Suffix Arrays

Justin Zhang

24 May 2017

1 Introduction

A *suffix array* is a type of data structure that stores the sorted suffixes of a string. Suffix arrays are used for a wide range of string problems; three well-known ones include pattern searching, finding the longest common substring, and finding the longest palindromic substring. A suffix array can solve these in $O(m)$, $O(n+m)$, and $O(n)$, respectively, which improves the $O(n+m)$ runtime of the Boyer-Moore and the KMP algorithms for pattern searching (given that the suffix array is precomputed) and the $O(nm)$ dynamic programming algorithm for longest common substring.

Construction of a suffix array can be done in $O(n)$ time, but in programming contests, the much simpler $O(n\log^2 n)$ solution will often suffice. Suffix arrays take $O(n)$ memory.

To understand the motivation for suffix arrays, we first look at suffix trees.

2 Suffix Trees

A *trie* is a tree which stores a set of strings. Each node has a number of children up to the size of the alphabet, and each letter of a string is represented as a child of the previous letter's node. A *suffix trie* is a trie that stores the set of suffixes of a string; a *suffix tree* is a compressed version of a *suffix trie* that eliminates chains that only contain nodes with out-degree 1, which saves memory and allows for faster construction.

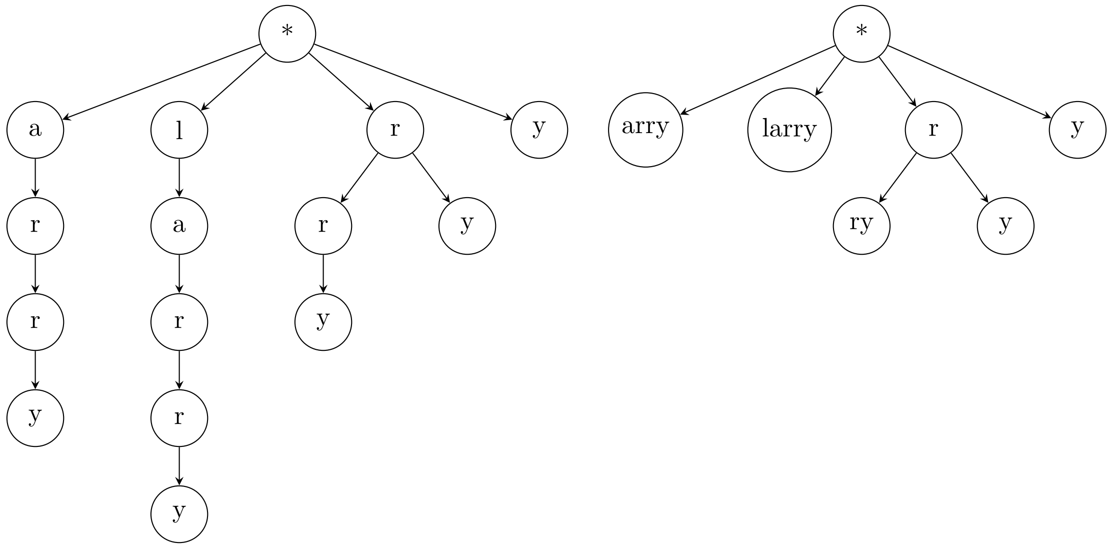

Figure 1: A suffix trie and suffix tree, respectively, for the string "larry"

We can do a number of operations with a suffix tree easily. Given a suffix tree of a string S:

- Check if a given string T is a substring of S in $O(|T|)$ time.

- Find the n-th lexicographically smallest suffix by finding the n-th leaf.

With more involved algorithms, we can also use a suffix tree to accomplish the tasks listed in the introduction. However, there are two major downsides to suffix trees:

- They have a large constant factor in time and memory; jump pointers have high overhead.
- They are difficult to construct efficiently; the $O(n)$ construction algorithm is rather involved.

This motivates us to look for an alternate data structure.

3 Suffix Arrays

A suffix array can be obtained by performing a DFS on a suffix tree. For our example in Figure 1, this would result in [`arry`, `larry`, `rry`, `ry`, `y`]. To save space, we can instead represent the suffixes by their starting index number: $[1, 0, 2, 3, 4]$.

