Chapter 14

Suffix Tree and Suffix Array

14.1 Summary

1. `Trie:` In this section, we introduce *trie*, a useful data structure to facilitate searching in a large text base.

2. `Suffix Tree:` In this section, we introduce suffix tree, a special type of trie used in searching particular pattern in large text.

3. `Suffix Array:` In this section, we describe suffix array, an alternative of suffix tree with more efficient searching ability.

14.2 Trie

14.2.1 What is trie?

A string is a list of characters over an alphabet $\sum = \{a, b, c, \ldots\}$ with $|\sum| = \sigma$. A *trie* is a multi-way tree structure for storing strings over an alphabet. The name of trie comes from the work re**trie**val. A formal definition is given as follows,

Definition 85. *A trie is a tree for storing strings in which there is one node for every common prefix. The strings are stored in extra leaf nodes.*

14.2.2 Why do we need trie?

The trie is a data structure that can be used to do a fast search in a large text. Given a collection of strings, we want to organize them in a manner, such that the searching for a particular pattern is fast.

14.2.3 Non-compact trie

In non-compact trie, each edge represent exactly one symbol in the alphabet. Given data "algorithm, algorithms, algebra, aaaah", figure 14.1 shows the corresponding non-compact trie, where the $ sign marks the end of a string.

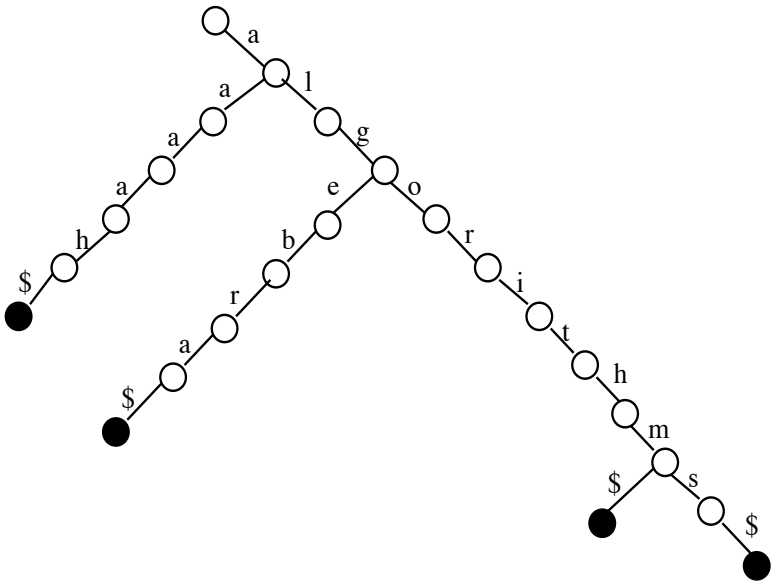

Figure 14.1: This is an example of a non-compact trie, in which four strings, $algorithm, algorithms, algebra, aaaah$ are stored. To search for a string, for example, $algorithm$, we start at the root and we follow the a edge, followed by the l edge, then g edge, etc., at last, we reach a leaf node corresponding to $algorithm$. However, if we are trying to search for another string $aloha$, after we traverse a edge and l edge we can go nowhere, which means the string $aloha$ is not in this text base.

14.2.4 Compact trie

This type of trie is different from the non-compact trie in that an edge can be labelled with more than one characters. An example is illustrated in figure 14.2, in which the same strings $algorithm, algorithms, algebra, aaaah$ are stored in a more compact way.

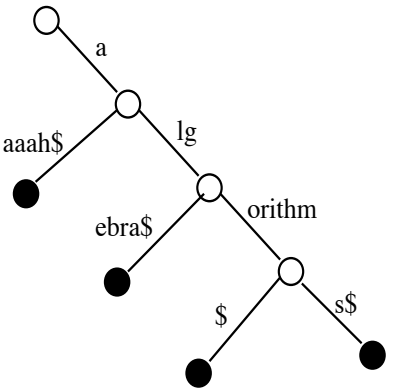

Figure 14.2: This is an example of a compact trie, in which the four strings, $algorithm, algorithms, algebra, aaaah$ are stored in a more compact way. Searching is the same as for non-compact trie, the only difference is in each step we may traverse multiple symbols.

