CME 211: Homework 1

Due: Friday, October 4, 2019 at 4:30pm (Pacific Daylight Time)

This assignment was designed by Patrick LeGresley and modified for the purposes of this course. It is a good idea to read the entire document before starting work.

Background

Gene sequencing technology has progressed dramatically over the past decade. Whereas it used to take hundreds of millions or even billions of dollars and many years to sequence a single human, today it can be done in a week for thousands of dollars. This has led to an explosion of data volume in the field of *bioinformatics*.

Current gene sequencing instruments typically use *shotgun sequencing*, resulting in short sequences called *reads* that are typically of length 50 - 1000 base pairs depending on the technology. For comparison the human genome is approximately 3 billion base pairs. However, the instruments generate hundreds of millions to billions of reads in parallel so these need to be *assembled* by computer algorithms to determine the full genome. An important parameter in this process is the *coverage*, computed as the number of reads times the read length divided by the total size of the genome, which quantifies the average redundancy in the data. Coverages of 15 to 30x are typical.

If an organism has never been sequenced before a *de-novo assembly* is performed. Humans have now been sequenced many times and there is a *reference genome* called HuRef (Human Reference). In this case assembly is typically done by *mapping* or *aligning* the reads to the reference. This is a process of taking each read and finding which location(s), if any, on the reference genome it best matches up. However, the mapping or aligning process has to accommodate the expected differences between the reference and the reads of the individual currently being sequenced. The instruments also aren't perfect so the reads can have errors.

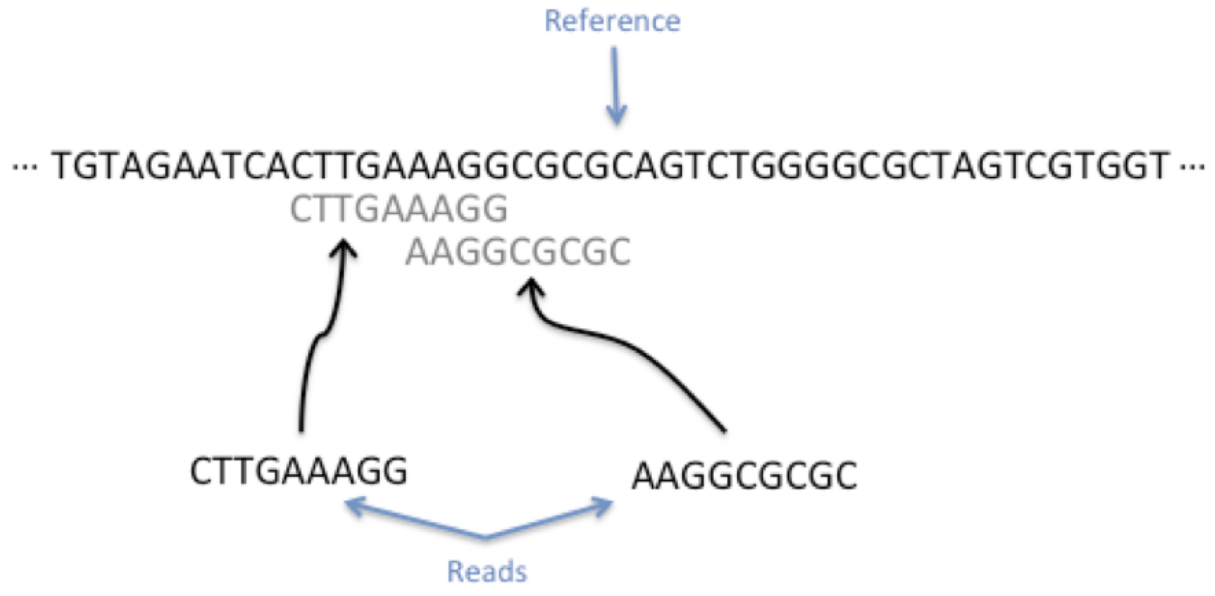

Figure 1: mapping (alignment) assembly

Assignment

In this assignment we will be implementing a simple alignment program that computes the position(s) where reads exactly align to the reference. We'll start by generating simulated data to develop and test our program. Simulated data is useful because we can create one or more datasets with properties that exercise different aspects of our program. For example a given read might align to the reference in no locations, exactly one location, or even multiple locations. Our program should correctly handle all of these situations.

