Live coding demo - week 3 (Thursday)

Version control with `git`

Preface

For this demo we will develop a script to identify perfect matches between a query sequence and subject sequence. This is the same script you wrote for the homework, but we will go through that development using `git` to track our version history.

For this demo I am going to develop the script in a text editor window and issue commands in the terminal. For this script, I will show snapshots of the script when changes are made. Commands issued in the terminal will have a prompt ($) symbol before them and will be in a separate code block from snapshots of the script. Snapshots of the script will always begin with the shebang to help you distinguish them here.

Setting up

First, let's activate a `mamba` or `conda` environment with `blast` installed. For this demo we also want `seqtk` (finally you will use my go-to example tool!)

To begin a `git`-tracked project, you must first initialize the repo. That's as simple as

```
(base) $ mamba activate blast
(blast) $ git commit
```

Now let's start our script.

```
(blast) $ touch find_perfect_hits.sh
```

We know we want to find perfect hits, but we're maybe not sure what the final product will look like yet.

Let's start off by using an approach we've covered in class that can identify perfect matches and might be an obvious candidate solution when starting this project: `grep`

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

grep -f $1 $2 > $3
```

Let's take a look at the output.

```
(blast) $ ./find_perfect_matches.sh CRISPR_1f.fna ERR430992.fna out.txt
(blast) $ head out.txt
TTTCGGAGCTAGTTCACTGCCGTGTAGGCAGCTAAGAAAAGGTCGAGGTGGGCTCGGCGG
CGATGATCGATGTTCACTGCCGTGTAGGCAGCTAAGAAAGGTACGTGGTTTCGACCAACA
GCACTGCCCAAGTTCACTGCCGTGTAGGCAGCTAAGAAATAAAGGAGATTGCCATGCTGA
```

```
TCAAACTTCCCGTTCACTGCCGTGTAGGCAGCTAAGAAAGTCAGGGTCGTGCATGACTCC
GATGTGGTGGCGTTCACTGCCGTGTAGGCAGCTAAGAAACGTCCAGAACGTCACACGCTC
GCCGTCGATGTGTTCACTGCCGTGTAGGCAGCTAAGAAAAACCGGAGCCTTCGGGCCGCG
TTGGGATCCACGTTCACTGCCGTGTAGGCAGCTAAGAAATTGACTGCTGGGGCCTGACGC
TCATCGCGCGGGTTCACTGCCGTGTAGGCAGCTAAGAAAGCGACCCTGGCCAGGGCGGCG
TCGCGCTCTGCGTTCACTGCCGTGTAGGCAGCTAAGAAATTGAGCACAACCGGCTGAGCC
AGCTGGTTGTCGTTCACTGCCGTGTAGGCAGCTAAGAAACTCGAACCCACCTCGGCCACA
```

That's a great start! We can see obvious repeats in there, but we're getting the whole line which isn't quite right. There's more work to do.

HOWEVER! We now have a unit of code that does something. Let's commit it. We're adding a new feature so we'll use the "feat" type in our commit message.

```
(blast) $ git add ./find_perfect_matches.sh
(blast) $ git commit -m "feat: add basic sequence finder" -m "write basic sequence
finder that pulls lines with perfect matches to query sequences from fasta subject"
```

Now let's fix the bug that we don't just get the match. That's sort of a bug in that we have the behaviour of extracting matches, but it is not quite performing as we want. We can fix that with `grep -o`

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

grep -o -f $1 $2 > $3
```

```
(blast) $ ./find_perfect_matches.sh CRISPR_1f.fna ERR430992.fna out.txt
(blast) $ head out.txt
GTTCACTGCCGTGTAGGCAGCTAAGAAA
GTTCACTGCCGTGTAGGCAGCTAAGAAA
GTTCACTGCCGTGTAGGCAGCTAAGAAA
GTTCACTGCCGTGTAGGCAGCTAAGAAA
GTTCACTGCCGTGTAGGCAGCTAAGAAA
GTTCACTGCCGTGTAGGCAGCTAAGAAA
GTTCACTGCCGTGTAGGCAGCTAAGAAA
GTTCACTGCCGTGTAGGCAGCTAAGAAA
GTTCACTGCCGTGTAGGCAGCTAAGAAA
GTTCACTGCCGTGTAGGCAGCTAAGAAA
```

Much better. Let's commit the bug fix as anything else we do next will not be related to that bug. This is a nice, small commit.

```
(blast) $ git add ./find_perfect_matches.sh
(blast) $ git commit -m "bug: fix extracting sequence around match" -m "use grep -o
flag to only extract match and not surrounding sequence"
```

Unsatisfied

We could probably leave it there for this script if this were the only input file we ever wanted to run and if we only wanted a yes or no answer to the question "are there matches in the subject file?"

However, we can make this script better. We can make sure it can match sequences in either orientation and sequences that don't happen to be on the same line of the fasta file.

