Contents

1

Exact Matching: Fundamental Preprocessing and First Algorithms

1.1. The naive method

Almost all discussions of exact matching begin with the *naive method*, and we follow this tradition. The naive method aligns the left end of P with the left end of T and then compares the characters of P and T left to right until either two unequal characters are found or until P is exhausted, in which case an occurrence of P is reported. In either case, P is then shifted one place to the right, and the comparisons are restarted from the left end of P. This process repeats until the right end of P shifts past the right end of T.

Using n to denote the length of P and m to denote the length of T, the worst-case number of comparisons made by this method is $\Theta(nm)$. In particular, if both P and T consist of the same repeated character, then there is an occurrence of P at each of the first $m - n + 1$ positions of T and the method performs exactly $n(m - n + 1)$ comparisons. For example, if $P = aaa$ and $T = aaaaaaaaaa$ then $n = 3$, $m = 10$, and 24 comparisons are made.

The naive method is certainly simple to understand and program, but its worst-case running time of $\Theta(nm)$ may be unsatisfactory and can be improved. Even the practical running time of the naive method may be too slow for larger texts and patterns. Early on, there were several related ideas to improve the naive method, both in practice and in worst case. The result is that the $\Theta(n \times m)$ worst-case bound can be reduced to $O(n + m)$. Changing "$\times$" to "$+$" in the bound is extremely significant (try $n = 1000$ and $m = 10{,}000{,}000$, which are realistic numbers in some applications).

1.1.1. Early ideas for speeding up the naive method

The first ideas for speeding up the naive method all try to shift P by more than one character when a mismatch occurs, but never shift it so far as to miss an occurrence of P in T. Shifting by more than one position saves comparisons since it moves P through T more rapidly. In addition to shifting by larger amounts, some methods try to reduce comparisons by skipping over parts of the pattern after the shift. We will examine many of these ideas in detail.

Figure 1.1 gives a flavor of these ideas, using $P = abxyabxz$ and $T = xabxyabxyabxz$. Note that an occurrence of P begins at location 6 of T. The naive algorithm first aligns P at the left end of T, immediately finds a mismatch, and shifts P by one position. It then finds that the next seven comparisons are matches and that the succeeding comparison (the ninth overall) is a mismatch. It then shifts P by one place, finds a mismatch, and repeats this cycle two additional times, until the left end of P is aligned with character 6 of T. At that point it finds eight matches and concludes that P occurs in T starting at position 6. In this example, a total of twenty comparisons are made by the naive algorithm.

A smarter algorithm might realize, after the ninth comparison, that the next three

```
   0        1             0        1             0        1
   1234567890123          1234567890123          1234567890123
T: xabxyabxyabxz       T: xabxyabxyabxz       T: xabxyabxyabxz
P: abxyabxz            P: abxyabxz            P: abxyabxz
   *                      *                      *
    abxyabxz               abxyabxz               abxyabxz
    ^^^^^^^*               ^^^^^^^*               ^^^^^^^*
     abxyabxz                  abxyabxz               abxyabxz
     *                         ^^^^^^^^                  ^^^^^
      abxyabxz
      *
       abxyabxz
       *
        abxyabxz
        ^^^^^^^^
```

Figure 1.1: The first scenario illustrates pure naive matching, and the next two illustrate smarter shifts. A caret beneath a character indicates a match and a star indicates a mismatch made by the algorithm.

comparisons of the naive algorithm will be mismatches. This smarter algorithm skips over the next three shift/compares, immediately moving the left end of P to align with position 6 of T, thus saving three comparisons. How can a smarter algorithm do this? After the ninth comparison, the algorithm knows that the first seven characters of P match characters 2 through 8 of T. If it also knows that the first character of P (namely a) does not occur again in P until position 5 of P, it has enough information to conclude that character a does not occur again in T until position 6 of T. Hence it has enough information to conclude that there can be no matches between P and T until the left end of P is aligned with position 6 of T. Reasoning of this sort is the key to shifting by more than one character. In addition to shifting by larger amounts, we will see that certain aligned characters do not need to be compared.

An even smarter algorithm knows the next occurrence in P of the first three characters of P (namely abx) begin at position 5. Then since the first seven characters of P were found to match characters 2 through 8 of T, this smarter algorithm has enough information to conclude that when the left end of P is aligned with position 6 of T, the next three comparisons must be matches. This smarter algorithm avoids making those three comparisons. Instead, after the left end of P is moved to align with position 6 of T, the algorithm compares character 4 of P against character 9 of T. This smarter algorithm therefore saves a total of six comparisons over the naive algorithm.

The above example illustrates the kinds of ideas that allow some comparisons to be skipped, although it should still be unclear how an algorithm can efficiently implement these ideas. Efficient implementations have been devised for a number of algorithms such as the Knuth-Morris-Pratt algorithm, a real-time extension of it, the Boyer–Moore algorithm, and the Apostolico–Giancarlo version of it. All of these algorithms have been implemented to run in linear time ($O(n + m)$ time). The details will be discussed in the next two chapters.

1.2. The preprocessing approach

Many string matching and analysis algorithms are able to efficiently skip comparisons by first spending "modest" time learning about the internal structure of either the pattern P or the text T. During that time, the other string may not even be known to the algorithm. This part of the overall algorithm is called the *preprocessing* stage. Preprocessing is followed by a *search* stage, where the information found during the preprocessing stage is used to reduce the work done while searching for occurrences of P in T. In the above example, the smarter method was assumed to know that character a did not occur again until position 5, and the even smarter method was assumed to know that the pattern abx was repeated again starting at position 5. This assumed knowledge is obtained in the preprocessing stage.

