Burrows-Wheeler Alignment — Paper Exercise

Start by reading the **Introduction** and **Methods** of Li and Durbin (2009) up to Section 2.3: *Suffix array interval and sequence alignment*. When you're done reading, work through the questions in this paper exercise. There are 5 in total, with an additional 3 questions for quick students. You do not have to hand this paper exercise in.

Question 1: Burrows-Wheeler Transform

Figure 2 from the paper shows how to construct the BWT string $B = lo\$oogg$ and suffix array $S = (6, 3, 0, 5, 2, 4, 1)$ out of the reference string $X = googol\$$.

Find the sorted rotations for $X = attac\$$, and write down the BWT string B and suffix array S. The unsorted rotations are given below for your convenience:

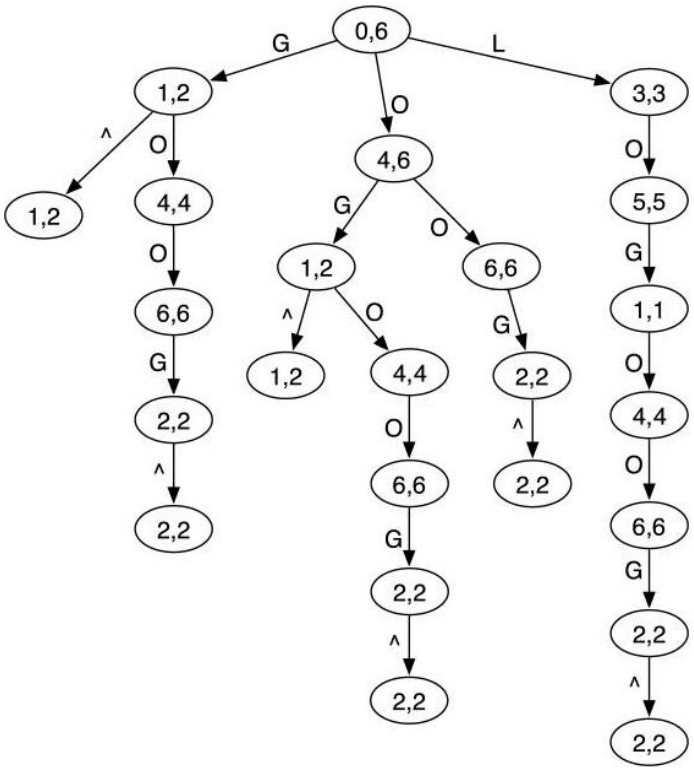
0	attac$
1	ttac$a
2	tac$at
3	ac$att
4	c$atta
5	$attac

Question 2: Prefix Tree

The figure to the right is a prefix tree, also known as a **trie**, for the string $X = googol\$$. Refer back to section 2.1 of the paper to learn how to interpret and use them. Draw the prefix tree for $X = attac\$$ below.

Remember every node represents a **string** (the concatenation of every edge's letter up to the root). The node's **suffix array (SA) interval** corresponds to the indices of your **sorted rotations** whose suffix **starts** with that node's string.

The tree's leaf nodes (terminal nodes, indicated by a carat: ^) each represent a **unique** prefix of X, which you can easily find back in the rotations!



Question 3: Exact Matching

Now that you have a trie, determining if a substring occurs within your original sequence is easy: *any* contained substring *must* be the prefix of a suffix (and the suffix of a prefix)! For example, the substring go occurs exactly twice in the string $googol$.

A. Which node from the trie on the previous page tells you this?

B. Which positions in the original string does the SA interval of that node point you towards?

C. What is the minimum number of edges you have to check to find a substring of *any* given length?

Question 4: Inexact Matching

Navigating the prefix tree with *bounded traversal* also allows you to find *inexact* matches of a substring. This is visualized by the dotted line in figure 1 of Li and Durbin (2009). The process is similar to exact matching, but you keep track of how many mistakes (mismatches) it takes to reach a given node, and stop evaluating a branch in the trie once the number of mistakes exceeds a given limit.

A. How many edges do you have to check to find all occurrences of lol in the $googol$ trie, when allowing up to 1 mismatch?

B. How many edges do you have to check to find all occurrences of tag in the $attac$ trie,
   a. when allowing 0 mismatches?
   b. at most 1 mismatch?
   c. 2 mismatches?
   d. 3?

