Chapter 3

Suffix Tree

3.1 Introduction

The suffix tree of a string is a fundamental data structure for pattern matching [305]. It has many biological applications. The rest of this book will discuss some of its applications, including

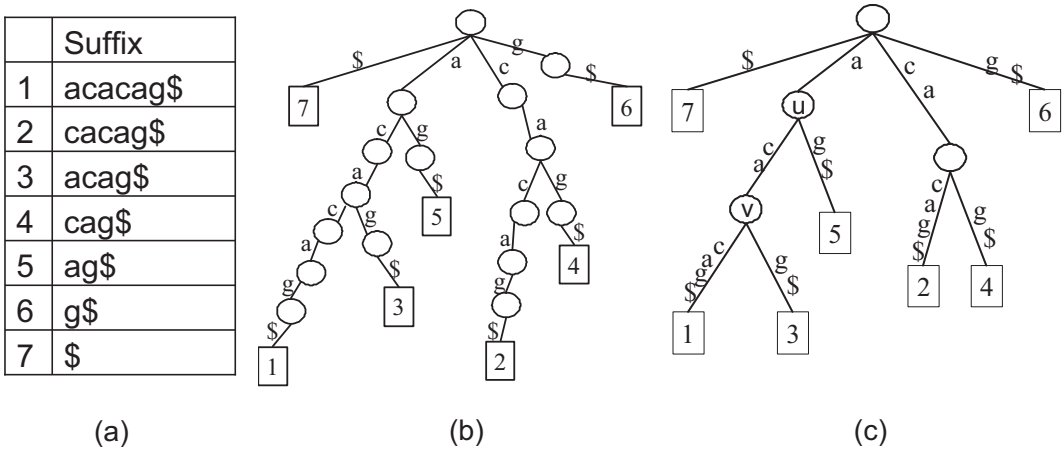
- Biological database searching (Chapter 5)
- Whole genome alignment (Chapter 4)
- Motif finding (Chapter 10)

In this chapter, we define a suffix tree and present simple applications of a suffix tree. Then, we discuss a linear suffix tree construction algorithm proposed by Farach. Finally, we discuss the variants of a suffix tree like suffix array and FM-index. We also study the application of suffix tree related data structures on approximate matching problems.

3.2 Suffix Tree

This section defines a suffix tree. First, we introduce a suffix trie. A trie (derived from the word re*trie*val) is a rooted tree where every edge is labeled by a character. It represents a set of strings formed by concatenating the characters on the unique paths from the root to the leaves of the trie. A suffix trie is simply a trie storing all possible suffixes of a string S. Figure 3.1(a) shows all possible suffixes of a string $S[1..7] = acacag\$$, where \$ is a special symbol that indicates the end of S. The corresponding suffix trie is shown in Figure 3.1(b). In a suffix trie, every possible suffix of S is represented as a path from the root to a leaf.

A suffix tree is a rooted tree formed by contracting all internal nodes in the suffix trie with single child and single parent. Figure 3.1(c) shows the suffix tree made from the suffix trie in Figure 3.1(b). For each edge in the suffix tree, the edge label is defined as the concatenation of the characters on the



	Suffix
1	acacag$
2	cacag$
3	acag$
4	cag$
5	ag$
6	g$
7	$

FIGURE 3.1: For $S = acacag\$$, (a) shows all suffixes of S, (b) shows the suffix trie of S, and (c) shows the suffix tree of S.

edges of the suffix trie which are merged. The path label of a node is defined as the concatenation of the edge labels from the root to the node. We define the string depth of a node to be the length of the path label of the node. For example, in Figure 3.1(c), the edge label of the edge (u, v) is ca, the path label of the node v is aca, and the string depth of the node v is 3.

Below, we analyze the space complexity of a suffix tree. Unlike in other chapters, the space complexity is analyzed in terms of bits to visualize the actual size in memory. Consider a string S of length n over the alphabet $\mathcal{A}$. Let σ be the size of $\mathcal{A}$ ($\sigma = 4$ for DNA and $\sigma = 20$ for protein). The suffix tree T of S has exactly n leaves and at most $2n$ edges. Since each character can be represented in $\log \sigma$ bits and every edge in T can be as long as n, the space required to store all edges can be as large as $O(n^2 \log \sigma)$ bits. Observe that every edge label is some substring of S. To avoid storing the edge label explicitly, the label of each edge can be represented as a pair of indices, (i, j), if the edge label is $S[i..j]$. For example, the edge "a" in Figure 3.1(c) could be represented by $(1, 1)$, the edge "ca" is represented by $(2, 3)$ and the edge "$cag\$$" is represented by $(4, 7)$. Each index is an integer between 1 and n and hence it can be represented by $\log n$ bits. Thus, each edge label can be stored in $O(\log n)$ bits and the suffix tree T can be stored in $O(n \log n)$ bits.

In general, we can store two or more strings using a generalized suffix tree. For example, the generalized suffix tree of $S_1 = acgat\#$ and $S_2 = cgt\$$ is shown in Figure 3.2. Note that different terminating symbols are used to represent the ends of different strings.

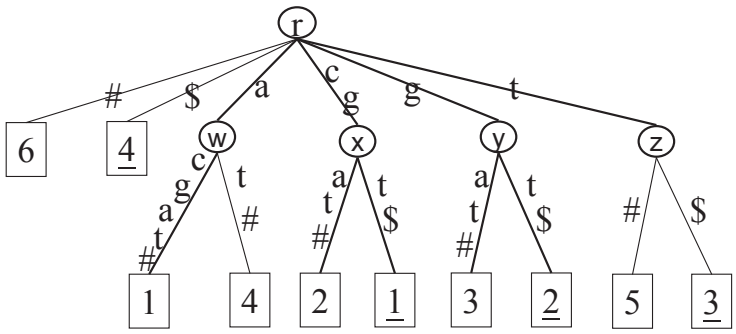


FIGURE 3.2: The generalized suffix tree of $S_1 = acgat\#$ and $S_2 = cgt\$$. The leaves with underlined integers represent the suffixes of S_2, and the rest of the leaves represent the suffixes of S_1.

3.3 Simple Applications of a Suffix Tree

Given a string S of length n, its suffix tree can be constructed in $O(n)$ time as shown in Section 3.4. Before we detail the construction algorithm, this section first discusses some basic applications of a suffix tree.

3.3.1 Exact String Matching Problem

Given a string S of length n, the exact string matching problem asks if a query pattern Q of length m exists in S. If yes, we would like to report all occurrences of Q in S.

This problem can be solved easily in $O(n+m)$ time using the Knuth-Morris-Pratt (KMP) algorithm. However, when n is big, the running time is long. (In genomic application, if S is human/mouse genome, n equals 3 billions.) If we can preprocess the string S, can we solve the exact string matching problem in a time independent of n? Here, we give a definite answer. We show that given the suffix tree for S, the occurrences of any pattern Q of length m can be found in $O(m + occ)$ time where occ is the number of occurrences of the pattern Q in S. The algorithm has two steps.

1: Starting from the root of the suffix tree, we traverse down to find a node x such that the path label of x matches Q. If such a path is found, Q exists in S; otherwise, Q does not occur in S.

2: If Q exists in S, all leaves in the subtree rooted at x are the occurrences of Q. By traversing the subtree rooted at x, we can list all occurrences of Q using $O(occ)$ time .

For example, consider the string $S = acacag\$$ whose suffix tree is shown in Figure 3.1(b). To find the occurrences of $Q = aca$, we traverse down the suffix tree of S along the path aca. Since such a path exists, we confirm Q

occurs in S. The positions of occurrences are 1 and 3, which are the leaves of the corresponding subtree.

Consider the case when $Q = acc$. We can find a path with label ac when we traverse down the suffix tree. However, from this point, we cannot extend the path for the character c; thus we report that the pattern acc does not exist in S.

Next, we give the time analysis. Consider the suffix tree T for S. For any query Q of length m, we can identify if there exists a path label in T matches Q in $O(m)$ time. Then, traversing the subtree to report all occurrences takes $O(occ)$ time where occ is the number of occurrences of Q in S. In total, all occurrences of Q in S can be found in $O(m + occ)$ time.

3.3.2 Longest Repeated Substring Problem

Two identical substrings in a sequence S are called repeated substrings. The longest repeated substring is the repeated substring which is the longest. In the past, without the use of a suffix tree, people solved this problem using $O(n^2)$ time. Moreover, with a suffix tree, we know that this problem can be solved in $O(n)$ time.

The basic observation is that if a substring is repeated at positions i and j in S, the substring is the common prefix of suffix i and suffix j. Hence, the longest repeated substring corresponds to the longest path label of some internal node in the suffix tree of S. Thus, the longest repeated substring can be found as follows.

1: Build a suffix tree T for the string S using $O(n)$ time.
2: Traverse T to find the deepest internal node. Since T has $O(n)$ nodes, this step can be done in $O(n)$ time. The length of the longest repeat is the length of the path label of the deepest internal node.

Thus, a suffix tree can be used to solve the longest repeated substring problem in $O(n)$ time. For example, consider $S = acacag\$$. As shown in Figure 3.1(c), the deepest internal node in the suffix tree of S is the node with the path label aca. Indeed, aca is the longest repeated substring of S.

3.3.3 Longest Common Substring Problem

Scientists are interested in finding similar regions between two sequences. This problem can be modeled as finding the longest common substring between those sequences. Given two strings P_1 and P_2 of total length n, the longest common substring problem asks for the longest common substring between P_1 and P_2. In 1970, Don Knuth conjectured that the longest common substring problem is impossible to solve in linear time. Below, using a generalized suffix tree, we show that the problem can be solved in linear time. The idea is to find a common prefix between the suffixes of P_1 and P_2. Below is the detail of the algorithm.

1: Build a generalized suffix tree for P_1 and P_2. This takes $O(n)$ time.
2: Mark each internal node with leaves representing suffixes of P_1 and P_2. This takes $O(n)$ time. This will ensure that the path labels of the marked internal nodes are common substrings of P_1 and P_2.
3: Report the deepest marked node. This step can be done in $O(n)$ time.

