maximal pairs, then the algorithm can be modified to run in $O(n)$ time. If only maximal pairs of a certain minimum length are requested (this would be the typical case in many applications), then the algorithm can be modified to run in $O(n + k_w)$ time, where $k_w$ is the number of maximal pairs of length at least the required minimum. Simply stop the bottom-up traversal at any node whose string-depth falls below that minimum.

In summary, we have the following theorem:

**Theorem 7.12.5.** All the maximal pairs can be found in $O(n + k)$ time, where $k$ is the number of maximal pairs. If there are only $k_w$ maximal pairs of length above a given threshold, then all those can be found in $O(n + k_w)$ time.

### 7.13. APL12: Circular string linearization

Recall the definition of a circular string $S$ given in Exercise 2 of Chapter 1 (page 11). The characters of $S$ are initially numbered sequentially from 1 to $n$ starting at an arbitrary point in $S$.

**Definition** Given an ordering of the characters in the alphabet, a string $S_1$ is lexicographically (or lexicographically) smaller than a string $S_2$ if $S_1$ would appear before $S_2$ in a normal dictionary ordering of the two strings. That is, starting from the left end of the two strings, if it is the first position where the two strings differ, then $S_1$ is lexicographically smaller than $S_2$ if and only if $S_1(i)$ precedes $S_2(i)$ in the ordering of the alphabet used in those strings.

To handle the case that $S_1$ is a proper prefix of $S_2$ (and should be considered lexicographically less than $S_2$), we follow the convention that a space is taken to be the first character of the alphabet.

The circular string linearization problem for a circular string $S$ of $n$ characters is the following: Choose a place to cut $S$ so that the resulting linear string is the lexicographically smallest of all the $n$ possible linear strings created by cutting $S$.

This problem arises in chemical data bases for circular molecules. Each such molecule is represented by a suffix tree with a prefix of the alphabet; to allow faster lookup and comparisons of molecules, one wants to store each circular string by a canonical linear string. A single circular molecule may itself be a part of a more complex molecule, so this problem arises in the “inner loop” of more complex chemical retrieval and comparison problems.

A natural choice for canonical linear string is the one that is lexicographically least. With suffix trees, this string can be found in $O(n)$ time.

### 7.13.1. Solution via suffix trees

Arbitrarily cut the circular string $S$, giving a linear string $L$. Then, double $L$, creating the string $LL$, and build the suffix tree $T$ for $LL$. As usual, affix the terminal symbol $S$ at the end of $LL$, but interpret it to be lexicographically greater than any character in the alphabet used for $S$. Intuitively, the purpose of doubling $L$ is to allow efficient consideration of strings that begin with a suffix of $L$ and end with a prefix of $L$. Next, traverse tree $T$ with the rule that, at every node, the traversal follows the edge whose first character is lexicographically smallest over all first characters on edges out of the node. This traversal continues until the traversed path has string-depth $n$. Such a depth will always be reached (with the proof left to the reader). Any leaf $T$ in the subtree at that point can be used to cut the string. If $1 < l < n$, then cutting $S$ between characters $l - 1$ and $l$ creates a lexicographically smallest 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 Sibolsh [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 $n$ either requires $O(m|\Sigma|)$ space or the minimum of $O(m \log m)$ and $O(n \log |\Sigma|)$ time. Similarly, searching for a pattern $P$ of length $n$ using a suffix tree can be done in $O(n)$ time only if $8m|\Sigma|$ 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, suffix trees 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 Martínez [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 lexicographically smallest suffix, and in general suffix $Pos(i)$ of $T$ is lexicographically smaller than suffix $Pos(i + 1)$. As usual, we will affix a terminal symbol $S$ to the end of $S$, but now interpret it to be lexicographically 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 of lexicographic order.
7.14. 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 \( T \) for \( T \) by performing a "lexical" depth-first traversal of \( T \). Once the suffix array is built, the suffix tree is discarded.