For the exact matching problem, all of the algorithms mentioned in the previous section preprocess pattern P. (The opposite approach of preprocessing text T is used in other algorithms, such as those based on suffix trees. Those methods will be explained later in the book.) These preprocessing methods, as originally developed, are similar in spirit but often quite different in detail and conceptual difficulty. In this book we take a different approach and do not initially explain the originally developed preprocessing methods. Rather, we highlight the similarity of the preprocessing *tasks* needed for several different matching algorithms, by first defining a *fundamental preprocessing* of P that is independent of any particular matching algorithm. Then we show how each specific matching algorithm uses the information computed by the fundamental preprocessing of P. The result is a simpler more uniform exposition of the preprocessing needed by several classical matching methods and a simple linear time algorithm for exact matching based only on this preprocessing (discussed in Section 1.5). This approach to linear-time pattern matching was developed in [202].

1.3. Fundamental preprocessing of the pattern

Fundamental preprocessing will be described for a general string denoted by S. In specific applications of fundamental preprocessing, S will often be the pattern P, but here we use S instead of P because fundamental preprocessing will also be applied to strings other than P.

The following definition gives the key values computed during the fundamental preprocessing of a string.

Definition Given a string S and a position $i > 1$, let $Z_i(S)$ be the *length* of the longest substring of S that *starts* at i and matches a prefix of S.

In other words, $Z_i(S)$ is the length of the longest *prefix* of $S[i..|S|]$ that matches a prefix of S. For example, when $S = aabcaabxaaz$ then

$$Z_5(S) = 3 \quad (aabc...aabx...),$$

$$Z_6(S) = 1 \quad (aa...ab...),$$

$$Z_7(S) = Z_8(S) = 0,$$

$$Z_9(S) = 2 \quad (aab...aaz).$$

When S is clear by context, we will use Z_i in place of $Z_i(S)$.

To introduce the next concept, consider the boxes drawn in Figure 1.2. Each box starts at some position $j > 1$ such that Z_j is greater than zero. The length of the box starting at j is meant to represent Z_j. Therefore, each box in the figure represents a maximal-length

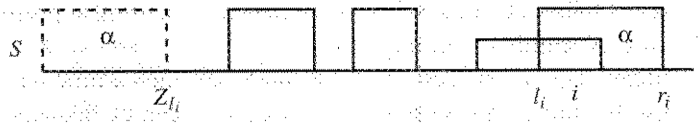

Figure 1.2: Each solid box represents a substring of S that matches a prefix of S and that starts between positions 2 and i. Each box is called a Z-*box*. We use r_i to denote the *right-most* end of any Z-box that begins at or to the left of position i and α to denote the substring in the Z-box ending at r_i. Then l_i denotes the left end of α. The copy of α that occurs as a prefix of S is also shown in the figure.

substring of S that matches a prefix of S and that does not start at position one. Each such box is called a *Z-box*. More formally, we have:

Definition For any position $i > 1$ where Z_i is greater than zero, the *Z-box* at i is defined as the interval starting at i and ending at position $i + Z_i - 1$.

Definition For every $i > 1$, r_i is the right-most endpoint of the Z-boxes that begin at or before position i. Another way to state this is: r_i is the largest value of $j + Z_j - 1$ over all $1 < j \leq i$ such that $Z_j > 0$. (See Figure 1.2.)

We use the term l_i for the value of j specified in the above definition. That is, l_i is the position of the *left end* of the Z-box that ends at r_i. In case there is more than one Z-box ending at r_i, then l_i can be chosen to be the left end of any of those Z-boxes. As an example, suppose $S = aabaabcaxaabaabcy$; then $Z_{10} = 7$, $r_{15} = 16$, and $l_{15} = 10$.

The linear time computation of Z values from S is the *fundamental* preprocessing task that we will use in all the classical linear-time matching algorithms that preprocess P. But before detailing those uses, we show how to do the fundamental preprocessing in linear time.

1.4. Fundamental preprocessing in linear time

The task of this section is to show how to compute all the Z_i values for S in linear time (i.e., in $O(|S|)$ time). A direct approach based on the definition would take $\Theta(|S|^2)$ time. The method we will present was developed in [307] for a different purpose.

The preprocessing algorithm computes Z_i, r_i, and l_i for each successive position i, starting from $i = 2$. All the Z values computed will be kept by the algorithm, but in any iteration i, the algorithm only needs the r_j and l_j values for $j = i - 1$. No earlier r or l values are needed. Hence the algorithm only uses a single variable, r, to refer to the most recently computed r_j value; similarly, it only uses a single variable l. Therefore, in each iteration i, if the algorithm discovers a new Z-box (starting at i), variable r will be incremented to the end of that Z-box, which is the right-most position of any Z-box discovered so far.

To begin, the algorithm finds Z_2 by explicitly comparing, left to right, the characters of $S[2..|S|]$ and $S[1..|S|]$ until a mismatch is found. Z_2 is the length of the matching string. If $Z_2 > 0$, then $r = r_2$ is set to $Z_2 + 1$ and $l = l_2$ is set to 2. Otherwise r and l are set to zero. Now assume inductively that the algorithm has correctly computed Z_i for i up to $k - 1 > 1$, and assume that the algorithm knows the current $r = r_{k-1}$ and $l = l_{k-1}$. The algorithm next computes Z_k, $r = r_k$, and $l = l_k$.

