COMPSCI 260, Introduction to Computational Genomics **30 September 2016**

Problem Set 3

Due: 5pm on 14 October 2016

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

Consider this simplified model of whole-genome shotgun sequencing:

1. many copies of a genome of length G are broken at many points uniformly at random (the probability of breakage at each location in the genome is uniform over the locations)
2. the location of each fragment so produced is 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 cloned fragment, exactly L nucleotides are sequenced from one end of the fragment, yielding a total of R sequence reads of length L (where L is much, much smaller than G)

a) The coverage C of a 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).

Hints: Remember that we ignore edge effects and we 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 and can perfectly assemble the reads it is given, without needing to satisfy any minimal length overlap requirements: it just knows precisely where every read should go in the genome (but of course, can learn nothing about nucleotides that are not in any read and thus remain unsequenced). What is the expected number of contigs reported by such an algorithm? What is the expected length of each contig?

[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).

Problem 2: Beyond comparison (20 points)

a) Write a Python program called `simulate.py` that simulates the sequencing process and computes (empirically) the quantities in parts a), b), and c) from the previous problem. In this simulation, you do not need actual DNA sequences. Instead, you will represent a genome of size G using a list with exactly G elements, where each element contains relevant information about that position, *e.g.*, how many times the nucleotide at that position was sequenced. To simulate the sequencing process once, follow these steps:

- Set all the elements in the list to 0 (to mark that each nucleotide has been sequenced 0 times).
- Randomly select starting locations for R reads of size L, and for each read update the number of times the nucleotides covered by the read were sequenced.

Setting $G = 3 \times 10^6$, $R = 4.5 \times 10^4$, and $L = 500$, simulate the sequencing process 20 times and each time answer the following questions:

- What is the coverage (the average number of times a nucleotide in the genome was sequenced)?
- How many nucleotides were not covered by any read?
- Assuming you can use the oracular assembly algorithm mentioned in Problem 1(c) to assemble all the reads, what is the number of contigs in the assembly and what is the average length of a contig?

Then, average the values computed above over the 20 simulations and compare them with the values computed using the expressions you derived in parts a), b), and c) from Problem 1. Do you get similar numbers? If not, what might be different?

b) 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?

During the first step of the assembly, each read needs to be compared with each other read in each of the two different relative orientations. 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 subsequence comparison operations that would need to be done while sliding one read along the other.)

Assume an assembler of that era could perform 50 million read comparisons per second. How long will it take to complete the first step of the assembly?

When the assembler is finished, 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?

Comment on the values you have computed in terms of sequencing the human genome. Are the numbers reasonable? What strikes you?

Problem 3: Transformers: More than meets the eye (35 points)

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 the 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 specific manner so as to make compression easier (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 comprising long runs of similar characters. To illustrate, the result of performing the BWT on `TATCGTACACTACGTACGA$` (where `$` is the EOF, or end-of-file character) is `AGTTTCTAATAACCCGGC$A`.

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

b) Suppose you are given a sequence `C$TTTACAG`, 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 will not be sufficient to simply report the original sequence.

Hint: If you wish to check your result, you may employ the BWT in the forward direction (i.e., the algorithm you employed in part (a)) on the answer you obtained here. Assuming you have performed the reversal correctly, you should find that your "original sequence" generates the permuted sequence given in this problem: `C$TTTACAG`.

c) In the Python program `bwt_structures.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 `$` character to signify the end of the string and returns the result of the BWT applied to that string. In `reverse_bwt`, write code that takes a string which has been permuted via the BWT and return the original string from which it was generated (if you've done things correctly, you should find that the original string ends in a `$` character). You may wish to test the correctness of your methods by using the example we provided at the top of this question and the sequences you permuted by hand in parts a) and b). You can also test using the output of one function as the input for the other function since each is the inverse transform of the other.

d) Recall the exact read matching algorithm which backtracks through a given query and calculates the range of rows beginning with successively longer suffixes of the query. We iterate backwards over characters in the query sequence. While there are still bases to check, we update the pointers corresponding to the beginning and the end of the sorted suffixes which start with the suffix of the query string we are currently examining. This can be accomplished with `new_beginning = counts[character]+rank[character][old_beginning]+1` and `new_end = counts[character]+rank[character][old_end]`.

