COMPSCI 260, Introduction to Computational Genomics **30 September 2022**

Problem Set 3

Due: 5pm on 14 October 2022

Problem 1: Genome assembly: Shotgun wedding? (20 points)

List of files to submit:

1. `README.problem1.[txt/pdf]`
2. `simulate.py`

Plan

Consider this simplified model of whole-genome shotgun sequencing:

1. billions of copies of a genome of length G are each fragmented at many locations uniformly at random (the probability of fragmentation at each location in the genome is uniform over the locations); within a genome, the fragmentation process occurs at a low enough rate that resulting fragments are all at least L nucleotides long (where L is much, much smaller than G)
2. the location of each fragment produced by this process is assumed to be independent of the locations of the other fragments
3. edge effects can be ignored[1]
4. the process of cloning the fragments into vectors works perfectly and introduces no bias or loss, so can be ignored entirely
5. from each of R different cloned fragments, exactly L nucleotides are sequenced from one end of the fragment, yielding a grand total of R sequence reads, each of length L

a) The coverage C of shotgun sequencing is the expected number of times each nucleotide in the genome has been sequenced during the procedure. Provide an expression for C in terms of other defined quantities.

b) What is the probability that a specific location in the genome will not be covered by any of the R reads (your answer must be a function of R and C)? Using this probability, write an expression for the expected number of nucleotides in the genome that remain unsequenced during this procedure (your answer must be a function of G and C).

[1] An 'edge effect' arises when an object has a different behavior or property by dint of the fact that it is near the boundary or edge of some structure. In our specific problem, edge effects can arise at the beginning and at the end of a linear chromosome or genome. For example, the first nucleotide will be sequenced only when the region from 1 to L is sequenced, while a nucleotide at some position i where $L \leq i \leq G - L + 1$ will be sequenced whenever one of the following regions is sequenced: from $i-(L-1)$ to i, from $i-(L-2)$ to $i+1$, from $i-(L-3)$ to $i+2$, ..., from i to $i+L-1$. In such a case, the probability of covering the first position in the genome is smaller than the probability of covering the i^{th} position. But in our simplified model of whole-genome shotgun sequencing we will ignore such 'edge effects' and assume that every position behaves as if it is not near the edge of the genome (as genomes become longer, a smaller and smaller fraction of positions are subject to edge effects anyway, so this is a reasonable approximation).

Hints: Remember that you can ignore edge effects and you know that $L \ll G$. For the second question you may wish to recall from calculus that:

$$\lim_{x\to\infty}\left(1-\frac{a}{x}\right)^x = e^{-a}.$$

c) The output of an assembly algorithm is a set of *contigs*, where a contig is defined to be a set of contiguously assembled nucleotides. Assume that you have access to an assembly algorithm that is oracular in the sense that it can perfectly assemble the reads it is given, without needing to satisfy any minimal length overlap requirements: it somehow knows precisely where every read it sees should go in the genome (but of course, it can learn nothing about genomic nucleotides that are not in any read and are thus unsequenced). What is the expected number of contigs reported by such an algorithm? What is the expected length of each contig?

Develop

d) Write a Python program called `simulate.py` that simulates the sequencing process and computes (empirically) the quantities in subproblems (**a**–**c**). In this simulation, you do not need actual nucleotide sequences. Instead, you will represent a genome of size G using a list with exactly G elements (where G is a fixed value passed to your function). As you run the simulation, each list element will store the number of times the nucleotide at that position has been sequenced. To simulate the sequencing process once, follow these steps:

- Initialize all the elements in the list of length G to 0 (to represent that each nucleotide has been sequenced 0 times).
- Randomly select starting locations for R reads of length L, and for each read, increment the values for the L nucleotides covered by the read (thereby incrementing the count of the number of times each of those L nucleotides has been sequenced). You will need to decide on a sensible approach to addressing edge effects in your simulation; make sure to state and explain/justify your choice in your `README` (multiple approaches are reasonable since the effect should be small when the value of G is very large).

Apply

e) Setting $G = 3 \times 10^6$, $R = 4 \times 10^4$, and $L = 400$, simulate the sequencing process 20 separate times and, for each simulation, **report** the following values:

- The empirical coverage (average number of times a nucleotide in the genome was sequenced)
- The number of nucleotides not covered by any read
- The number of contigs assuming you can use the oracular assembly algorithm mentioned in (c) to assemble all the reads
- The average length of these contigs

f) Average the values computed above over the 20 simulations and compare them with the values computed using the expressions you derived in subproblems (**a**–**c**).