For example, consider $P_1 = acgat$ and $P_2 = cgt$, the longest common substring between P_1 and P_2 is cg. See Figure 3.3.

Note that this solution can be generalized to find the longest common substring for more than 2 sequences (see Exercise 13). By the way, the longest common substring is not the same as the longest common subsequence. The longest common subsequence is a sequence of characters that are not necessary contiguous, whereas the longest common substring is a contiguous substring! Besides, the longest common substring problem can be solved in $O(n)$ time using a suffix tree, whereas the longest common subsequence takes $O(n^2)$ time for a general alphabet (see Section 2.2.4).

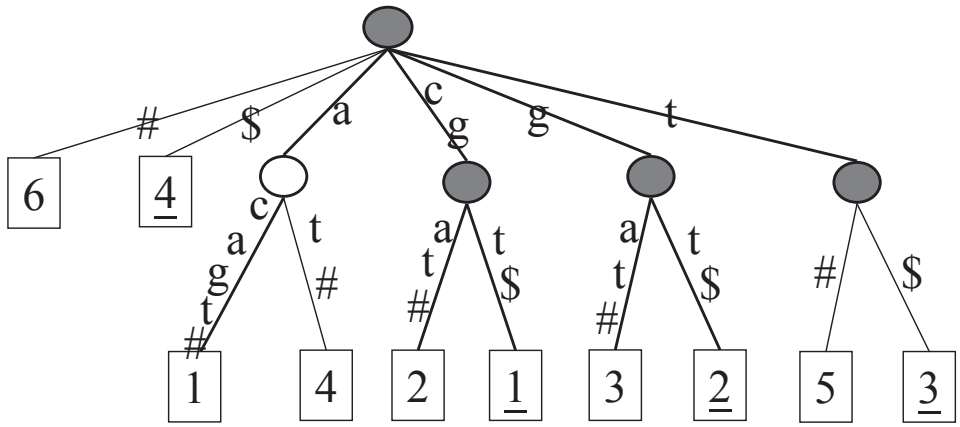


FIGURE 3.3: This is the generalized suffix tree for $P_1 = acgat\#$ and $P_2 = cgt\$$. All square nodes are leaves representing suffixes. For suffixes of P_2, the corresponding integers are underlined. The gray colored internal nodes are nodes whose subtrees contain suffixes of both P_1 and P_2. The path label of the deepest gray colored node is of depth 2, whose path label is cg, which is the longest common substring of P_1 and P_2.

3.3.4 Longest Common Prefix (LCP)

Given a string S of length n, for any $1 \leq i, j \leq n$, we denote $LCP(i, j)$ as the length of the longest common prefix of suffixes i and j of S. For example, for $S = acacag\$$, the longest common prefix of suffix 1 and suffix 3 is aca; hence, $LCP(1, 3) = 3$. This section describes an $O(n)$-time preprocessing to

create a data-structure such that, for any i, j, $LCP(i, j)$ can be reported in $O(1)$ time.

Let T be the suffix tree for S. The key observation is that $LCP(i, j)$ equals the length of the path label of the lowest common ancestor of the two leaves representing suffixes i and j (this lowest common ancestor is denoted as $LCA(i, j)$).

The problem of finding the lowest common ancestor (LCA) of two nodes in a tree is well studied. Harel and Tarjan [137] showed that for a tree of size n, after an $O(n)$-time preprocessing, the LCA for any two nodes in the tree can be computed in $O(1)$ time. The solution was then simplified by Schieber and Vishkin [259] and Bender and Farach-Colton [21]. Based on the $O(n)$-time preprocessing for the lowest common ancestor, the longest common prefix of any two suffixes can be computed in $O(1)$ time.

The LCP data-structure finds many applications. Section 3.3.5 demonstrates one such application.

3.3.5 Finding a Palindrome

In DNA, a complemented palindrome is a sequence of base pairs that reads the same backward and forward across the double strand. As described in Section 1.7.1, those sequences may be specific sites which are cut by restriction enzymes. This section describes an algorithm to locate all palindromes in a string.

First, we formally define the term palindrome and complemented palindrome. Given a string S, a palindrome is a substring u of S such that $u = u^r$ where u^r denotes the reverse string of u. For example, *acagaca* is a palindrome since $acagaca = (acagaca)^r$. A palindrome $u = S[i..i + |u| - 1]$ is called a maximal palindrome in S if $S[i - 1..i + |u|]$ is not a palindrome. A complemented palindrome is a substring u of S, such that $u = \bar{u}^r$ where $\bar{u}$ is a complement of u. For example, *acaugu* is a complemented palindrome. A complemented palindrome $u = S[i..i + |u| - 1]$ is said to be maximal if $S[i - 1..i + |u|]$ is not a complemented palindrome.

With the definition of maximal palindrome, every palindrome is contained in a maximal palindrome. In other words, a maximal palindrome is a compact way to represent all palindromes. Similarly, a maximal complemented palindrome is a compact way to represent all complemented palindromes. Below, we give a solution to find all maximal palindromes of a string in linear time based on a suffix tree. Note that maximal complemented palindromes can be found in a similar manner.

The basic observation of the solution is that every palindrome can be divided into two halves where the first half equals the reversal of the second half. Precisely, $u = S[i - (k - 1)..i + (k - 1)]$ is a length $(2k - 1)$ palindrome if $S[i + 1..i + (k - 1)] = (S[i - (k - 1)..i - 1])^r = S^r[n - i + 2..n - i + k]$. Similarly, $u = S[i - k..i + k - 1]$ is a length $2k$ palindrome if $S[i..i + k - 1] =$

$(S[i-k..i-1])^r = S^r[n-i+2..n-i+k+1]$. Utilizing this observation, we can find all maximal palindromes in four steps.

1: Build a generalized suffix tree for S and S^r.
2: Enhance the suffix tree so that we can answer the longest common prefix query in $O(1)$ time (see Section 3.3.4).
3: For $i = 1, \ldots, n$, find the longest common prefix for (S_i, S^r_{n-i+1}) in $O(1)$ time. If the length of the longest prefix is k, we have found an odd length maximal palindrome $S[i-k+1..i+k-1]$.
4: For $i = 1, \ldots, n$, find the longest common prefix for (S_i, S^r_{n-i+2}) in $O(1)$ time. If the length of the longest prefix is k, we have found an even length maximal palindrome $S[i-k..i+k-1]$.

Finally, we analyze the time complexity. The generalized suffix tree and the longest common prefix data structure can be constructed in $O(n)$ time. Since there are $O(n)$ longest common prefix queries and each such query can be answered in $O(1)$ time, we can find all the maximal palindromes in $O(n)$ time.

3.3.6 Extracting the Embedded Suffix Tree of a String from the Generalized Suffix Tree

Given a generalized suffix tree T for K strings $S_1, \ldots, S_K$, we aim to extract the suffix tree T_k for one particular string S_k.

Note that, for every $1 \leq k \leq K$, the suffix tree T_k for S_k is actually embedded in the generalized suffix tree T. Precisely, such an embedded suffix tree T_k consists of the leaves corresponding to S_k and their lowest common ancestors. The edges of T_k can be inferred from the ancestor-descendant relationship among the nodes. For example, in Figure 3.2, the embedded suffix tree for S_1 consists of r, w, and six leaves whose integers are not underlined. The edges are (r, w), $(r, 6)$, $(r, 2)$, $(r, 3)$, $(r, 5)$, $(w, 1)$, and $(w, 4)$. The embedded suffix tree for S_2 consists of r and four leaves whose integers are underlined. It has four edges, each attaches r to each of the four leaves.

To construct the embedded suffix tree T_k from T, we assume we have (1) the generalized suffix tree T for $S_1, \ldots, S_K$, (2) the lexicographical ordering of the leaves in T, and (3) the lowest common ancestor data structure [21, 137, 259]. Then, the embedded suffix trees T_k can be constructed in $O(|S_k|)$ time as follows. Figure 3.4 demonstrates an example.

1: Let the set of leaves be $a_1, \ldots, a_{|S_k|}$ ordered in lexicographical order;
2: We generate a tree R with the leaf a_1 and the root;
3: **for** $i = 2$ to $|S_k|$ **do**
4: We insert the internal node $LCA(a_i, a_{i+1})$ into R if it does not exist and attach a_i to this node;
5: **end for**

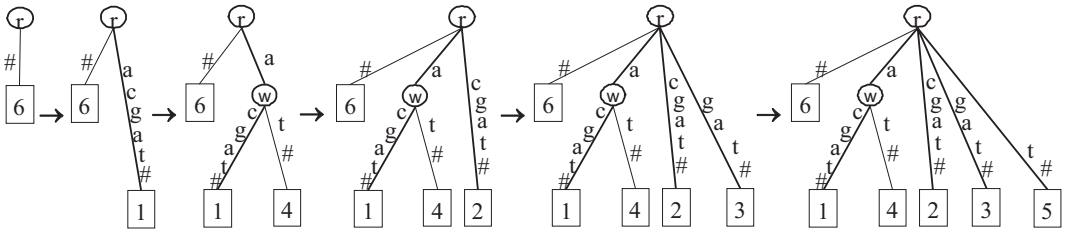


FIGURE 3.4: Constructing the embedded suffix tree of S_1 from the generalized suffix tree in Figure 3.2 for $S_1 = acgat\#$ and $S_2 = cgt\$$.

3.3.7 Common Substring of 2 or More Strings

Consider K strings $S_1, S_2, \ldots, S_K$ whose total length is n. Let $\ell(k)$ be the length of the longest substring common to at least k of these strings, for every $2 \leq k \leq K$.

For example, consider a set of 5 strings {sandollar, sandlot, handler, grand, pantry}. The longest substring appearing in all five strings is "an"; hence, $\ell(5) = 2$. $\ell(4) = 3$ and the corresponding substring is "and", which appears in {sandollar, handlot, handler, grand}. Using the same method, we can check that $\ell(3) = 3$ and $\ell(2) = 4$. The corresponding substrings are "sand" and "and", respectively.