14.3 Suffix Tree

14.3.1 What is suffix tree?

Definition 86. *A suffix tree is a trie of all the suffixes of a string.*

The above definition can be best illustrated by figure 14.3, which shows the suffix tree for a string *bananaba*\$.

14.3.2 Space cost of suffix tree

The space cost of suffix tree is $O(n^2)$, where n is the length of the string.
The worst case happens when the string has no duplicate. (Consider the suffix tree for a string *abcde*\$).

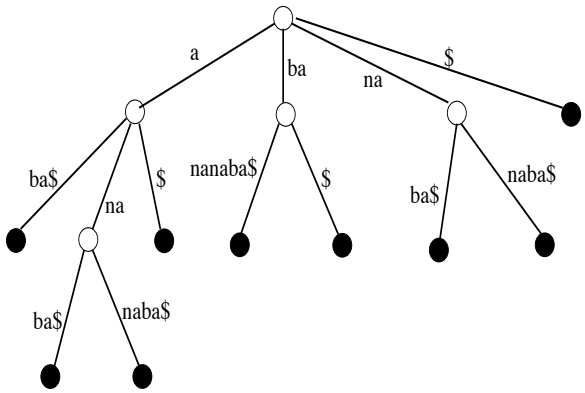

Figure 14.3: This figure shows a suffix tree for the string *bananaba*\$. It is a compact trie composed of all the suffixes of the string *bananaba*\$.

A more efficient way of storing suffix tree

Figure 14.4 shows a more efficient way to store a suffix tree. At here we replace every string by a pair of indices, (a, b), where a is the index of the beginning of the string and b the index of the end of the string.

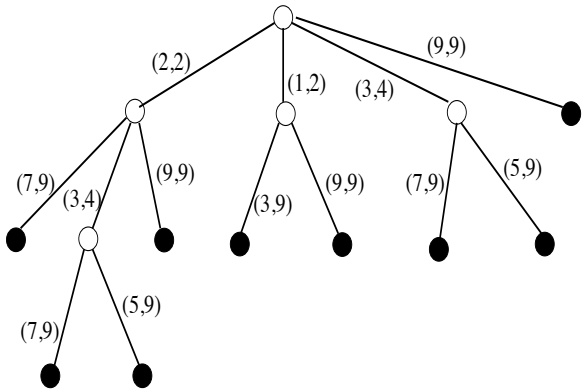

Figure 14.4: This figure shows a efficient way for storing suffix tree for the string *bananaba*\$. Note that every substring is represented as a pair of indices. For example, *aba* is $(6, 8)$, *a*\$ is $(8, 9)$, etc.

Theorem 87. *The space cost of the efficient suffix tree is of $O(n)$, where n is the length of the string.*

Proof. Consider figure 14.4, there are $n+1$ leaf nodes in this tree. Also, every inner node has

more than one children, which means the total number of inner nodes is of $O(n)$. Besides, each edge has constant storage cost. Hence, the space cost of this kind of suffix tree is of $O(n)$. □

14.3.3 Construction of suffix tree

The question here is how long does it take to build a suffix tree?

In general case

Naive way takes time $O(n^2 \log \sigma)$. In 1972, Wiener: $O(n)$ construction algorithm for constant size σ. It can be easily generalized to $O(n \log \sigma)$ for general alphabets.

For deterministic comparion-based algorithm

After sorting, we can construct a suffix tree in linear time. e.g., $cost = sort + O(n)$.

14.3.4 Application of suffix trees

Now we have known what is a suffix tree, and how to build suffix tree. Nonetheless, what good are suffix trees anyway?

String matching

Given pattern P, with $|P| = m$; text T, with $|T| = n$, searching for P in T can be solved in $O(m)$ time. (after the suffix tree for T has been built in $O(n) + sort$ time)

This can also be used to determine how many time does P appear in T (with its location)? (by counting how many leaves have common prefix P)

Longest palendrome

Palindrome is a string, P, such that $P = P^R$. For example, $mom = reverse(mom)$. Here, the problem is to find the longest palendrome in a text W. For example, *ississi* is the longest palindrome in *mississippi*.

