

linearization of the circular string. If $l = 0$ or $l = n + 1$, then cut the circular string between character n and character 1. Each leaf in the subtree of this point gives a cutting point yielding the same linear string.

The correctness of this solution is easy to establish and is left as an exercise.

This method runs in linear time and is therefore time optimal. A different linear-time method with a smaller constant was given by Shiloach [404].

7.14. APL13: Suffix arrays – more space reduction

In Section 6.5.1, we saw that when alphabet size is included in the time and space bounds, the suffix tree for a string of length m either requires $\Theta(m|\Sigma|)$ space or the minimum of $O(m \log m)$ and $O(m \log |\Sigma|)$ time. Similarly, searching for a pattern P of length n using a suffix tree can be done in $O(n)$ time only if $\Theta(m|\Sigma|)$ space is used for the tree, or if we assume that up to $|\Sigma|$ character comparisons cost only constant time. Otherwise, the search takes the minimum of $O(n \log m)$ and $O(n \log |\Sigma|)$ comparisons. For these reasons, a suffix tree may require too much space to be practical in some applications. Hence a more space efficient approach is desired that still retains most of the advantages of searching with a suffix tree.

In the context of the substring problem (see Section 7.3) where a fixed string T will be searched many times, the key issues are the time needed for the search and the space used by the fixed data structure representing T . The space used during the preprocessing of T is of less concern, although it should still be “reasonable”.

Manber and Myers [308] proposed a new data structure, called a *suffix array*, that is very space efficient and yet can be used to solve the exact matching problem or the substring problem almost as efficiently as with a suffix tree. Suffix arrays are likely to be an important contribution to certain string problems in computational molecular biology, where the alphabet can be large (we will discuss some of the reasons for large alphabets below). Interestingly, although the more formal notion of a suffix array and the basic algorithms for building and using it were developed in [308], many of the ideas were anticipated in the biological literature by Martinez [310].

After defining suffix arrays we show how to convert a suffix tree to a suffix array in linear time. It is important to be clear on the setting of the problem. String T will be held fixed for a long time, while P will vary. Therefore, the goal is to find a space-efficient representation for T (a suffix array) that will be held fixed and that facilitates search problems in T . However, the amount of space used during the construction of that representation is not so critical. In the exercises we consider a more space efficient way to build the representation itself.

Definition Given an m -character string T , a *suffix array* for T , called Pos , is an array of the integers in the range 1 to m , specifying the lexicographic order of the m suffixes of string T .

That is, the suffix starting at position $Pos(1)$ of T is the lexically smallest suffix, and in general suffix $Pos(i)$ of T is lexically smaller than suffix $Pos(i + 1)$.

As usual, we will affix a terminal symbol $\$$ to the end of S , but now we interpret it to be lexically *less* than any other character in the alphabet. This is in contrast to its interpretation in the previous section. As an example of a suffix array, if T is *mississippi*, then the suffix array Pos is 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3. Figure 7.4 lists the eleven suffixes in lexicographic order.

```

11: i
 8: ippi
 5: issippi
 2: ississippi
 1: mississippi
10: pi
 9: ppi
 7: sippi
 4: sisippi
 6: ssippi
 3: ssissippi

```

Figure 7.4: The eleven suffixes of *mississippi* listed in lexicographic order. The starting positions of those suffixes define the suffix array *Pos*.

Notice that the suffix array holds only integers and hence contains no information about the alphabet used in string T . Therefore, the space required by suffix arrays is modest – for a string of length m , the array can be stored in exactly m computer words, assuming a word size of at least $\log m$ bits.

When augmented with an additional $2m$ values (called *Lcp* values and defined later), the suffix array can be used to find all the occurrences in T of a pattern P in $O(n + \log_2 m)$ single-character comparison and bookkeeping operations. Moreover, this bound is independent of the alphabet size. Since for most problems of interest $\log_2 m$ is $O(n)$, the substring problem is solved by using suffix arrays as efficiently as by using suffix trees.

7.14.1. Suffix tree to suffix array in linear time

We assume that sufficient space is available to build a suffix tree for T (this is done once during a preprocessing phase), but that the suffix tree cannot be kept intact to be used in the (many) subsequent searches for patterns in T . Instead, we convert the suffix tree to the more space efficient suffix array. Exercises 53, 54, and 55 develop an alternative, more space efficient (but slower) method, for *building* a suffix array.

