1   Overview

In this lecture, we consider the string matching problem - finding some or all places in a text where the query string occurs as a substring. From the perspective of a one-shot approach, we can solve string matching in $O(|T|)$ time, where $|T|$ is the size of our text. This purely algorithmic approach has been studied extensively in the papers by Knuth-Morris-Pratt [6], Boyer-Moore [1], and Rabin-Karp [4].

However, we entertain the possibility that multiple queries will be made to the same text. This motivates the development of data structures that preprocess the text to allow for more efficient queries. We will show how to construct, use, and analyze these string data structures.

2   Predecessor Problem

Before we get to string matching we will solve an easier problem which is necessary to solve string matching. This is the predecessor problem among strings. Assume we have k strings, $T_1, T_2, ..., T_k$ and the query is: given some pattern P, we would like to know where P fits among the k strings in lexicographical order. More specifically we would like to know the predecessor of P among the k strings, according to the lexicographical order of the strings.

Using our already developed data structures for this problem is not efficient as strings can be quite long and comparing strings is slow. So we need to use some other data structure that takes into account this fact.

2.1   Tries and Compressed Tries

To solve the predecessor problem we will use a structure called a **trie**. A **trie** is a rooted tree where each child branch is labeled with letters in the alphabet Σ. We will consider any node v to store a string which represents the concatenation of all brach labels on the path from the root r to the v. More specifically given a path of increasing depth $p = r, v_1, v_2, ..., v$ from the root r to a node v, the string stored at node v_i is the concatenation of the string stored in v_{i-1} with the letter stored on the branch $v_{i-1}v_i$. We will denote the strings stored in the leaves of the trie as words, and the strings stored in all other nodes as prefixes. The root r represents the empty string.

We order the child branches of every node alphabetically from left to right. Note that the fan-out of any node is at most $|\Sigma|$. Also an inorder traversal of the trie outputs the stored strings in sorted

order.

It is common practice to terminate strings with a special character $\$ \notin \Sigma$, so that we can distinguish a prefix from a word. The example trie in Figure 1 stores the four words ana\$, ann\$, anna\$, and anne\$.

If we assign only one letter per edge, we are not taking full advantage of the trie's tree structure. It is more useful to consider **compact** or **compressed** tries, tries where we remove the one letter per edge constraint, and contract non-branching paths by concatenating the letters on these paths. In this way, every node branches out, and every node traversed represents a choice between two different words. The compressed trie that corresponds to our example trie is also shown in Figure 1.
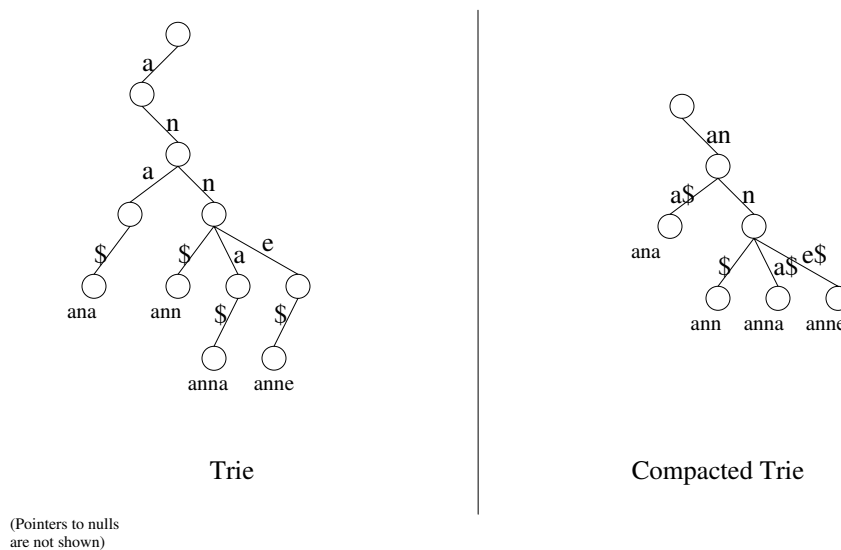

Trie   Compacted Trie

(Pointers to nulls
are not shown)

Figure 1: Trie and Conmpacted Trie Examples

2.2   Solving the Predecessor Problem

We will store our k strings $T_1, T_2, ..., T_k$ in a trie data structure as presented above. Given a query string P, in order to find it's predecessor we walk down the trie from the root, following the child branches. At some point we either fall off the tree or reach a leaf. If we reach a leaf then we are done, because the pattern P matches one of our words exactly. If we fall of the three then the predecessor is the maximum string stored in the tree imediately to the left of where we fell off. For example if we search for the word anb in the tree if Figure 1, we fall of the tree after we traverse branches a and n downward. Then b is between branches a and n of the node where we fell of, and the predecessor is the maximum string stored in the subtree following branch a.

