2. *Preliminaries*

2.1. Properties of Suffix Trees. Let $S \in \Sigma^n$ be a string formed by n characters drawn from Σ. The suffix tree T_S of S is the compacted trie[*] of all the suffixes of $S\$$, where $\$ \notin \Sigma$. Throughout the paper, we will assume that suffix trees are represented as follows. Leaf l_i represents suffix $S[i, n]$, and array entry $l[i]$ points to l_i. Each internal node v has a length $L(v)$ that is the sum of the edge lengths on the path from the root to v. Then the string at v, denoted $\sigma(v)$, is $S[i, i + L(v) - 1]$ where l_i is any leaf below v. The children of node v are stored in a list sorted by the first character on the edge from v. Neither these first characters, nor any other part of the string represented by any edge is stored, since it can be retrieved in linear time. See Figure 1 for an example. In this representation, the size of the suffix tree is $O(n)$.

[*] A compacted trie differs from a trie in that maximal branch-free paths are replaced by edges labeled by the appropriate substring.

```
/*
  label      | edge label start address
  label_len  | size of edge label
  target_node | target of directed edge
*/
struct Edge {
  int* label;
  int label_len;
  Node* target_node;
};
```

```
/*
  lv         | Depth of Node in Tree
  edge_table | array used for O(1) insert
  children   | vector of child edges
  suffix_index | leaf identifier, 0 for internal
  _color     | internal use during contract
  _next_edge | internal use during traversal
*/
struct Node {
  int lv;
  Edge** edge_table = nullptr;
  vector <Edge*> children;
  int suffix_index;
  int _color;
  int _next_edge;
};
```

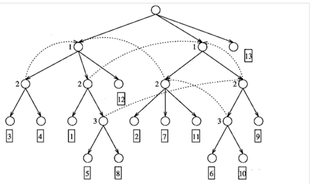


Fig. 1. Suffix tree of string S = 121112212221¥. Numbers in boxes represent which suffix is denoted by that leaf. Numbers next to internal nodes represent the string length $L(\cdot)$ of that node. Dotted lines are suffix links (see Lemma 2.1).

The following well-known lemma gives suffix trees a nice structure.

Lemma 2.1. [Weiner 1973]. *Let $\alpha \in \Sigma$ and $\alpha \in \Sigma^*$. If there is a node v in T_S such that $\sigma(v) = \alpha\alpha$, then there is a node w in T_S such that $\sigma(w) = \alpha$.*

Given this lemma we can define, for every node v in T_S, the *suffix link* $sl(v) = w$, where v and w are defined as in Lemma 2.1. Notice that $sl(\cdot)$ links form a tree rooted at the root of T_S. The depth of any node v in this $sl(\cdot)$ tree is then just $L(v)$.

Given a string $\alpha \in \Sigma^m$, let $|\alpha|$ denote the string length m. We use $\mathtt{lcp}(\alpha, \beta)$ to indicate the *length of the longest common prefix* of any two strings α and β, and $\mathtt{lca}(v, w)$ to denote the *least common ancestor* of any two nodes v and w in a tree. The property of suffix trees most often exploited algorithmically is the following relationship between $\mathtt{lcp}$ in S and $\mathtt{lca}$ in T_S.

for all nodes v, w in suffix tree T_S of S

$$\forall v, w \in T_S, \quad \mathtt{lcp}(\sigma(v), \sigma(w)) = |\sigma(\mathtt{lca}(v, w))|. \tag{1}$$

length of longest common prefix of (string@v, string@w) = length of the string@(least common ancestor of node v, w)

In all computational models, we will use the fact that it is easy to find the least common ancestors of two nodes [Harel and Tarjan 1984; Bender and Farach-Colton 2000; Chiang et al. 1995].