Reflect

g) Consider $G = 3 \times 10^9$, $C = 7.5$, and $L = 500$, numbers reasonably reflective of the situation faced by Celera in the late 1990s. How many nucleotides should we expect will remain unsequenced? How many reads will need to be generated in order to achieve the given coverage? (Please calculate numerical values using the formulas you derived above.)

h) After all the reads are sequenced, the first step of genome assembly requires that every read needs to be compared with every other read in both its original orientation as well as its reverse complement orientation. How many such read comparisons will the assembler need to undertake? (Let us define the process of checking whether or not two reads in a certain relative orientation overlap by a sufficient amount to be *one* comparison. Do not worry about counting the individual nucleotide comparison operations that would need to be done while sliding one read along the other.)

i) Assume an assembler of that era could perform 40 million read comparisons per second. How long will it take to complete the first step of the assembly?

j) When the assembler is finished with all of the steps of assembly (you can assume it is oracular, as above), how many contigs will be produced? What will be the average length of a contig? What will be the average length of an unsequenced region between two adjacent contigs?

k) *Comment on the set of values you have computed in subproblems (**g**–**k**), those regarding the Celera assembly task for sequencing the human genome. Are the numbers reasonable? What strikes you about them, or about the task?*

Please comment your code clearly, especially when changing code that has been provided for you, but please don't change function signatures (the expected parameters and output, which we try to be clear about in each function's comments). Do your best to submit code that runs without errors, even if you don't completely finish the problem.

Problem 2: Transformers: More than meets the eye (15 points)

List of files to submit:

1. `README.problem2.[txt/pdf]`
2. `bwt.py`

Plan

The Burrows-Wheeler Transform (BWT), also referred to as block-sorting compression, is a powerful tool in the field of genomics. As discussed in class, formulations of the BWT are particularly important in multiple contexts, including the following two:

- *Short-read mapping*: Several modern short-read aligners, including Bowtie, BWA, and SOAP2, efficiently build an index of a given reference genome using algorithms based on the BWT.
- *Data compression*: The BWT serves as the foundation for a number of lossless data compression algorithms (you may have heard of bzip or bzip2), some of which are employed for maintaining massive amounts of genetic sequencing data.

Essentially, the BWT manipulates a block of input text using a reversible transformation that does not itself compress the text, but reorders (permutes) it in a manner that facilitates compression (and other nice properties).

In the case of DNA sequence data, for example, the content of the original sequence (represented as a string of `A`s, `C`s, `G`s, and `T`s) is rearranged so that the output of the transformation is a sequence that contains runs of similar characters. To illustrate, the result of performing the BWT on `TATCGTACACTACGTACGA$` (where `$` is the special EOF (end-of-file) character that indicates the end of the string) is `AGTTTCTAATAACCCGGC$A`.

a) Perform, by hand, the BWT on the sequence `TCCTGTAG$`. Be sure to show your work, including details pertaining to each step in your solution. It is not sufficient to simply report the permuted sequence.

b) Suppose you are given a sequence `GTCA$ATC`, which has been permuted by way of the BWT. Perform, by hand, the reverse of the BWT in order to obtain the original sequence from which it was generated. Be sure to show your work, including details pertaining to each step in your solution. It is not sufficient to simply report the original sequence.

You can use either the append and sort method, or you can do LF mapping, or you can try your hand at both. It's up to you, and you can receive full credit for either approach, provided you do it correctly and show your work (including intermediate steps). But note that in part (c), we will ask you to implement reverse BWT using append and sort, so it might be sensible to practice that method here.

Hint: If you wish to check the result you obtain from the reverse BWT, you can apply the BWT in the forward direction (i.e., the algorithm you employed in part (a)) on your answer. Assuming you have performed the reversal correctly, you should find that applying the BWT to your answer yields the transformed sequence at the start of this subproblem: `GTCA$ATC`.

Develop

c) In the Python program `bwt.py`, there are two largely empty functions called `forward_bwt` and `reverse_bwt`. Write code so that `forward_bwt` takes as input a string ending in a `$` character to signify the end of the string and returns the result of the BWT applied to that string. Your code should handle arbitrary strings—not just those composed of DNA bases—but you may assume that these strings will contain only standard printable ASCII characters, and you may further assume that they contain no characters that would sort before the `$` character[2], and that the only `$` character they contain is the one at the end.

In `reverse_bwt`, write code that takes a string which has been permuted via the BWT and returns the original string from which it was generated (if you've done things correctly, you should find that the original string ends in a `$` character). In this function, you must implement the append and sort method: That is what we'll be looking for when grading.