Definition: Define an edge \((v, u)\) to be lexicographically 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 the \((v, w)\). (In this application, the end of string character \( S \) 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 \( T \) following the lexicographically smallest edge out of each encountered node leads to a leaf of \( T \) representing the lexicographically smallest suffix of \( T \). More generally, a depth-first traversal of \( T \) that traverses the edges out of each node \( v \) in their lexical order will encounter the leaves of \( T \) 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 \( T \) 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.

For example, the suffix tree for \( T = tartar \) 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, \( P = issi \) 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 lexicographically less than the suffix in the middle position of \( Pos \) (i.e., suffix \( Pos[l \cdot m/2] \)). 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 lexicographically greater than suffix \( Pos[(m-1)/2] \), 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 \( Pos(i + 1) \).

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.
7.14.3. A simple accelerator

As the binary search proceeds, let L and R denote the left and right boundaries of the "current search interval". At the start, L equals I and R equals n. Then in each iteration of the binary search, a query is made at location \( M = \lceil (R + L)/2 \rceil \) of Pos. The search algorithm keeps track of the longest prefixes of Pos(L) and Pos(R) that match a prefix of P. Let l and r denote those two prefix lengths, respectively, and let mrl = \( \min(l, r) \).

The value mrl can be used to accelerate the lexicical comparison of P and suffix Pos(M). Since array Pos gives the lexicographical ordering of the suffixes of T, if i is an index between L and R, the first mrl characters of suffix Pos(i) must be the same as the first mrl characters of suffix Pos(L) and hence of P. Therefore, the lexicographical comparison of P and suffix Pos(M) can begin from position mrl + 1 of the two strings, rather than starting from the first position.

Maintaining mrl during the binary search adds little additional overhead to the algorithm but avoids many redundant comparisons. At the start of the search, when L = 1 and R = m, explicitly compare P to suffix Pos(1) and suffix Pos(m) to find l, r, and mrl. However, the worst-case time for this revised method is still \( O(n \log m) \). Myers and Numer report that the use of mrl alone allows the search to run as fast in practice as the \( O(n + \log m) \) worst-case method that we first advertised. Still, if only because of its elegance, we present the full method that guarantees that better worst-case bound.

7.14.4. A super-accelerator

Call an examination of a character in P redundant if that character has been examined before. The goal of the acceleration is to reduce the number of redundant character examinations to at most one per iteration of the binary search—hence \( O(n \log m) \) in all. The desired time bound, \( O(n + \log m) \), follows immediately. The use of mrl alone does not achieve this goal. Since mrl is the minimum of l and r, whenever \( l \neq r \), all characters in P from mrl + 1 to the maximum of l and r will have already been examined. Thus any comparisons of those characters will be redundant. What is needed is a way to begin comparisons at the maximum of l and r.

Definition Lep(i, j) is the length of the longest common prefix of the suffixes specified in positions i and j of Pos. That is, Lep(i, j) is the length of the longest prefix common to suffix Pos(i) and suffix Pos(j). The term Lep stands for longest common prefix.

For example, when \( T = \text{mississippi} \), suffix Pos(3) is isissipi, suffix Pos(4) is isissipi, and so Lep(3, 4) is four (see Figure 7.4).

To speed up the search, the algorithm uses Lep(L, M) and Lep(M, R) for each triple \((L, M, R)\) that arises during the execution of the binary search. For now, we assume that these values can be obtained in constant time when needed and show how they help the search. Later we will show how to compute the particular Lep values needed by the binary search during the preprocessing of T.

How to use Lep values

Simplest case In any iteration of the binary search, if \( l = r \), then compare P to suffix Pos(M) starting from position mrl + 1 = \( l + 1 = r + 1 \), as before.

General case When \( l \neq r \), let us assume without loss of generality that \( l > r \). Then there are three subcases:

- If \( \text{Lep}(L, M) > r \), then the common prefix of suffix Pos(L) and suffix Pos(M) is longer than the common prefix of P and Pos(L). Therefore, P agrees with suffix Pos(M) up through character r. In other words, characters l + 1 of suffix Pos(L) and suffix Pos(M) are identical and lexically less than character l + 1 of P (the last fact follows since P is lexically greater than suffix Pos(L)). Hence all (if any) starting locations of P in T must occur to the right of position M in Pos. So in any iteration of the binary search where this case occurs, no examinations of P are needed; L just gets changed to M, and l and r remain unchanged. (See Figure 7.6.)

