1.

* Create a pointer to a child node, *firstNode*, for McCreights algorithm
* Build a suffix tree using McCreights algorithm
* Begin at the leaf, *firstNode*:
* Let edge label be α.
* While current node is not root:
  + Move to the parent node
    - Let the edge label of this parent node be beta
    - If beta + ‘$’ != alpha:
      * Return “not periodic”
* Print string[α]

McCreights algorithm builds the tree in O(n) time.

Traversal up the edge path of leaf node 1 is linear time.

Evaluation of internal nodes is linear time.

Total time complexity = O(n) + 1 + 1 = O(n)

2.

* Reverse the string
* Add $ to the end
* Use a modified version of McCreights algorithm to build a suffix tree as follows:
  + Add an integer vector to the node data structure that will be used to store the string depth of all leaf nodes below an internal node
  + While string is not empty:
  + Follow McCreights algorithm to build the tree in linear time and add the following steps:
  + As a new internal node is added:
    - Create a pointer to the current, new, internal node:
      * Visit all children down to the leaves, get the string-depth of the leaf and store it in the new internal node, in ascending order
    - If the step to create the new internal node also created a new leaf node that stores a ‘$’ character:
      * Test the stored string-depths at this internal node:
        + Let the first string depth value be n
        + If there is an internal node that has a string depth that is 2n:

Store the edge label of the new leaf node, minus the dollar sign, as beta

Reverse beta

Return beta

* McCreights algorithm builds the tree in O(n) time.
* Traversal down the path of internal node to add string-depths is O(n) time.
  + (For each new internal node that is added, visit each child once and read their internal node depth vector and store in new internal node string depth vector, for n children, O(n) time)
* Evaluation of internal node’s string-depths is linear time.
  + E.g. let node k’s depth vector be: node->depthList<2,4,6>
  + n = node->depthList[0] = 2
  + for (int i = 0; i < node->depthList.size(); i++)
    - if node->depthList[i] == 2n:
      * beta = node->depthList->edgeLabel
      * betaReturn = reverse (beta) - time = O(n)
      * return betaReturn
    - time = O(n)
* Total time complexity = O(n) + O(n) + O(n) = O(3n) = O(n)