s/\n/&&/; /^\([ -~]*\n\).*\n\1/d; s/\n//; h; P'

# delete all lines except duplicate lines (emulates "uniq -d").
sed '$!N; s/^\(.*\)\n\1$/\1/; t; D'

# delete the first 10 lines of a file
sed '1,10d'

# delete the last line of a file
sed '$d'

# delete the last 2 lines of a file
sed 'N;$!P;$!D;$d'

# delete the last 10 lines of a file
sed -e :a -e '$d;N;2,10ba' -e 'P;D'   # method 1
sed -n -e :a -e '1,10!{P;N;D;};N;ba'  # method 2

# delete every 8th line
gsed '0~8d'                           # GNU sed only
sed 'n;n;n;n;n;n;n;d;'                # other seds

# delete lines matching pattern
sed '/pattern/d'

# delete ALL blank lines from a file (same as "grep '.' ")
sed '/^$/d'                           # method 1
sed '/./!d'                           # method 2

# delete all CONSECUTIVE blank lines from file except the first; also
```

```
# deletes all blank lines from top and end of file (emulates "cat -s")
sed '/./,/^$/!d'       # method 1, allows 0 blanks at top, 1 at EOF
sed '/^$/N;/\n$/D'      # method 2, allows 1 blank at top, 0 at EOF

# delete all CONSECUTIVE blank lines from file except the first 2:
sed '/^$/N;/\n$/N;//D'

# delete all leading blank lines at top of file
sed '/./,$!d'

# delete all trailing blank lines at end of file
sed -e :a -e '/^\n*$/{$d;N;ba' -e '}'  # works on all seds
sed -e :a -e '/^\n*$/N;/\n$/ba'     # ditto, except for gsed 3.02.*

# delete the last line of each paragraph
sed -n '/^$/{p;h;};/./{x;/./p;}'
```

16.7 SPECIAL APPLICATIONS

```
# remove nroff overstrikes (char, backspace) from man pages. The 'echo'
# command may need an -e switch if you use Unix System V or bash shell.
sed "s/.`echo \\\b`//g"   # double quotes required for Unix environment
sed 's/.^H//g'          # in bash/tcsh, press Ctrl-V and then Ctrl-H
sed 's/.\x08//g'        # hex expression for sed 1.5, GNU sed, ssed

# get Usenet/e-mail message header
sed '/^$/q'            # deletes everything after first blank line

# get Usenet/e-mail message body
sed '1,/^$/d'          # deletes everything up to first blank line

# get Subject header, but remove initial "Subject: " portion
sed '/^Subject: */!d; s///;q'

# get return address header
sed '/^Reply-To:/q; /^From:/h; /./d;g;q'
```

```
# parse out the address proper. Pulls out the e-mail address by itself
# from the 1-line return address header (see preceding script)
sed 's/ *(.*)//; s/>.*//; s/.*[:<] *//'

# add a leading angle bracket and space to each line (quote a message)
sed 's/^/> /'

# delete leading angle bracket & space from each line (unquote a message)
sed 's/^> //'

# remove most HTML tags (accommodates multiple-line tags)
sed -e :a -e 's/<[^>]*>//g;/</N;//ba'

# extract multi-part uuencoded binaries, removing extraneous header
# info, so that only the uuencoded portion remains. Files passed to
# sed must be passed in the proper order. Version 1 can be entered
# from the command line; version 2 can be made into an executable
# Unix shell script. (Modified from a script by Rahul Dhesi.)
sed '/^end/,/^begin/d' file1 file2 ... fileX | uudecode  # vers. 1
sed '/^end/,/^begin/d' "$@" | uudecode                  # vers. 2

# sort paragraphs of file alphabetically. Paragraphs are separated by blank
# lines. GNU sed uses \v for vertical tab, or any unique char will do.
sed '/./{H;d;};x;s/\n/={NL}=/g' file | sort | sed '1s/={NL}=//;s/={NL}=/\n/g'
gsed '/./{H;d};x;y/\n/\v/' file | sort | sed '1s/\v//;y/\v/\n/'

# zip up each .TXT file individually, deleting the source file and
# setting the name of each .ZIP file to the basename of the .TXT file
# (under DOS: the "dir /b" switch returns bare filenames in all caps).
echo @echo off >zipup.bat
dir /b *.txt | sed "s/^\(.*\)\.TXT/pkzip -mo \1 \1.TXT/" >>zipup.bat
```

16.8 TYPICAL USE

Sed takes one or more editing commands and applies all of them, in sequence, to each line of input. After all the commands have been applied to the first input line, that line is output and a second input line is taken for processing, and the cycle repeats. The preceding examples assume that input comes from the standard input device (i.e, the console, normally this will be piped input). One or more filenames can be appended to the command line if the input does not come from stdin. Output is sent to stdout (the screen). Thus:

