Problem Set 1: Exact Pattern Matching

Handed out Friday, January 24. Due at the start of class Monday, February 3.

Homework Information: Some of the problems are probably too long to attempt the night before the due date, so plan accordingly. No late homework will be accepted. You must cite any sources used outside of the course materials and textbook, including anyone you discussed the problems with.

1. (2 pts) Construct a suffix tree for the string CACATGCAC$. If you wish, you may write code to do this, but it is not required. You are welcome to do this by hand. If you write code for this, please email me a copy of your code. If you do this by hand, please show all intermediate steps.

2. (2 pts) Construct the pattern repeat table T as used by the Knuth-Morris-Pratt algorithm for the string CATCATCAG.

3. (6 pts) Given a long nucleotide sequence of length n, we want to find the number of occurrences of a given l-mer of arbitrary length m.

a. What would be the appropriate data structure - Hash-table or Suffix tree?

b. Design an efficient algorithm for this problem using the data structure chosen in (a). Note: The running time should be independent of the length of the nucleotide sequence.

c. What is the time complexity for the query search?

4. (4 pts) Often times when analyzing DNA sequencing reads, we'll want to find sets of reads that "overlap" each other. Overlap here means that some suffix of one read matches some prefix of another read. The longer the overlap, the more likely it is that the pair of reads originated from the same region in the genome. For example the strings A<u>CGTTCGT</u> and <u>CGTTCGT</u>A have a large overlap of 7 characters (underline). Describe an algorithm that uses a suffix tree that solves the following computational problem.

Input: Two strings $s = s_1 s_2 ... s_n$ and $t = t_1 t_2 ... t_m$.

Output: The starting index of all suffixes of s that match some prefix of t.

What is the asymptotic runtime of your algorithm? You may assume the use of Ukkonen's algorithm for building the suffix tree.

5. (6 pts) Boyer-Moore Algorithm. Boyer-Moore is another algorithm for exact pattern matching. Boyer-Moore can be implemented to run in $O(m + n)$. We won't go into Boyer-Moore in detail. Instead, here are some of the key ideas behind the algorithm.

- We will move the pattern **p** along text **t** from the left-to-right (as done in KMP), but at each location where we put the pattern, we will compare **p** and **t** right-to-left.
- Boyer-Moore has two different rules for shifting the pattern by more than 1 character at a time. These heuristics will guarantee that no match will be missed, but they may not always apply. Whichever shift is the largest is the one used. If you are curious about these heuristics then check out this `http://www.cs.jhu.edu/~langmea/resources/lecture_notes/04_boyer_moore_v2.pdf`, but you don't need to know them for this problem.

Download the zip file `Boyer-Moore.zip` from Moodle required for this problem. Make sure to unzip this file to obtain three files: `compareMatching.py`, `chr19.fasta` and `Alu.fasta`. The `compareMatching.py` file contains Python code that runs the Boyer-Moore algorithm already. Take a quick look through the code, but again, you don't need to understand how Boyer-Moore works. You will be completing two functions in this code to be able to compare Boyer-Moore to the brute force algorithm and counting how many individual character comparisons each algorithm does. Once this is done, you'll run the code using chromosome 19 from the human reference genome and looking for a particular Alu sequence (`https://en.wikipedia.org/wiki/Alu_element`)(a particular sequence of DNA that appears many times in primate genomes and has been implicated in some human diseases).

Your Task

1. Look through the code in `compareMatching.py` and trying running it to see what happens.
2. Finish implementing the following two functions: `naive` and `read_single_fasta`. Complete specifications on these functions are included in the comments in the code.

   For the Naive algorithm you don't need to consider capitalization. So for example, g should match to G. When writing the code to read FASTA files, make sure to convert all characters of the resulting strings to upper or lower case. The reference genome uses both cases, but we don't want to use that information right now. You'll need to do this for how Boyer-Moore is currently written.
3. Make sure your code is working correctly by modifying the test case located in the `main` function. (For comparison, my naive algorithm did 24 comparisons on the original test case and only 10 comparisons using the Boyer-Moore algorithm.)
4. Run your code on the real data by uncommenting the second section of the `main` function. **NOTE:** Even though we are only running on chromosome 19, your code WILL NOT RUN INSTANTANEOUSLY. My code took about 8 minutes to run on my laptop.
5. Turn in the following: (1) The number of comparisons done by both the naive algorithm and the Boyer-Moore algorithm on the real data. (2) The code for only your completed `naive` and `read_single_fasta` function. Do this by printing out a hard copy of these two functions and including it with your written homework submission. DO NOT turn in the entire python file.