The longest palindrome of W can be found in $O(n)$ time by building the suffix tree on $W\$W^R$. The longest palendrome can be found by searching for the longest common prefix of one suffix starting in W and another in W^R.

Hamming distance matching

Definition 88. *The hamming distance between two patterns, P and Q, is K, means Q can be obtained by changing the value of K characters of P. For example, the hamming distance between 'glass' and 'blast' is two, because 'glass' becomes 'blast' after changing two characters, g to b and s to t.*

Now, our problem is that given a pattern of text P of length m and a text W of length n, we want to find a place in W that is hamming distance at most K from P.

This problem can be solved in a naive way in time $O(nm)$.

We can build suffix tree on $W\$P@$, then solve this problem in time $O(nK)$.

14.4 Suffix Array

In previous lectures we introduced suffix tree, a compact trie-like data structure whose nodes correspond to all suffixes of a given string. Given the question that whether or not a text S of length n over an alphabet σ contains a substring q of length m, A suffix tree provides you the answer in $O(m \log |\sigma|)$ time, because any substring of S is the prefix of some suffix in this suffix tree. Such a suffix tree can be built in $O(n \log |\sigma|)$ time and $O(n)$ space.

Now we are going to introduce a new data structure called suffix array. Basically, a suffix array is a sorted list of all the suffixes of S. Similar as suffix trees, suffix arrays help lookup of any substring of a text and indentification of repeated substrings. And it is more compact than a suffix tree and suitable for storing in secondary memory.

14.4.1 What is suffix array

Definition 89. *Given a text $S = s_1, ..., s_n$, a suffix array for S is an array of integers of range 0 to n specifying the lexicographix ordering of the $n + 1$ suffixes of the string $S\$$.*

Next we show an example to explain this concept.

Assume we have such a text S : *acaaacatat*\$, where \$ is the ending symbol, the basic suffix array of this text is:

	suffixes
0	*aaacatat*\$
1	*aacatat*\$
2	*acaaacatat*\$
3	*acatat*\$
4	*atat*\$
5	*at*\$
6	*caaacatat*\$
7	*catat*\$
8	*tat*\$
9	*t*\$
10	\$

All suffixes of S are listed in the above table. Obviously this representation is not concise: totally we need $n(n+1)/2$ bytes to store the array and $O(n \log n)$ time to sort them.

We need a more clever representation.

First we assign index points to the sample text. In this example, index points are assigned character by character and hence we can search the sample text with the suffix array at any positions later:

text	a	c	a	a	a	c	a	t	a	t	\$
index	0	1	2	3	4	5	6	7	8	9	10

Now we can sort the index points according to their corresponding suffixes. After sorting we get:

Sorted Suffix	Index
aaacatat\$	2
aacatat\$	3
acaaacatat\$	0
acatat\$	4
atat\$	6
at\$	8
caaacatat\$	1
catat\$	5
tat\$	7
t\$	9
\$	10

Thus we get the resulting index points to be the suffix array for the sample text:

$$suffix\ array:\ 2\ 3\ 0\ 4\ 6\ 8\ 1\ 5\ 7\ 9\ 10$$

14.4.2 Coupling suffix array with LCPs

To binary search a pattern in S takes $O(\log n)$ steps. In each step we need to compare m characters of the text and the pattern. This leads to a running time of $O(m \log n)$.

Our goal is to reduce the search time, with very little construction time overhead. Specifically we want the search time to be within $O(m + \log n)$.

Recall the definition of Least Common Prefix (LCP):

Definition 90. *LCP(i, j) is the length of the longest common prefix of the suffixes specified in positions i and j in suffix tree's sorted order.*

E.g. for $S = acaaacatat$, LCP$(0, 1)$ is thus 2, the length of the longest common prefix of *aaacatat*\$ and *aacatat*\$.

Now we get a new definition of suffix arrays:

Definition 91. *Suffix Array is sorted order of suffixes with pairwise LCP of neighboring suffixes.*

14.4.3 How to search

How can we use the LCP information to help the search in suffix array?

Let's say an examination of a character of P is redundant if that character has been examined before. Our goal is to limit the number of redundant character comparisons to one per iteration of the binary search.