2.2. Suffix Arrays and Euler Tours. Let $\Delta = \{S_i \mid S_i \in \Sigma^n\}$ be a set of strings of arbitrary lengths and assume, for the sake of exposition, that no string in Δ is a prefix of another. We denote by T the sorted compacted trie of the strings in Δ, and adopt the same notation introduced for suffix trees in Section 2.1 to refer to T's structure. In particular, leaf l_j will now represent the string S_j, and array entry $l[i]$ will point to l_i. The size of T is $O(|\Delta|)$ in addition to the overall length of the strings in Δ. Since T is sorted, the children of any node v are stored in a list sorted by the first character on the edges from v. Observe that the in order traversal of the leaves of T gives the lexicographically ordered sequence of the strings in Δ. We denote by $s_1, \ldots, s_{|\Delta|}$ the permutation of $1, \ldots, |\Delta|$ such that S_{s_i} is lexicographically smaller than S_{s_j} if and only if $s_i < s_j$.

The suffix array of T consists of two arrays, the *sort array* A_T and the *longest common prefix array* $\mathtt{LCP}_T$. A_T is the lexicographically ordered sequence of the strings in Δ, that is $A_T[i] = s_j$. The array $\mathtt{LCP}_T$ stores the lengths of the common prefix of adjacent strings in A_T, that is $\mathtt{LCP}_T[i] = \mathtt{lcp}(S_{j_i}, S_{j_{i-1}})$.

Suppose we are given the sorted compacted trie T and we wish to compute A_T and $\mathtt{LCP}_T$. A_T is simply the inorder listing of the leaves of T, whereas $\mathtt{LCP}_T$ can be obtained by means of Equality (1). Hence, both of them can be computed simultaneously during an inorder traversal of T. Conversely, given A_T and $\mathtt{LCP}_T$ it is a straightforward exercise to reconstruct the trie T in linear time on a RAM [Farach and Muthukrishnan 1996]. We give other model-specific complexities below.

Given a rooted tree $T' = (V, E)$, the *Euler Tour* $\mathtt{ET}(T')$ of T' is a sequence of nodes. $\mathtt{ET}(T')$ has length $2|E| + 1$ and is obtained by performing a depth-first search (DFS) on T' and outputting each node every time it is visited. In the RAM model, $\mathtt{ET}(T')$ is computed via an explicit DFS of the tree T', whereas in the DAM model $\mathtt{ET}(T')$ is efficiently computed by simulating known PRAM-algorithms [Chiang et al. 1995].

```
/*
  root       | Root of compacted trie
  input_string| raw input string
  rank_string | post-sort string
  LCP_ext    | final LCP array
  A_ext      | final Sort array
*/
struct S {
  Node* root;
  vector<int> input_string;
  vector<int> rank_string;
  vector<int> LCP_ext;
  vector<int> A_ext;
};
```

3. *RAM Algorithm*

We start by developing a few tools that will simplify our presentation. Recall that $\mathtt{lca}_T(u, v)$ denotes the least common ancestor of a pair of nodes u, v in T. We will drop the T-subscript whenever there is no ambiguity.

Theorem 3.1 [Harel and Tarjan 1984; Bender and Farach-Colton 2000]: *A tree T with m nodes can be preprocessed in $O(m)$ time so that, for any pair of its nodes u, v, $\mathtt{lca}(u, v)$ can be computed in constant time.*

Expectation is to replace LCA to RMQ using a Sparsetable to answer LCP=LCAqueries.

Given a sorted compacted trie T built on a set Δ of k strings, we can compute the arrays A_T and $\mathtt{LCP}_T$ in $O(k)$ time via a traversal of T (see Section 2.2). Conversely, given A_T and $\mathtt{LCP}_T$ we can reconstruct the sorted compacted trie T in linear time by exploiting Theorem 3.1 and Eq. (1). Notice that both transformations are independent on the length of the strings stored in the two data structures. Therefore, we can state:

Theorem 3.2. *Given a sorted compacted trie T built on k strings, A_T and $\mathtt{LCP}_T$ can be computed in $O(k)$ time. The inverse transformation also takes $O(k)$ time.*

We have now all the ingredients to detail the implementation of the three steps of our divide-and-conquer approach for the case of the RAM model.