- If \( \text{Lep}(L, M) < l \), then the common prefix of suffix Pos(L) and suffix Pos(M) is smaller than the common prefix of P and Pos(L) and Pos(M). Therefore, P agrees with suffix Pos(M) up through character Lep(L, M). The Lep(L, M) + 1 characters of P and suffix Pos(M) are identical and lexically less than character Lep(L, M) + 1 of suffix Pos(M). Hence all (if any) starting locations of P in T must occur to the left of position M in Pos. So in any iteration of the binary search where this case occurs, no examinations of P are needed; r is changed to Lep(L, M), M remains unchanged, and L and R change along with the corresponding change of l or r.

Theorem 7.14.3 Using the Lep values, the search algorithm does at most \( O(n + \log m) \) comparisons and runs in that time.

Proof First, by simple case analysis it is easy to verify that neither l nor r ever decrease during the binary search. Also, every iteration of the binary search terminates the search, examines no characters of P, or ends after the first mismatch occurs in that iteration.

In the two cases \( l = r \) or \( \text{Lep}(L, M) = l > r \) where the algorithm examines a character during the iteration, the comparisons start with character max(l, r) of P. Suppose there are \( k \) characters of P examined in that iteration. Then there are \( k - 1 \) matches during the iteration and, at the end of the iteration max(l, r) increases by \( k - 1 \) (either l or r is changed to that value). Hence at the start of any iteration, character max(l, r) of P may have already been examined, but the next character in P has not been. That means at most one redundant comparison per iteration is done. Thus no more than \( \log m \) redundant comparisons are done overall. There are at most \( n \) nonredundant comparisons of characters
7.14.5. How to obtain the Lcp values

The Lcp values needed to accelerate searches are precomputed in the preprocessing phase during the creation of the suffix array. We first consider how many possible Lcp values are ever needed (over any possible execution of binary search). For convenience, assume m is a power of two.

Definition Let T be a complete binary tree with m leaves, where each node of T is labeled with a pair of integers (i, j), 1 ≤ i ≤ j ≤ m. The root of T is labeled (1, m). Every non-leaf node (i, j) has two children: the left one is labeled (i, (i + j)/2] and the right one is labeled ([i + j/2], j). The leaves of T are labeled (1, 1) (plus one labeled (1, i)) and are ordered left to right in increasing order of i. (See Figure 7.7.)

Essentially, the node labels specify the endpoints (L, R) of all the possible search intervals that could arise in the binary search of an ordered list of length m. Since T is a binary tree with m leaves, B has 2m − 1 nodes in total. So there are only O(m) Lcp values that need be precomputed. It is therefore plausible that those values can be accumulated during the O(m)-time preprocessing of T; but how exactly? In the next lemma we show that the Lcp values at the leaves of T are easy to accumulate during the lexical depth-first traversal of T.

Lemma 7.14.1. In the depth-first traversal of T, consider the internal nodes visited between the visits to leaf Pos(i) and leaf Pos(i + 1), that is, between the ith leaf visited and the next leaf visited. From among those internal nodes, let v denote the one that is closest to the root. Then Lcp(i, i + 1) equals the string depth of node v.

For example, consider again the suffix tree shown in Figure 7.5 (page 151). Lcp(5, 6) is the string-depth of the parent of leaves 4 and 1. That string-depth is 3, since the parent of 4 and 1 is labeled with the string tar. The values of Lcp(i, i + 1) are 2, 0, 1, 0, 3 for i from 1 to 5.

The hardest part of Lemma 7.14.1 involves parsing it. Once done, the proof is immediate from properties of suffix trees, and it is left to the reader.