```
cat filename | sed '10q'        # uses piped input
sed '10q' filename           # same effect, avoids a useless "cat"
sed '10q' filename > newfile    # redirects output to disk
```

For additional syntax instructions, including the way to apply editing commands from a disk file instead of the command line, consult "sed & awk, 2nd Edition," by Dale Dougherty and Arnold Robbins (O'Reilly, 1997; `http://www.ora.com`), "UNIX Text Processing," by Dale Dougherty and Tim O'Reilly (Hayden Books, 1987) or the tutorials by Mike Arst distributed in U-SEDIT2.ZIP (many sites). To fully exploit the power of sed, one must understand "regular expressions." For this, see "Mastering Regular Expressions" by Jeffrey Friedl (O'Reilly, 1997). The manual ("man") pages on Unix systems may be helpful (try "man sed", "man regexp", or the subsection on regular expressions in "man ed"), but man pages are notoriously difficult. They are not written to teach sed use or regexps to first-time users, but as a reference text for those already acquainted with these tools.

16.9 QUOTING SYNTAX

The preceding examples use single quotes ('...') instead of double quotes ("...") to enclose editing commands, since sed is typically used on a Unix platform. Single quotes prevent the Unix shell from intrepreting the dollar sign ($) and backquotes (`...`), which are expanded by the shell if they are enclosed in double quotes. Users of the "csh" shell and derivatives will also need to quote the exclamation mark (!) with the backslash (i.e., !) to properly run the examples listed above, even within single quotes. Versions of sed written for DOS invariably require double quotes ("...") instead of single quotes to

enclose editing commands.

16.10 USE OF ‘’ IN SED SCRIPTS

For clarity in documentation, we have used the expression ‘’ to indicate a tab character (0x09) in the scripts. However, most versions of sed do not recognize the ‘’ abbreviation, so when typing these scripts from the command line, you should press the TAB key instead. ‘’ is supported as a regular expression metacharacter in awk, perl, and HHsed, sedmod, and GNU sed v3.02.80.

16.11 VERSIONS OF SED

Versions of sed do differ, and some slight syntax variation is to be expected. In particular, most do not support the use of labels (:name) or branch instructions (b,t) within editing commands, except at the end of those commands. We have used the syntax which will be portable to most users of sed, even though the popular GNU versions of sed allow a more succinct syntax. When the reader sees a fairly long command such as this:

```
sed -e '/AAA/b' -e '/BBB/b' -e '/CCC/b' -e d
```

it is heartening to know that GNU sed will let you reduce it to:

```
sed '/AAA/b;/BBB/b;/CCC/b;d'      # or even
sed '/AAA\|BBB\|CCC/b;d'
```

In addition, remember that while many versions of sed accept a command like “/one/ s/RE1/RE2/”, some do NOT allow “/one/! s/RE1/RE2/”, which contains space before the ‘s’. Omit the space when typing the command.

16.12 OPTIMIZING FOR SPEED

If execution speed needs to be increased (due to large input files or slow processors or hard disks), substitution will be executed more quickly if the “find” expression is specified before giving the “s/…/…/” instruction. Thus:

```
sed 's/foo/bar/g' filename         # standard replace command
sed '/foo/ s/foo/bar/g' filename   # executes more quickly
sed '/foo/ s//bar/g' filename      # shorthand sed syntax
```

On line selection or deletion in which you only need to output lines from the first part of the file, a "quit" command (q) in the script will drastically reduce processing time for large files. Thus:

```
sed -n '45,50p' filename         # print line nos. 45-50 of a file
sed -n '51q;45,50p' filename     # same, but executes much faster
```

If you have any additional scripts to contribute or if you find errors in this document, please send e-mail to the compiler. Indicate the version of sed you used, the operating system it was compiled for, and the nature of the problem. To qualify as a one-liner, the command line must be 65 characters or less. Various scripts in this file have been written or contributed by:

- Al Aab # founder of "seders" list
- Edgar Allen # various
- Yiorgos Adamopoulos # various
- Dale Dougherty # author of "sed & awk"
- Carlos Duarte # author of "do it with sed"
- Eric Pement # author of this document
- Ken Pizzini # author of GNU sed v3.02
- S.G. Ravenhall # great de-html script
- Greg Ubben # many contributions & much help