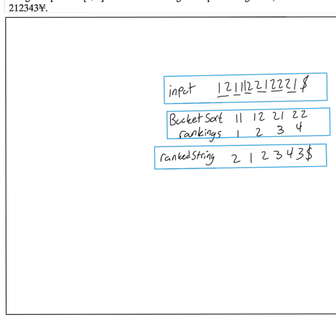
3.1. Building the Odd Tree. We show how to compute T_o, the compacted trie of all suffixes of S beginning in odd positions. Note that we can extend S with any character so that it has length a power of two. At the end of the computation, we simply prune the tree in linear time to remove the extra suffixes. The padding simplifies some boundary cases below, by making sure that the length of the string is always divisible by two. The padding at most doubles the length of S and so does not affect the time complexity. While the string can be padded with any character, padding with ¥'s makes it easier to strip away unwanted suffixes at the end of the computation.

I'm unsure about padding bits for now, we merely add 1 or 2 as necessary for the example case. To be revisited in future update

1

Step 1. Map pairs of characters into single characters as follows. For $i = 1$ to $n/2$, form pairs $(S[2i - 1], S[2i])$. Lexicographically sort them by radix sort in linear time, remove duplicates, and compute the new string S' such that $S'[i]$ is the rank of $(S[2i - 1], S[2i])$ in the sorted list. S' is a string of length $n/2$ over the integer alphabet $[n/2]$. For our running example in Figure 1, we have that S' = 212343¥.

Radix sort is scheduled for optimized version. For ease of implementation, std::mAP was used to perform a variation of bucket sort.



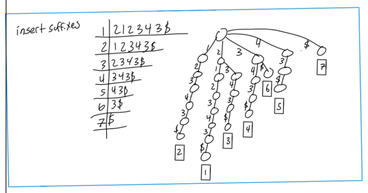

```
transformString(s.input_string, s.rank_string);
```

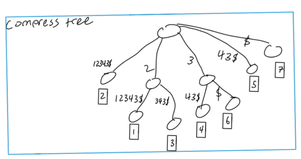
```
/*
  transform input string t_str to a ranked string
  bucket sort using nested maps, keys of length 2
  iterate over map, assign monotonic increasing int
  eliminates duplicates: asserts counter < input len
  copy into return vector rv
*/
void transformString(vector<int> &t_str, vector<int> &rv) {
  map<int, map<int, int>> bucket_map;
  vector<int*> rank_vec;
  int input_str_len = t_str.size();

  for (int i = 0; i < input_str_len/2; i++)
    rank_vec.push_back(&(bucket_map[t_str[2 * i]][t_str[2 * i + 1]]));

  int counter = 1;
  for (auto a : bucket_map) {
    for (auto b : a.second) {
      bucket_map[a.first][b.first] = counter;
      counter++;
    }
  }

  for (auto a : rank_vec)
    rv.push_back(*a);
}
```

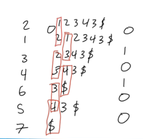
2

Step 2. Recursively compute $T_{S'}$, the suffix tree of S', and derive $A_{T_{S'}}$ and $\mathtt{LCP}_{T_{S'}}$. See Figure 2 for $T_{S'}$ for our example string. $A_{T_{S'}}$ = [2, 1, 3, 4, 6, 5, 7] and $\mathtt{LCP}_{T_{S'}}$ [0, 1, 0, 1, 0, 0].

iterative implementation is more observable than recursion. Personal preference of auxiliary stack & queue







Longest common prefix Between adjacent strings [0 1 0 1 0 0]

serialize sorted tree [2 1 3 4 6 5 7]

