CSE 584 Algorithms for Biosequence Comparison Spring Semester 2016

Homework 1

Assigned: 2/1/2016 *Due Date: 2/22/2016*

This homework must be completed and submitted electronically. Formatting standards, submission procedures, and (optional) document templates for homeworks may be found at

`http://classes.engineering.wustl.edu/cse584/ehomework/ehomework-guide.html`

Advice on how to compose homeworks electronically, with links to relevant documentation for several different composition tools, may be found at

`http://classes.engineering.wustl.edu/cse584/ehomework/composing-tips.html`

Please remember to

- **create a separate PDF file (typeset or scanned) for each problem;**
- **include a header with your name, WUSTL key, and the homework number at the top of each page of each solution;**
- **include any figures (typeset or hand-drawn) inline or as floats;**
- **upload and submit your PDFs to Blackboard before class time on the due date.**

Always show your work.

1. (**20%**) When performing pattern matching in DNA, a useful type of imprecision is the ability to specify *residue classes*. As one example, the temperature at which a DNA molecule "melts" (i.e. the two strands of the double helix separate) is determined not only by its length – longer strands melt at higher temperatures – but also by the proportion of "weak" (A or T) vs. "strong" (C or G) bases that it containts. Strands of a given length with a higher proportion of strong residues melt at higher temperatures.

   In this problem, we want to search a DNA database for matches to a pattern P in which each character is either a single DNA base or one of the symbols W or S, denoting the weak and strong classes respectively. A symbol W or S in the pattern matches either element of its class. For example, the pattern AWG would match either of the text strings AAG or ATG, while the pattern $SSSS$ would match any of the 16 possible 4-mers composed entirely of C's and G's.

   (a) We can easily extend the definition of sp_i values in the KMP algorithm to include prefix-suffix matches in which one or both substrings contain residue classes. Show, however, that if the pattern contains a residue class at even a single position, the pattern-shifting rule used by the KMP algorithm can yield incorrect answers. Explain why the rule fails.

   (b) Describe how to extend the basic KMP algorithm to work correctly using a pattern with a residue class at exactly one position. *Hint*: make your *sp*-values conditional on which character matched the S or W. Argue that your revised method is still correct.

   Don't worry about explaining how to compute *sp*-values in this revised algorithm, but do prove that, once you have these values, your revised algorithm preserves KMP's property of performing at most $2|T|$ comparisons to the text.

   (c) How much space (asymptotically) do you need to store *sp*-values for a pattern with residue classes in k positions? Justify your answer.

2. (**20%**) Give an algorithm to construct the table of alphabet-dependent failure links $g_{i,c}$ for the improved KMP algorithm, given a pattern P. Your method should run in worst-case time $O(|P||\Sigma|)$ for patterns over an alphabet Σ. Prove the correctness and running time of your algorithm.

3. (**20%**)

   (a) Describe how to extend the Aho-Corasick algorithm presented in class to work when one pattern can occur as a *prefix* of another pattern, but not as an arbitrary substring of it. For example, the pattern set "he, here, ham, hambone" satisfies this property. What additions do you have to make to the Aho-Corasick automaton to find all occurrences of each pattern?

   (b) Describe how to extend the Aho-Corasick algorithm to work when one pattern can occur as an *arbitrary substring* of another. For example, the pattern set "he, she, they, them" satisfies this property. Again, what additions do you have to make to the Aho-Corasick automaton to find all occurrences of each pattern?

   For each subproblem above, any extensions you make should preserve the automaton's asymptotic size (proportional to at most the sum of the pattern lengths). The search must still complete in time proportional to the length of the text *plus the number of matches found*.

4. (**40%**) This programming problem asks you to implement the Aho-Corasick algorithm to discover instances of a set of patterns in a long DNA sequence.

   The input for this problem will consist of two files: a *corpus file*, containing a single large DNA sequence, and a *pattern file*, containing one or more short DNA patterns. DNA sequences will consist of the ASCII letters A, C, G, and T (please ignore the case); patterns will be separated by newlines.

   Your goal is to identify all matches between the corpus and any pattern string. Pattern matches may occur with respect to *either strand* of the text (forward or reverse-complement). For each match found, you should emit a triple of integers the form "t p s", where

   - t is the position at which the match starts in the corpus file. The first character is position 1.
   - p is the line number of the matching pattern the pattern file. The first pattern is line 1.
   - s is 1 if the match is on the forward strand of the corpus file, or -1 if it is on the reverse-complement strand.

   Please sort your output first by t, then by p, and finally by s.

   You should use the Aho-Corasick algorithm, using the language of your choice, to construct a suitable automaton from the pattern set and then make a single pass over the corpus to identify all matches. You may use either the version with failure links f_q or the version with character-dependent transitions $g_{q,c}$, as you desire.

   You must support the general case in which one pattern can occur as a substring of another. (This requires solving Problem 3 above.)

   Test Cases: (forthcoming)

   Where to Turn In Your Solution: each student has an SVN repository at the following URL:

   `https://svn.seas.wustl.edu/repositories/yourid/cse584a_sp16/hwk1`

   where "yourid" should be replaced by your WUSTL Key ID (e.g. "jbuhler"). To turn in your solution, please check the following items into this directory before the due date and time:

   - your code
   - its output on each of the supplied test cases. Name the output files `case1.out`, `case2.out`, and so forth
   - a README file describing any interesting properties of your implementation (including bugs, if known)

   Please do not check in copies of the test case files, especially the (very large) corpus files!

   Advice on Implementation: In addition to implementing the construction and search algorithms, you will need code to read the corpus and the pattern file. The patterns can all be read in at the start of the program, but you should *not* read the entire corpus string into memory at once. Use your language's streaming I/O (e.g. `ifstream` for C++, `FileInputStream` or similar for Java) to read only a limited amount of sequence into memory at a time. You can either read single characters or read chunks of data into a large (up to a few megabytes) finite-sized buffer.

   Your life will probably be easier if you map the ASCII input characters to a small, contiguous range of integer values as you read them in, e.g. $A = 0$, $C = 1$, etc. You should also write simple procedures to map a DNA base to its complement and to compute the reverse-complement of a pattern string.

   There will *not* be an "autograder" for this assignment, and I do not expect to compile or run your code myself (though I will read it and look at the output you provide). Be sure to devise your own small test cases with known output to check that your code is working correctly.