A given dataset will be made up of two files. The first file will contain the reference. Although an implementation capable of handling human genome scale data would want to use a very efficient representation for memory capacity reasons, for our purposes we will use one large string using the upper case letters `A`, `C`, `G`, and `T`. A second file will contain the reads to be mapped with one read per line. The reads will also be represented by strings.

The alignment program will load the reference file and reads file and output an alignment file. Each line of the alignment file should contain the string representing the read, a space, and then the alignment(s), each separated by a space. Each alignment is an integer indicating the `0` based index of the position in the reference where an alignment of the read begins. If the read does not align a single alignment of value `-1` should be reported. The format for the alignments is shown in Figure 2.

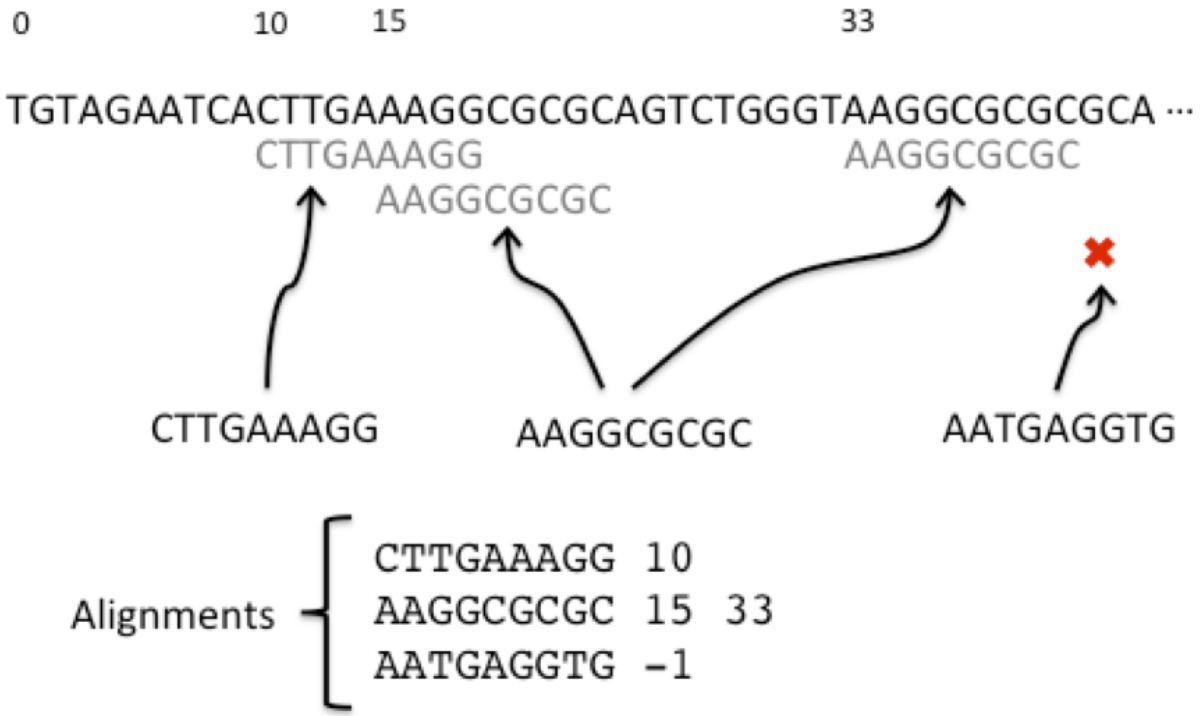

Figure 2: representing the alignments in a file

Preparation

This assignment requires that you have completed CME211 HW0. If you have not completed HW0, please do so (even if it is past the HW0 deadline). Part of HW0 is the creation of your homework repository, which will be used throughout the quarter. You will not be able to complete HW1 (or any future CME211 homework) without completing HW0. (In the event that when you signed up for a student repository, a

spurious one suffixed with "-1" was created, please make sure to only work off the "original", as this is where the TA's will pull your submissions from; spurious repositories will be removed by TA's soon.)

These instructions assume that you are logged into `corn.stanford.edu` and have cloned your CME211 homework repository to your Farmshare user directory. First, navigate to your homework repository:

```
$ cd /farmshare/user_data/[sunet_id]/cme211-[github_user]
$ ls
README.md  STUDENT  hw0
```

In the above `cd` command, `[sunet_id]` must be replaced with your SUNetID and `[github_user]` must be replaced with your GitHub username.