If we assume that the string-depths of the nodes are known (these can be accumulated in linear time), then by the lemma, the values Lcp(i, i + 1) for i from 1 to m − 1 are easily accumulated in O(m) time. The rest of the Lcp values are easy to accumulate because of the following lemma:

Lemma 7.14.2. For any pair of positions i, j, where j is greater than i + 1, Lcp(i, j) is the smallest value of Lcp (k, k + 1), where k ranges from i to j − 1.

Proof Suffix Pos(i) and Suffix Pos(j) of T have a common prefix of length Lcp(i, j). By the properties of lexical ordering, for every k between i and j, suffix Pos(k) must also have that common prefix. Therefore, Lcp(k, k + 1) ≥ Lcp(i, j) for every k between i and j.

Now by transivity, Lcp(i, i + 2) must be at least as large as the minimum of Lcp(i, i + 1) and Lcp(i + 1, i + 2). Extending this observation, Lcp(i, j) must be at least as large as the smallest Lcp(k, k + 1) for k from i to j − 1. Combined with the observation in the first paragraph, the lemma is proved.

Given Lemma 7.14.2, the remaining Lcp values for B can be found by working up from the leaves, setting the Lcp value at any node v to the minimum of the Lcp values of its two children. This clearly takes just O(n) time.

In summary, the O(n + log m)-time string and substring matching algorithm using a suffix array must precompute the 2m − 1 Lcp values associated with the nodes of binary tree B. The leaf values can be accumulated during the linear-time, lexical, depth-first traversal of T used to construct the suffix array. The remaining values are computed from the leaf values in linear time by a bottom-up traversal of B, resulting in the following:

Theorem 7.14.4. All the needed Lcp values can be accumulated in O(m) time, and all occurrences of P in T can be found using a suffix array in O(n + log m) time.

7.14.6. Where do large alphabet problems arise?

A large part of the motivation for suffix arrays comes from problems that arise in using suffix trees when the underlying alphabet is large. So it is natural to ask where large alphabets occur.

First, there are natural languages, such as Chinese, with large “alphabets” (using some computer representation of the Chinese pictograms.) However, most large alphabets of interest to us arise because the string contains numbers, each of which is treated as a character. One simple example is a string that comes from a picture where each character in the string gives the color or gray level of a pixel.

String and substring matching problems where the alphabet contains numbers, and where P and T are large, also arise in computational problems in molecular biology. One example is the map matching problem. A restriction enzyme map for a single enzyme specifies the locations in a DNA string where copies of a certain substring (a restriction enzyme recognition site) occurs. Each such site may be separated from the next one by many thousands of bases. Hence, the restriction enzyme map for that single enzyme is represented as a string consisting of a sequence of integers specifying the distances between successive enzyme sites. Considered as a string, each integer is a character of a (huge) underlying alphabet. More generally, a map may display the sites of many different patterns of interest (whether or not they are restriction enzyme sites), so the string (map)
consists of characters from a finite alphabet (representing the known patterns of interest) alternating with integers giving the distances between such symbols. The alphabet is huge because the range of integers is huge, and since distances are often known with high precision, the numbers are not rounded off. Moreover, the variety of known patterns of interest is itself large (see [435]).

It often happens that a DNA substring is obtained and studied without knowing where that DNA is located in the genome or whether that substring has been previously researched. If both the new and the previously studied DNA are fully sequenced and put in a database, then the issue of previous work or locations would be solved by exact string matching. But most DNA substrings that are studied are not fully sequenced—maps are easier and cheaper to obtain than sequences. Consequently, the following matching problem on maps arises and translates to an matching problem on strings with large alphabets:

Given an established (restriction enzyme) map for a large DNA string and a map from a smaller string, determine if the smaller string is a substring of the larger one.

Since each map is represented as an alternating string of characters and integers, the underlying alphabet is huge. This provides one motivation for using suffix arrays for matching or substring searching in place of suffix trees. Of course, the problems become more difficult in the presence of errors, when the integers in the strings may not be exact, or when sites are missing or spuriously added. That problem, called map alignment, is discussed in Section 16.10.

7.15. APL14: Suffix trees in genome-scale projects

Suffix trees, generalized suffix trees and suffix arrays are now being used as the central data structures in three genome-scale projects.