Both of those are obvious things you will always need to handle when working with fasta files. They relate to two fundamental issues with the format:

1. DNA is not a single string of characters
    - DNA sequence is (usually) double stranded
    - the two strand are complements of one another (i.e., A in one strand matches T in the other; C with G)
2. Fasta has some readabilitiy and feasibility design choices
    - fasta usually uses lines of 60 characters. DNA is usually longer than that
    - fasta doesn't have a good way to handle circular genetic elements like plasmids or prokaryotic genomes. The start position in your file might actually be part of a circle of DNA without beginning or end

We can see that out repeats happen to be in the same orientation in our query and subject because we know that otherwise grep wouldn't find them. However, we can see what would happen if our assembly happened to be in the opposite orientation using `seqtk` (assemblies pick one of the two DNA strands basically at random.)

```
(blast) $ # reverse complement and stick to 60 character lines
(blast) $ seqtk seq -r -Cl60 ERR430992.fna >ERR430992_rev.fna # reverse complement
and stick to 60 character lines
(blast) $ head -2 ERR430992*
==> ERR430992.fna <==
>NODE_1_length_922990_cov_42.400140
CTTCGCCGTCGCCGGGAGTGGTGCGCATTATAGGGAGATAGAAACTGGCGTCAACACTTA

==> ERR430992_rev.fna <==
>NODE_1_length_922990_cov_42.400140
CTTCGTCGTCGCCGGGAGTGGTGCGCATTATAGGGAGATAGAAACTGGCGTCAACGATTA
(blast) $ ./find_perfect_matches.sh CRISPR_1f.fna ERR430992_rev.fna out.txt
(blast) $ head out.txt
(blast) $
```

As you can see in the about head output, the fasta headers are the same, but the sequence has changed. If you were to look closely you would see that the sequence is reverse complemented. Try it out on the CRISPR repeats sequence for a managable test case.

With reverse complemented sequence, our script fails even though the repeats are present. Nightmare!!!

We definitely need to fix this. However, we currently have a program that works fine some (maybe half) the time. We don't want to break that while we work on a fix to this important bug!

Making a branch to fix the issue

This is where `git branch` comes in. We can keep the version on our master branch intact and instead work in a contained development branch. That way if anyone wants to use our code, they can download the bad, but at least working version from master, even while we are modifying the script in our branch.

```
(blast) $ git checkout -b "fix/handle_revcomp"
Switched to a new branch 'fix/handle_revcomp'
(blast) $ # that was short for the following alternative
(blast) $ # git branch "fix/handle_revcomp"
(blast) $ # git checkout "fix/handle_revcomp"
(blast) $ # our currently active branch can be checked with git branch
(blast) $ git branch
* fix/handle_revcomp
  master
```

As we make changes to the script, you can switch back and forth between branches using `git branch <name>`. You'll see that committed changes only apply to the branch you are working in. The history shown by `git log` is also isolated. We can later merge branches or just individual files from branches into other branches (like master). There's no need to commit a new branch, so we can get right to it. Right now our new branch and our master branch are at the same commit. i.e., until we make changes they are the same so there is nothing to commit yet.

First let's make a start on fixing our issue. We know that grep isn't able to handle some of the features of fasta. We could maybe search for both the query and its reverse complement, but that doesn't deal with formatting issues like line lengths. Fortunately, I told you to install BLAST at the start of the demo so we can just use that!

BLAST is a specifically designed tool for finding sequence matches between fasta-formatted queries and fasta-formatted subjects. It handles all the issues with fasta that I mentioned except one: it doesn't consider the possibility that your subject sequence is actually circular. If a match really spans the start and end of your sequence, then BLAST won't find it. That's just something to be aware of, but we're not going to worry about that here. Indeed most circular genetic elements are thousands or millions of base pairs long so your hits will rarely have that issue.

Replacing `grep` with `blastn` would look something like this

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

blastn -query $1 -subject $2 -task blastn-short -outfmt 6 > $3
```

Let's see if it works

```
(blast) $ ./find_perfect_matches.sh CRISPR_1f.fna ERR430992_rev.fna out.txt
(blast) $ head -2 out.txt
1F      NODE_18_length_84619_cov_36.005061      100.000 28      0       0       1
28      64088   64061   1.41e-09        56.0
1F      NODE_18_length_84619_cov_36.005061      100.000 28      0       0       1
28      64148   64121   1.41e-09        56.0
(blast) $ ./find_perfect_matches.sh CRISPR_1f.fna ERR430992.fna out.txt
```