This section aims to compute $\ell(k)$ for all $2 \leq k \leq K$. We solve the problem using a generalized suffix tree. Let T be the generalized suffix tree for all K strings. Then, for every internal node v in T, we let (1) $d(v)$ be the length of its path label and (2) $C(v) = |\mathcal{C}(v)|$ where $\mathcal{C}(v)$ is the set of distinct terminating symbols in the subtree rooted at v. Note that $\ell(k) = \max_{j=k}^{K} \max_{v:C(v)=j} d(v)$.

We observe that, for any node u in T, $d(u) = d(v) + len(u, v)$ where v is the parent of u and $len(u, v)$ is the length of the edge label of (u, v). Also, $\mathcal{C}(u) = \cup_{v \text{ is a child of } u} \mathcal{C}(v)$. Below, we detail the steps for computing $\ell(k)$ for $2 \leq k \leq K$.

1: Build a generalized suffix tree T for all K strings, each string having a distinct terminating symbol. This can be done in $O(n)$ time.

2: For every internal node u in T traversing in pre-order, we compute $d(u) = d(v) + len(u, v)$ where v is the parent of u. This step takes $O(n)$ time.

3: For every internal node u in T traversing in post-order, we compute $\mathcal{C}(u) = \cup\{\mathcal{C}(v) \mid v \text{ is a child of } u\}$ and set $C(u) = |\mathcal{C}(u)|$. This step takes $O(Kn)$ time.

4: This step computes, for $2 \leq k \leq K$, $V(k) = \max_{v:C(v)=k} d(v)$. The detail is as follows. We initialize $V(k) = 0$ for $2 \leq k \leq K$. For every internal node v in T, if $V(C(v)) < d(v)$, set $V(C(v)) = d(v)$. This step takes $O(n)$ time.

5: Set $\ell(K) = V(K)$. For $k = K - 1$ down to 2, set $\ell(k) = \max\{\ell(k + 1), V(k)\}$.

For time analysis, Step 3 runs in $O(Kn)$ time and all other steps take $O(n)$

time. Hence, the overall time complexity of the above algorithm is $O(Kn)$.

Can we further speed up the algorithm? Below, we give a definite answer by showing a way to reduce the running time of Step 3 to $O(n)$. Thus, we can compute $\ell(k)$ for $k = 2, \ldots, K$ using $O(n)$ time.

Prior to describe a faster solution for Step 3, we need some definitions. For every internal node v in T, let $N(v)$ be the number of leaves in the subtree of v. Let $n_k(v)$ be the number of leaves in the subtree of v representing suffixes of S_k. Intuitively, $n_k(v) - 1$ is the number of duplicated suffixes of S_k in the subtree of v. Hence $U(v) = \sum_{k:n_k(v)>0}(n_k(v) - 1)$ is the number of duplicate suffixes in the subtree of T rooted at v. Therefore, $C(v) = N(v) - U(v)$.

The difficulty is in computing $U(v)$ efficiently for all internal nodes v of T. Let $deg_k(v)$ be the degree of v in T_k where T_k is the embedded suffix tree of S_k in T. We observe that, for every internal node v in T, $n_k(v) - 1$ equals $\sum\{deg_k(u) - 1 \mid u$ is a descendant of v in T and u is in $T_k\}$. Based on this observation, we define $h(u)$ to be $\sum_{k=1..K,u\in T_k}(deg_k(u) - 1)$. Then $U(v) = \sum_{k:n_k(v)>0}(n_k(v) - 1) = h(v) + \sum\{U(u) \mid u$ is a child of $v\}$.

Note that, for any node v in T, $N(v) = \sum\{N(u) \mid u$ is a child of $v\}$. Therefore, $C(v)$ for all $v \in T$ can be computed as follows using $O(n)$ time.

1: Compute all the embedded suffix trees T_k in T for $k = 1, \ldots, K$ using the method in Section 3.3.6;
2: **for** every internal node v in T visiting in post-order **do**
3: compute $h(v) = \sum_{k=1..K,v\in T_k}(deg_k(v) - 1)$;
4: compute $U(v) = h(v) + \sum\{U(u) \mid u$ is a child of $v\}$;
5: compute $N(v) = \sum\{N(u) \mid u$ is a child of $v\}$;
6: compute $C(v) = N(v) - U(v)$;
7: **end for**

3.4 Construction of a Suffix Tree

Given a string $S = s_1 s_2 \ldots s_n$, where $s_n = \$$. We denote S_i be the i-th suffix of S, that is, $s_i s_{i+1} \ldots s_n$. A straightforward approach to build the suffix tree T for S is as follows. We first initialize a tree T with only a root. Then, the suffixes S_i are inserted into T one by one for $i = n$ downto 1. To insert the suffix S_i into T, we identify the maximal path in T which matches the prefix of S_i. Then, a new leaf edge is created whose label is the remaining part of S_i. It will take $O(n)$ time for adding suffix S_i. Since there are n suffixes, the straightforward algorithm will take $O(n^2)$ time. Figure 3.5 shows an example for constructing the suffix tree for $S = acca\$$ and Figure 3.6 shows an example for constructing the generalized suffix tree for $S = acca\$$ and $S' = c\#$.

The straightforward method for constructing the suffix tree is not efficient.

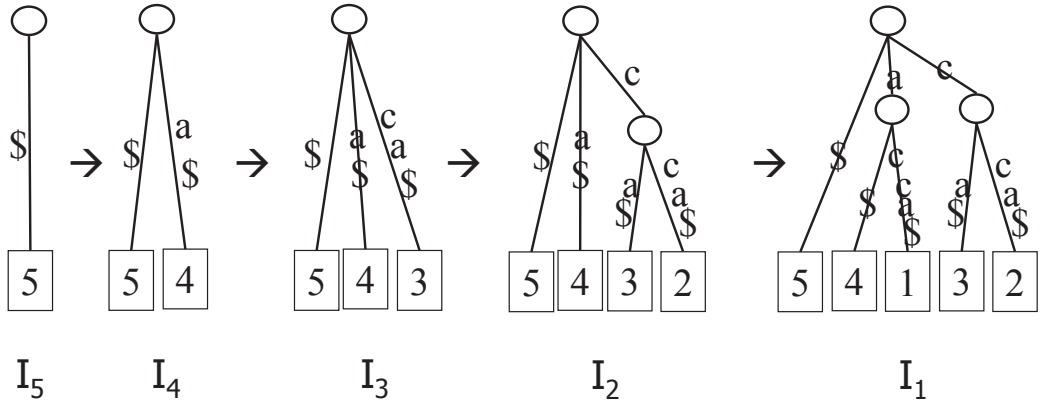


FIGURE 3.5: Constructing the suffix tree for $S = acca\$$.

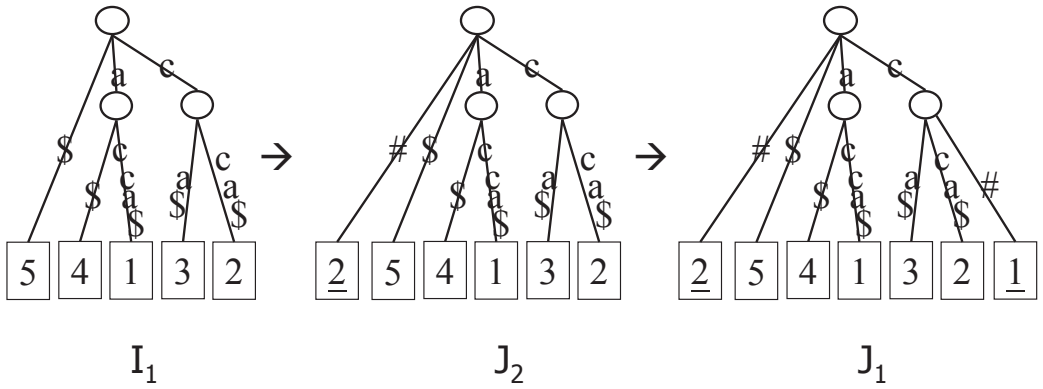


FIGURE 3.6: Constructing the generalized suffix tree for $S = acca\$$ and $S' = c\#$.

Many works tried to address this problem and we can now construct a suffix tree using $O(n)$ time. As early as 1973, Weiner [305] introduced the suffix tree data structure and he gave the first linear time suffix tree construction algorithm when the alphabet size σ is constant. Weiner's algorithm, however, requires a lot of memory. McGreight [211] improved the space complexity to $O(n^2)$ in 1976. Later, in 1995, Ukkonen [295] presented a simplified on-line variant of the algorithm. In 1997, Farach [100] showed that, even for a general alphabet of unbounded size, a suffix tree can be constructed in linear time. The rest of this section will discuss the Farach suffix tree construction algorithm.

Before presenting Farach's algorithm, we first introduce the concept of suffix link. Let T be the suffix tree of S. For any internal node u in T whose path label is ap where a is a character and p is a string, a suffix link is a pointer from u to another node whose path label is p. Based on Lemma 3.1, every internal node has a suffix link. Figure 3.7 shows the suffix links for all internal nodes of the suffix tree of the string $acacag\$$.

LEMMA 3.1

Let T be a suffix tree. If the path label of an internal node v in T is ap, then there must be another internal node w in T whose path label is p. [305]

PROOF Since the path label of v is ap, there exist two suffixes S_i and S_j such that the longest common prefix between them is ap. Then, the longest common prefix between S_{i+1} and S_{j+1} is p. Hence, there exists a node w in T whose path label is p. ∎

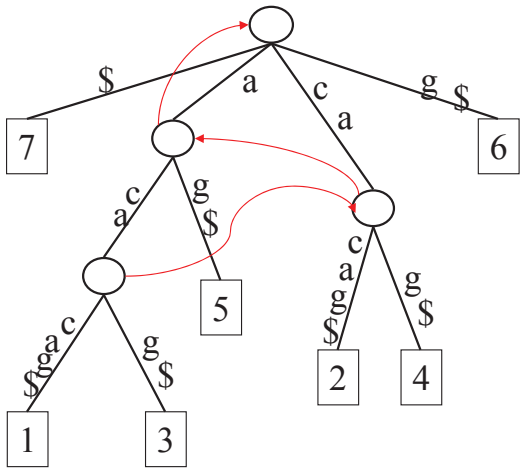


FIGURE 3.7: The arrows indicate the suffix links of the suffix tree for $S = acacag\$$.