The main idea is to use the already computed Z values to accelerate the computation of Z_k. In fact, in some cases, Z_k can be deduced from the previous Z values without doing any additional character comparisons. As a concrete example, suppose $k = 121$, all the values Z_2 through Z_{120} have already been computed, and $r_{120} = 130$ and $l_{120} = 100$. That means that there is a substring of length 31 starting at position 100 and matching a prefix of S (of length 31). It follows that the substring of length 10 starting at position 121 must match the substring of length 10 starting at position 22 of S, and so Z_{22} may be very helpful in computing Z_{121}. As one case, if Z_{22} is three, say, then a little reasoning shows that Z_{121} must also be three. Thus in this illustration, Z_{121} can be deduced without any additional character comparisons. This case, along with the others, will be formalized and proven correct below.

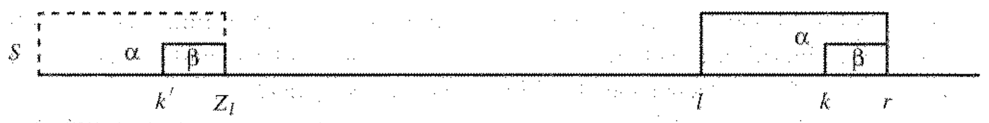

Figure 1.3: String $S[k..r]$ is labeled β and also occurs starting at position k' of S.

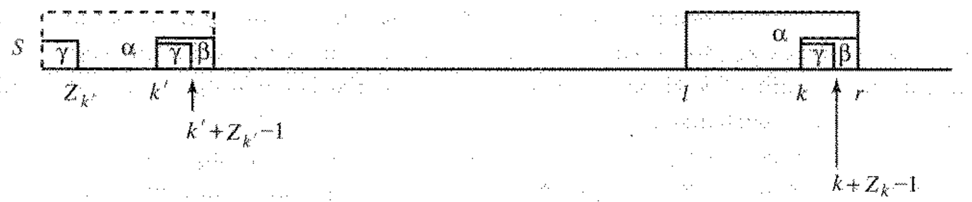

Figure 1.4: Case 2a. The longest string starting at k' that matches a prefix of S is shorter than $|\beta|$. In this case, $Z_k = Z_{k'}$.

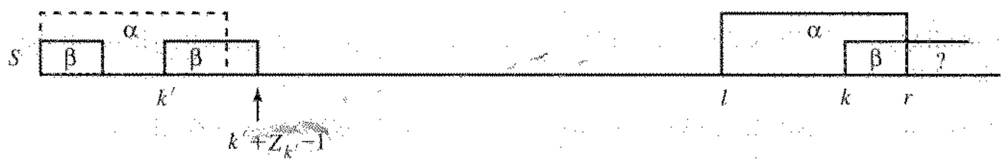

Figure 1.5: Case 2b. The longest string starting at k' that matches a prefix of S is at least $|\beta|$.

The Z algorithm

Given Z_i for all $1 < i \leq k-1$ and the current values of r and l, Z_k and the updated r and l are computed as follows:

Begin

1. If $k > r$, then find Z_k by explicitly comparing the characters starting at position k to the characters starting at position 1 of S, until a mismatch is found. The length of the match is Z_k. If $Z_k > 0$, then set r to $k + Z_k - 1$ and set l to k.

2. If $k \leq r$, then position k is contained in a Z-box, and hence $S(k)$ is contained in substring $S[l..r]$ (call it α) such that $l > 1$ and α matches a prefix of S. Therefore, character $S(k)$ also appears in position $k' = k - l + 1$ of S. By the same reasoning, substring $S[k..r]$ (call it β) must match substring $S[k'..Z_l]$. It follows that the substring beginning at position k must match a prefix of S of length at least the *minimum* of $Z_{k'}$ and $|\beta|$ (which is $r - k + 1$). See Figure 1.3.

   We consider two subcases based on the value of that minimum.

   2a. If $Z_{k'} < |\beta|$ then $Z_k = Z_{k'}$ and r, l remain unchanged (see Figure 1.4).

   2b. If $Z_{k'} \geq |\beta|$ then the entire substring $S[k..r]$ must be a prefix of S and $Z_k \geq |\beta| = r - k + 1$. However, Z_k might be strictly larger than $|\beta|$, so compare the characters starting at position $r + 1$ of S to the characters starting a position $|\beta| + 1$ of S until a mismatch occurs. Say the mismatch occurs at character $q \geq r + 1$. Then Z_k is set to $q - k$, r is set to $q - 1$, and l is set to k (see Figure 1.5).

End

Theorem 1.4.1. *Using Algorithm Z, value Z_k is correctly computed and variables r and l are correctly updated.*

PROOF In Case 1, Z_k is set correctly since it is computed by explicit comparisons. Also (since $k > r$ in Case 1), before Z_k is computed, no Z-box has been found that starts between positions 2 and $k - 1$ and that ends at or after position k. Therefore, when $Z_k > 0$ in Case 1, the algorithm does find a new Z-box ending at or after k, and it is correct to change r to $k + Z_k - 1$. Hence the algorithm works correctly in Case 1.