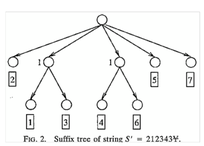


Fig. 2. Suffix tree of string S' = 212343¥.

```
for (int i = 0; i < s.rank_string.size(); i++)
  insertSuffix(root, s.rank_string.data() + i, i + 1);

collectNodes(root);
contractTrie(root);

vector<int> c_LCP;
vector<int> c_A;
buildArrays(root, &c_A, &c_LCP);

deleteCollectedTrie(root);
```

```
/*
  inserts a substring, id = suffix_index
*/
void insertSuffix(Node* root, int* start, int suffix_index) {
  Node* curr_node = root;
  int* curr_char = start;
  int lv = 1;

  while (*curr_char != TERM) {
    // current node must be leaf node, therefore must have edge_table
    if (!curr_node->edge_table[*curr_char])
      curr_node->edge_table[*curr_char] = createTrieEdge(createTrieNode(lv),
                                                         curr_char,
                                                         1);
    curr_node = curr_node->edge_table[*curr_char]->target_node;
    curr_char++;
    lv++;
  }
  // suffixes are guaranteed to be different lengths, therefore no collisions possible
  curr_node->edge_table[ALPHABET_SIZE - 1] = createTrieEdge(createTrieNode(lv, true, suffix_index),
                                                            curr_char,
                                                            1);
}
```

```
/*
  consolidates the edges of each node in BFS manner
  moves Edges from Node.edge_table to Node.children
  deletes edge_table
*/
void collectNodes(Node* root) {
  queue<Node*> node_q;
  node_q.push(root);
  Node* curr = nullptr;

  while (!node_q.empty()) {
    curr = node_q.front();
    node_q.pop();

    if (curr->edge_table == nullptr)
      continue;

    for (int i = 0; i < ALPHABET_SIZE; i++) {
      if (curr->edge_table[i] != nullptr) {
        curr->children.push_back(curr->edge_table[i]);
        node_q.push(curr->edge_table[i]->target_node);
      }
    }

    delete[] curr->edge_table;
  }
}
```

```
/*
  replace simple path p = (u,...,v) with single edge(u,v)
  maximal branch-free paths are replaced by edges labeled
    by the appropriate substring
*/
void contractTrie(Node* root) {
  stack<Edge*> edge_s;
  Edge* base = createTrieEdge(root);
  edge_s.push(base);
  Node* curr = nullptr;

  while (!edge_s.empty()) {
    curr = edge_s.top()->target_node;

    if (curr->children.size() == 1) {
      edge_s.top()->label_len = edge_s.top()->label_len + curr->children[0]->label_len;
      edge_s.top()->target_node = curr->children[0]->target_node;
      delete curr->children[0];
      delete curr;

    } else if (curr->_next_edge < curr->children.size()) {
      edge_s.push(curr->children[curr->_next_edge]);
      curr->_next_edge++;

    } else {
      curr->_next_edge = 0;
      edge_s.pop();
    }
  }
  delete base;
}
```

```
/*
  Traverses the Trie in DFS manner
  builds sort_array containing suffix positions in lexicographic order
  builds LCP array, least common ancestor of adjacent strings in sort_array

  Derivative of the visit leaf depth first traversal
*/
void buildArrays(Node* root, vector<int> *sort_array, vector<int> *LCP_array){
  int last_peak = -1;
  bool peak_set = true;
  stack<Node*> node_s;
  node_s.push(root);
  Node* curr = nullptr;

  while (!node_s.empty()) {
    curr = node_s.top();
    if (curr->_next_edge < curr->children.size()) {
      if (!peak_set) {
        //save highest point
        last_peak = curr->lv;
        peak_set = true;
      }
      node_s.push(curr->children[curr->_next_edge]->target_node);
      curr->_next_edge++;
      continue;

    } else if (curr->suffix_index != 0){
      //making sure LCP.size() == sort_array.size() - 1
      if (last_peak != -1)
        LCP_array->push_back(last_peak);
      sort_array->push_back(curr->suffix_index);
      peak_set = false;
    } else {
      curr->_next_edge = 0;
    }
    node_s.pop();
  }
}
```

3

Step 3. Notice that any odd suffix $S[2i - 1] \cdots S[n]$¥ is equivalent to the suffix $S'[i] \cdots S'[n/2]$¥. Thus, by the lexicographic ordering of the characters of S', we have that $A_{T_o}[i] = 2 \times A_{T_{S'}}[i] - 1$, and so this computation takes linear time. For our example, A_{T_o} = [3, 1, 5, 7, 11, 9, 13].