Farach's algorithm performs 3 steps to construct the suffix tree T of S. First, the algorithm constructs the suffix tree for all the odd suffixes of S, denoted as odd suffix tree T_o. Second, the algorithm constructs the suffix tree for all the even suffixes of S, denoted as even suffix tree T_e. Last, the algorithm merge T_o and T_e to construct the suffix tree T of S. The algorithm is summarized as follows.

Algorithm **Construct_Suffix_Tree**(S)

1: Rank $s_{2i-1}s_{2i}$ for $i = 1, 2, \ldots, n/2$. Let $S'[1..n/2]$ be a string such that $S'[i]$ = rank of $s_{2i-1}s_{2i}$. By recursively call Construct_Suffix_Tree(S'), we compute the suffix tree T' of S'. Then, we refine T' to generate the odd suffix tree T_o.
2: From T_o, compute the even suffix tree T_e.
3: Merge T_o and T_e to form the suffix tree T of S.

Below, we detail all three steps.

3.4.1 Step 1: Construct the Odd Suffix Tree

Step 1 pairs up the characters of S and forms a string $PS[1 \cdots n/2]$, in which $PS[i] = s_{2i-1}s_{2i}$. By bucket sort on PS (see Exercise 10), the rank of each character pair can be computed. Then a new string $S' = s'_1 \ldots s'_{n/2}$ is generated, where s'_i equals the rank of $PS[i]$ in the sorted list. Now S' is a string of length $n/2$ over integers $[1 \cdots n/2]$. By calling Construct_Suffix_Tree(S'), the suffix tree T' for S' can be constructed recursively.

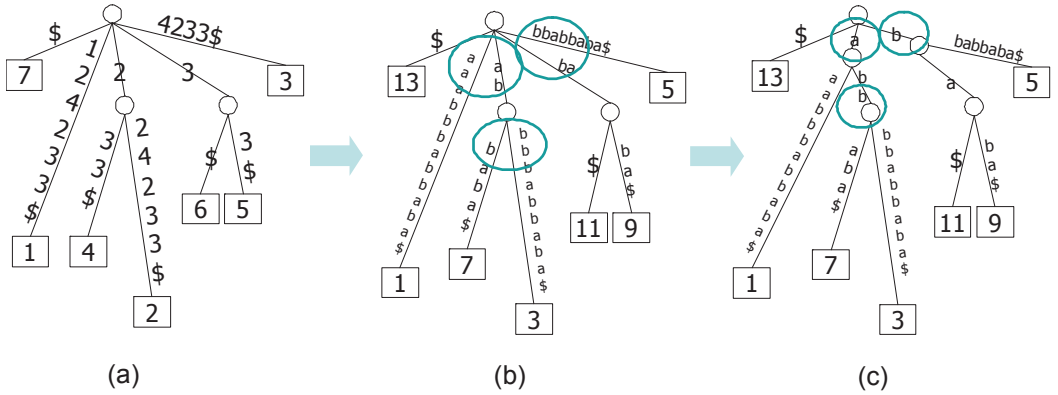


FIGURE 3.8: Given the string $S = aaabbbabbaba\$$, by replacing every character pair by its rank, we obtain $S' = 123233\$$. (a) The suffix tree T' of S'. By (1) replacing every leaf j by $2j-1$ and (2) replacing the ranks on each edge by the corresponding character pairs, we obtain the tree in (b). Note that this is not a suffix tree since some edges attached to the same internal node start with the same character (as shown in the circle). After resolving the problems in the edges, we obtain the odd suffix tree T_o for S in (c).

For example, for the string $S = aaabbbabbaba\$$, its PS is (aa, ab, bb, ab, ba, ba). By bucket sort, these character pairs could be sorted to $aa < ab < ba < bb$ and the rank of each pair is $\{Rank(aa) = 1, Rank(ab) = 2, Rank(ba) = 3, Rank(bb) = 4\}$. Now we can form a new string $S' = 124233\$$ and get the suffix tree of S' as shown in Figure 3.8(a) by recursion.

The last job of Step 1 is to refine T' into the odd suffix tree T_o. To do this, the characters s'_i on every edge of T' should be replaced by the corresponding character pairs and the leaf labels j in T' should be replaced by the leaf labels $2j-1$. For example, after replacement, the suffix tree of S' in Figure 3.8(a) becomes a tree in Figure 3.8(b). However, the tree after replacement may not be a suffix tree. For example, in Figure 3.8(b), the root node has two edges whose labels are $aaabbbabbaba\$$ and ab and both labels start with a. There are two other faults in the same tree. To correct the tree into a suffix tree, for each internal node u, if some of its child edges whose labels start with the same character, say x, a new node v is introduced between u and these

child edges and the edge (u, v) is labeled by x. Because the edges coming from any node are lexicographically sorted, we can identify and correct all those faults by checking the first characters on the adjacent edges. Hence, the correction takes linear time. Figure 3.8(c) shows the odd suffix tree of $S = aaabbbabbaba\$$ generated by Step 1.

For time analysis, the bucket sorting and the refinement of T' take $O(n)$ time in Step 1. Suppose $Time(n)$ is the time to build the suffix tree for a string of length n, then Step 1 takes $Time(n/2) + O(n)$ time in total.

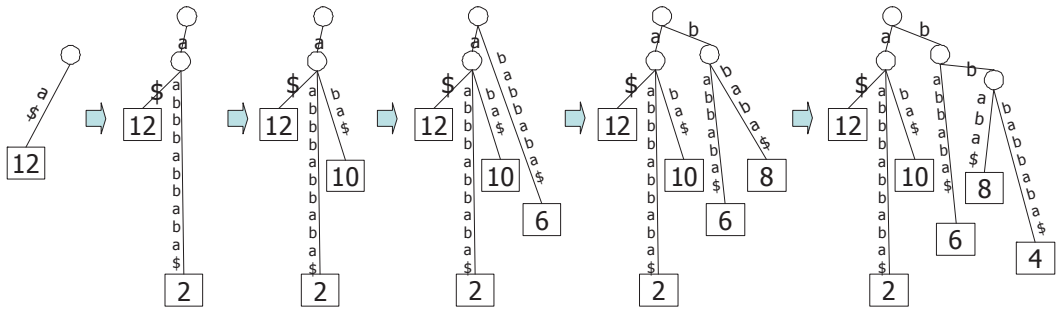


FIGURE 3.9: Process of constructing the even suffix tree T_e of $S = aaabbbabbaba\$$.

3.4.2 Step 2: Construct the Even Suffix Tree

Step 2 constructs the even suffix tree T_e. Interestingly, Farach observed that T_e can be constructed from T_o. The process consists of three substeps: (i) Find the lexicographical ordering of the even suffixes, (ii) for adjacent even suffixes, we compute their longest common prefixes, and (iii) construct T_e using information from (i) and (ii).

For (i), we notice that $S_{2i} = s_{2i}S_{2i+1}$. Given the lexicographical order of all the odd suffixes, we can generate the lexicographical order of all even suffixes by bucket sorting on the pairs (s_{2i}, S_{2i+1}) in linear time. For instance, the even suffixes $S_2, S_4, \cdots, S_{12}$ of $S = aaabbbabbaba\$$ could be represented as $(a, S_3), (b, S_5), (b, S_7), (b, S_9), (a, S_{11}), (a, S_{13})$, respectively. Note that the lex-ordering of the suffixes in T_o is $S_{13} < S_1 < S_7 < S_3 < S_{11} < S_9 < S_5$; so after bucket sorting, the lex-ordering of the pairs is $(a, S_{13}), (a, S_3), (a, S_{11}), (b, S_7), (b, S_9), (b, S_5)$, which can be generated in $O(n)$ time. Hence, the lex-ordering of the leaves of T_e is $S_{12} < S_2 < S_{10} < S_6 < S_8 < S_4$.

For (ii), we compute $LCP(2i, 2j)$ for every adjacent suffixes S_{2i} and S_{2j}. Observe that for any two adjacent suffixes S_{2i} and S_{2j} in T_e, if $S_{2i} = S_{2j}$, then $LCP(2i, 2j) = LCP(2i+1, 2j+1)+1$; otherwise $LCP(2i, 2j) = 0$. Hence, we can calculate the LCP for a pair of adjacent suffixes S_{2i} and S_{2j} in linear time. In our example, the LCP for the adjacent leaves of T_e is $\{LCP(12, 2) = 1,$

$LCP(2,10) = 1$, $LCP(10,6) = 0$, $LCP(6,8) = 1$, $LCP(8,4) = 2\}$.

For (iii), given the order of the leaves and the LCP information, we can construct T_e incrementally from left to right in linear time using an approach similar to Section 3.3.6. Figure 3.9 shows the process of constructing the even suffix tree for S. In summary, Step 2 takes $O(n)$ time to construct the even tree T_e for S.

3.4.3 Step 3: Merge the Odd and the Even Suffix Trees

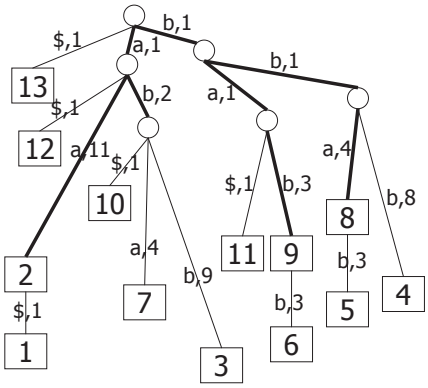


FIGURE 3.10: The over-merged tree of the odd and even suffix trees of $S = aaabbbabbaba\$$. Every edge is labeled by the first character of the edge and the length of the edge. All the thick edges are formed by merging between the odd and even suffixes.

After the first two steps, we get the odd suffix tree T_o and the even suffix tree T_e of S. In the final step, T_o and T_e will be merged to generate the suffix tree T of S.