It is readily apparent that we can use this data structure to find the n-th smallest suffix in $O(1)$ time. We can also use this along with its corresponding LCP array to do text searching in $O(m)$ time.

Note that a prefix of any suffix in the array is a substring of the original string.

3.1 LCP

The LCP (longest common prefix) array computes the longest common prefix of adjacent elements (i and $i-1$) in the suffix array. For our example in Figure 1, our LCP array would be $[-, 0, 0, 1, 0]$. Element 3 has a value of 1 since `rry` and `ry` share only `r` as their longest common prefix.

This LCP array can be used alongside the suffix array to easily simulate most operations of a suffix tree.

3.2 Construction

An $O(n \log^2 n)$ suffix array construction algorithm and an $O(n \log n)$ LCP construction algorithm for the string S is as follows.

Algorithm 1 Suffix Array Construction

```
function COMPARE(int $i$, int $j$)
    $lo \leftarrow 0$
    $hi \leftarrow$ MIN($|S| - i$, $|S| - j$)
    while $lo \leq hi$ do
        $mid \leftarrow (lo + hi)/2$
        if HASH($S[i..i + mid]$) = HASH($S[j..j + mid]$) then
            $lo \leftarrow mid + 1$
        else
            $hi \leftarrow mid - 1$
        end if
    end while
    if $i + lo = |S|$ then
        return True
    else if $j + lo = |S|$ then
        return False
    end if
    return $S[i + lo] < S[j + lo]$
end function
function SUFFIXARRAY
    $SA \leftarrow$ SORT($[0, 1, 2, ..., |S| - 1]$, COMPARE)
    return $SA$
end function
```

Algorithm 2 LCP Construction

```
function firstDifference(int $i$, int $j$)
    $lo \leftarrow 0$
    $hi \leftarrow$ min($|S| - i$, $|S| - j$)
    while $lo \leq hi$ do
        $mid \leftarrow (lo + hi)/2$
        if hash($S[i..i + mid]$) = hash($S[j..j + mid]$) then
            $lo \leftarrow mid + 1$
        else
            $hi \leftarrow mid - 1$
        end if
    end while
    return $lo$
end function
function LCP
    $SA \leftarrow$ SuffixArray()
    $lcp \leftarrow [0, 0, ..., 0]$
    for $i = 1..|SA|$ do
        $lcp[i] \leftarrow$ firstDifference($SA[i]$, $SA[i - 1]$)
    end for
    return $lcp$
end function
```

The SuffixArray method relies on an $O(n \log n)$ sort. Both SuffixArray and LCP use a rolling hash (which gets substring hashes in $O(1)$) to optimize comparisons between strings. As mentioned above, faster algorithms ($O(n \log n)$ and $O(n)$ for the suffix array and $O(n)$ for the LCP array) are possible, but not usually necessary for competitive programming.

4 Applications

We use the notation LCP[i] to denote the i-th entry of the LCP array and SA[i] to denote the suffix represented by the suffix array at index i.

4.1 Pattern Matching

Here we describe an $O(|m| + \log |n|)$ (per query) algorithm for string searching, where n is the text and m is the pattern to search for. An $O(|m|)$ algorithm is also possible, but is more involved, and is left up to the reader. In practice, $\log n$ will be within one order of magnitude of m or smaller given a reasonably sized text.

We first compute the suffix array of n. We can then search for m in $O(|m| \log |n|)$ easily by binary searching for the suffix that contains m. Using a rolling hash, we can speed this up to $O(|m| + \log |n| \log |m|)$ (the m factor comes from computing the hash itself, and the $\log |n| \log |m|$ is from doing nested binary searches – one for the suffix that contains m and the other for comparing m to a given suffix).

We define the *middle value* to be the suffix at the middle of a given search range that may occur during the binary search. For instance, the first middle value is the middle suffix in the suffix array, the next is either at the first or third quarter, and so on.