You may wish to test the correctness of your methods by using the example we provided at the top of this question as well as the sequences you permuted by hand in parts (a) and (b). You can also create your own test cases by using the output of one function as the input for the other, since each is the inverse transform of the other.

Extra challenge: You are welcome to (additionally) write a separate function that implements reverse BWT using LF mapping. This function should still take only the transformed text as input; from that input text, it can then compute any other data structures it would need to implement the LF mapping. Any LF-mapping implementation is for your own edification (though it might be helpful practice before building your `fm_index.py` code in Problem 2). If you do write such a function, consider how the two reverse BWT approaches differ in time and space (including the time it takes to compute other data structures before you can do the LF mapping). You might even compare empirical run-times on different sized inputs with `timeit`, which you saw in Problem Set 2.

Apply

[2]If you know ASCII or consult an ASCII table, this assumption means that the string should not contain `!`, `#`, `"`, or ␣ (the space character).

d) Run your `forward_bwt` function on the following input and **report** the output.

```
CGGACTAACGGACTAACGGACTAACGGACTAC$
```

e) While on a train headed north from the nation's capitol, you intercept a mysterious note that is clearly of high importance but seems like gobbledygook. The text of the note is provided in the file `mystery.txt`: take a look. Top spy that you are, you decide to study the gibberish closely. You notice that many of the characters are clumped into runs, and there's also only a single `$` character in the text, a clue which leads you to cry, "Egad, man! Someone has encoded their message with the transform of Burrows and Wheeler!". Run your `reverse_bwt` function on the mystery text, and **report** the output. Your code should read in the text from the file. In your reported output, you should replace underscore characters with spaces.[3]

Problem 3: I have a gut feeling... (25 points)

List of files to submit:

1. `README.problem3.[txt/pdf]`
2. `bwt.py`
3. `fm_index.py`
4. `infection_investigator.py`
5. `FLAG.txt`

⋆ **Step 1:** Using genomic sequencing to quantify the prevalence of various gut microbes

Plan

You are a hot-shot gastroenterologist, at the cutting edge of modern medical technology. Whether to generate new research data, improve your diagnoses, reduce the chances of being sued for medical malpractice, or just keep your billing rates obscenely high, you have adopted a policy that any time a patient comes in to see you, you will order a routine analysis of his/her gut microbiome. The analysis allows you to profile the diverse community of microbes living in each patient's gut. Patients are therefore asked to provide a stool sample (and not the three-legged variety). Sample in hand—or, more accurately, safely in a sample tube—your crackerjack lab team goes to work: purifying the collective genomic DNA of all the different kinds of microbes that happen to be mixed together in different proportions within a patient's sample, then randomly fragmenting all that DNA, and finally, generating short reads from the fragments that result.

Today has been a slow day in the clinic, but things are about to pick up. All of a sudden, three patients come in to see you. Patients 1 and 3 are asymptomatic, but patient 2 is complaining of abdominal pain and severe diarrhea, which has led to dehydration and electrolyte imbalance. Worried about a possible bacterial infection, you immediately order your lab team to profile each patient's gut microbiome.

a) To keep things simpler, you decide to focus your attention on these 10 common species of gut microbe:

- *Bacteroides ovatus*
- *Bacteroides thetaiotaomicron*
- *Bifidobacterium longum*
- *Eubacterium rectale*
- *Lactobacillus acidophilus*
- *Peptoniphilus timonensis*
- *Prevotella copri*
- *Roseburia intestinalis*

[3] As mentioned in the previous footnote, in ASCII, the space character comes before the `$` character, so by default, Python will sort spaces before `$`s, which would break your BWT code. To prevent this, we replaced all the space characters in this particular message with underscores.

- *Ruminococcus bromii*
- *Vibrio cholerae*

To help your patients parse their lab reports, you ask your trusty assistant to prepare a quick summary for each of these 10 microbes, discussing its prevalence in the human gut and how it participates in the gut (the functions or roles it plays, which could be beneficial, harmful, or neutral). Your assistant returns with five descriptions and promptly quits: not so trusty after all, it seems. **I guess it's left to you to complete the set by describing the other five in your README**. Of course, you already know this material cold from your days in medical school, but if you happen to find yourself a little rusty, you are welcome to search the Internet for a quick refresher on this specific task. Here are the five short descriptions your assistant left you before quitting:

Bacteroides species are usually mutualistic, benefiting their host by excluding potential pathogens from colonizing the gut. The genus as a whole comprises a considerable portion of the human gastrointestinal (GI) microbiota and individual species play fundamental roles in processing various complex molecules in the host intestine. Despite this mutualistic behavior, *Bacteroides* species are capable of exhibiting opportunistic pathogenicity, which can prove troublesome given their resistance to a wide variety of antibiotics. While both *Bacteroides thetaiotaomicron* and *Bacteroides ovatus* are critical in the breakdown of plant polysaccharides in the human gut, the former has a emerged as a popular model to study human-bacterial symbiosis due to its implication in postnatal gut development, its characterized contributions to host physiology, and its apparent ability to perform environmental sensing in order to track nutrient sources.

Bifidobacterium longum is a non-pathogenic, mutualistic bacterium that is believed to prevent growth of pathogenic organisms via production of lactic acid. Although it is not especially prevalent in the human adult gut, it is considered to be one of the earliest colonizers of the infant GI tract.

Eubacterium rectale is a normal component of the human gut microbiota, thought to produce butyrate, which is an important energy source for cells in the colonic epithelium. Its prevalence in the gut appears to be similar to that of individual species from the *Bacteroides* genus.

Roseburia intestinalis breaks down sugars in order to produce butyrate, thus possessing a protective effect against colon disease by nourishing colonocytes (similar to *E. rectale*). Depressed levels of *R. intestinalis* are associated with various bowel conditions, while elevated levels may provide protection against atherosclerosis (at least in mice) and may promote weight loss.

Develop

b) Recall the FM-index, the exact read-matching algorithm we discussed in class which iterates through a given query sequence in reverse and repeatedly calculates the ever-shrinking range of rows whose prefixes match successively longer suffixes of the query. While there are still characters in the query sequence to check, we update a pair of indices demarcating the beginning and end of the range of sorted suffixes that start with the suffix of the query corresponding to this iteration. After the final step of the iteration (in which we have finished considering the largest possible query suffix, i.e., the entire query sequence itself), if we find that the range of sorted suffixes contains at least one row with a match to the query, we can use the range indices to obtain the actual locations of those matches in the reference genome, locations which are themselves stored in the suffix array.

Based on this, write a function called `find()` inside `fm_index.py` which can take a query sequence and the necessary data structures containing various information about the text to be searched, and returns a list containing all locations of the query in the original text.

Note: You will be using your function throughout the rest of this problem, where the text will be a reference genome and the queries will be genomic reads. However, as with the BWT above, your FM-index should be able to handle arbitrary texts and queries—not just those composed of DNA bases. You may make the same assumptions about the strings as above with the BWT, and you may further assume that the queries

come from the same alphabet as the text (i.e., no query will contain a character that didn't appear in the original text).

c) Your crackerjack lab team is back! They've managed to produce 100,000 reads from each of your three patients. Unfortunately, you fired your bioinformatician last month—a short-sighted move, if ever there was one—so it's now on you to do something with all these reads. Hazily remembering back to your time in college, you recall a problem set where you had to code up a read aligner in Python ("Ah, it seems like it was only yesterday!").

Write Python code within the `infection_investigator.py` file that leverages (by importing its functions) your `fm_index.py` code to map all these reads: For each of the 3 patients, your code will need to take each of the 100,000 reads from that patient's gut, and figure out to which bacterial genome(s) each read aligns.

To enable this, you should write a helper function called `find_aligned_reads_for_patient()` within `infection_investigator.py`. This helper function will take as input a list of patient reads and BWT data representing each bacterial genome, and it will output a dictionary whose keys name the bacterial species and whose values list the start positions of reads that *uniquely* map to each bacterium for that patient. [*Note*: The end positions are not necessary to store in this case because each read is assumed to be the same length, 50 bases.]

The returned dictionary should look something like:

```
{
    'Bacteroides_ovatus': [8, 124, 179, ... ]
    ...
}
```

In building your helper function, you should use the following information:

- We provide you with the genomes of the 10 bacterial species above in FASTA files. However, to reduce time and space usage, the FASTA files we provide only contain a 15 kbp chunk from each genome. So in this problem, you'll pretend that each genome is just 15 kbp long, whereas in reality, they're longer. *Note:* **Please do not include these files when submitting your work to Gradescope.** Even these abridged genomes take up a fair bit of space in memory.
- You may not assume that every one of the 100,000 reads will align to one or more of these 10 genomes. For any reads that fail to align to any of the genomes, simply ignore those reads when estimating the prevalence of each bacterial species, as described below. However, for those reads that do align to one or more of the 10 genomes, you can be assured there have been no sequencing errors (i.e., you will only be looking for perfect matches).
- Furthermore, the sequencing protocol is such that somehow, all the reads are complementary to the genome sequences we provide. So you don't need to check both strands, just one. The `reverse_complement` function in `compsci260lib.py` should be helpful to you in this regard. You can reverse-complement the reads before you align them to the genomes, or reverse-complement the genomes before you give them to your read aligning code. You might want to think about whether one of these might be preferable to the other (for reasons of efficiency or simplicity); you should be careful about reporting coordinates properly in terms of the reference genome sequences being queried.
- You may not assume that each read aligns uniquely, which is to say that it is present in only one genome. It is certainly possible for a read to appear in more than one genome. If that happens, since you don't know what genome it came from, the read should not be used when estimating the prevalence of each bacterial species, as described below, and `find_aligned_reads_for_patient()` should not return it.
- As a tip, while you are testing your code, you'll probably want to use only the first 100 or 1000 reads to save a little run time until you're confident everything is working correctly. As another tip, it might

help to have your program write its results to a file and not only to the console so that you don't have to re-run your code every time you want to look at its output.

Apply

d) Once you know where all the reads map, you can use the reads that map uniquely to estimate the prevalence of each microbe in each patient. In particular, for each patient, the estimated prevalence (proportion) of each microbe is just the number of reads mapping uniquely to that microbe's genome, divided by the total number of reads that map uniquely to any of the 10 genomes.

For example, say there are 10 reads sequenced from a patient's stool sample, and 3 of those reads map to *Bacteroides ovatus*, 3 map to *Bacteroides thetaiotaomicron*, and 4 map to more than one bacterial genome. In this example, the prevalence of both *Bacteroides ovatus* and *Bacteroides thetaiotaomicron* is 0.5 (3 of 6 uniquely mapping reads), and the prevalences of the remaining microbial species are 0.

Report the prevalences of the 10 microbes in the guts of each of your 3 patients.

Reflect

e) *As a world-leading gastroenterologic scientist, what stands out to you?*

⋆ **Step 2:** Toxic waste: Understanding precisely what ails your patient

Plan

Wait, something funny seems to be going on. Among other things, you definitely notice a marked elevation of a particularly species in patient 2 (which makes total sense), but one of the asymptomatic patients *also* seems to be highly elevated for the same species (which makes no sense). You are going to have to dig a little deeper.

An idea suddenly—and rather miraculously, you must admit—pops into your head: You remember that sometimes a bacterium can be infected by viruses (such a virus is called a bacteriophage, or phage), and that these viruses can insert their own genes into the host bacterium's genome, in some cases adding genes coding for toxin proteins. On a hunch, you decide that maybe patient 2 has a strain of this species with a genome that contains a toxin gene of viral origin, while the other patient has a strain with a genome that is missing this toxin gene. You confirm that the reference genome you've been using for this species contains a copy of the toxin protein in it. So, if your hunch is correct, you should see patient 2's genomic reads distributed all throughout the reference genome, while for the asymptomatic patient—since a segment of the reference genome is missing from the strain's genome—when you map that patient's genomic reads to the reference, there should be a segment of the reference genome with no reads mapping to it at all.

Develop

f) Add more code to your `infection_investigator.py` program that allows you, for a patient and species of interest, to count the number of the patient's reads mapping not just to that genome as a whole (which you did before), but to each of the 15,000 locations within that genome. So your output will be a list of counts of length 15,000. We have provided a skeleton function called `read_mapper()` for you to complete (please don't change the name or arguments). Unlike in part **(c)**, here you can safely assume that each read will map to only one location in the genome.

Apply

g) Use your new code to produce the count vectors for the suspicious species for the two patients in question. Then call the function we have already provided for you called `longest_zeros()` which will take a count vector as input and will return the start and stop positions (inclusive) of the longest string of 0's within the vector. To avoid issues that might arise from edge effects, this function ignores any leading or trailing 0's in the count vector. That is, it looks for the longest *internal* string of 0's.

Report the outputs of this function for the two patients in question, respectively.

Reflect

h) If your hunch is correct, and there is a long string of zeroes in the asymptomatic patient's count vector, have your code extract the corresponding sequence from the reference genome and **report** it. Now use BLAST to identify it. *What do you see? What have you learned? What do you tell your patients?*

Now go home, put your feet up, and bask in your own glow, you metagenomic superhero.