Consider the uncompressed version of both T_o and T_e so that every edge is labeled by one character (that is, they are treated as suffix tries). We can merge T_o and T_e using depth first search (DFS). Precisely, we start by the two roots of both trees. Then, we simultaneously take the edges in both trees whose labels are the same and recursively merge the two subtrees. If only one tree has an edge labeled a, then we do nothing since there is nothing to merge. Merging trees in this way may take $O(n^2)$ time since there may be $O(n^2)$ characters in the suffix tree.

Merging the uncompacted version of T_o and T_e is time consuming. Farach introduced a method which merges the original T_o and T_e directly in $O(n)$ time. The method is also based on DFS. Moreover, two edges are merged as long as they start with the same character. The merge is ended when the label of one edge is longer than that of the other. Figure 3.10 shows an example of an over-merged tree M, which merges our example odd and even suffix

trees T_o and T_e of $S = aaabbbabbaba\$$. The thick edges in Figure 3.10 are the merged edges and they may be over-merged. What remains is to unmerge these incorrectly merged parts of the tree. Before explaining the unmerging process, let's first introduce two definitions d and L.

For any node v created by merging, its incoming edge may be over-merged. We let $L(v)$ be the correct depth of v.

Since v is a node created by merging, there must exist one even suffix S_{2i} and one odd suffix S_{2j-1} such that $v = LCA(2i, 2j-1)$ in M. The suffix link $d(v)$ is the node $LCA(2i+1, 2j)$ if v is not a root; otherwise $d(v)$ is undefined. The dotted edges in Figure 3.11 show the d relationship of the vertices in M. Since every vertex has at most one out-going dotted edge, the dotted edges form a tree, which is denoted as d tree. By construction, $L(v) = 1 + L(d(v))$ if v is not the root; otherwise $L(v) = 0$. Hence, $L(v)$ equals the depth of v in the d tree.

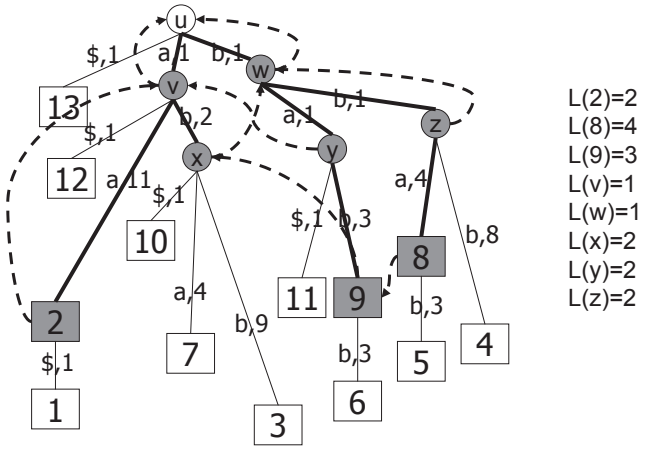


FIGURE 3.11: The dotted edges form the d tree for the over-merged tree M in Figure 3.10. We also show the L values for the nodes on the d tree, which are the depths of the nodes with respect to the d tree.

The depth of every node in the d tree can be determined in linear time by DFS. Then, we identify all the real over-merged nodes. In Figure 3.11, the real over-merged nodes are 2, 8, 9, and x, whose L values are smaller than the depth in M. For any over-merged node v, we adjust its depth to $L(v)$. Then, all the children of v are appended under v' in the lexicographical order. The final tree computed is just the suffix tree T for S. Figure 3.12 shows the suffix tree for $S = aaabbbabbaba\$$.

When these three steps are completed, we get the suffix tree T for S.

Finally, we analyze the time complexity of Farach's algorithm. Let $Time(n)$ be the time required to construct a suffix tree for a sequence of length n. Then, Step 1 takes $Time(n/2) + O(n)$ time. Steps 2 and 3 take $O(n)$ time. In total,

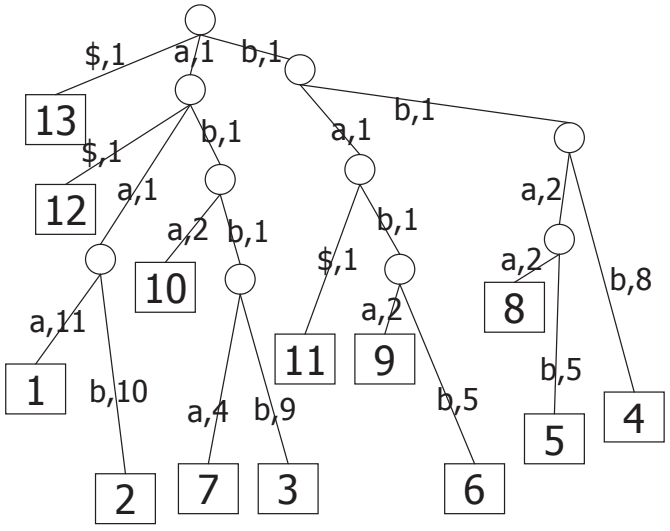


FIGURE 3.12: The suffix tree for $S = aaabbbabbaba\$$.

the running time of the whole algorithm is $Time(n) = Time(n/2) + O(n)$. By solving this equation, we have $Time(n) = O(n)$. In conclusion, given a string $S = s_1 s_2 \cdots s_n$, Farach's algorithm can construct the suffix tree for S using $O(n)$ time and space.

3.5 Suffix Array

Although a suffix tree is a very useful data structure, it has limited usage in practice. This is mainly due to its large space requirement. For a length-n sequence over an alphabet $\mathcal{A}$, the space requirement is $O(n|\mathcal{A}| \log n)$ bits. (Note that the size of $|\mathcal{A}|$ is 4 for DNA and 20 for protein.)

To solve this problem, Manber and Myers [204] proposed a new data structure called a suffix array, in 1993, which has a similar functionality as a suffix tree but only requires $n \log n$ bits space.

Let $S[1..n]$ be a string of length n over an alphabet $\mathcal{A}$. We assume that $S[n] = \$$ is a unique terminator which is alphabetically smaller than all other characters. The suffix array ($SA[1..n]$) stores the suffixes of S in a lexicographically increasing order. Formally, $SA[1..n]$ is an array of integers such that $S[SA[i]..n]$ is lexicographically the i-th smallest suffix of S. For example, consider $S = acacag\$$. Its suffix array is shown in Figure 3.13(b).

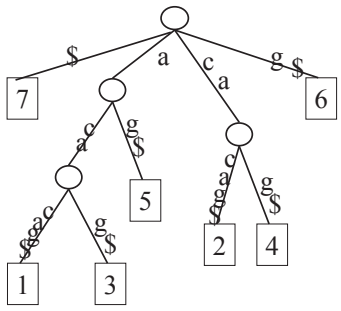
For the space complexity, as each integer in the suffix array is less than n and can be stored using $\log n$ bits, the whole suffix array of size n can be stored in $n \log n$ bits.

Suffix	Position
acacag\$	1
cacag\$	2
acag\$	3
cag\$	4
ag\$	5
g\$	6
\$	7

(a)

i	$SA[i]$	Suffix
1	7	\$
2	1	*acacag*\$
3	3	*acag*\$
4	5	*ag*\$
5	2	*cacag*\$
6	4	*cag*\$
7	6	*g*\$

(b)



(c)

FIGURE 3.13: (a) The set of suffixes of $S = acacag\$$, (b) the corresponding suffix array, and (c) the corresponding suffix tree. Note that the lexicographical order of the leaves in the suffix tree equals the order of the suffixes in the suffix array.

3.5.1 Construction of a Suffix Array

Observe that when the leaves of a suffix tree are traversed in lexicographical depth-first search order, they form the suffix array of that string. This is shown in Figure 3.13(b,c). Thus the suffix array of $S[1..n]$ can be constructed in $O(n)$ time by first constructing the suffix tree T and then the suffix array is generated by traversing T using lexicographical depth-first traversal.

However, this naïve approach requires a large working space, since it needs to build the suffix tree itself which requires $O(n|\mathcal{A}|\log n)$ bits space. Thus this defeats the purpose of using a suffix array.

To date, if we only have $O(n)$ bits of working memory available, the best known technique for constructing a suffix array takes $O(n)$ time. Please refer to [152] for more details.

3.5.2 Exact String Matching Using a Suffix Array

Most applications using a suffix tree can be solved using a suffix array with some overhead. This section demonstrates how to use a suffix array to solve the exact string matching problem. Assume the suffix array of a length-n string S is given. For any query Q of length m, our aim is to check if Q exists in S.

We first state a property. Let Q be a string. Suppose suffix $SA[i]$ and suffix $SA[j]$ are lexicographically the smallest and largest suffixes, respectively, having Q as their prefix. This implies that Q occurs at positions $SA[i], SA[i+1], \ldots, SA[j]$ in S. The interval $[i..j]$ is called the SA range of Q. Notationally, we denote $[i..j]$ as $range(S, Q)$. For example, considering the text $S = acacag\$$, ca occurs in $SA[5]$ and $SA[6]$. In other words, $range(S, ca) = [5..6]$.

The idea is to perform a binary search on the suffix array of S. The algorithm is shown in Figure 3.14. Initially, Q is expected to be in the SA range

$L..R = 1..n$. Let $M = (L + R)/2$. If Q matches $SA[M]$, report Q exists. If Q is smaller than suffix $SA[M]$, Q is in the SA range $L..M$; otherwise, Q is in the SA range $M..R$. Using binary search on the suffix array, we will perform at most $\log n$ comparisons. Each comparison between the query Q and a suffix takes at most $O(m)$ time. Therefore, in the worst case, the algorithm takes $O(m \log n)$ time.

Comparison between the query Q and the suffix $SA[M]$ is time consuming. We have the following observation which reduces the amount of comparisons. Suppose l is the length of the longest common prefix between Q and the suffix $SA[L]$ and r is the length of the longest common prefix between Q and the suffix $SA[R]$. Then, the length of the longest common prefix between Q and the suffix $SA[M]$ is at least $mlr = \min\{l, r\}$.