To speed up our algorithm further, we notice that precomputing the LCP for each successive middle value in the binary search can speed up the runtime. Namely, given a middle value M, let's assume that the next middle value, M', is to the right (i.e. m is greater than M). We know the value of $k \leftarrow$lcp(M, m) (the number of characters that matched the pattern to search for). We have three cases:

- lcp(M, M') $< k$: this means that m matches M more than it matches M'. The $(k + 1)$-th character of M' must be greater than the $(k + 1)$-th character of m, and we continue the binary search in the left half.

- lcp(M, M') $> k$: this means that the $(k+1)$-th character of M' must still be less than the $(k+1)$-th character of m, and we continue the binary search in the right half.
- lcp(M, M') $= k$: we don't know whether m is less than or greater than M' since the LCP is still the same. We compare m to M' starting from the $(k+1)$-th character, and decide whether to go left or right from there.

The case in which M' is to the left of M is analogous. However, this raises the question: how do we precompute these successive LCP values?

We observe that in doing the binary search for the suffix that contains m, we always compare m in a consistent way: first to the middle suffix, then either the suffix at the first or the third quarter of the suffix array, and so on. We notice that any given element in the suffix array only has one possible range that we can encounter through the binary search process such that the element is in the middle of the range. For instance, the middle element of the suffix array can only occur with the range 0 to $|n|$, the element at the first quarter can only occur with the range 0 to $|n|/2$, and so on. Thus, we only have $O(2 * |n|) = O(|n|)$ ways to transition middle values. This means that our precomputed table is linear in size.

To specifically compute the values, we need to make one more observation: the LCP of two suffixes indexed i and j in the suffix array is equal to the minimum element between i and j, inclusive. As such, we can precompute the necessary LCP values in $O(|n|)$ time by using the same concept behind the segment tree construction algorithm.

4.2 Longest Repeated Substring

This is simply equal to the maximal value in the LCP array. We can use the suffix array to obtain the substring itself.

4.3 Longest Common Substring

Say we want to find the longest common substring of two strings, n and m. An $O(|n| + |m|)$ algorithm is as follows:

Define a new string $q \leftarrow n\#m$; that is, n concatenated with "#" and m (we assume that neither of the strings contains "#"). "#" is defined to be lexicographically less than all characters in the alphabet. Compute the suffix array and LCP of q. The length of the longest common substring is the largest value in LCP[i] such that exactly one of the suffixes SA[i] and SA[$i-1$] contains the divider "#".

Suffix Array	LCP	Eligible LCP entries
`#panacea`	-	-
`a`	0	0
`a#panacea`	1	1
`acea`	1	1
`ana#panacea`	1	1
`anacea`	3	3
`anana#panacea`	3	3
`banana#panacea`	0	-
`cea`	0	0
`ea`	0	-
`na#panacea`	0	0
`nacea`	2	2
`nana#panacea`	2	2
`panacea`	0	0

Figure 2: Finding the longest common substring for "banana" and "panacea"

In this example, the maximum value in the third column, 3, is the length of the longest common substring. From the suffix array, we can see that the substring is `ana`.

We can extend this algorithm to n strings with an $O(T)$ runtime, where T is the total length of the strings. We do this by creating a suffix array of the concatenated strings separated by unique delimiters, as described above, and then using the sliding window minimum on ranges of the suffix array that contain at least one suffix of each of the n strings. The maximum of the ranges is our answer.

5 Problems

1. Hidden password (ACM 2003, abridged)
   Consider a string of length n ($1 \leq n \leq 100000$). Determine its minimum lexicographic rotation. For instance, the rotations of ‘‘cba’’ are [`cba`, `bac`, `abc`] and the smallest is ‘‘abc’’.

2. substr (training camp 2003, abridged)
   Given a string of length n ($1 \leq n \leq 100000$) and an integer k ($1 \leq k \leq 20000$), find the longest substring that appears at least k times.

3. guess (training camp 2003, abridged)
   Given a string of length n ($1 \leq n \leq 100000$), find the number of distinct substrings.

4. the longest palindrome (USACO training gate)
   Given a string of length n ($1 \leq n \leq 100000$), determine the longest palindromic substring.