```
(blast) $ head -2 out.txt
1F      NODE_18_length_84619_cov_36.005061      100.000 28      0       0       1
28      19992   20019   1.41e-09        56.0
1F      NODE_18_length_84619_cov_36.005061      100.000 28      0       0       1
28      20052   20079   1.41e-09        56.0
```

The new script gets hits in both the original and reverse complemented script. The only difference is now the locations of the hits is different

We now have a basic blast command that gets hits. It doesn't do everything we want yet, but it's a unit of code we can sensibly commit. Let's do that now.

I'm going to use the feat tag for this one. It's up to you (or your team in a collaborative git setup) to define what each tag should be used for

```
(blast) $ git add ./find_perfect_matches.sh
(blast) $ git commit -m "feat: implement search using blastn" -m "replace grep-
based search with blastn using outfmt 6 to allow downstream processing with shell
tools"
```

Now we have matches, but we haven't yet filtered out any that are imperfect. The above terminal output shows only perfect matches because BLAST sorts its output by query ID (alphabetical) and then by hit quality (bit score - column 12 in outfmt 6). You'll see the bad hits at the bottom of the output

```
(blast) $ tail -2 out.txt
1F      NODE_33_length_9858_cov_47.713455       100.000 9       0       0       20
28      9539    9531    307     18.3
1F      NODE_24_length_53696_cov_38.948696      100.000 9       0       0       3
11      8751    8743    307     18.3
```

How do we know it's a bad hit? Well column 4 contains hit length and you can see above that the good hits were 28 bases long while these are 9 bases long. Another measure of hit good-ness in this view is column 11. That column contains the "E-value" which is the frequency you would expect to see a hit this good by chance given a random query and subject of the lengths seen here. The good hits are one in one billion. The bad hits are quite likely.

Note that neither of those data alone tell us if the hit is perfect. Judging a hit based on the best hit seen only works if you know the best hit in this output is perfect (which you can't rely on). Using the E-value won't work because E-value depends on the length of your query and subject. If you search a much larger database, then the liklihood of seeing good hits by chance increases. The same hit will have different E-value in different searches. You should checkout the videos linked in the homework if you want to understand more about how BLAST works.

So which data do we need to determine if the hit is perfect? If we consult the `blastn` help message, there are a couple of candidates that jump out in the section describing output fields for outfmt 6. nident, and qlen.

nident tells us the number of identical bases between query and subject. However, we already have that information in the current view. It's the pident (percent identity) and len (match length) columns. i.e., nident =

pident*len.

qlen adds information the view doesn't already have: the length of our query sequence. If we compare that information to the length of our match (column 4) we will know what proportion of our query length matched. We already know percent identity (column 3) so with those two pieces of information we will be able to identify perfect hits.

We know that awk can quickly and easily output lines of a file that match some set of criteria, so now we have a plan to implement a BLAST-based search for perfect hits. Let's implement it now

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

blastn -query $1 -subject $2 -task blastn-short -outfmt '6 std qlen' | awk '$3==100
&& $4==$13' > $3
```

Column 13 is the newly added qlen, while "std" in the outformat 6 specification indicates we want the standard columns. std is just an abbreviation of std, just like the other column names are abbreviations.

With this new version of the script our worst hits kept are still perfect because we only have perfect hits now.

```
(blast) $ ./find_perfect_matches.sh CRISPR_1f.fna ERR430992.fna out.txt
(blast) $ tail -2 out.txt
1F      NODE_18_length_84619_cov_36.005061      100.000 28      0       0       1
28      20472   20499   1.41e-09        56.0    28
1F      NODE_18_length_84619_cov_36.005061      100.000 28      0       0       1
28      20532   20559   1.41e-09        56.0    28
```

Merging the fix into master

let's commit that to git and merge it back into our master branch to replace the now defunct grep-based approach. The tag for this commit is debateable too. Perhaps it could be described as a bug that out results included imperfect hits. I'll stick with feat for this as we're really adding new functionality to our script. As long as you stick to agreed upon rules with the people collaborating on the same codebase, you can use whatever guidelines you like for git commit messages.

```
(blast) $ git add ./find_perfect_matches.sh
(blast) $ git commit -m "feat: implement perfect hit filtering" -m "add qlen to
blast output and filter with awk"
```

Now let's head back to our master branch which still has the old script version.

```
(blast) $ git checkout master
Switched to branch 'master'
(blast) $ git branch
  fix/handle_revcomp
* master
(blast) $ # our script version in master is still the old one
```

```
(blast) $ cat find_perfect_matches.sh
#!/usr/bin/env bash

grep -o -f $1 $2 > $3
(blast) $ git merge fix/handle_revcomp
Updating ea53a59..e8e543e
Fast-forward
 find_perfect_matches.sh | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
(blast) $ # note we're "fast-forwarding" here because new commits
(blast) $ # in our branch are being copied into master
(blast) $ # which brings its head forward through that history
(blast) $ # git log in master now shows the commits from our branch
(blast) $ # and you can now delete the branch as master is updated
(blast) $ git branch -d fix/handle_revcomp
Deleted branch fix/handle_revcomp (was e8e543e).
```