PROGRAMMING PROJECT 2

Cpt S 471/571, Spring 2020

Due: March 23, 2020 (Monday), 11:59pm PST @ Blackboard Project #2 dropbox

---

General Guidelines:

- For this programming project, you ***are allowed to work in teams of size (up to) 2 each***. Teaming up with someone else is not mandatory but highly encouraged. Note that regardless of whether you decide to work in teams or not, the assignment will be graded strictly on its merit and the same grade will be given to all team members.
- Reproduction/reuse of source codes from online resources or other's people's assignments in any shape or form, is *strictly not* allowed. All the source code should be solely yours. You are allowed to use any standard library functions supported by the underlying programming language, but if that directly implements any of the functions posed in the question of the assignment then you should **not** use those libraries and instead write your own code.
- It is recommended that you use the same language that you used for the Project 1, because you will be using both the codes in Project 3. To this effect, it is also recommended that you design a structured interface (API) and modularized code structure, suitable for reuse later.
- Also, you are allowed to reuse any code that you have written from Project 1 (this will be particularly useful for Input reading).
- Submissions without the source code will NOT be graded and will be returned with 0 points.
- Grading will be based on a combination of factors such as correctness, coding style, implementation efficiency, exception handling, source code documentation, and code reusability.

Submission: The assignment should be zipped folder and both the folder *and* the zip file name should have your name on it. The folder **naming convention** is as follows: Program2-X.zip (if you are working alone and if your name is X) or Program2-X_Y.zip (if you are working as part of a team and the member names are X and Y - in any order). The zip file is the one that should be uploaded onto the WSU Blackboard dropbox (learn.wsu.edu) for Project #2. **Submissions not following this naming convention risk being not graded!** So please pay close attention.
Submissions are due by 11:59pm PST on the due date. A 24-hour grace period is allowed although it will incur a late penalty of 10%. Submissions that are more than 24 hours late will be returned without grading.

- Note: If you are submitting as a team (of 2 people) then both of you should submit TWO IDENTICAL COPIES separately on Blackboard. Just make sure you add the information of your team participants in the COVER SHEET.

Assignment: SUFFIX TREE CONSTRUCTION

The goal of this programming assignment is to implement the **McCreight's suffix tree construction algorithm.**

You are expected to follow the detailed algorithmic pseudocode provided in the lecture notes (along with the scribes):

http://www.eecs.wsu.edu/~ananth/CptS571/Lectures.index.htm (please refer to the notes on McCreight's algorithm; follow the same conventions to help code readability).

Please also try to follow, to the extent possible, the same conventions and notation used in those notes (for variable and function naming). Deviation from this convention could make debugging and code comprehension hader. Here are the notation I expect you to use the same way (not limited to):

node *u* : suffix leaf i-1's parent)

node *v* : suffix link SL(u)

node *u'*: parent of u (if exists)

node *v'*: SL(u') - may or may not be the parent of v

Case labels:

IA) SL(u) is known and u is not the root

IB) SL(u) is known and u is the root

IIA) SL(u) is unknown and u' is not the root

IIB) SL(u) is unknown and u' is the root.

Function naming:
FindPath(args): finds the path starting at the specified node argument that spells out the longest possible prefix of the specified string argument, and then insert the next suffix.

NodeHops (args): does node hopping child to child until string Beta (or Beta') is

exhausted, depending on the case.

In addition to that algorithm, however, there are some additional instructions for this project which are detailed below.

Input: An input string s (in the FASTA file format) and a file containing the input alphabet (format details below).

Goal: To build a suffix for the input string adhering to following API specifications.

- The API should have a header file in which the suffix tree data structure (or class if you are using Java/C++) is defined. The struct/class should support the following *public* methods:
  i) ST construction method that takes an input string and alphabet as function parameters;
  ii) given a pointer to a specific node *u* in the tree, display *u*'s children from left to right;
  iii) enumerate nodes using DFS traversal (i.e., visit all children from left to right AFTER visiting the parent). Note: this will enumerate nodes of the tree in a top-down fashion. As a result of this enumeration you should display the STRING DEPTH information from each node. No need to display any other information. As this display can get too long (== number of nodes in the tree) you can simply break the output into lines with say 10 entries in each.
  iv) BWT index: Modify the DFS procedure to print the BWT index for the input string s. This can be done by enumerating ONLY the leaf node id's from left to right (lexicographically smallest to largest). The BWT index is an array B of size n, given by B[i] = s[leaf(i)-1], where leaf(i) is the suffix id of the i^{th} leaf in the lexicographical order.

