Warmup and Setup

CS181, Fall 2024

Out: Sept. 6
Due: Sept. 20, 11:59 PM

1 Environment and Docker Setup

For two projects in this course, you will be running bioinformatics analyses using existing packages. In order to have an easier environment setup, we would like you to use a Docker container setup.

Please follow the tutorial here to complete the installation process.

As the tutorial explains, any package you might need during the course is pre-installed in the container. Therefore, you may run the Shell scripts in this project and future projects within your container! Please reach out to course staff if you run into any issues regarding this setup process. We are not grading your container creation. However, it will be necessary for Project 1 and future assignments.

2 Warmup Exercises

The first two exercises are selected or adapted from Rosalind (http://rosalind.info), an online collection of bioinformatics problems for beginners.

For each problem, write a program that outputs the solution. This is a homework to help you get your feet wet in your programming language of choice for this class.

Specifications:

- Our Gradescope autograder is currently configured to accept solutions written in Python 2, Python 3, Java, R, and Julia. Please note that the Docker Container does not currently support Java and Julia - they can be installed manually using `sudo` should you desire to use either of these languages.
- Your solutions generally should not require the installation of any packages that do not come in the standard installations of your chosen programming language. However, if you are using Python, you will also have access to numpy and pandas.
- To facilitate anonymized & automated grading, each of your solutions must be accompanied by a shell script. Make sure each problem is able to output the correct result using the shell script provided. If you are using Python 3, your shell script must run your program using the command "python3" rather than "python", as "python" will run your code with Python 2.

- Your shell scripts should print exactly what is shown in the examples given for each problem. If you print any extra text, you will fail our autograder and lose points. This means that if you are coding in R, you may need to print text using "cat()" instead of "print()".
- Be sure to print any terminal output to stdout, which is the channel that the standard print functions write to in most programming languages. If you print to stderr, your solution will be interpreted as an error message and fail the autograder.
- To hand in this assignment, upload your shell scripts along with all files needed to run your code on Gradescope. If you have any subdirectories or folders that you would like to preserve in your handin, you will need to compress your submission into a zip file and upload the zip file to Gradescope. However, to ensure that the autograder runs your handin properly, make sure that all shell scripts are present at the root of your handin; in other words, do not place your shell scripts inside any folders.
- When you hand in your solution, our autograder will immediately run on your submission. We have made the test cases provided in the project handout immediately visible to you so that you can ensure your solution runs properly with the autograder. If you fail the autograder and cannot determine why, notify a TA and we will help you to diagnose the problem. If your final submission is not compatible with our autograder, it will be very difficult for us to give you credit for this assignment.

For this project, we will provide you with stencil shell scripts. Please come to TA hours if you have any questions on shell scripts.

If you need help setting up remote access on your personal computer, see the *Working From Home: Using SSH Remotely* document on the Resources page of the course website.

Problem 0: Reading

On the Resources page of the course website, there is a section titled "Essential Background Reading". This section contains primers on molecular biology, probability and statistics, and Python that provide essential information for the course. Please read each of the resources posted here.

Problem 1: Hamming Distance

(Rosalind: HAMM) Given two strings s and t of equal length, the *Hamming distance* between s and t, denoted $d_H(s,t)$, is the number of positions where s and t differ.

For example, the Hamming distance between the pair of strings `GAGCCTACTAACGGGAT` / `CATCGTAATGACGGCCT` is 7.

Given: Two DNA strings s and t of equal length.

Return: The Hamming distance $d_H(s,t)$, printed to standard output.

Specification: `sh hamming.sh STRING1 STRING2`

```
> sh hamming.sh GAGCCTACTAACGGGAT CATCGTAATGACGGCCT
7
```

Problem 2: Reverse Complement

(Rosalind: REVC) In DNA, nucleotides `A` and `T` on opposite strands form a Watson-Crick base pair, as do nucleotides `C` and `G`. We say that `A` and `T` are *complementary* nucleotides, and `C` and `G` are complementary nucleotides. By convention, the string of nucleotides representing one strand of a DNA molecule is written in the $5' \rightarrow 3'$ direction, the direction of DNA synthesis. Thus, the string representing the other (complementary strand) is the *reverse complement* string.

The reverse complement of a DNA string s is the string s^C formed by reversing the symbols of s, then taking the complement of each symbol.

For example, the reverse complement of `GTCA` is `TGAC`.

Given: A DNA string s.

Return: The reverse complement s^C of s, printed to standard output.

Specification: `sh reverse.sh STRING`

```
> sh reverse.sh GTCA
TGAC
```

***Note**: Your implementation must not contain any usage of a built-in `reverse()` function on strings.*

Problem 3: Counting *k*-mers

When analyzing DNA sequences, it is often important to ask ourselves what properties we expect a "random" DNA sequence to have. If we assume, for example, that DNA strings are generated by independently drawing nucleotides from the set $\{A, C, G, T\}$ uniformly at random in succession, what do we expect the resulting string to look like? We'll refrain from doing any statistical analysis in this problem. Instead, we will simply count the number of distinct k-mers in a gene for a range of k values. (A "k-mer" is simply a string of length k.)

For example, `AAATC` has three distinct 2-mers: `AA`, `AT`, and `TC`.

Given: A file, `Tthermophilus.txt`, containing the sequence of a real gene.

Note: `Tthermophilus.txt` may or may not have a newline at the end of the file. Your code should be able to handle either case.

Return: The number of distinct k-mers in the gene for each of $k = 1, \ldots, 10$, printed to standard output. Print each count on its own line.

Specification: `sh kmers.sh Tthermophilus.txt`

```
> sh kmers.sh Tthermophilus.txt
4
16
...
```

As an optional exercise, think about the following. You won't be graded on these questions, but feel free to post your thoughts on Ed!

- How many possible distinct k-mers are there for each value of k?
- For which values of k do you fail to observe all possible k-mers in the gene?
- Generate a large number of random DNA strings of length 1098, which is the length of the given gene. (Use the definition of "random" given above.) Use these strings to compute empirical distributions on the counts of distinct k-mers for each value of k. How do the counts of distinct k-mers from the real gene sequence compare to the empirical distributions?
- What does this tell you about our model of random DNA sequences?
- The file we gave you happens to contain the sequence of a gene from *Thermus thermophilus*, an extreme thermophile. What does this tell you about the nature of the sequence?