A suffix array for T can be obtained from the suffix tree \mathcal{T} for T by performing a “lexical” depth-first traversal of \mathcal{T} . Once the suffix array is built, the suffix tree is discarded.

Definition Define an edge (v, u) to be *lexically less* than an edge (v, w) if and only if the first character on the (v, u) edge is lexicographically less than the first character on (v, w) . (In this application, the end of string character $\$$ is lexicographically less than any other character.)

Since no two edges out of v have labels beginning with the same character, there is a strict lexical ordering of the edges out of v . This ordering implies that the path from the root of \mathcal{T} following the lexicographically smallest edge out of each encountered node leads to a leaf of \mathcal{T} representing the lexicographically smallest suffix of T . More generally, a depth-first traversal of \mathcal{T} that traverses the edges out of each node v in their lexical order will encounter the leaves of \mathcal{T} in the lexical order of the suffixes they represent. Suffix array *Pos* is therefore just the ordered list of suffix numbers encountered at the leaves of \mathcal{T} during the lexical depth-first search. The suffix tree for T is constructed in linear time, and the traversal also takes only linear time, so we have the following:

Theorem 7.14.1. *The suffix array *Pos* for a string T of length m can be constructed in $O(m)$ time.*

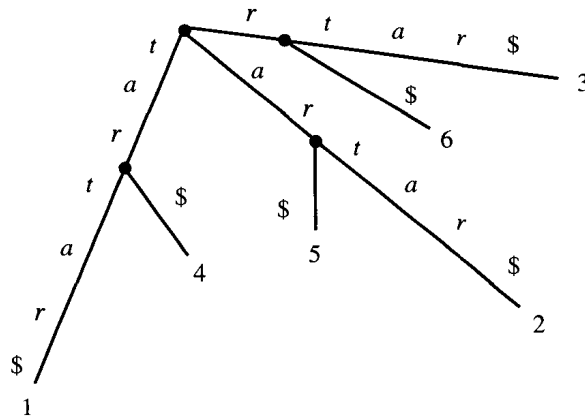


Figure 7.5: The lexical depth-first traversal of the suffix tree visits the leaves in order 5, 2, 6, 3, 4, 1.

For example, the suffix tree for $T = \text{tartar}$ is shown in Figure 7.5. The lexical depth-first traversal visits the nodes in the order 5, 2, 6, 3, 4, 1, defining the values of array Pos .

As an implementation detail, if the branches out of each node of the tree are organized in a *sorted* linked list (as discussed in Section 6.5, page 116) then the overhead to do a lexical depth-first search is the same as for any depth-first search. Every time the search must choose an edge out of a node v to traverse, it simply picks the next edge on v 's linked list.

7.14.2. How to search for a pattern using a suffix array

The suffix array for string T allows a very simple algorithm to find all occurrences of any pattern P in T . The key is that if P occurs in T then all the locations of those occurrences will be grouped consecutively in Pos . For example, $P = \text{issi}$ occurs in *mississippi* starting at locations 2 and 5, which are indeed adjacent in Pos (see Figure 7.4). So to search for occurrences of P in T simply do binary search over the suffix array. In more detail, suppose that P is lexically less than the suffix in the middle position of Pos (i.e., suffix $Pos(\lceil m/2 \rceil)$). In that case, the first place in Pos that contains a position where P occurs in T must be in the first half of Pos . Similarly, if P is lexically greater than suffix $Pos(\lceil m/2 \rceil)$, then the places where P occurs in T must be in the second half of Pos . Using binary search, one can therefore find the smallest index i in Pos (if any) such that P exactly matches the first n characters of suffix $Pos(i)$. Similarly, one can find the largest index i' with that property. Then pattern P occurs in T starting at every location given by $Pos(i)$ through $Pos(i')$.

The lexical comparison of P to any suffix takes time proportional to the length of the common prefix of those two strings. That prefix has length at most n ; hence

Theorem 7.14.2. *By using binary search on array Pos , all the occurrences of P in T can be found in $O(n \log m)$ time.*

Of course, the true behavior of the algorithm depends on how many long prefixes of P occur in T . If very few long prefixes of P occur in T then it will rarely happen that a specific lexical comparison actually takes $\Theta(n)$ time and generally the $O(n \log m)$ bound is quite pessimistic. In “random” strings (even on large alphabets) this method should run in $O(n + \log m)$ expected time. In cases where many long prefixes of P do occur in T , then the method can be improved with the two tricks described in the next two subsections.