The above observation allows us to reduce the redundant comparisons and we obtain the algorithm shown in Figure 3.15. Figure 3.16 demonstrates how to find the occurrence of $Q = acag$ in the text $S = acacag\$$ by the algorithm. First, as shown in Figure 3.16(a), we initialize L and R to the smallest index 1 and the biggest index 7, respectively. Also we initialize l as the length of the longest common prefix between Q and suffix $SA[L]$ and r as that between Q and suffix $SA[R]$. Second, as shown in Figure 3.16(b), we compute the mid-point $M = (L + R)/2 = 4$. Note that Q and suffix $SA[M]$ must match at least the first $mlr = \min\{l, r\} = 0$ character. We check the rest of the strings to compute the length of the longest common prefix between Q and $SA[M]$, which equals $m = 1$. Since the $(m + 1)$-th character of suffix $SA[M]$ is g and the $(m + 1)$-th character of Q is c, we have suffix $SA[M] > Q$. Hence, we can halve the range (L, R) by setting $R = M = 4$ and $r = m = 1$. Third, as shown in Figure 3.16(c), we compute $M = (L + R)/2 = 2$. Note that Q and suffix $SA[M]$ must match the first $mlr = \min\{l, r\} = 0$ characters. We check the rest of the strings to compute the length of the longest common prefix between Q and $SA[M]$, which equals $m = 3$. Since the $(m + 1)$-th character of suffix $SA[M]$ is c and the $(m+1)$-th character of Q is g, we have suffix $SA[M] < Q$. Hence, we can halve the range (L, R) by setting $L = M = 2$ and $l = m = 3$. Fourth, as shown in Figure 3.16(d), we compute $M = (L + R)/2 = 3$. Note that Q and suffix $SA[M]$ must match at least the first $mlr = \min\{l, r\} = 1$ characters. We check the rest of the strings to compute the length of the longest common prefix between Q and $SA[M]$, which equals $m = 4$. Hence, Q is found at $SA[2] = 3$.

Using binary search on a suffix array of size n, we will perform at most $\log n$ comparisons. Each comparison takes at most $O(m)$ time. Therefore, in the worst case, the algorithm takes $O(m \log n + occ)$ time. Moreover, since the number of comparisons is reduced, the practice running time is reported to be $O(m + \log n + occ)$ for most of the cases [204]. When space is limited, solving the exact string matching problem using a suffix array is a good alternative to a suffix tree.

Furthermore, if we can afford to store the information $\{LCP(1, 2), LCP(2, 3), \ldots, LCP(n - 1, n)\}$ using an additional $nlogn$-bit space, the exact string

SA_binary_search(Q)

```
1: Let L = 1 and R = n;
2: while L ≤ R do
3:    Let M = (L + R)/2.
4:    Find the length m of the longest common prefix of Q and suffix SA[M];
5:    if m = |Q| then
6:       Report suffixes SA[M] contain the occurrence of Q;
7:    else if suffix SA[M] > Q then
8:       set R = M;
9:    else
10:      set L = M;
11:   end if
12: end while
13: if L > R then
14:   Report Q does not exist in S;
15: end if
```

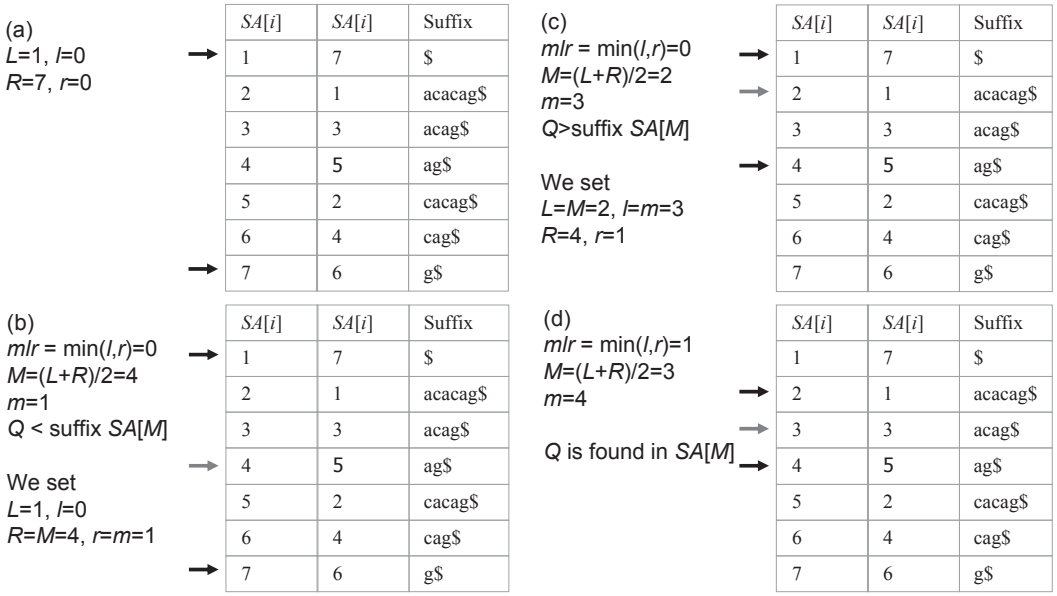
FIGURE 3.14: Checking the existence of a query Q through binary search on the suffix array of S.

SA_binary_search_with_lcp(Q)

```
1: Let L = 1 and R = n;
2: Let l be the length of the longest common prefix of Q and suffix_{SA[1]};
3: Let r be the length of the longest common prefix of Q and suffix_{SA[n]};
4: while L ≤ R do
5:    Let M = (L + R)/2.
6:    mlr = min(l, r);
7:    Starting from the position mlr of Q, find the length m of the longest
      common prefix of Q and suffix SA[M];
8:    if m = |Q| then
9:       Report suffixes SA[M] contain the occurrence of Q;
10:   else if suffix_{SA[M]} > Q then
11:      set R = M and r = m;
12:   else
13:      set L = M and l = m;
14:   end if
15: end while
16: if L > R then
17:   Report Q does not exist in S;
18: end if
```

FIGURE 3.15: Checking the existence of a query Q through binary search on the suffix array of S. In this algorithm, we speed up the computation using the longest common prefix.

.



$SA[i]$	$SA[i]$	Suffix
1	7	$
2	1	acacag$
3	3	acag$
4	5	ag$
5	2	cacag$
6	4	cag$
7	6	g$

FIGURE 3.16: This example demonstrates how to check the existence of $Q = acag$ in the text $S = acacag\$$ by performing binary search on the suffix array of S.

matching of a length-n pattern can be found in $O(m + \log n + occ)$ worst case time (see Exercise 5).

3.6 FM-Index

Although the space requirement of a suffix array is less than that of a suffix tree, it is still unacceptable in many real life situations. For example, to store the human genome, with its 3 billion base pairs, it takes up to 40 gigabytes using a suffix tree and 13 gigabytes using a suffix array. This is clearly beyond the capacity of a normal personal computer. A more space-efficient solution is required for practical solutions. Two such alternatives were proposed. They are the compressed suffix array by Grossi and Vitter [123] and the FM-index by Ferragine and Manzini [106]. Both data structures require only $O(n)$ bits where n is the length of the string. For instance, we can store the FM-index of the whole human genome using less than 1.5 gigabytes. We will introduce the FM-index data structure below.

3.6.1 Definition

The FM-index is a combination of the Burrows-Wheeler (BW) text [45] and an auxiliary data structure. Let $S[1..n]$ be a string of length n, and $SA[1..n]$ be its suffix array. The FM-index for S stores the following three data structures:

1. The BW text is defined as a string of characters $BW[1..n]$ where

$$BW[i] = \begin{cases} S[SA[i]-1] & \text{if } SA[i] \neq 1 \\ S[n] & \text{if } SA[i] = 1 \end{cases}$$

   In other words, the BW text is an array of preceding characters of the sorted suffixes. For example, for $S = acacag\$$, its BW text is $g\$ccaaa$ as shown in Figure 3.17.

2. For every $x \in \mathcal{A}$, $C[x]$ stores the total number of occurrences of characters which are lexicographically less than x. For example, for $S = acacag\$$, we have $C[a] = 1$, $C[c] = 4$, $C[g] = 6$, and $C[t] = 7$.

3. A data structure which supports $O(1)$ computation of $occ(x, i)$, where $occ(x, i)$ is the number of occurrences of $x \in \mathcal{A}$ in $BW[1..i]$. For example, for the BW text $g\$ccaaa$, we have $occ(a, 4) = 1$ and $occ(c, 3) = 2$. Section 3.6.2 will detail a succinct way to store the occ data structure using $O(\frac{n \log\log n}{\log n})$ bits.

We will now analyze the space complexity of the FM-index. In the case of the DNA sequence, structure one can be stored in $2n$ bits because we are storing n characters, each character having four possible choices ($\log 4 = 2$). Structure 2 can be stored in $O(\log n)$ bits and structure 3 can be stored in $O(\frac{n \log\log n}{\log n})$ bits. Therefore, in total the size of the FM-index is $O(n)$ bits.

i	$SA[i]$	Suffix	$S[SA[i]-1]$
1	7	\$	g
2	1	$acacag\$$	\$
3	3	$acag\$$	c
4	5	$ag\$$	c
5	2	$cacag\$$	a
6	4	$cag\$$	a
7	6	$g\$$	a

FIGURE 3.17: Consider the string $S = acacag\$$. The second column shows the suffix array $7, 1, 3, 5, 2, 4, 6$. The last column is the BW text, which is the list of preceding characters of the sorted suffixes $g\$ccaaa$.