For every node v of the trie we will store the maximum string in the subtree defined by v. Then, to allow fast predecessor queries we need a way to store every node of the trie, that allows both fast traversals of edges (going down the tree) and fast predecessor queries within the node.

2.3 Trie Node Representation

Depending on the way in which we choose to store a node in the trie, we will get various query time and space bounds.

2.3.1 Array

We can store every node of the trie as an array of size $|\Sigma|$, where each cell in the array represents one branch. In this case following a down branch takes $O(1)$ time. To find the predecessor within an array, we simply precompute all possible $|\Sigma|$ queries and store them. Thus, predecessor queries within a node also take $O(1)$ time. Unfortunately, the space used for every node is $O(|\Sigma|)$, so the lexicographical total space used is $O(|T||\Sigma|)$, where $|T|$ is the number of nodes in the trie $|T_1| + |T_2| + ... + |T_k|$. The total query time is $O(|P|)$.

2.3.2 Binary Search Tree

We can store a trie node as a binary search tree. More specifically, we store the branches of a node in a binary search tree. This will reduce the total space of the trie to $O(|T|)$, but will increase the query time to $O(|P| \log \Sigma)$.

2.3.3 Hash Table

Using a hash table for each node gives us an $O(|T|)$ space trie with $O(P)$ query time. Unfortunately with hashing we can only do exact searches, as hashes don't support predecessor.

2.3.4 Van Emde Boas

To support predecessor queries we can use a Van Emde Boas data structure to store each node of the trie. With this total space used to store the trie is $O(|T|)$ and query time is $O(P \log \log \Sigma)$.

2.3.5 Weight Balanced BST

We will store the children of the node v in a weight balanced binary search tree, where the weight of each child x is equal to the total number of descendand leaves in subtree x.

To construct such a tree we split the children of node v in two, such that the number of descendant leaves of the children on the left is as close as possible to the number of descendant leaves of the
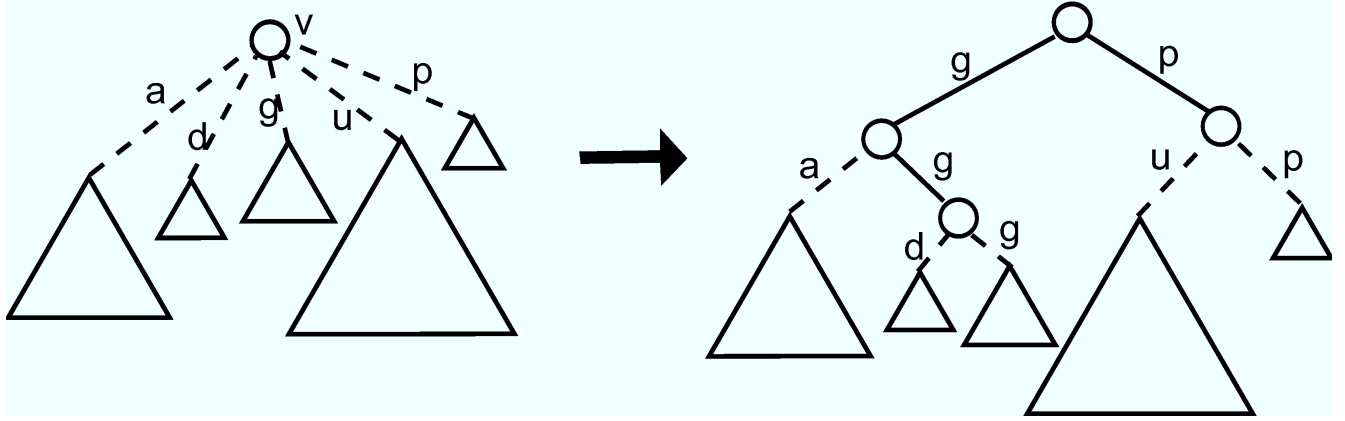
Figure 1: On the left we have node v with 5 children. Each triangle represents a subtree, and the size of the triangle represents the weight of the subtree (the number of descendand leaves of the subtree). On the right we can see the weight balanced BST. The solid lines are the edges of the weight balanced BST and the dashed lines represent the initial edges to the children of v.