At this point, it is good to check if your local repository is clean and up-to-date with the remote repository on GitHub. This is achieved by running:

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean
```

If you get a different message, make sure to commit (or revert) all modified files or perform a `$ git pull` to retrieve remote changes. Now, create a `hw1` directory (it must be lower case):

```
$ mkdir hw1
$ ls
README.md  STUDENT  hw0  hw1
```

Now, create a basic `README` inside of `hw1`, commit it to the repository, and push to GitHub:

```
$ nano hw1/README
# add some basic info to the README
# check status of the repo with $ git status
$ git add hw1
$ git commit -m "add hw1 readme"
# ... output omitted ...
$ git push origin master
# ... output omitted ...
```

All HW1 files must go inside of the `hw1` directory you just created. So, `$ cd hw1` and you are ready to go.

Part 1

Scoring: 10 points

Manually create a very small dataset that you will use for the initial development and testing of your program. You should also manually create the correct alignment file so that you can compare it with the output of your program. The length of the reference should be `10` and there should be `5` reads of length `3`. One read should not align to the reference, one read should align to the reference in two positions, and the remaining `3` reads should align to the reference in one position. Name your files `reference0.txt`, `reads0.txt`, and `alignments0_ref.txt`.

Once you have created the files, it is a good idea to commit them to your repository and push to GitHub.

Part 2

Scoring: 45 points total: 30 code correctness, 5 code style, 10 writeup

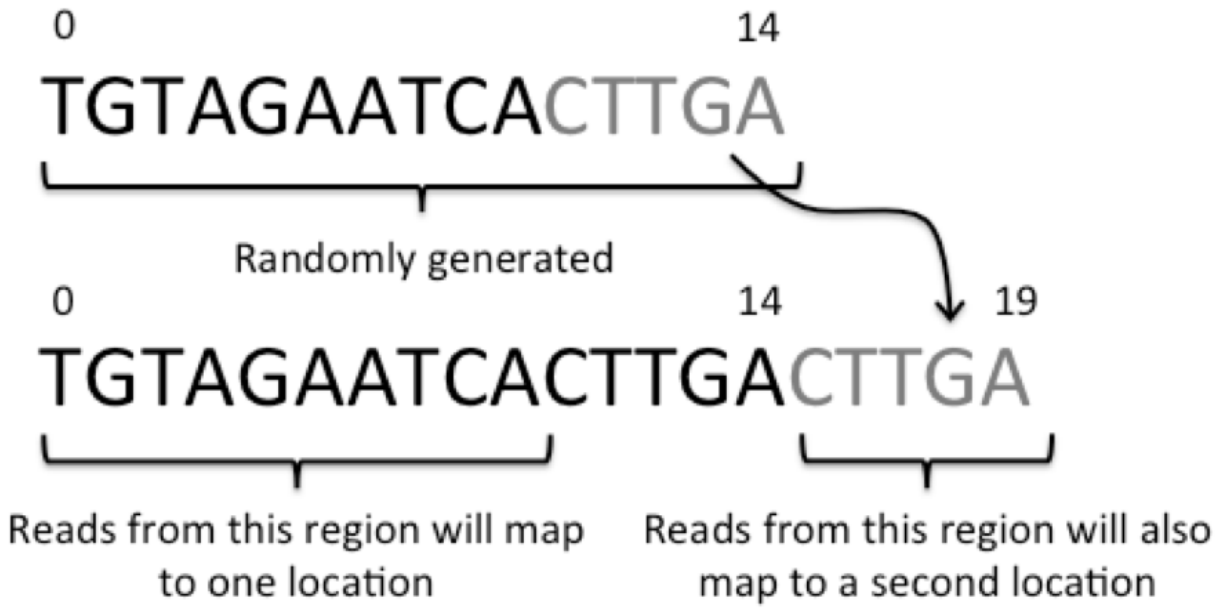

Figure 3: constructing a reference of length 20

It isn't feasible to manually create larger datasets of the scale that your alignment program will actually process so next we will write a program to create some additional datasets. We recommend that you (at least) generate datasets with the sizes listed in Table 1, because they match the sizes of the datasets that we will use to test your program. You do not need to commit the datasets that your program generates to your repository (in fact, we prefer that you don't).

dataset number	reference length	number of reads	read length	coverage
1	1000	600	50	30
2	10000	6000	50	30
3	100000	60000	50	30

Table 1: sizes of test datasets

The references are to be generated randomly using the `randint()` function from the random module. Each letter in the reference can be selected using `random.randint(0,3)` and assigning the `4` different integer outcomes to the `4` different letters. However, reads that will align twice require repeats in the reference. To get repeats in the reference we will generate the first 75% of the reference randomly. The last 25% of the reference should be a copy of the last part of the randomly generated partial reference as shown in the following Figure 3.

The order of the reads that align different numbers of times should be random and we would like approximately 75% of them to align at one position in the reference, 10% to align at two positions in the reference, and the remaining 15% should not align to the reference. This can be achieved by using `random.random()` which returns a floating point value in the interval `[0,1)`. Use 15% of the interval for reads that align zero times, 75% for the reads that align one time, and 10% for the reads that align two times. Each time you need to create a read, generate a new random number and use the location in the interval to decide how many times the new read should align. Keep track of the actual number of zero, one, and two alignment reads generated using the three methods described below.

The reads that align once can be created by randomly picking a starting position in the first 50% of the reference (the portion that was created entirely randomly and was not copied). From the starting position

copy a substring out of the reference that has a length equal to the read length.

The reads that align twice are then created by randomly picking a starting position (again using `random.randint()`) in the last 25% of the full reference and copying a substring out of the reference that has a length equal to the read length. Be careful not to pick a position that is so close to the end of the reference that the substring would extend past the end of the reference.

To generate a read that does not align first create a random read using the same approach that you used to create the random part of the reference and then check if the read is present in the reference using the `find()` function from the string module. `string.find()` returns the lowest index in a string where a substring is found. A return value of `-1` indicates the substring was not found in the string. Note that in our case the string is the reference and the substring is a read. If the read is not present in the reference you can use it, otherwise generate a new random read and repeat the process until you get a read that does not align.

Program specifications Name your Python file `generatedata.py`. It should accept as arguments from the command line the following parameters *in the following order*: `reference_length`, `nreads`, `read_length`, `reference_file` and `reads_file`. If one or more parameters are missing, the program should return usage information and immediately exit.

