CSE 584A Algorithms for Biosequence Comparison Spring Semester 2018

Homework 3

Assigned: 3/19/2016 *Due Date: 4/9/2016*

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

`https://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

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

Please remember to

- **create a separate PDF file (typeset or scanned) for each problem;**
- **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%**) Suppose we have the BWT of a text T, but *not* the original text, and that we've augmented the BWT to support constant-time rprev computations. Given a constant k, show how to further augment the BWT to answer queries of the form "what is substring $T[i..j]$?" in worst-case time $\Theta(j - i + k)$, using additional space at most $(n \log n)/k$ bits.

   Note that you cannot just keep a copy of T around, because $(n \log n)/k < n \log |\Sigma|$ for large enough k.

2. (**30%**) Consider the algorithm for enumerating the internal nodes of a suffix tree using the 2BWT. We gave a recursive formulation for simplicity of exposition, but this problem will consider how to code the algorithm with an explicit stack instead.

   (a) Consider this (simplified) version of the EnumTree code from the class notes:

   ```
   enumTree(I, d)
     if I is right-diverse
       for c ∈ Σ do                          ▷ possible inverse links in τ'
         extend I backwards by char c to obtain maximal bi-interval I'
         enumTree(I', d + 1)
   ```

   Write an iterative version of enumTree using an explicit stack to replace the recursive calls. Your code should start from the root of the tree. To process a right-diverse maximal bi-interval I, it should first find *all* the maximal bi-intervals I' obtained by extending I, then push them onto the stack.

   (b) A problem with the the above code (and with the recursive implementation) is that the stack depth can become very large – as large as the depth of the tree, which is bounded only by the length of the text T. This means that we'd have to spend $\Theta(|T|)$ space in the worst case to store the stack in memory, which is a Bad Thing if we are trying to save space by using the 2BWT in the first place!

   Consider the following heuristic improvement to the iterative algorithm. *When enumerating extensions I' for a given I, push the largest bi-interval I' onto the stack first.* Prove that this heuristic guarantees that the stack depth never exceeds $\Theta(\log |T|)$.

3. (**50%**) One way to evaluate the complexity of a DNA sequence T is to count the number of distinct k-mers that occur as substrings of T, for some fixed length k. Very repetitive sequences have only a small number of distinct k-mers, while (for long enough k) non-repetitive sequences have a number of distinct k-mers close to their length.

   Suppose we have a suffix tree τ for a string T. Let L be the number of leaves in τ, and let V_k be the set of internal nodes whose label has length at least k. For any internal node $v \in \tau$, let $c(v) > 1$ be the number of v's immediate children. Then the number N_k of distinct k-mers in τ is given by the following formula (see Belazzougui and Cunial, *Proc. 2015 Conf. on Combinatorial Pattern Matching*):

   $$N_k = L - k + 1 + \sum_{v \in V_k} (1 - c(v)).$$

   The justification for this formula is roughly as follows. The number of distinct k-mers is equal to the number of tree nodes at depth $\geq k$ whose parent is at depth $< k$. We begin our estimate of N_k with the number of leaves of depth $\geq k$ in the tree; then, for each internal node at depth $\geq k$, we add one for this node *and subtract one for each of its children.* In this way, the sum includes a net contribution of 0 $(1 - 1)$ for each node at depth $\geq k$ whose parent is also at depth $\geq k$, and a contribution of 1 for

each node at depth $\geq k$ whose parent is at depth $< k$. Note that we can enumerate the nodes of V_k in *any* order and still get the right answer.

Your job in this assignment is to implement the computation of N_k for a very large DNA sequence. Since we don't want to spend the space required to build the actual suffix tree for such a sequence, you should instead convert the sequence to its 2BWT, using your BWT constructor from Homework 2, and then use the 2BWT-based algorithm for enumerating all internal nodes of a sequence's suffix tree to implement the formula for computing N_k. Use the iterative, bounded-stack version of the enumeration algorithm from the previous problem!

After validating your implementation on some small strings for which you can check the answer by hand, apply it to the human chromosome 1 sequence from Homework 1. Report the number of distinct (case-insensitive) k-mers in this sequence for each $15 \leq k \leq 40$. You need consider only k-mers that appear on the **forward strand**, not the reverse-complement. Since your BWT construction appends a "$" character to the sequence, be sure to correct your output so as not to count the k-mer that ends with this extra character.

Where to Turn In Your Solution: each student will have a Git repository through GitHub Classroom.

To turn in your solution, please check the following items into the assignment repo before the due date and time:

- your code
- the requested output
- a README file describing any interesting properties of your implementation (including bugs, if known)

Note that GitHub will not let you check the entire BWT of the input string into your repo, because it's too big.

Implementation Advice: to implement the enumeration of suffix tree nodes, you need to compute C and *occ* for each direction of the 2BWT. You do *not* need to compute or store a compact suffix array for your sequence. Moreover, you do not have to print the nodes of the tree (unless you are using these yourself for debugging). Finally, you're working with DNA, so don't bother with a wavelet tree or similar fancy thing – a simple $4 \times |T|$ table of *occ* values, decimated by a decent factor (I suggest between 32 and 128), will suffice.

The right-diversity check that forms part of the traversal algorithm will also give you $c(v)$ for suffix tree nodes v.

If your BWT constructor from Homework 2 is broken, let me know, and I'll help you get the necessary BWTs a different way.