Arabidopsis thaliana. An Arabidopsis thaliana genome project, by the Michigan State University and the University of Minnesota is initially creating an EST map of the Arabidopsis genome (see Section 3.5.1 for a discussion of ESTs and Chapter 16 for a discussion of mapping). In that project generalized suffix trees are used in several ways [63, 64, 65].

First, each sequenced fragment is checked to catch any contamination by known vector sequences. The vector sequences are kept in a generalized suffix tree, as discussed in Section 7.5.

Second, each new sequenced fragment is checked against fragments already sequenced to find duplicate sequences or regions of high similarity. The fragment sequences are kept in an expanding generalized suffix tree for this purpose. Since the project will sequence about 36,000 fragments, each of length about 400 bases, the efficiency of the searches for duplicates and for contamination is important.

Third, suffix trees are used in the search for biologically significant patterns in the obtained Arabidopsis sequences. Patterns of interest are often represented as regular expressions, and generalized suffix trees are used to accelerate regular expression pattern matching, where a small number of errors in a match are allowed. An approach that permits suffix trees to speed up regular expression pattern matching (with errors) is discussed in Section 12.4.

Yeast. Suffix trees are also the central data structure in genome-scale analysis of Saccharomyces cerevisiae (brewer’s yeast), done at the Max-Planck Institute [320]. Suffix trees are “particularly suitable for finding substring patterns in sequence databases” [320]. So in that project, highly optimized suffix trees called hashed position trees are used to solve problems of “clustering sequence data into evolutionary related protein families, structure prediction, and fragment assembly” [320]. (See Section 16.15 for a discussion of fragment assembly.)

Borrelia burgdorferi. Borrelia burgdorferi is the bacterium causing Lyme disease. Its genome is about one million bases long, and is currently being sequenced at the Brookhaven National Laboratory using a directed sequencing approach to fill in gaps after an initial shotgun sequencing phase (see Section 16.14). Chen and Skiena [100] developed methods based on suffix trees and suffix arrays to solve the fragment assembly problem for this project. In fragment assembly, one major bottleneck is overlap detection, which requires solving a variant of the suffix-prefix matching problem (allowing some errors) for all pairs of strings in a large set (see Section 16.15.1). The Borrelia work [100] consisted of 4,612 fragments (strings) totaling 2,032,740 bases. Using suffix trees and suffix arrays, the needed overlaps were computed in about fifteen minutes. To compare the speed and accuracy of the suffix tree methods to pure dynamic programming methods for overlap detection (discussed in Section 11.6.4 and 16.15.1), Chen and Skiena closely examined comb-sized data. The test established that the suffix tree approach gives a 1,000 times speedup over the (slightly) more accurate dynamic programming approach, finding 99% of the significant overlaps found by using dynamic programming.

Efficiency is critical.

In all three projects, the efficiency of building, maintaining, and searching the suffix trees is extremely important, and the implementation details of Section 6.5 are crucial. However, because the suffix trees are very large (approaching 20 million characters in the case of the Arabidopsis project) additional implementation effort is needed, particularly in organizing the suffix tree on disk, so that the number of disk accesses is reduced. All three projects have deeply explored that issue and have found somewhat different solutions. See [320], [100] and [63] for details.

7.16. APL15: A Boyer–Moore approach to exact set matching

The Boyer–Moore algorithm for exact matching (single pattern) will often make long shifts of the pattern, examining only a small percentage of all the characters in the text. In contrast, Knuth-Morris-Pratt examines all characters in the text in order to find all occurrences of the pattern.

In the case of exact set matching, the Aho-Corasick algorithm is analogous to Knuth-Morris-Pratt; it examines all characters of the text. Since the Boyer–Moore algorithm for a single string is far more efficient in practice than Knuth-Morris-Pratt, one would like to have a Boyer–Moore type algorithm for the exact set matching problem, that is, a method for the exact set matching problem that typically examines only a sublinear portion of T. No known simple algorithm achieves this goal and also has a linear worst-case running