In Case 2a, the substring beginning at position k can match a prefix of S only for length $Z_{k'} < |\beta|$. If not, then the next character to the right, character $k + Z_{k'}$, must match character $1 + Z_{k'}$. But character $k + Z_{k'}$ matches character $k' + Z_{k'}$ (since $Z_{k'} < |\beta|$), so character $k' + Z_{k'}$ must match character $1 + Z_{k'}$. However, that would be a contradiction to the definition of $Z_{k'}$, for it would establish a substring longer than $Z_{k'}$ that starts at k' and matches a prefix of S. Hence $Z_k = Z_{k'}$ in this case. Further, $k + Z_k - 1 < r$, so r and l remain correctly unchanged.

In Case 2b, β must be a prefix of S (as argued in the body of the algorithm) and since any extension of this match is explicitly verified by comparing characters beyond r to characters beyond the prefix β, the full extent of the match is correctly computed. Hence Z_k is correctly obtained in this case. Furthermore, since $k + Z_k - 1 \geq r$, the algorithm correctly changes r and l. □

Corollary 1.4.1. Repeating Algorithm Z for each position $i > 2$ correctly yields all the Z_i values.

Theorem 1.4.2. *All the $Z_i(S)$ values are computed by the algorithm in $O(|S|)$ time.*

PROOF The time is proportional to the number of iterations, $|S|$, plus the number of character comparisons. Each comparison results in either a match or a mismatch, so we next bound the number of matches and mismatches that can occur.

Each iteration that performs any character comparisons at all ends the first time it finds a mismatch; hence there are at most $|S|$ mismatches during the entire algorithm. To bound the number of matches, note first that $r_k \geq r_{k-1}$ for every iteration k. Now, let k be an iteration where $q > 0$ matches occur. Then r_k is set to $r_{k-1} + q$ at least. Finally, $r_k \leq |S|$, so the total number of matches that occur during any execution of the algorithm is at most $|S|$. □

1.5. The simplest linear-time exact matching algorithm

Before discussing the more complex (classical) exact matching methods, we show that fundamental preprocessing alone provides a simple linear-time exact matching algorithm. This is the simplest linear-time matching algorithm we know of.

Let $S = P\$T$ be the string consisting of P followed by the symbol "\$" followed by T, where "\$" is a character appearing in neither P nor T. Recall that P has length n and T has length m, and $n \leq m$. So, $S = P\$T$ has length $n + m + 1 = O(m)$. Compute $Z_i(S)$ for i from 2 to $n + m + 1$. Because "\$" does not appear in P or T, $Z_i \leq n$ for every $i > 1$. Any value of $i > n + 1$ such that $Z_i(S) = n$ identifies an occurrence of P in T starting at position $i - (n + 1)$ of T. Conversely, if P occurs in T starting at position j of T, then $Z_{(n+1)+j}$ must be equal to n. Since all the $Z_i(S)$ values can be computed in $O(n + m) = O(m)$ time, this approach identifies all the occurrences of P in T in $O(m)$ time.

The method can be implemented to use only $O(n)$ space (in addition to the space needed for pattern and text) independent of the size of the alphabet. Since $Z_i \leq n$ for all i, position k' (determined in step 2) will always fall inside P. Therefore, there is no need to record the Z values for characters in T. Instead, we only need to record the Z values for the n characters in P and also maintain the current l and r. Those values are sufficient to compute (but not store) the Z value of each character in T and hence to identify and output any position i where $Z_i = n$.

There is another characteristic of this method worth introducing here: The method is considered an *alphabet-independent* linear-time method. That is, we never had to assume that the alphabet size was finite or that we knew the alphabet ahead of time – a character comparison only determines whether the two characters match or mismatch; it needs no further information about the alphabet. We will see that this characteristic is also true of the Knuth-Morris-Pratt and Boyer–Moore algorithms, but not of the Aho–Corasick algorithm or methods based on suffix trees.

1.5.1. Why continue?

Since function Z_i can be computed for the pattern in linear time and can be used directly to solve the exact matching problem in $O(m)$ time (with only $O(n)$ additional space), why continue? In what way are more complex methods (Knuth-Morris-Pratt, Boyer–Moore, real-time matching, Apostolico–Giancarlo, Aho–Corasick, suffix tree methods, etc.) deserving of attention?

For the exact matching problem, the Knuth-Morris-Pratt algorithm has only a marginal advantage over the direct use of Z_i. However, it has historical importance and has been generalized, in the Aho–Corasick algorithm, to solve the problem of searching for a *set* of patterns in a text in time linear in the size of the text. That problem is not nicely solved using Z_i values alone. The real-time extension of Knuth-Morris-Pratt has an advantage in situations when text is input on-line and one has to be sure that the algorithm will be ready for each character as it arrives. The Boyer–Moore method is valuable because (with the proper implementation) it also runs in linear worst-case time but typically runs in *sublinear* time, examining only a fraction of the characters of T. Hence it is the preferred method in most cases. The Apostolico–Giancarlo method is valuable because it has all the advantages of the Boyer–Moore method and yet allows a relatively simple proof of linear worst-case running time. Methods based on suffix trees typically preprocess the text rather than the pattern and then lead to algorithms in which the search time is proportional to the size of the pattern rather than the size of the text. This is an extremely desirable feature. Moreover, suffix trees can be used to solve much more complex problems than exact matching, including problems that are not easily solved by direct application of the fundamental preprocessing.