children to the right. We then recurse on the children on the left and on the children on the right. To support predecessor searches in this new tree, we can label each edge e of the constructed tree with the maximum edge label corresponding to some child x of v, in the subtree determined by following e. To better understand this construction look at Figure 2.

Lemma 1. *In a trie where we store each node as a weight balanced BST as above, a search takes $O(P + \log k)$ time.*

Proof. We will prove this by showing that at every edge followed in the tree, we either advance one letter in P or we decrease the number of candidate T_i's by 1/3. Assume that in our traversal we are at node y in the weight balanced BST representing node x in the trie. Consider all trie children $c_1, c_2, ..., c_m$ of node x present in y's subtree. Let D be the total number of descendant leaves of $c_1, c_2, ..., c_m$. If the number of descendant leaves of any child c_i is at most $D/3$ then following the next edge from y reduces the number of candidate leaves decreases by at least 1/3. Otherwise, there is some child c_i that has at least $D/3$ descendants. Then from the construction of the weight balanced BST, c_i has to be at most a grandchild of y. Then, by following the next two edges, we either move to node c_i in the trie, in which case we advance one letter in P, or we don't move do node c_i in which case we reduce the number of potential leaves by 1/3. $\square$

Note that k is the number of strings stored in the trie. Even though our data structure achieves $O(P + \log k)$ query time, we would however like to achieve $O(P + \log \Sigma)$ time, to make query time independent of the number of strings in the trie. We can do this by using indirection and leaf trimming, similar to what we used to solve the level ancestor problem in constant time and linear space.

We will trim the tree by cutting edges below all maximally deep nodes that have at least $|\Sigma|$ descendant leaves. We will thus get a top tree, which has at most $|T|/|\Sigma|$ leaves and thus at most $|T|/|\Sigma|$ branching nodes. On this tree we can store trie nodes using arrays, to get linear query and $O(T)$ space. Note that we can only store arrays for the branching nodes and leaves, as there may

be more nodes than $|T|/|\Sigma|$. But for the non-branching nodes we can simply store a pointer to the next node, or use the compressed trie from the start instead.

The bottom trees have less than $|\Sigma|$ descendant leaves, which means we can store trie nodes using weight balanced BSTs, which use linear space, but now have $O(P + \log \Sigma)$ query time since there are less than Σ leaves.

In total we get a data structure with $O(T)$ space and $O(P + \log \Sigma)$ query time. Another data structure that obtains these bounds is the suffix trays [9].

3   Suffix Trees

A **suffix tree** is a compressed trie built on all $|T|$ suffixes of T, with $ appended. For example, if our text is the string $T = banana\$$, then our suffix tree will be built on $\{banana\$, anana\$, nana\$, ana\$, na\$, a\$\}$. The suffix starting at the ith index is denoted $T[i :]$. For a non-leaf node of the suffix tree, define the letter depth of the node as the length of the prefix stored in the node.

Storing strings on the edges and in the nodes is potentially very costly in terms of space. For example, if all of the characters in our text are different, storage space is quadratic in the size of the text. To decrease storage space of the suffix tree to $O(|T|)$, we can replace the strings on each edge by the indices of its first and last character, and omit the strings stored in each node. We lose no information, as we are just removing some redundancies.

3.1   Applications

Suffix trees are versatile data structures that have myriad applications to string matching and related problems:

3.1.1   String Matching

To solve the string matching problem, note that a substring of T is simply a prefix of a suffix of T. Therefore, to find a pattern P, we walk down the tree, following the edge that corresponds to the next set of characters in P. Eventually, if the pattern matches, we will reach a node v that stores P. Finally, report all the leaves beneath v, as each leaf represents a different occurrence. There is a unique way to walk down the tree, since every edge in the fan out of some node must have a distinct first letter. The runtime of a search, however, depends on how the nodes are stored. If the nodes are stored as a hash table, this method achieves $O(P)$ time. Using trays, we can achieve $O(P + \lg \Sigma)$, and using a hash table and a van Emde Boas structure, we can achieve $O(P + \lg \lg \Sigma)$ time. These last two results are useful because they preserve the sorted order of the nodes, where a hash table does not preserve sorting.