Question 5: BWA Workflow

You now know the basics of how BWA works, and are almost ready to use it for some real read mapping. Your reference sequence is the entire human genome. The substrings you look for are reads. Sketch a workflow describing how you would use Burrows-Wheeler Alignment to map reads to a reference genome. It should contain **at least** the following elements (add more as necessary!):

- Read
- Reference genome
- Prefix trie
- Suffix array
- Suffix array interval
- Position(s) of read in reference

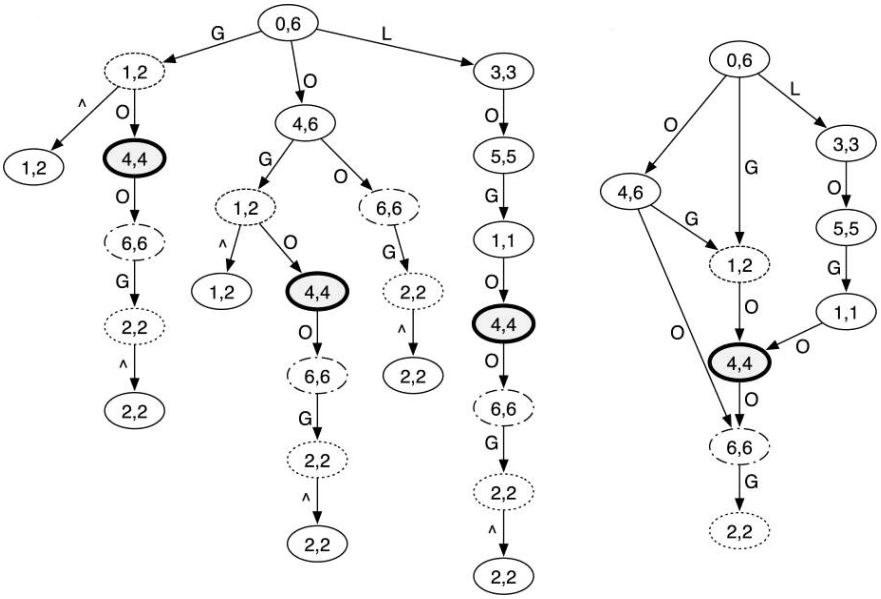
Hint: Consider enumerating the arrows in your workflow to indicate the order in which each element is used or traversed. You may also find it helpful to ***describe*** *the arrows, in terms of what step, method or algorithm they represent.*

Bonus Question 1: DAWG

A Directed Acyclic Word Graph, or DAWG, can be used to simplify (compress) tries, making them more memory-efficient. The figure below comes from Li and Durbin (2010), which shows how they merge multiple nodes with *identical SA intervals* into one, while retaining the incoming and outgoing edges. For the prefix trie of $X = googol\$$, this is the resulting DAWG:

Note 1) edges labeled with **^** are no longer present in the DAWG; and 2) nodes with the same SA interval have the same border in both the trie on the left, and the DAWG on the right.

Draw the DAWG for the prefix trie of $X = attac\$$ below.



Bonus Question 2: Algorithmic Complexity

You must by now have developed an intuitive understanding for how BWA accelerates string matching. This intuition can be formalized. Algorithmic complexity describes how "worst case" computation times of algorithms scale with respect to their input. For example, we've mentioned before that Dynamic Programming has a complexity of $O(m \cdot n)$.

A. Read back your answer to question 3C. Use "Big O" notation to express the complexity of exact mapping, with respect to (the length of) its input, i.e. the query substring.
Hint: Li and Durbin (2009) discuss it in section 2.4!

B. Describe the algorithmic complexities of the following steps:
    a. Calculating the Burrows-Wheeler Transform (BWT) of a string
    b. Making a prefix trie out of the sorted rotations
    c. **Inexact** mapping against a trie
    d. Converting a prefix trie into a DAWG
    e. Mapping against a DAWG

C. Does performing all of the above steps in order *really* speed up read mapping compared to how long it would take with Dynamic Programming?

Bonus Question 3: Inverse BWT

The Burrows-Wheeler Transform was originally published in 1994, as a linear-complexity compression method also known as block-sorting. The BWT string usually contains more repeated characters than the string it's constructed from, which is easier to compress. Importantly, the original string can be reconstructed from the transformed string. Look up how to do this online, and find out which string X was transformed into $B = eelnppi\$pa$.
Hint: You might want to write a little Python script to help out here!