1.6. Exercises

The first four exercises use the fact that fundamental processing can be done in linear time and that all occurrences of *P* in *T* can be found in linear time.

1. Use the existence of a linear-time exact matching algorithm to solve the following problem in linear time. Given two strings α and β, determine if α is a circular (or cyclic) rotation of β, that is, if α and β have the same length and α consists of a suffix of β followed by a prefix of β. For example, *defabc* is a circular rotation of *abcdef*. This is a classic problem with a very elegant solution.

2. Similar to Exercise 1, give a linear-time algorithm to determine whether a linear string α is a substring of a *circular string* β. A circular string of length n is a string in which character n is considered to precede character 1 (see Figure 1.6). Another way to think about this

2

Exact Matching: Classical Comparison-Based Methods

2.1. Introduction

This chapter develops a number of classical comparison-based matching algorithms for the exact matching problem. With suitable extensions, all of these algorithms can be implemented to run in linear worst-case time, and all achieve this performance by preprocessing pattern P. (Methods that preprocess T will be considered in Part II of the book.) The original preprocessing methods for these various algorithms are related in spirit but are quite different in conceptual difficulty. Some of the original preprocessing methods are quite difficult.[1] This chapter does not follow the original preprocessing methods but instead exploits fundamental preprocessing, developed in the previous chapter, to implement the needed preprocessing for each specific matching algorithm.

Also, in contrast to previous expositions, we emphasize the Boyer–Moore method over the Knuth-Morris-Pratt method, since Boyer–Moore is the practical method of choice for exact matching. Knuth-Morris-Pratt is nonetheless completely developed, partly for historical reasons, but mostly because it generalizes to problems such as real-time string matching and matching against a set of patterns more easily than Boyer–Moore does. These two topics will be described in this chapter and the next.

2.2. The Boyer–Moore Algorithm

As in the naive algorithm, the Boyer–Moore algorithm successively aligns P with T and then checks whether P matches the opposing characters of T. Further, after the check is complete, P is shifted right relative to T just as in the naive algorithm. However, the Boyer–Moore algorithm contains three clever ideas not contained in the naive algorithm: the right-to-left scan, the bad character shift rule, and the good suffix shift rule. Together, these ideas lead to a method that typically examines fewer than $m + n$ characters (an expected sublinear-time method) and that (with a certain extension) runs in linear worst-case time. Our discussion of the Boyer–Moore algorithm, and extensions of it, concentrates on *provable* aspects of its behavior. Extensive experimental and practical studies of Boyer–Moore and variants have been reported in [229], [237], [409], [410], and [425].

2.2.1. Right-to-left scan

For any alignment of P with T the Boyer–Moore algorithm checks for an occurrence of P by scanning characters from *right to left* rather than from left to right as in the naive

[1] Sedgewick [401] writes "Both the Knuth-Morris-Pratt and the Boyer–Moore algorithms require some complicated preprocessing on the pattern that is difficult to understand and has limited the extent to which they are used". In agreement with Sedgewick, I still do not understand the original Boyer–Moore preprocessing method for the *strong* good suffix rule.

algorithm. For example, consider the alignment of P against T shown below:

```
                1         2
       12345678901234567
T:     xpbctbxabpqxctbpq
P:       tpabxab
```

To check whether P occurs in T at this position, the Boyer–Moore algorithm starts at the *right* end of P, first comparing $T(9)$ with $P(7)$. Finding a match, it then compares $T(8)$ with $P(6)$, etc., moving right to left until it finds a mismatch when comparing $T(5)$ with $P(3)$. At that point P is shifted *right* relative to T (the amount for the shift will be discussed below) and the comparisons begin again at the right end of P.

Clearly, if P is shifted right by one place after each mismatch, or after an occurrence of P is found, then the worst-case running time of this approach is $O(nm)$ just as in the naive algorithm. So at this point it isn't clear why comparing characters from right to left is any better than checking from left to right. However, with two additional ideas (the *bad character* and the *good suffix* rules), shifts of more than one position often occur, and in typical situations large shifts are common. We next examine these two ideas.

2.2.2. Bad character rule

To get the idea of the bad character rule, suppose that the last (right-most) character of P is y and the character in T it aligns with is $x \neq y$. When this initial mismatch occurs, if we know the right-most position in P of character x, we can safely shift P to the right so that the right-most x in P is below the mismatched x in T. Any shorter shift would only result in an immediate mismatch. Thus, the longer shift is correct (i.e., it will not shift past any occurrence of P in T). Further, if x never occurs in P, then we can shift P completely past the point of mismatch in T. In these cases, some characters of T will never be examined and the method will actually run in "sublinear" time. This observation is formalized below.

Definition For each character x in the alphabet, let $R(x)$ be the position of right-most occurrence of character x in P. $R(x)$ is defined to be zero if x does not occur in P.

It is easy to preprocess P in $O(n)$ time to collect the $R(x)$ values, and we leave that as an exercise. Note that this preprocessing does not require the fundamental preprocessing discussed in Chapter 1 (that will be needed for the more complex shift rule, the good suffix rule).