3.1.2   First k Occurrences

Instead of reporting all matching subsequences of T, we can report the first k occurrences of P in only $O(k)$ additional time. To do this, add a pointer from each node to its leftmost descendant

leaf, and then connect all the leaves via a linked list. Then, to determine the first k occurrences, perform a search as above, and then simply follow pointers to find the firsk k leaves that match that pattern.

3.1.3 Counting Occurrences

In this variant of the string matching problem, we must find the number of times the pattern P appears in the text. However, we can simply store in every node the size of the subtree at that node.

3.1.4 Longest Repeating Substring

To find the longest repeated substring, we look for the branching node with maximum letter depth. We can do this in $O(T)$ time. Suppose instead we are given two indices i and j, and we want to know the longest common prefix of $T[i:]$ and $T[j:]$. This is simply an LCA query, which from L15 can be accomplished in $O(1)$ time.

3.1.5 All occurences of $T[i:j]$

When the pattern P is a known substring of the text T, we can interpret the problem as a LA query: We'd like to find the $(j-i)$th ancestor of the leaf for $T[i:]$. We must be careful because the edges do not have unit length, so we must interpret the edges as weighted, by the number of characters that are compressed onto that edge.

We must make a few modifications to the LA data structure from L15 to account for the weighted edges. We store the nodes in the long path/ladder DS of L15 in a van Emde Boas predecessor DS, which uses $O(\lg \lg T)$ space. Additionally, we can't afford the lookup tables at the bottom of the DS from L15. Instead, we answer queries on the bottom trees in the indirection by using a ladder decomposition. This requires $O(\lg \lg n)$ ladders, to reach height $O(\lg n)$. Then, we only need to run a predecessor query on the last ladder. This gives us $O(\lg \lg T)$ query and $O(T)$ space. This result is dues to Abbott, Baran, Demaine and others in Spring 2005's 6.897 course.

3.1.6 Multiple Documents

When there are multiple texts in question, we can concatenate them with $\$_1, \$_2, ..., \$_n$. Whenever we see a $\$_i$, we trim below it to make it a leaf and save space. Then to find the longest common substring, we look for the node with the maximum letter depth with greater than one distinct $\$$ below. To count the number of documents containing a pattern P, simply store at each node the number of distinct $\$$ signs as a descendant of that node.

3.1.7 Document Retrieval

We can find d distinct documents T_i containing a pattern P in only an additional $O(d)$ time, for a total of $O(P + d)$. This is ideal, because if P occurs an astronomical number of times in one

document, and only once or twice in a second document, a more naive algorithm could spend time finding every occurrence of P when the result will only be 2 documents. This DS, due to Muthukrishnan [8], will avoid that problem.

To do this, augment the data structure by having each $\$_i$ store the leaf number of the previous $\$_i$. Then, suppose we have searched for the pattern P and come to the node v. Suppose the leaf descendants of v are numbered l through n. So, we want the first occurrence of $\$_i$ in the range $[l, n]$, for each i. With the augmentation, this is equivalent to looking for the $\$_i$ whose stored value is $< l$, because this indicates that the previous $\$_i$ is outside of the interval $[l, n]$.

We can solve this problem with an RMQ query from L15. Find the minimum in $O(1)$ time. Suppose the minimum is found at position m. Then, if the stored value of that leaf is $< l$, that is an answer, so output that the pattern occurs in document i. Then, recurse on the remaining intervals, $[l, m-1]$ and $[m + 1, n]$. Thus, each additional output we want can be found in $O(1)$ time, and we can stop any time we want.

4   Suffix Arrays

Suffix trees are powerful data structures with applications in fields such as computational biology, data compression, and text editing. However, a **suffix array**, which contains most of the information in a suffix tree, is a simpler and more compact data structure for many applications. The only drawback of a suffix array is that is that it is less intuitive and less natural as a representation. In this section, we define suffix arrays, show that suffix arrays are in some sense equivalent to suffix trees, and provide a fast algorithm for building suffix arrays.