- Each node in the ST should be given a unique ID (an integer identifier) between 0 and N-1, where N is the total number of nodes (internal nodes + leaves).
- Each internal node should be a structure with at least the following information: { node ID, suffix-link pointer, parent pointer, parent edge label (i.e., label of the incoming edge from the parent), children pointers, string-depth}. You can have other fields as needed but I don't expect you will need anything more.
  Note: If the node is a leaf, then you can use the node ID to store the suffix number of the leaf. For internal nodes, it does not matter what ID you store as long as its some unique integer in the allowed interval.
- In your construction code, remember to concatenate your original input

string with the $ sign before building the tree.
- During construction, make sure to order your children in the lexicographic order of their edge-labels. As a convention, assume that the $ symbol is lexicographically SMALLER than all the other symbols in your alphabet.
- Write a test "user" code that includes this header file and calls the *main(...)* function to test all the above functions. Basically it should read in an input FASTA and alphabet files, build the ST, then print some basic statistics about the tree (#internal nodes, #leaves, #total number of nodes, size of the tree (in bytes), average string-depth of an internal node, string-depth of the deepest internal node). After this, for further testing, printout a couple of randomly handpicked node's children list, and then call the function to enumerate the nodes in DFS and post-order traversals and output the results into two corresponding output files.

$ <test executable> <input file containing sequence s> <input alphabet file>

Input File Formats:

- The FASTA file format is the same format used in PA#1. For this assignment you can assume that there is only one string in the file. But that string can be spread over multiple (contiguous) lines in the input FASTA file.

- For the input alphabet, specify a file with all the symbols in a single line delimited by space. For coding convenience, you can assume the contents of this file to be case sensitive. For example, if you specify {A,C,G,T}in the alphabet file, you can safely assume that the input file has its sequence written in uppercase. Conversely, if you find a lowercase letter somewhere in the input sequence then you could exit with an error.

Test inputs:

Input #1: string s1 English alphabet (BWT index)

Input #2: string s2 English alphabet (BWT index)

Input #3: Opsin gene in human (3,302 bp) DNA alphabet

Input #4: Opsin gene in mouse (3,843 bp) DNA alphabet

Input #5: Human BRCA2 gene (11,382 bp) DNA alphabet (BWT index)

Input #6: Tomato's chloroplast genome (155,461 bp) DNA alphabet

Input #7: Yeast's Chromosome 12 (1,078,175 bp) DNA alphabet

Testing:

- Inputs are ordered by their size. For testing/demo purposes, try to find out how much and how well your implementation scales across these inputs (until memory is not sufficient). This means measuring the total runtime for tree construction alone (i.e., excluding the times to load the input or print output) for these input sizes. If you can also measure the peak memory usage of your code while its running (in Unix you can do this by running and monitoring the *top* command).
- Since inputs #5 and above are large strings, calling the DFS and post-order functions is NOT needed as the outputs will be very large. But make sure you test the DFS & post-order display parts for the smaller inputs.

SEE DETAILED NOTES ON HOW TO TEST YOUR CODE - I SUGGEST HERE 4 WAYS TO INCREMENTALLY TEST YOUR CODE.

Report:

In a separte Word or PDF document, report the following:

1. System configuration: CPU used, Clock rate, RAM, Cache size (if you know it).
2. Construction performance: For all the inputs successfully tested, report the *running times for ST construction* (do not include other times such as I/O read time or print times). Time can be reported in seconds or milliseconds or microseconds. If you want to report any other performance statistics you are welcome to do so in your report.
3. Justification: Do the performance observations made above, meet your expectations? Please explain.
4. Implementation constant: Report your implementation's space constant for your code - i.e., how many bytes does your code consume for *every input byte?* This can obviously be a roughly estimate (need not be exact) intended to give the end user an idea of your code's peak memory usage behavior.
5. BWT index: . For this component, please output the BWT index for the Opsin mouse, Tomato Chloropast genome and Yeast Chromosome 12 inputs, following the example shown here for the string banana$. Name the files: Tomato_BWT.txt and Yeast_BWT.txt and attach it with your submission. (For your reference I have generated the Human_BRCA2_BWT.txt here which you can compare and check your answer with.)
6. Exact matching repeat: For each of the input sequence, what is the *length* and the *<start> coordinates* of the longest exact matching repeat? The repeat occurrences may or may not be overlapping as long as they don't share the exact same start and end coordinates. For example, for

sequence "banana" it should be "ana" starting at positions 2 and 4. In your report, also comment on how you found this repeat - i.e., your algorithm to find this repeat.

-

CHECKLIST FOR SUBMISSION:

___ Cover sheet

___ Source code

___ A helpful readme file saying how to compile and run the code

___ Output file showing the statistics asked (don't display or dump the entire suffix tree please!) for all the test inputs (or at least up to the largest input you could run on your machine)

___ Report (Word or PDF)

___ All the above zipped into an archive folder that follows the naming convention mentioned above in the instructions.

-