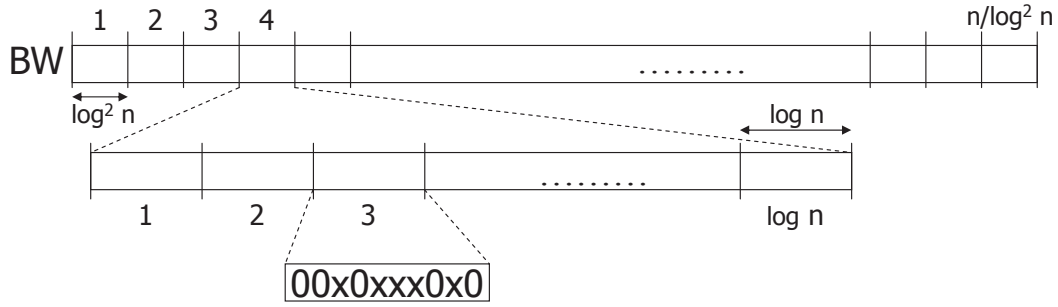


FIGURE 3.18: The data structure for answering the $occ(x, i)$ query.

3.6.2 The *occ* Data Structure

To describe the *occ* data structure, we need to conceptually divide the text $BW[1..n]$ into $\frac{n}{\log^2 n}$ buckets, each of size $log^2 n$. Each bucket is further subdivided into $\log n$ sub-buckets of size $\log n$. Figure 3.18 demonstrates how the BW text is partitioned into buckets and sub-buckets.

The *occ* data structure stores an integer for each bucket and each sub-bucket. For each bucket $i = 1, \cdots, \frac{n}{\log^2 n}$, we store $P[i]$ = number of x's in $BW[1..i \log^2 n]$. For each sub-bucket j of the bucket i, we store $Q[i][j]$ = number of x's in the first j sub-buckets of the bucket i. In addition, the *occ* data structure also requires a lookup table $rank(b, k)$ for every string b of length $\log n/2$ and $1 \leq k \leq \log n/2$. Each entry $rank(b, k)$ stores the number of x's occurring in the first k characters of b.

In total, the *occ* data structure needs to store two arrays and one lookup table. The array $P[1..n/\log^2 n]$ stores $\frac{n}{\log^2 n}$'s $(\log n)$-bit integers, which uses $O(\frac{n}{\log n})$-bit space. The array $Q[1..n/\log^2 n][1..\log n]$ stores $\frac{n}{\log n}$'s $(\log \log n)$-bit integers, which uses $O(\frac{n \log \log n}{\log n})$-bit space. The table $rank(b, k)$ has $2^{\frac{1}{2}\log n}\frac{\log n}{2} = \sqrt{n}\log n/2$ entries of $\log \log n$-bit integers, which uses $o(n)$-bit space. Thus, the *occ* data structure uses $O(\frac{n \log \log n}{\log n})$ bits.

How can we compute $occ(x, i)$ using the *occ* data structure? Observe that, for any i,

$$i = i_1 \log^2 n + i_2 \log n + i_3$$

where $i_1 = \lfloor \frac{i}{\log^2 n} \rfloor$, $i_2 = \lfloor \frac{i \mod \log^2 n}{\log n} \rfloor$, and $i_3 = i \mod \log n$.

Based on the *occ* data structure, the number of x's in $BW[1..i - i_3]$ is $P[i_1]+Q[i_1+1][i_2]$. The number of x's in $BW[i-i_3+1..i]$ equals $rank(BW[i-i_3+1..i], x_3)$ if $i_3 < \log n/2$; otherwise, it equals $rank(BW[i - i_3 + 1..i - i_3 + \log n/2], \log n/2) + rank(BW[i - i_3 + \log n/2 + 1..i], i_3 - \log n/2)$. Hence, we can compute $occ(x, i)$ in $O(1)$ time. For example, suppose $\log n = 10$. As shown in Figure 3.18, to compute $occ(x, 327)$, the result is $P[3] + Q[4][2] + rank(00x0x, 5) + rank(xx0x0, 2)$.

Algorithm BW_search($Q[1..m]$)

1: $x = Q[m]$; $st = C[x] + 1$; $ed = C[x + 1]$;
2: $i = m - 1$;
3: **while** $st \leq ed$ and $i \geq 1$ **do**
4: $x = Q[i]$;
5: $st = C[x] + occ(x, st - 1) + 1$;
6: $ed = C[x] + occ(x, ed)$;
7: $i = i - 1$;
8: **end while**
9: if $st > ed$, then pattern not found else report $[st..ed]$.

FIGURE 3.19: Given the FM-index of S, the backward search algorithm finds $range(S, Q)$.

3.6.3 Exact String Matching Using the FM-Index

Here, we revisit the exact string matching again and we describe the application of the FM-index to the exact string matching problem. Consider the FM-index of a string $S[1..n]$. For any query string Q, our task is to determine if Q exists in the string S. This section gives the backward search algorithm to perform the task. First, we state a simple property of the FM-index.

LEMMA 3.2

The number of suffixes which are lexicographically smaller than or equal to $xT[SA[i]..n]$ equals $C[x] + occ(x, i)$, where $x \in \mathcal{A}$.

PROOF A suffix $S[j..n]$ which is smaller than $xS[SA[i]..n]$ has two types: (1) $S[j] < x$ and (2) $S[j] = x$ and $S[j + 1..n] < S[SA[i]..n]$. The number of suffixes of type 1 equals $C[x]$. The number of suffixes of type 2 equals $occ(x, i)$. The lemma follows. ∎

As an example, for the FM-index of $S = acacag\$$, the number of suffixes smaller than $cS[SA[5]..n] = ccag\$$ is equal to $C[c] + occ(c, 5) = 4 + 2 = 6$. Based on Lemma 3.2, we can derive the key lemma for backward search.

LEMMA 3.3

Consider a string $S[1..n]$ and a query Q. Suppose $range(S, Q)$ is $[st..ed]$. Then, we have $range(S, xQ) = [p..q]$ where $p = C[x] + occ(x, st - 1) + 1$ and $q = C[x] + occ(x, ed)$.

PROOF By definition, $p-1$ equals the number of suffixes strictly smaller than xQ. By Lemma 3.2, we have $p - 1 = C[x] + occ(x, st - 1)$. For q, it

equals the number of suffixes smaller than or equal to $xS[SA[st-1]..n]$. By Lemma 3.2, we have $q = C[x] + occ(x, ed)$. □

Now, we describe the backward search algorithm. It first determines $range(T, Q[m])$ which is $[C[x]+1..C[x+1]]$ where $x = Q[m]$. Then, we compute $range(T, Q[i..m])$ through applying Lemma 3.3 iteratively for $i = m-1$ to 1. Figure 3.19 gives the detail of the backward search algorithm. Figure 3.20 gives an example to demonstrate the execution of the algorithm.

We now analyze the time complexity of backward search. To find a pattern $Q[1..m]$, we need to process the characters in Q one by one iteratively. Each iteration takes $O(1)$ time. Therefore, the whole backward search algorithm takes $O(m)$ time.

(a) First iteration (initial values),
$Q[3..3] = a$
$range(S, Q[3..3]) = [sp..ep]$
$sp = C[a] + 1 = 1 + 1 = 2$
$ep = C[c] = 4$

	$SA[i]$	Suffix	$BW[i]$
	7	\$	g
→	1	$acacag$\$	\$
	3	$acag$\$	c
→	5	ag\$	c
	2	$cacag$\$	a
	4	cag\$	a
	6	g\$	a

(b) Second iteration,
$Q[2..3] = ca$
$range(S, Q[2..3]) = [sp'..ep']$
$sp' = C[c] + occ(c, sp-1) + 1 = 4 + 0 + 1 = 5$
$ep' = C[c] + occ(c, ep) = 4 + 2 = 6$

	$SA[i]$	Suffix	$BW[i]$
	7	\$	g
	1	$acacag$\$	\$
	3	$acag$\$	c
	5	ag\$	c
→	2	$cacag$\$	a
→	4	cag\$	a
	6	g\$	a

(c) Third iteration,
$Q[1..3] = aca$
$range(S, Q[1..3]) = [sp''..ep'']$
$sp'' = C[a] + occ(a, sp'-1) + 1 = 1 + 0 + 1 = 2$
$ep'' = C[a] + occ(a, ep') = 1 + 2 = 3$

	$SA[i]$	Suffix	$BW[i]$
	7	\$	g
→	1	$acacag$\$	\$
→	3	$acag$\$	c
	5	ag\$	c
	2	$cacag$\$	a
	4	cag\$	a
	6	g\$	a

FIGURE 3.20: Given the FM-index for the text $S = acacag\$$. This figure shows the three iterations for searching pattern $Q = aca$ using backward search. (a) $range(S, a) = [2..4]$, (b) $range(S, ca) = [5..6]$, and (c) $range(S, aca) = [2..3]$.

i	$SA[i]$	$SA^{-1}[i]$	Suffix
1	7	2	\$
2	1	5	$acacag\$$
3	3	3	$cacag\$$
4	5	6	$acag\$$
5	2	4	$cag\$$
6	4	7	$ag\$$
7	6	1	$g\$$

FIGURE 3.21: The suffix array and inverse suffix array of $S = acacag\$$.

3.7 Approximate Searching Problem

Although the suffix tree, the suffix array, and their variants are very useful, they are usually used to solve problems related to exact string matching. In bioinformatics, many problems involve approximate matching. This section describes how to apply suffix tree related data structures to solve an approximate matching problem. In particular, we describe the 1-mismatch problem.

Given a pattern $Q[1..m]$, the 1-mismatch problem finds all occurrences of Q in the text $S[1..n]$ that has hamming distance at most 1. For example, for $Q = acgt$ and $S = a\underline{acgt}ggcca\underline{acttg}ga$, the underlined substrings of S are the 1-mismatch occurrences of Q.

A naïve solution for solving the 1-mismatch problem is to construct the suffix tree for T and find the occurrences of every 1-mismatch pattern of Q in the suffix tree. Note that the number of possible 1-mismatch patterns of Q is $(|\mathcal{A}| - 1)m$ and it takes $O(m)$ time to find occurrences of each 1-mismatch pattern. Hence the naïve algorithm takes $O(|\mathcal{A}|m^2 + occ)$ time in total, where occ is the total number of occurrences.

Many sophisticated solutions for solving the indexed 1-mismatch problem have been proposed in the literature. Below, we present the algorithm by Trinh et al. [158].