4.1   Example

Let us store the suffixes of a text T in lexicographical order in an intermediate array. Then the suffix array is the array that stores the index corresponding to each suffix. For example, if our text is $banana\$$, the intermediate array is $[\$, a\$, ana\$, anana\$, banana\$, na\$, nana\$]$ and the suffix array is $[6, 5, 3, 1, 0, 4, 2]$, as in Figure 2. Since suffixes are ordered lexicographically, we can use binary search to search for a pattern P in $O(|P| \log |T|)$ time.

We can compute the length of the longest common prefix between neighboring entries of the intermediate array. If we store these lengths in an array, we get what is called the **LCP array**. This is illustrated in Figure 3. These LCP arrays can be constructed in $O(|T|)$ time, a result due to Kasai et al [5]. In the example above, the LCP array constructed from the intermediate array is $[0, 1, 3, 0, 0, 2]$. Using LCP arrays, we can improve pattern searching in suffix arrays to $O(|P| + \log |T|)$.
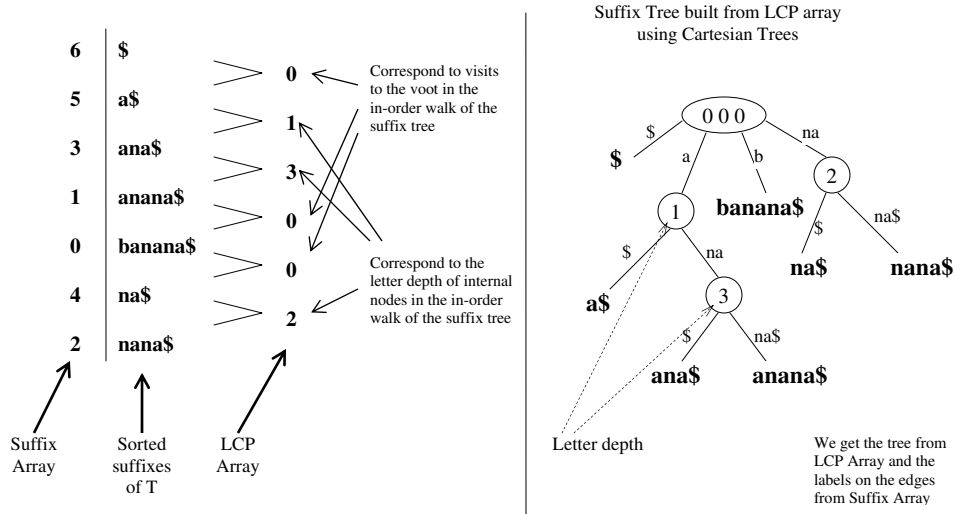
6	$	0	Correspond to visits to the voot in the in-order walk of the suffix tree
5	a$	1	
3	ana$	3	
1	anana$	0	
0	banana$	0	Correspond to the letter depth of internal nodes in the in-order walk of the suffix tree
4	na$	2	
2	nana$		

Suffix Array   Sorted suffixes of T   LCP Array

Suffix Tree built from LCP array using Cartesian Trees

Letter depth

We get the tree from LCP Array and the labels on the edges from Suffix Array

Figure 2: Suffix Array and LCP Array examples and their relation to Suffix Tree

4.2 Suffix Arrays and Suffix Trees

To motivate the construction of a suffix array and a LCP array, note that a suffix array stores the leaves of the corresponding suffix tree, and the LCP array provides information about the height of internal nodes, as seen in Figure 2. Our aim now is to show that suffix trees can be transformed into suffix arrays in linear time and vice versa.