```
$ python3 generatedata.py
Usage:
  $ python3 generatedata.py <ref_length> <nreads> <read_len> <ref_file> <reads_file>
```

Please review HW0 or course notes for information on how to read arguments from the command line. The datasets created by the main program must be written to disk, using the file names specified by `reference_file` and `reads_file`. The integer arguments `nreads`, `reference_length`, and `read_length` are used to specify the values of the respective parameters.

For example, one run of your program must adhere to the following format (i.e. the numbers may change):

```
$ python3 generatedata.py 1000 600 50 "reference_test.txt" "reads_test.txt"
reference length: 1000
number reads: 600
read length: 50
aligns 0: 0.151666666667
aligns 1: 0.76
aligns 2: 0.0883333333333
```

For code style we will be looking at how you name variables, consistency of your indentation, and the descriptiveness and consistency of your comments compared to what the code actually does. No need to comment every line of code, but include descriptions of what small subsections of code are doing. For example, describe what happens in a loop body, why and what you are doing inside an if clause, etc.

Writeup In your HW1 `README` file, include a command line log of using your program to generate the recommended test datasets. This should include the following commands followed by the output. For example:

```
$ python3 generatedata.py 1000 600 50 "ref_1.txt" "reads_1.txt"
# ... include output from command ...
$ python3 generatedata.py 10000 6000 50 "ref_2.txt" "reads_2.txt"
# ... include output from command ...
$ python3 generatedata.py 100000 60000 50 "ref_3.txt" "reads_3.txt"
# ... include output from command ...
```

In your HW1 `README` file, please also provide answers to the following questions:

- Describe in a paragraph or so the considerations you used in designing your handwritten test data. If your code works properly for your handwritten data, will it always work correctly for other datasets?

- Should you expect an exact 15% / 75% / 10% distribution for the reads that align zero, one, and two times? What other factors will affect the exact distribution of the reads?
- How much time did you spend writing the code for this part? Keeping track of the time you spend on coding is also a useful habit. Depending on your career path, in the future you may have to keep track of time for billing purposes or be asked to scope projects and estimate the hours required to complete them. The amount of time spent will not affect your grade. We are gathering this information to help gauge the difficulty level of the assignment. If you've spent over 5 hours on the assignment and have not made very substantial progress towards finishing it you should come to office hours to discuss your approach and implementation progress.

So why a writeup in a programming class?

When applying for jobs that require software development as part of the position it is very common for employers to ask for "code samples". Along with the source code you are usually required to provide a text description of the function and implementation of your code, interesting challenges or design decisions you made, etc. One or more of your assignments could be submitted as code samples. And writing a text description of your program is good practice for communicating your design and development process to interviewers.

Part 3

Scoring: **45 points total: 30 code correctness, 5 code style, 10 writeup**

Now we will write a simple alignment program to process our datasets. For this implementation we will again make use of the `string.find()` function. Be sure you handle the case where a read aligns at multiple positions in the reference. The optional start and end arguments to `string.find()` may be useful for this.

Your program should load a reference and corresponding reads file, perform the alignment, and then write the alignments file. Also keep track of the number of reads that align zero, one, and two or more times. Make sure you get the correct results for your manually created dataset by referring to your `alignments0_ref.txt` file before attempting the other datasets. You might find the Linux `diff` command to be useful.

Compute the amount of time your program spends performing the alignment. You don't need to include the time to load the reference and the reads. Right before you start the actual alignment portion of your program call the `time.time()` function and record the returned time (which is a float representing the number of seconds since the Unix Epoch). After the completion of the alignment portion of your program make a second call to `time.time()`. The difference between the two values is the elapsed time in seconds required to compute the alignments. For dataset 1 you should expect an elapsed time of approximately `0.01` seconds.