Assume in a binary search process L and R denote the left and right boundaries of the current search interval. At the beginning $L = 0$ and $R = n - 1$. In each iteration of the binary search a query is made at location $M = \lceil (R + L)/2 \rceil$. Let $suf[i]$ be the i-th suffix in the suffix array. Let l and r denote the longest prefixes of $suf[L]$ and $suf[R]$ that match a prefix of pattern P. Assume we already have the suffix array for text S and LCP(i, j) for all i, j in $[0, n]$.

In an iteration of the improved binary search algorithm, if $l = r$ we compare P to $suf[M]$. In this case the minimum of l and r is also the maximum of the two and no redundant character comparisons are made.

If $l \neq r$, there are three cases (without loss of generity, assume $l > r$):

1. LCP$(L, M) > l$
   Then the common prefix of the suffixes $suf[L]$ and $suf[M]$ is longer than the common prefix of P and $suf[L]$. Therefore P agrees with the suffix $suf[M]$ up through character l. Characters $l + 1$ of $suf[L]$ and $suf[M]$ are identical and lexically less than character $l + 1$ of P. So any possible starting position must start to the right of M in suffix array. In this case no examination of P is needed. L is set to M while l, r and R remain unchanged.

2. LCP$(L, M) < l$
   Then the common prefix of suffix $suf[L]$ and $suf[M]$ is smaller than the common prefix of $suf[L]$ and P. Therefore P agrees with $suf[M]$ up through character $LCP[L, M]$. Character LCP$(L, M) + 1$ of P and $suf[L]$ are identical and lexically less than the character LCP$(L, M) + 1$ of $suf[M]$. So any possible starting position must start to the left of M in the suffix array. In this case no examination of P is needed. R is set to M, r is set to LCP(L, M). r and R remain unchanged.

3. LCP$(L, M) = l$
   Then P agrees with $suf[M]$ up to character l. Then we should compare P to $suf[M]$

starting from character $l+1$. The result determinses which of L and R change along with the corresponding change of l and r.

We draw a tabular here to better illustrate the three cases,

$$suf[L] = abcdefg\ldots, suf[R] = abcwxyz\ldots, P = abcdemn\ldots, l = 5, r = 3$$

	$suf[M]$	LCP(L, M)	Next Step
Case 1	$abcdef\ldots$	7	$L \longleftarrow M$, no change toR, r, l
Case 2	$abcdgg\ldots$	4	$R \longleftarrow M, r \longleftarrow 4$, no change to$l, L$
Case 3	$abcdeg\ldots$	5	$L \longleftarrow M, l \longleftarrow 5$, no change to$r, R$

Theorem 92. *Using the LCP values, the search algorithm does at most $O(m + \log n)$ comparisons.*

Proof. Notice neither l nor r decrease along iterations. Let δ be the difference between the value of $\max(l, r)$ at the beginning and at the end of the iteration. For each iteration the number of single character comparisons is limited to $\delta + 1$. Sum up $\delta + 1$ over all $\log n$ iterations and use the fact that $\Sigma\delta \leq m$, we get that the number of total comparisons is at most $m + \log n$. □

14.4.4 Construction of suffix array

How do we construct such a suffix array efficiently?

1. Building suffix array from suffix tree takes linear time by DFS. So the time complexity is $O(n \log \sigma)$.
2. Implement by bucket sort (radix sort). Each call runs in $O(n)$ time. The sorting algorithm runs in $O(n \log n)$ time and consumes $O(n)$ space.

14.4.5 How do we compute LCPs effectively?

Seemingly we need to compute n^2 LCP values. However we observe only the LCP values of the values of L and R we encounter in our binary search are needed, These values are

constant for each search and there are only linearly many of them.

To compute all LCP values needed we take these two steps:

1. Compute the LCP values for pairs of neighboring suffixes in the suffix array.
2. For the fixed binary search tree compute the LCP values for its internal nodes using the result we get at previous step.

This can be done during the construction of the suffix array, without additional overhead, or alternatively in linear time with a scan over the suffix array as follows:

LCP Algorithm: Consider a suffix array $SA[1, ldots, n]$. If we were to compute the LCP of $SA[1]$ with $SA[2]$, then $SA[2]$ with $SA[3]$, etc., we'd end up with an $O(n^2)$ algorithm. Instead, we proceed in a different order, which allows for a linear-time algorithm.