To formalize this intuition, we use **Cartesian Trees** from L15. To build a Cartesian tree, store the minimum over all LCP array entries at the root of the Cartesian tree, and recurse on the remaining array pieces. For a concrete example, see Figure 3 below.
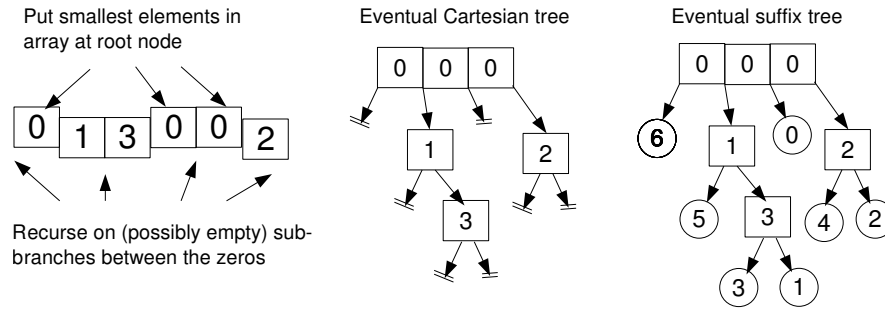
Figure 3: Constructing the Cartesian tree and suffix tree from LCP array

4.2.1 From Suffix Trees to Suffix Arrays

To transform a suffix tree into a suffix array, we simply run an in-order traversal of the suffix tree. As noted earlier, this process returns all suffixes in alphabetical order.

4.2.2 From Suffix Arrays to Suffix Trees

To transform a suffix array back into a suffix tree, create a Cartesian tree from the LCP array associated to the suffix array. Unlike in L15, use all minima at the root. The numbers stored in the nodes of the Cartesian tree are the letter depths of the internal nodes! Hence, if we insert the entries of the suffix array in order into the Cartesian tree as leaves, we recover the suffix tree. In fact, we are rewarded with an augmented suffix tree with informationa bout the letter depths. From L15, this construction is possible in linear time.

5 DC3 Algorithm for Building Suffix Arrays

Here, we give a description of the DC3 (Difference Cover 3) divide and conquer algorithm for building a suffix array in $O(|T| + sort(\Sigma))$ time. We closely follow the exposition of the paper by Karkkainen-Sanders-Burkhardt [3] that originally proposed the DC3 algorithm. Because we can create suffix trees from suffix arrays in linear time, a consequence of DC3 is that we can create suffix trees in linear time, a result shown independently of DC3 by Farach [2], McCreight [7], Ukkonen [10], and Weiner [11]

1. Sort the alphabet Σ. We can use any sorting algorithm, leading to the $O(\text{sort}(\Sigma))$ term.

2. Replace each letter in the text with its rank among the letters in the text. Note that the rank of the letter depends on the text. For example, if the text contains only one letter, no matter what letter it is, it will be replaced by 1. This operation is safe, because it does not change any relations we are interested in. We also guarantee that the size of the alphabet being used is no larger than the size of the text (in cases where the alphabet is excessively large), by ignoring unused alphabets.

3. Divide the text T into 3 parts and package triples of letters into *megaletters*. More formally,

form T_0, T_1, and T_2 as follows:

$$T_0 = \; < (T[3i], T[3i+1], T[3i+2]) \quad \text{for} \quad i = 0, 1, 2, \dots >$$
$$T_1 = \; < (T[3i+1], T[3i+2], T[3i+3]) \quad \text{for} \quad i = 0, 1, 2, \dots >$$
$$T_2 = \; < (T[3i+2], T[3i+3], T[3i+4]) \quad \text{for} \quad i = 0, 1, 2, \dots >$$

Note that T_i's are just texts with $n/3$ letters of a new alphabet Σ^3. Our text size has become a third of the original, while the alphabet size has cubed.

4. Recurse on $< T_0, T_1 >$, the concatenation of T_0 and T_1. Since our new alphabet is of cubic size, and our original alphabet is pre-sorted, radix-sorting the new alphabet only takes linear time. When this recursive call returns, we have all the suffixes of T_0 and T_1 sorted in a suffix array. Then all we need is to sort the suffixes of T_2, and to merge them with the old suffixes to get suffixes of T, because
$$\texttt{Suffixes}(T) \cong \texttt{Suffixes}(T_0) \cup \texttt{Suffixes}(T_1) \cup \texttt{Suffixes}(T_2)$$
If we can do this sorting and merging in linear time, we get a recursion formula $T(n) = T(2/3n) + O(n)$, which gives linear time.