We use the R values in the following way, called the *bad character shift rule*:

Suppose for a particular alignment of P against T, the right-most $n - i$ characters of P match their counterparts in T, but the next character to the left, $P(i)$, mismatches with its counterpart, say in position k of T. The *bad character rule* says that P should be shifted right by $\max[1, i - R(T(k))]$ places. That is, if the right-most occurrence in P of character $T(k)$ is in position $j < i$ (including the possibility that $j = 0$), then shift P so that character j of P is below character k of T. Otherwise, shift P by one position.

The point of this shift rule is to shift P by more than one character when possible. In the above example, $T(5) = t$ mismatches with $P(3)$ and $R(t) = 1$, so P can be shifted right by two positions. After the shift, the comparison of P and T begins again at the right end of P.

Extended bad character rule

The bad character rule is a useful heuristic for mismatches near the right end of P, but it has no effect if the mismatching character from T occurs in P to the right of the mismatch point. This may be common when the alphabet is small and the text contains many similar, but not exact, substrings. That situation is typical of DNA, which has an alphabet of size four, and even protein, which has an alphabet of size twenty, often contains different regions of high similarity. In such cases, the following *extended bad character rule* is more robust:

> When a mismatch occurs at position i of P and the mismatched character in T is x, then shift P to the right so that the closest x to the left of position i in P is below the mismatched x in T.

Because the extended rule gives larger shifts, the only reason to prefer the simpler rule is to avoid the added implementation expense of the extended rule. The simpler rule uses only $O(|\Sigma|)$ space (Σ is the alphabet) for array R, and one table lookup for each mismatch. As we will see, the extended rule can be implemented to take only $O(n)$ space and at most one extra step per character comparison. That amount of added space is not often a critical issue, but it is an empirical question whether the longer shifts make up for the added time used by the extended rule. The original Boyer–Moore algorithm only uses the simpler bad character rule.

Implementing the extended bad character rule

We preprocess P so that the extended bad character rule can be implemented efficiently in both time and space. The preprocessing should discover, for each position i in P and for each character x in the alphabet, the position of the closest occurrence of x in P to the left of i. The obvious approach is to use a two-dimensional array of size n by $|\Sigma|$ to store this information. Then, when a mismatch occurs at position i of P and the mismatching character in T is x, we look up the (i, x) entry in the array. The lookup is fast, but the size of the array, and the time to build it, may be excessive. A better compromise, below, is possible.

During preprocessing, scan P from right to left collecting, for each character x in the alphabet, a list of the positions where x occurs in P. Since the scan is right to left, each list will be in decreasing order. For example, if $P = abacbabc$ then the list for character a is 6, 3, 1. These lists are accumulated in $O(n)$ time and of course take only $O(n)$ space. During the search stage of the Boyer–Moore algorithm if there is a mismatch at position i of P and the mismatching character in T is x, scan x's list from the top until we reach the first number less than i or discover there is none. If there is none then there is no occurrence of x before i, and all of P is shifted past the x in T. Otherwise, the found entry gives the desired position of x.

After a mismatch at position i of P the time to scan the list is at most $n - i$, which is roughly the number of characters that matched. So in worst case, this approach at most doubles the running time of the Boyer–Moore algorithm. However, in most problem settings the added time will be vastly less than double. One could also do binary search on the list in circumstances that warrant it.

2.2.3. The (strong) good suffix rule

The bad character rule by itself is reputed to be highly effective in practice, particularly for English text [229], but proves less effective for small alphabets and it does not lead to a linear worst-case running time. For that, we introduce another rule called the *strong*

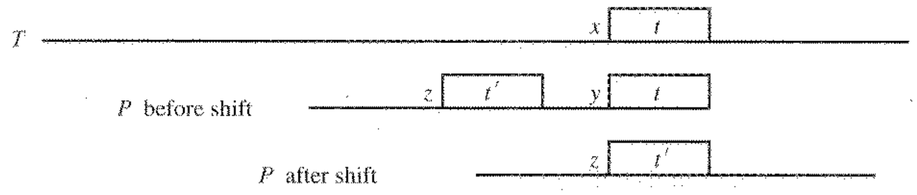

Figure 2.1: Good suffix shift rule, where character x of T mismatches with character y of P. Characters y and z of P are guaranteed to be distinct by the good suffix rule, so z has a chance of matching x.

good suffix rule. The original preprocessing method [278] for the strong good suffix rule is generally considered quite difficult and somewhat mysterious (although a weaker version of it is easy to understand). In fact, the preprocessing for the strong rule was given incorrectly in [278] and corrected, without much explanation, in [384]. Code based on [384] is given without real explanation in the text by Baase [32], but there are no published sources that try to fully explain the method.[2] Pascal code for strong preprocessing, based on an outline by Richard Cole [107], is shown in Exercise 24 at the end of this chapter.

In contrast, the fundamental preprocessing of P discussed in Chapter 1 makes the needed preprocessing very simple. That is the approach we take here. The *strong good suffix rule* is:

Suppose for a given alignment of P and T, a substring t of T matches a suffix of P, but a mismatch occurs at the next comparison to the left. Then find, if it exists, the right-most copy t' of t in P such that t' is not a suffix of P and *the character to the left of t' in P differs from the character to the left of t in P*. Shift P to the right so that substring t' in P is below substring t in T (see Figure 2.1). If t' does not exist, then shift the left end of P past the left end of t in T by the least amount so that a prefix of the shifted pattern matches a suffix of t in T. If no such shift is possible, then shift P by n places to the right. If an occurrence of P is found, then shift P by the least amount so that a *proper* prefix of the shifted P matches a suffix of the occurrence of P in T. If no such shift is possible, then shift P by n places, that is, shift P past t in T.