Expand sorted Suffix array

$A_{T_o}[i] = 2 * A_{T_{S'}}[i] - 1$

$A_{T_{S'}} = [2\ 1\ 3\ 4\ 6\ 5\ 7]$

$A_{T_o} = [3\ 1\ 5\ 7\ 11\ 9\ 13]$

```
/*
  expand lexicographically ordered suffix array
*/
void expand_sorted_suffix_array(vector<int>* A_ext, vector<int>* A) {
  for (int i = 0; i < A->size(); i++) {
    A_ext->push_back((2 * (*A)[i]) - 1);
  }
}
```

4

Step 4. *Since each character in S'* represents two characters in S, in order to compute $\mathtt{LCP}_{T_o}$ we need only to check whether the $\mathtt{lcp}$ between two adjacent suffixes in A_{T_o} must be extended by one unit. Thus,

$$\mathtt{LCP}_{T_o}[i] = 2 * \mathtt{LCP}_{T_{S'}}[i] + \begin{cases} 1 & \text{if } S[A_{T_o}[i] + 2 * \mathtt{LCP}_{T_{S'}}[i]] = S[A_{T_o}[i+1] + 2 * \mathtt{LCP}_{T_{S'}}[i]]; \\ 0 & \text{otherwise.} \end{cases}$$

This computation takes linear time. For our example, $\mathtt{LCP}_{T_o}[i]$ = [1, 2, 0, 2, 1, 0].

$LCP_{T_o} = 2 * LCP_{T_o}[i] + \begin{cases} 1, S[A_{T_o}[i] + 2 * LCP_{T_{S'}}[i]] = S[A_{T_o}[i+1] + 2 * LCP_{T_{S'}}[i]] \\ 0, \text{otherwise} \end{cases}$

$LCP_{T_o} = [2\ 0\ 1\ 0\ 1\ 0\ 0]$

$S = 121112212221\$$

$A_{T_o} = [3\ 1\ 5\ 7\ 11\ 9\ 13]$

$0 \Rightarrow 1 = 2 * (0) + \begin{cases} 1, S[3 + 2 * (2)] = S[1 + 2 * (0)] = S[5] > S[3] \Rightarrow 1 = 1 \\ 0 \end{cases}$

$1 \Rightarrow 2 = 2 * (1) + \begin{cases} 1, S[1 + 2 * (0)] = S[5] \neq S[3] \Rightarrow 1 \neq 2 \\ 0 \end{cases}$

$2 \Rightarrow 0 = 2 * (2) + \begin{cases} 1 & S[5 + 2 * (0)] = S[7 + 2 * (0)] \\ 0 & S[5] = S[7] \Rightarrow 1 \neq 2 \end{cases}$

$3 \Rightarrow 2 = 2 * (1) + \begin{cases} 1 & S[7 + 2 * (1)] = S[11 + 2 * (1)] \\ 0 & S[9] = S[13] \Rightarrow 2 \neq \$ \end{cases}$

$4 \Rightarrow 1 = 2 * (0) + \begin{cases} 1 & S[11 + 2 * (0)] = S[9 + 2 * (0)] \\ 0 & S[11] = S[9] \Rightarrow 2 = 2 \end{cases}$

$5 \Rightarrow 0 = 2 * (0) + \begin{cases} 1 & S[9 + 2 * (0)] = S[13 + 2 * (0)] \\ 0 & S[9] = S[13] \Rightarrow 2 \neq \$ \end{cases}$