5. Sort suffixes of T_2 using radix sort. This is straight forward to do once we note that

$$T_2[i :] \cong T[3i+2 :] \cong (T[3i+2], T[3i+3 :]) \cong (T[3i+2], T_0[i+1 :]).$$

The logic here is that once we rewrite $T_2[i :]$ in terms of T, we can pull off the first letter of the suffix and pair it with the remainder. We end up with something where the index $3i+3$ corresponds with the start of a triplet in T_0, specifically, $T_0[i+1]$, which we already have in sorted order from our recursive call.

Thus, we can radix sort on two coordinates, the triplet $T_0[i+1]$ and then the single alphabet $T[3i+2]$, both of which we know the sorted orders of. This way, we get $T_2[i :]$ in sorted order. Specifically, the radix sort is just on two coordinates, where the second coordinate is already sorted.

6. Merge the sorted suffixes of T_0, T_1, and T_2 using standard linear merging. The only problem is finding a way to compare suffixes in constant time. Remember that suffixes of T_0 and T_1 are already sorted together, so comparing a suffix from T_0 and a suffix from T_1 takes constant time. To compare against a suffix from T_2, we will once again decompose it to get a suffix from either T_0 or T_1. There are two cases:

- Comparing T_0 against T_2:

$$
\begin{aligned}
T_0[i :] \quad &\texttt{vs} \quad T_2[j :] \\
\cong \quad T[3i :] \quad &\texttt{vs} \quad T[3j+2 :] \\
\cong \quad (T[3i], T[3i+1 :]) \quad &\texttt{vs} \quad (T[3j+2], T[3j+3 :]) \\
\cong \quad (T[3i], T_1[i :]) \quad &\texttt{vs} \quad (T[3j+2], T_0[j+1 :])
\end{aligned}
$$

So we just compare the first letter and then, if needed, compare already sorted suffixes of T_0 and T_1.

- Comparing T_1 against T_2:

$$
\begin{aligned}
& & T_1[i :] \quad &\texttt{vs} \quad T_2[j :] \\
&\cong & T[3i + 1 :] \quad &\texttt{vs} \quad T[3j + 2 :] \\
\cong & \quad (T[3i + 1], T[3i + 2], T[3i + 3 :]) \quad &\texttt{vs} \quad (T[3j + 2], T[3j + 3], T[3j + 4 :]) \\
\cong & \quad (T[3i + 1], T[3i + 2], T_0[i + 1 :]) \quad &\texttt{vs} \quad (T[3j + 2], T[3j + 3], T_1[j + 1 :])
\end{aligned}
$$

So we can do likewise by first comparing the two letters in front, and then comparing already sorted suffixes of T_0 and T_1 if necessary.

References

[1] R. Boyer and J. Moore. *A fast string searching algorithm.* Communications of the ACM, 20(10):762772, 1977.

[2] M. Farach. *Optimal suffix tree construction with large alphabets.* In Proc. 38th Annual Symposium on Foundations of Computer Science, pages 137-143. IEEE, 1997.

[3] Juha Karkkainen, Peter Sanders, *Simple linear work suffix array construction,* In Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 03). LNCS 2719, Springer, 2003, pp. 943-955

[4] R. M. Karp and M. O. Rabin. *Efficient randomized pattern-matching algorithms.* IBM Journal of Research and Development, 31:249260, March 1987.

[5] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. *Linear-time longest-common-prefix computation in suffix arrays and its applications.* In Proc.12th Symposium on Combinatorial Pattern Matching (CPM 01), pages 181192. Springer-Verlag LNCS n. 2089, 2001.

[6] D. E. Knuth, J. H. Morris, and V. R. Pratt. *Fast pattern matching in strings.* SIAM Journal of Computing, 6(2):323350, 1977.

[7] E. M. McCreight. *A space-economic suffix tree construction algorithm.* J. ACM,23(2):262-272, 1976.

[8] S. Muthukrishnan. *Efficient algorithms for document retrieval problems.* SODA 2002:657-666

[9] Richard Cole, Tsvi Kopelowitz, Moshe Lewenstein. *Suffix Trays and Suffix Trists: Structures for Faster Text Indexing.* ICALP 2006: 358-369

[10] E. Ukkonen. *On-line construction of suffix trees.* Algorithmica, 14(3):249-260, 1995.

[11] P. Weiner. *Linear pattern matching algorithm.* In Proc. 14th Symposium on Switching and Automata Theory, pages 1-11. IEEE, 1973.