Program specifications Name your Python file `processdata.py`. Your program should accept from command line the following arguments in the following order: `reference_file`, `reads_file`, and `align_file`. The main program will read input data from the reference and reads datafiles, and will output results to a file with name specified by `align_file`. Each line of the alignments output file will contain the read, followed by a space, followed by space-separated 0-based indices of read locations. If the read is not found in the reference data, the read is followed by `-1`. To simplify things if a read appears more than twice we can simply write the first two starting indices we find. In this way, we have provided a "certificate", in the sense that one can look at a read in the output file, followed by a couple indices which specify the location of the reads; the auditor can then quickly examine these locations in the simulated data and make sure the proposed read is found!

The running of your program must adhere to the following format:

```
$ python3 processdata.py
Usage:
  $ python3 processdata.py <ref_file> <reads_file> <align_file>
```

```
$ python3 processdata.py reference0.txt reads0.txt align0.txt
reference length: 1000
number reads: 600
aligns 0: 0.151666666667
aligns 1: 0.76
aligns 2: 0.0883333333333
elapsed time: 0.008469
```

The same criteria for code style mentioned in Part 2 also apply to Part 3.

Performance There are two tasks: generating the data, and processing it. A reasonable solution should be able to *generate* data for our **largest** specified input size within two seconds, and to *process* the corresponding data in about 35 seconds. We will start penalizing programs which take longer than 10x these cutoffs, and we will not allow programs to run which require more than 25x compute time relative to the target cutoffs.

Writeup In your HW1 `README` file, include a command line log of using your program to perform alignment on the recommended test datasets. This should include the following commands followed by the output. For example:

```
$ python3 processdata.py ref_1.txt reads_1.txt align_1.txt
# ... include output from command ...
$ python3 processdata.py ref_2.txt reads_2.txt align_2.txt
# ... include output from command ...
$ python3 processdata.py ref_3.txt reads_3.txt align_3.txt
# ... include output from command ...
```

In your HW1 `README` file, please also provide answers to the following questions:

- Does the distribution of reads that align zero, one, and two or more times exactly match the distributions you computed as you were creating the datasets? Why or why not?
- Using your timing results what can you say about the scalability of your implementation as the size of the reference and read length varies? Estimate the time to align the data for a human at 30x coverage and a read length of 50. Put differently, we'd like you estimate the growth rate of the time it takes your program to process inputs as a function of input size. Is it feasible to actually analyze all the data for a human using your program?
- How much time did you spend writing the code for this part?

Checklist & submission

The following files must be present in the `hw1` directory of your CME211 GitHub homework repository by the deadline:

- `README`: text document with command line logs and answers to questions from Parts 2 and 3. You may also call this file `README.txt` or `README.md` (if you want to use Markdown formatting).
- `reference0.txt`: manually constructed reference data (Part 1)
- `reads0.txt`: manually constructed read data (Part 1)
- `alignments0_ref.txt`: manually constructed alignment data (Part 1)
- `generatedata.py`: code implemented for Part 2
- `processdata.py`: code implemented for Part 3

Notes:

- Take care to follow the file and directory names exactly. Everything is case sensitive.

- Please avoid committing other data files to the repository. We only want to see your `README`, Python code, and manually constructed data. For HW1 it is acceptable if you end up committing more data files. In the future we will deduct points for excess files.
- Please avoid committing editor temporary files to the repository.
- Your homework is not complete until you have pushed the above files to GitHub.
- Late work is not accepted.