Expand and correct Longest Common Prefix array

$LCP_{T_o} = [2\ 0\ 1\ 0\ 1\ 0\ 0]$

$LCP_{T_o} = [1\ 2\ 0\ 2\ 1\ 0]$

```
/*
  expand and correct longest common prefix array
*/
void expand_LCP_array(vector<int>* LCP_ext, vector<int>* LCP, vector<int> *A_ext, int* S) {
  int val = 0;
  for (int i = 0; i < LCP->size(); i++) {
    val = 2 * (*LCP)[i] + (S[((*A_ext)[i] + 2 * (*LCP)[i]) - 1] == S[((*A_ext)[i + 1] + 2 * (*LCP)[i]) - 1] ? 1 : 0);
    LCP_ext->push_back(val);
  }
}
```

5

Step 5. Construct T_o from A_{T_o} and $\mathtt{LCP}_{T_o}$ in linear time (Theorem 3.2). Figure 3 gives the odd tree for our example string.

The equivalence between LCA and LCP was used to properly place and correctly label nodes on an Euler tour-like traversal

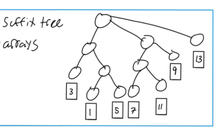
$LCP_{T_o} = [2\ 0\ 1\ 0\ 1\ 0\ 0]$

$A_{T_o} = [3\ 1\ 5\ 7\ 11\ 9\ 13]$

$S = 121112212221\$$

LCP[i]	1	2	0	2	1	0	0	ε
$A_{T_o}[3], A_S[i+1]$	3,1	1,5	5,7	7,11	11,9	9,13	13,$	

Construct full odd suffix tree from expanded arrays



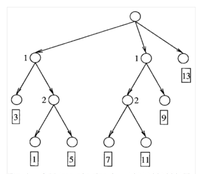


Fig. 3. Odd tree of string S = 121112212221¥.

```
/*
  construct NEW full odd trie using LCP and sorted array
*/
void constructFullOddTrie(Node* root, vector<int>& LCP, vector<int>& A, int* S) {
  Node* curr_node = nullptr;
  int curr_depth = 0;
  int LHS = 0;
  int END = A[A.size() - 1];

  stack<Node*> node_stack;
  node_stack.push(root);
  curr_node = node_stack.top();

  while (LHS < LCP.size()) {
    while (curr_depth < LCP[LHS]) {
      // if curr_depth < trie_depth, avoid edge_table leaks
      Edge* e = createTrieEdge(createTrieNode(curr_depth + 1, true),
                               S + (A[LHS] - 1 + curr_depth),
                               1);
      curr_node->children.push_back(e);
      node_stack.push(e->target_node);
      curr_node = node_stack.top();
      curr_depth++;
    }
    if (curr_depth == LCP[LHS]) {
      Edge* e = createTrieEdge(createTrieNode(curr_depth + 1, true, A[LHS]),
                               S + (A[LHS] - 1 + curr_depth),
                               END - (A[LHS] - 1 + curr_depth));
      curr_node->children.push_back(e);
      LHS++;
    }
    if (LHS == LCP.size()) {
      break;
    }
    if (curr_depth > LCP[LHS]) {
      Edge* e = createTrieEdge(createTrieNode(curr_depth + 1, true, A[LHS]),
                               S + (A[LHS] - 1 + curr_depth),
                               END - (A[LHS] - 1 + curr_depth));
      curr_node->children.push_back(e);
    }
    while (curr_depth > LCP[LHS]) {
      node_stack.pop();
      curr_node = node_stack.top();

      curr_depth--;
      if (curr_depth == LCP[LHS]) {
        LHS++;
        break;
      }
    }
  }
  Edge* e = createTrieEdge(createTrieNode(curr_depth + 1, true, A[LHS]),
                           S + (A[LHS] - 1 + curr_depth),
                           END - (A[LHS] - 1 + curr_depth));
  curr_node->children.push_back(e);
}
```