Lemma 93. *Given an SA array, the LCP array can be computed in linear time.*

proof Let $P[i]$ be the j such that $SA[j] = i$. That is, $P[i]$ is the rank of the ith suffix. This can be computed form SA in linear time.

Now compute the LCP of $SA[P[1]]$ with $SA[P[1]-1]$ by character-by-character brute force. Suppose this is k. If $k = 0$, then this took constant time and we move on. Otherwise $k > 0$.

Now compute the LCP of $SA[P[2]]$ with $SA[P[2]] - 1$. We know that this LCP is at least $k - 1$ so we skip these characters and continue character-by-character as before.

And continue with the third suffix and so on.

Why is this linear time? The total work spent on finding mismatched characters is $O(n)$, since each mismatched character pair moves us to the next LCP computation, and there are only $n - 1$ LCP computations to perform.

As for the time spent on matched characters: this is why we compute on the first suffix, then the second, etc. Imagine a pointer that moves along the string. When we match the characters during the computation of the LCP of $SA[P[1]]$ with $SA[P[1] - 1]$, we move the pointer to the right for each matched pair found. The next computation starts with the pointer where it is and move it (potentially) further to the right. Thus, each matched character moves this pointer to the right by one space, and there are only n characters to match in total.

Thus the total computation time is $O(n)$. QED

14.4.6 Summary of suffix array

1. Suffix arrays are more space-efficient than suffix trees.
2. Suffix arrays can be build in $O(n \log n)$ time and requires $O(n)$ space.
3. Simple binary search in a suffix array takes $O(m \log n)$ time. When coupled with LCP information it can be done in $O(m + \log n)$ time.
4. The LCP values can be computed in linear time and requirs extra linear space.

14.5 Equivalence of Suffix Trees and Suffix Arrays

Theorem 94. *Given a suffix tree, the corresponding suffix array can be computed in linear time.*

Proof: A traversal of the suffix tree gives the sorted order of the leaves. The LCPs can be computed during the tree traversal.

Theorem 95. *Given a suffix array, the corresponding suffix tree can be computed in linear time.*

Proof: First, compute the LCP array in linear time. Now interleave the suffix array and the LCP array, assigning numerical values to the suffix array entries equal to their string length. The suffix tree is given by the cartesian tree of the interleaved array, which we've seen can be computed in linear time.

14.6 Computing Suffix Arrays in Linear Time (plus sorting)

We noted that suffix arrays can be computed in time equal to the sorting time of the underlying alphabet. Here we show how to do this.

Observation: Let $S = \Sigma^n$ be an n character string from alphabet Σ. Let $\Phi : \Sigma \to \Sigma'$ be a monotone function from alphabet Σ to alphabet Σ'. Let $S' = \Phi(S) = \Phi(S_1)\Phi(S_2)\ldots\Phi(S_n)$. Then the suffix array of S is identical to the suffix array of $\S'$. This is because the suffix array of S depends on the sorted order of the suffixes of S, but a monotone mapping of the characters doesn't change the order, so it doesn't change the suffix array.

Example: $S = banana\$$.

==========

This is the MIT stuff, but it needs the figure back in, and it needs to be weeded down.

==========

14.7 Overview

In this lecture, we consider the string matching problem - finding all places in a text where some query string occurs. 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 [**?**], Boyer-Moore [**?**], and Rabin-Karp [**?**].

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.

14.8 Storing Strings and String Matching

First, we introduce some notation. Throughout these notes, Σ will denote a finite alphabet. An example of a finite alphabet is the standard set of English letters $\Sigma = \{a, b, c, ..., z\}$. A fixed string of characters $T \in \Sigma^*$ will comprise what we call a *text*. Another string of characters $P \in \Sigma^*$ will be called a *search pattern*.

For integers i and j, define $T[i : j]$ as the substring of T starting from the i^{th} character and ending with the j^{th} character inclusive. We will often omit j and write $T[i :]$ to denote the suffix of T starting at the i^{th} character. Finally, we let the symbol $\circ$ denote concatenation. As a simple illustration of our notation, $(abcde[0 : 2]) \circ (cde[1 :]) = abcde$.