For a specific example consider the alignment of P and T given below:

```
       0         1
       123456789012345678
    T: prstabstubabvqxrst
                   *
    P:   qcabdabdab
         1234567890
```

When the mismatch occurs at position 8 of P and position 10 of T, $t = ab$ and t' occurs in P starting at position 3. Hence P is shifted right by *six* places, resulting in the following alignment:

[2] A recent plea appeared on the internet newsgroup comp. theory:

I am looking for an elegant (easily understandable) proof of correctness for a part of the Boyer–Moore string matching algorithm. The difficult-to-prove part here is the algorithm that computes the dd_2 (good-suffix) table. I didn't find much of an understandable proof yet, so I'd much appreciate any help!

```
   0         1
   123456789012345678
T: prstabstubabvqxrst
P:          qcabdabdab
```

Note that the extended bad character rule would have shifted P by only one place in this example.

Theorem 2.2.1. *The use of the good suffix rule never shifts P past an occurrence in T.*

PROOF Suppose the right end of P is aligned with character k of T before the shift, and suppose that the good suffix rule shifts P so its right end aligns with character $k' > k$. Any occurrence of P ending at a position l strictly between k and k' would immediately violate the selection rule for k', since it would imply either that a closer copy of t occurs in P or that a longer prefix of P matches a suffix of t. □

The original published Boyer–Moore algorithm [75] uses a simpler, weaker, version of the good suffix rule. That version just requires that the shifted P agree with the t and does not specify that the next characters to the left of those occurrences of t be different. An explicit statement of the weaker rule can be obtained by deleting the italics phrase in the first paragraph of the statement of the strong good suffix rule. In the previous example, the weaker shift rule shifts P by three places rather than six. When we need to distinguish the two rules, we will call the simpler rule the *weak* good suffix rule and the rule stated above the *strong* good suffix rule. For the purpose of proving that the search part of Boyer–Moore runs in linear worst-case time, the weak rule is not sufficient, and in this book the strong version is assumed unless stated otherwise.

2.2.4. Preprocessing for the good suffix rule

We now formalize the preprocessing needed for the Boyer–Moore algorithm.

Definition For each i, $L(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L(i)]$. $L(i)$ is defined to be zero if there is no position satisfying the conditions. For each i, $L'(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L'(i)]$ and such that the character preceding that suffix is not equal to $P(i-1)$. $L'(i)$ is defined to be zero if there is no position satisfying the conditions.

For example, if $P = cabdabdab$, then $L(8) = 6$ and $L'(8) = 3$.

$L(i)$ gives the right end-position of the right-most copy of $P[i..n]$ that is not a suffix of P, whereas $L'(i)$ gives the right end-position of the right-most copy of $P[i..n]$ that is not a suffix of P, with the stronger, added condition that its preceding character is unequal to $P(i-1)$. So, in the strong-shift version of the Boyer–Moore algorithm, if character $i-1$ of P is involved in a mismatch and $L'(i) > 0$, then P is shifted right by $n - L'(i)$ positions. The result is that if the right end of P was aligned with position k of T before the shift, then position $L'(i)$ is now aligned with position k.

During the preprocessing stage of the Boyer–Moore algorithm $L'(i)$ (and $L(i)$, if desired) will be computed for each position i in P. This is done in $O(n)$ time via the following definition and theorem.

Definition For string P, $N_j(P)$ is the length of the longest *suffix* of the substring $P[1..j]$ that is also a *suffix* of the full string P.

For example, if $P = cabdabdab$, then $N_3(P) = 2$ and $N_6(P) = 5$.

Recall that $Z_i(S)$ is the length of the longest substring of S that starts at i and matches a prefix of S. Clearly, N is the reverse of Z, that is, if P^r denotes the string obtained by reversing P, then $N_j(P) = Z_{n-j+1}(P^r)$. Hence the $N_j(P)$ values can be obtained in $O(n)$ time by using *Algorithm Z* on P^r. The following theorem is then immediate.

Theorem 2.2.2. *$L(i)$ is the largest index j less than n such that $N_j(P) \geq |P[i..n]|$ (which is $n-i+1$). $L'(i)$ is the largest index j less than n such that $N_j(P) = |P[i..n]| = (n-i+1)$.*

Given Theorem 2.2.2, it follows immediately that all the $L'(i)$ values can be accumulated in linear time from the N values using the following algorithm:

Z-based Boyer–Moore

```
for $i := 1$ to $n$ do $L'(i) := 0$;
for $j := 1$ to $n - 1$ do
    begin
    $i := n - N_j(P) + 1$;
    $L'(i) := j$;
    end;
```

The $L(i)$ values (if desired) can be obtained by adding the following lines to the above pseudocode:

```
$L(2) := L'(2)$;
for $i := 3$ to $n$ do $L(i) := \max[L(i-1), L'(i)]$;
```

Theorem 2.2.3. *The above method correctly computes the L values.*

PROOF $L(i)$ marks the right end-position of the right-most substring of P that matches $P[i..n]$ and is not a suffix of $P[1..n]$. Therefore, that substring begins at position $L(i)-n+i$, which we will denote by j. We will prove that $L(i) = \max[L(i-1), L'(i)]$ by considering what character $j-1$ is. First, if $j = 1$ then character $j-1$ doesn't exist, so $L(i-1) = 0$ and $L'(i) = 1$. So suppose that $j > 1$. If character $j-1$ equals character $i-1$ then $L(i) = L(i-1)$. If character $j-1$ does not equal character $i-1$ then $L(i) = L'(i)$. Thus, in all cases, $L(i)$ must either be $L'(i)$ or $L(i-1)$.