Consider a text $S[1..n]$. The data structures used in [158] are the suffix array (SA) of S and the inverse suffix array (SA^{-1}) of S, where $SA[SA^{-1}[i]] = i$. SA can be constructed as shown in Section 3.5.1. Given SA, obtaining SA^{-1} in $O(n)$ time is trivial. Both SA and SA^{-1} can be stored in $O(n \log n)$ bits space. See Figure 3.21 for an example of the suffix array and the inverse suffix array of $S = acacag\$$.

Trinh et al. showed that, given the SA and SA^{-1}, we can compute all 1-mismatch occurrences of a query $Q[1..m]$ in $S[1..n]$ using $O(|\mathcal{A}|m \log n + occ)$ time. Precisely, the algorithm tries to compute the SA range $range(S, Q) = [st..ed]$.

The key trick used in the algorithm is that, for any strings P_1 and P_2,

we can compute $range(S, P_1P_2)$ in $O(\log n)$ time given $range(S, P_1)$ and $range(S, P_2)$. The lemma below shows the correctness of the trick.

LEMMA 3.4
Suppose $[st_1..ed_1] = range(S, P_1)$ *and* $[st_2..ed_2] = range(S, P_2)$ *then we can compute* $[st..ed] = range(S, P_1P_2)$ *in* $O(\log n)$ *time.*

PROOF Let the length of P_1 be k. Observe that $S[SA[st_1]..n], S[SA[st_1+1]..n], \ldots, S[SA[ed_1]..n]$ are lexicographically increasing. Since they share the common prefix P_1, $S[SA[st_1]+k..n], S[SA[st_1+1]+k..n], \ldots, S[SA[ed_1]+k..n]$ are also lexicographically increasing. Thus, we have

$$SA^{-1}[SA[st_1] + k] < SA^{-1}[SA[st_1 + 1] + k] < \ldots < SA^{-1}[SA[ed_1] + k]$$

Therefore, to find st and ed, we need to find

- the smallest st such that $st_2 < SA^{-1}[SA[st] + k] < ed_2$ and
- the largest ed such that $st_2 < SA^{-1}[SA[ed] + k] < ed_2$

This can be done using binary search. □

Suppose we want to find the occurrence of the pattern formed by replacing the character at position j of the query pattern $Q[1..m]$ by x, we can compute $range(S, Q[1..j-1]xQ[j+1..m])$ from $range(S, Q[1..j-1])$, $range(S, x)$, and $range(S, Q[j+1..m])$ by applying Lemma 3.4 twice. Applying this trick on all m positions, we can identify all 1-mismatch occurrences of Q. The detail of the algorithm is shown in Figure 3.22.

Below, we analyze the running time of Trinh et al.'s algorithm. Steps 1 to 3 compute $range(S, x)$ for all $x \in \mathcal{A}$, which takes $O(|\mathcal{A}|)$ time. Steps 4 to 6 compute $range(S, Q[j..m])$ for $1 \leq j \leq m$, which takes $O(m \log n)$ time. Steps 7 to 9 compute $range(S, Q[1..j])$ for $1 \leq j \leq m$, which takes $O(m \log n)$ time. For Steps 11 to 18, we enumerate all possible $(|\mathcal{A}| - 1)m$ mismatches, and for each mismatch it takes $O(\log n)$ time to find the corresponding 1-mismatch SA range. Reporting all occurrences takes occ time. Hence the time complexity of the algorithm is $O(|\mathcal{A}|m \log n + occ)$. Note that this algorithm can be generalized to handle k mismatches in $O(|\mathcal{A}|^k m^k \log n + occ)$ time.

3.8 Exercises

1. Given three DNA sequences S_1, S_2, and S_3 of total length n.

Algorithm 1-**mismatch_search**($Q[1..m]$)

1: **for** each character $x \in \mathcal{A}$ **do**
2: Set $range(S, x) = [C[x] + 1..C[x + 1]]$;
3: **end for**
4: **for** $j = m$ to 1 **do**
5: Compute $range(S, Q[j..m])$ from $range(S, Q[j])$ and $range(S, Q[j + 1..m])$ using Lemma 3.4;
6: **end for**
7: **for** $j = 1$ to m **do**
8: Compute $range(S, Q[1..j])$ from $range(S, Q[1..j - 1])$ and $range(S, Q[j])$ using Lemma 3.4;
9: **end for**
10: Report all occurrences in $range(S, Q[1..m])$;
11: **for** $j = 1$ to m **do**
12: Let $P_1 = Q[1..j - 1]$ and $P_2 = Q[j + 1..m]$;
13: **for** each character $x \in \mathcal{A} - \{Q[j]\}$ **do**
14: By Lemma 3.4, find $range(S, P_1x)$;
15: Compute $range(S, P_1x)$ from $range(S, P_1)$ and $range(S, x)$ using Lemma 3.4;
16: Compute $range(S, P_1xP_2)$ from $range(S, P_1x)$ and $range(S, P_2)$ using Lemma 3.4;
17: Report all occurrences in $range(S, P_1xP_2)$;
18: **end for**
19: **end for**

FIGURE 3.22: 1-mismatch search.

(a) Suppose $S_1 = acgatca$, $S_2 = gattact$, $S_3 = aagatgt$. What is the longest common substring of S_1, S_2, and S_3?

(b) Describe an efficient algorithm that computes the length of the longest common substring. What is the time complexity?

(c) It is possible that the longest common substring for S_1, S_2, and S_3 is not unique. Can you describe an efficient algorithm to report the number of possible longest common substrings of S_1, S_2, and S_3? What is the time complexity?

2. Please give the generalized suffix tree for $S_1 = ACGT\$$ and $S_2 = TGCA\#$.

3. Given a DNA sequence S and a pattern P, can you describe an $O(|P|^2 + |S|)$ time algorithm to find all occurrences of P in S with hamming distance ≤ 1?

4. Consider the string $S = ACGTACGT\$$.

(a) What is the suffix array for S?

(b) Report the values of (1) $LCP(k, k+1)$ for $k = 1, 2, \ldots, 8$ and (2) $LCP(k, k+4)$ for $k = 1, 2, 3, 4$.

5. Consider a string S which ends at \$. Suppose we are given the suffix array $SA[1..|S|]$ of S and all $LCP(i, j)$ values. Can you propose an algorithm for searching pattern P which takes $O(|P| + log|S|)$ time?

6. Given a DNA sequence S of length n.

(a) Given another DNA sequence W of length smaller than n, describe an efficient algorithm that reports the number of occurrences of W in S. What is the time complexity?

(b) Describe an efficient algorithm that computes the length of the longest substring which appears exactly k times in S. What is the time complexity?

(c) Consider a DNA sequence X= "*acacagactacac*". What is the number of occurrences of "*aca*" in X? What is the length of the longest substring which appears exactly 3 times in X?

7. Given a string $S = S[1..n]$ and a number k, we want to find the smallest substring of S that occurs in S exactly k times, if it exists. Show how to solve this problem in $O(n)$ time.

8. For the suffix tree for a DNA sequence S and a query Q of length m, describe an algorithm to determine whether there exists a substring x of S such that the hamming distance between Q and x is smaller than or equal to 1. What is the time complexity?

9. For the suffix tree for S, describe a linear time algorithm for searching maximal complementary palindromes.

10. Consider a set Z of n pairs $\{(a_i, b_i) \mid i = 1, 2, \ldots, n\}$, where each symbol a_i or b_i can be represented as a positive integer smaller than or equal to n. Can you describe an $O(n)$-time algorithm to sort the pairs into a sequence $(a_{(1)}, b_{(1)}), \ldots, (a_{(n)}, b_{(n)})$?

11. Given only the FM-index without the text, can you recover the text from the FM-index? If yes, can you describe the detail of the algorithm? What is the time complexity?

12. Given the FM-index, can you generate the suffix array in $O(n)$ time using $O(n)$ bits working space? (For counting the working space, we do not include the space for outputting the suffix array. Also, we are allowed to write the output once. Based on this model, the output can be written directly to the secondary storage, without occupying the main memory.)

13. Given a set of k strings of total length n, give an $O(n)$-time algorithm which computes the longest common substring of all k sequences.

14. Given a set of k strings, we would like to compute the longest common substring of each of the $\binom{k}{2}$ pairs of strings.

    (a) Assume each string is of length n. Show how to find all the longest common substrings in $O(k^2 n)$ time.

    (b) Assume the string lengths are different but sum to m. Show how to find all the longest common substrings in $O(km)$ time.

15. Given a text of length n with constant alphabet, propose an $O(n \log n)$-bit space index and a query algorithm such that, for any pattern P of length m, the 2-mismatch query can be answered in $O(m^2 \log n + occ)$ time.

16. Let T be a DNA sequence of length n. Suppose we preprocess T and get its suffix array SA, its inverse suffix array SA^{-1}, and its FM-index. Consider a pattern P of length m whose alphabet is $\{a, c, g, t, *\}$, where $*$ is a wild card (which represents any DNA base). Assuming P contains x wild cards, can you propose an $O(m + 4^x \log n + occ)$-time algorithm to find all the occurrences of P? (occ is the number of occurrences of P in T.)

17. Consider two DNA sequences S and T of total length n. Describe an $O(n)$ time algorithm which detects whether there exists a substring T' of T such that the score of the global alignment between S and T' is bigger than or equal to -1. (Suppose the score for a match is 0 and the score for insert, delete, or mismatch is -1.)

18. For a sequence S, give an efficient algorithm to find the length of the longest pattern which has at least two non-overlapping occurrences in S. What is the running time?

19. Prove or disprove each of the following.

    (a) Let T be a suffix tree for some string. Let the string α be the label of an edge in T and β be a proper prefix of α. It is not possible to have an internal node in T with path label β.

    (b) Let u be an internal node with path label $xy\alpha$ (where x and y are single characters and α is a string) in a suffix tree. An internal node with path label α always exists.

20. Given a text $T = acgtcga\$$. Can you create a suffix array for T? Can you create $BW[1..8]$? Can you create $C[a]$, $C[c]$, $C[g]$, and $C[t]$? Also, can you demonstrate the steps for finding the pattern $P = cg$ using backward search?