Now we can formally state the string matching problem: Given an input text $T \in \Sigma^*$ and a pattern $P \in \Sigma^*$, we want to find all occurrences of P in T. Closely related variants of the string matching problem ask for the first, first k, or some occurrences, rather than for all occurrences.

14.8.1 Tries and Compressed Tries

A commonly used string data structure is called a **trie**, a tree where each edge stores a letter, each node stores a string, and the root stores the empty string. The recursive relationship between the values stored on the edges and the values stored in the nodes is as follows: 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 $v_{i-1}v_i$. We will denote the strings stored in the leaves of the trie as words, and the strings stored in all other nodes as prefixes.

If there is a natural lexicographical ordering on the elements in Σ, we order the edges of every node's fan-out alphabetically, from left to right. With respect to this ordering, in order traversal of the leaves gives us every word stored in the trie in alphabetical order. In particular, it is easy to see that the fan-out of any node must be bounded above by the size of the alphabet $|\Sigma|$.

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.

gure

14.8.2 Suffix Trees

A **suffix tree** is a compact trie built on the $|T| + 1$ suffixes of T. For example, if our text is the string $T = banana\$$, then our suffix tree will be built on $\{banana\$, anana\$, nana\$, ana\$, na\$, a\$\}$. 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.

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

- *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. Therefore, the total runtime of string matching is $O(|P| + k)$, where k denotes the size of the output.
- *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.
- *Longest Repeating Substring* To find the longest repeated substring, we look for the branching node with maximum letter depth.
- *Multiple Documents* When there are multiple texts in question, we can concatenate them with $\$_1, \$_2, ..., \$_n$. Then to find the longest common substring, we look for the node with the maximum letter depth with greater than one distinct \$ below.

Compared to binary search trees, suffix trees allow for faster queries and utilize less storage, especially for texts with a large number of suffixes. Compared to hash tables, suffix trees allow for faster worst case queries, and avoid problems caused by collisions and computing hash functions.

14.9 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.

14.9.1 Creating Suffix Arrays

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 $[6, 5, 3, 1, 0, 4, 2]$. 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**. These LCP arrays can be constructed in $O(|T|)$ time, a result due to Kasai et al [**?**]. 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 $O(|P| + \log |T|)$ (see homework).

gure

14.9.2 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 define the **Cartesian Tree** of an LCP array. 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.

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.

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. 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.

Fix this figure

14.9.3 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 [**?**] 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 [**?**], McCreight [**?**], Ukkonen [**?**], and Weiner [**?**]

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:

$$T_0 = \ < (T[3i], T[3i+1], T[3i+2]) \quad \text{for} \quad i = 0, 1, 2, \ldots >$$

$$T_1 = \ < (T[3i+1], T[3i+2], T[3i+3]) \quad \text{for} \quad i = 0, 1, 2, \ldots >$$

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

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:

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

  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:

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

  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.

14.10 Document Retrieval

The statement of the document retrieval problem is as follows: Given a collection of texts $\{T_1, T_2, ..., T_n\}$, we want to find all distinct documents that contain an instance of a query pattern P. Naively, we can store the suffixes of the text $T = T_1 \circ \$_1 \circ T_2 \circ \$_2 \circ ... \circ T_n \circ \$_n$ in a suffix tree and query for the pattern. However, the result of this query will include multiple matches of P within the same text, whereas we only want to find which texts contain P. Hence, a simple query doesn't give us the answer we want, and is costlier than we would like. The fix we present will follow the paper by Muthukrishan [**?**].

Consider the suffix array A of the augmented text T. The node corresponding to a pattern P in the suffix tree corresponds to some interval $[i, j]$ in the suffix array. Now the texts containing the pattern are precisely the indices of all the dollar signs appearing in the interval $[i, j]$ in A. Hence, solving document retrieval is equivalent to finding the position of the first dollar sign of each type within the interval $[i, j]$, as the suffixes at those positions cannot contain repeats of P.

To this end, augment the suffix array by connecting each dollar sign of a given type to the previous dollar sign of the same type. Then the queries can be answered with a **Range**

Minimum Data Structure. In particular, this gives us a $O(|P| + d)$ runtime, where d is the number of texts to report.