CSE 584 Algorithms for Biosequence Comparison Spring Semester 2016

Homework 2

Assigned: 2/22/2016 *Due Date: 3/9/2016*

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. (**30%**)

   (a) Let τ be a suffix tree for a string S. Suppose you are at a node x of τ, and you know that some descending path that starts from x is labeled with the string α. Show how to find the end of this path in time proportional to the number of *edges* on the path, independent of the number of characters in α.

   (b) Let τ be a suffix tree for a string S augmented with its *suffix links*; that is, for each node x s.t. the path from the root to x is labeled with a string $c\alpha$, there exists a pointer $x \rightarrow y$ to a node y whose path from the root is labeled with α.

   Show how to extend the trivial algorithm for matching a pattern P in τ into an algorithm for *substring matching*, i.e. an algorithm to find the longest prefix match to each suffix of P in τ. Prove that your algorithm runs in worst-case time $\Theta(|P|)$, independent of the tree size.

   Hint: use the suffix links and the trick from part (a) to figure out where to restart the matching algorithm after each suffix of P. You can separately bound the number of character comparisons and the number of edge traversals performed by the algorithm.

2. (**20%**) A repeat α in a string S is said to be *supermaximal* if no instance of α occurs as substrings of an instance of some larger repeat. For example, in the string $S = acgtcacgtgacgtc$, the string "acgt" is *not* a supermaximal repeat, since one of its instances appears as substrings of an instance of the longer repeat "acgtc". However, *acgtc* itself *is* a supermaximal repeat, since its sole instance is maximal.

   Show how to extend the suffix tree-based repeat finding algorithm we discussed in class to enumerate all instances of supermaximal repeats in a string S in time proportional to their number, plus at most linear extra work in $|S|$.

3. (**50%**) Implement the linear-time DC3 suffix array construction algorithm, using the language of your choice. Try to use as little memory as possible for your implementation; I think you can do it in well under 20 bytes per character of the input string. You may use any 3-difference cover you like; I think $\{0, 1\}$ is a good choice, as is $\{1, 2\}$ (which works even though it is not formally a difference cover by our definition).

   Using the chromosome 1 string S from Homework 1 as your input, you should

   (a) Append a terminal "$" character to S;

   (b) Compute its suffix array A;

   (c) Emit the corresponding *Burrows-Wheeler transform* of S, i.e. a string whose ith character is $S[A[i] - 1]$. (If $A[i] = 1$, the *ith* character of the BWT should be "$".)

   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/hwk2`

   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
   - the requested BWT, preferably compressed with ZIP, gzip, or bzip2
   - a README file describing any interesting properties of your implementation (including bugs, if known)

Please do not check in copies of the input file.

Implementation Advice: To save memory, do not keep around arrays for longer than you need them. In Java, explicitly set your reference to an array to `null` when you are done using it, so that the garbage collector will recycle it. In C++, use `delete[]` to free arrays at the appropriate time. It is possible to reuse arrays in some cases, but I would not recommend it for reasons of code clarity.

You can compute the *pos* function, which maps indices in the original string S to indices in the shorter string T for the recursive computation, without creating an array. You can also avoid explicitly computing the arrays I_x, or even the array B mapping suffixes in S to bin numbers – just do a linear sweep over the sorted version of S_D to find the bin boundaries, and fill in the characters of the recursive string T out of order during this sweep. Be sure to use integer arithmetic, not floating point, for all operations.

I can't see an easy way to avoid allocating all of R_D, A_D, A_L, and A concurrently.