Here `counts` tells us how many characters in the sequences are lexicographically smaller than the character we are currently examining (i.e., how many suffixes in the sorted suffixes precede any suffix which begins with the character we are currently examining). Meanwhile `rank` tells us, for a position in the BWT-permuted reference, how many times a given character has appeared up to and including that position (think about the last-first property). If we get to the end of the query and we find that the sorted suffixes contain at least one match to our query, than we can use the beginning and end positions in order to obtain their locations in the reference genome which are stored in the suffix array: `suffix_array[begin:end]`. Based on this, write a program titled `read_aligner.py` which can take a query and the necessary data structures containing various information about the reference genome and returns a list containing all locations of the query in the original reference genome.

Problem 4: I just have a gut feeling (25 points)

You are a hot-shot gastroenterologist, at the cutting edge of modern medical technology. Whether to generate research data, improve your diagnoses, reduce the chances of being sued for medical malpractice, or just keep your billing rates high, you have adopted a policy that any time a patient comes in to see you, you order a routine analysis of his/her gut microbiome. This will allow you to profile the diverse community of microbes living in a patient's gut. Each patient is 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 genomic DNA of all the different kinds of microbes mixed together in different proportions within the sample, randomly fragmenting all that DNA, and then generating short reads from the fragments that result.

Today has been a slow day, but all of a sudden, three patients have come in to see you in the last hour. Patients 1 and 2 are asymptomatic, but patient 3 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 each patient's gut microbiome to be profiled.

a) To keep things simpler, let's say you decide to focus your attention on only 10 common species of gut microbe, as indicated below. For each of these 10 microbes, briefly discuss (imagine 1-2 sentences, depending on whether you write like Faulkner or Hemingway, respectively) its prevalence in the human gut and how it participates in the gut (whether it's commensal or mutualistic and the functions or roles it plays). Of course, you already know this 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 here.

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

b) 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 create a read aligner based on the BWT ("Ah, it seems like it was just yesterday!").

Write Python code within the `infection_investigator.py` file that leverages your `read_aligner.py` code to map all these reads. Your code should do the following: for each of the 3 patients, take each of the 100,000 reads from that patient's gut, and figure out to which bacterial genome(s) each read aligns, using 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.
- You can assume that every read was generated from one or more of these 10 genomes, without any sequencing errors. So you'll be looking for perfect matches. In other words, each read should be a substring of one or more of these genomes.
- 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 provided `reverse_complement` function 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 these might be preferable to the other (for reasons of efficiency or simplicity).

- 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, do not use the read when estimating the prevalence of each bacterial species, as described below.
- As a tip, while you are testing your code, you may 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.

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.

What are the microbe prevalences for each of your 3 patients? As a world-leading gastroenterologic scientist, what stands out to you?

c) Wait, something funny seems to be going on. Among other things, you definitely notice a marked elevation of a particularly species in patient 3 (which makes total sense), but patient 2, who is showing no symptoms, seems to *also* 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 3 has a strain of this species with a genome that contains a toxin gene of viral origin, while patient 2 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 supposedly contains the toxin protein in it. So, if your hunch is correct, you should see patient 3's genomic reads distributed all across the reference genome, while for patient 2, since a segment of the reference genome is missing from the strain genome, when you map patient 2's genomic reads to the reference, there should be a segment of the reference genome that has no reads mapping to it at all.

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.

Use your new code to produce the count vectors for the suspicious species for patients 2 and 3. Then create a function which will take a count vector and will return the start and stop positions of the longest string of 0's within the vector. To avoid issues that might arise from edge effects, this function should ignore any leading or trailing 0's in the count vector. That is, it should look for the longest *internal* string of 0's.

What are the outputs of this new function for patients 2 and 3, respectively?

If your hunch is correct, and there is a long string of 0's in patient 2's count vector, extract the corresponding sequence and 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 superhero.