However, $L(i)$ must certainly be greater than or equal to both $L'(i)$ and $L(i-1)$. In summary, $L(i)$ must either be $L'(i)$ or $L(i-1)$, and yet it must be greater or equal to both of them; hence $L(i)$ must be the maximum of $L'(i)$ and $L(i-1)$. □

Final preprocessing detail

The preprocessing stage must also prepare for the case when $L'(i) = 0$ or when an occurrence of P is found. The following definition and theorem accomplish that.

Definition Let $l'(i)$ denote the length of the largest suffix of $P[i..n]$ that is also a prefix of P, if one exists. If none exists, then let $l'(i)$ be zero.

Theorem 2.2.4. *$l'(i)$ equals the largest $j \leq |P[i..n]|$, which is $n-i+1$, such that $N_j(P) = j$.*

We leave the proof, as well as the problem of how to accumulate the $l'(i)$ values in linear time, as a simple exercise. (Exercise 9 of this chapter)

2.2.5. The good suffix rule in the search stage of Boyer–Moore

Having computed $L'(i)$ and $l'(i)$ for each position i in P, these preprocessed values are used during the search stage of the algorithm to achieve larger shifts. If, during the search stage, a mismatch occurs at position $i-1$ of P and $L'(i) > 0$, then the good suffix rule shifts P by $n - L'(i)$ places to the right, so that the $L'(i)$-length prefix of the shifted P aligns with the $L'(i)$-length suffix of the unshifted P. In the case that $L'(i) = 0$, the good suffix rule shifts P by $n - l'(i)$ places. When an occurrence of P is found, then the rule shifts P by $n - l'(2)$ places. Note that the rules work correctly even when $l'(i) = 0$.

One special case remains. When the first comparison is a mismatch (i.e., $P(n)$ mismatches) then P should be shifted one place to the right.

2.2.6. The complete Boyer–Moore algorithm

We have argued that neither the good suffix rule nor the bad character rule shift P so far as to miss any occurrence of P. So the Boyer–Moore algorithm shifts by the largest amount given by either of the rules. We can now present the complete algorithm.

The Boyer–Moore algorithm

```
{Preprocessing stage}
    Given the pattern $P$,
    Compute $L'(i)$ and $l'(i)$ for each position $i$ of $P$,
    and compute $R(x)$ for each character $x \in \Sigma$.
{Search stage}
    $k := n$;
    while $k \leq m$ do
        begin
        $i := n$;
        $h := k$;
        while $i > 0$ and $P(i) = T(h)$ do
            begin
            $i := i - 1$;
            $h := h - 1$;
            end;
        if $i = 0$ then
            begin
            report an occurrence of $P$ in $T$ ending at position $k$.
            $k := k + n - l'(2)$;
            end
        else
            shift $P$ (increase $k$) by the maximum amount determined by the
            (extended) bad character rule and the good suffix rule.
        end;
```

Note that although we have always talked about "shifting P", and given rules to determine by how much P should be "shifted", there is no shifting in the actual implementation. Rather, the index k is increased to the point where the right end of P would be "shifted". Hence, each act of shifting P takes constant time.

We will later show, in Section 3.2, that by using the strong good suffix rule alone, the

Boyer–Moore method has a worst-case running time of $O(m)$ provided that the pattern does not appear in the text. This was first proved by Knuth, Morris, and Pratt [278], and an alternate proof was given by Guibas and Odlyzko [196]. Both of these proofs were quite difficult and established worst-case time bounds no better than $5m$ comparisons. Later, Richard Cole gave a much simpler proof [108] establishing a bound of $4m$ comparisons and also gave a difficult proof establishing a tight bound of $3m$ comparisons. We will present Cole's proof of $4m$ comparisons in Section 3.2.

When the pattern does appear in the text then the original Boyer–Moore method runs in $\Theta(nm)$ worst-case time. However, several simple modifications to the method correct this problem, yielding an $O(m)$ time bound in all cases. The first of these modifications was due to Galil [168]. After discussing Cole's proof, in Section 3.2, for the case that P doesn't occur in T, we use a variant of Galil's idea to achieve the linear time bound in all cases.

At the other extreme, if we only use the bad character shift rule, then the worst-case running time is $O(nm)$, but assuming randomly generated strings, the expected running time is sublinear. Moreover, in typical string matching applications involving natural language text, a sublinear running time is almost always observed in practice. We won't discuss random string analysis in this book but refer the reader to [184].

Although Cole's proof for the linear worst case is vastly simpler than earlier proofs, and is important in order to complete the full story of Boyer–Moore, it is not trivial. However, a fairly simple extension of the Boyer–Moore algorithm, due to Apostolico and Giancarlo [26], gives a "Boyer–Moore–like" algorithm that allows a fairly direct proof of a $2m$ worst-case bound on the number of comparisons. The Apostolico–Giancarlo variant of Boyer–Moore is discussed in Section 3.1.

2.3. The Knuth-Morris-Pratt algorithm

The best known linear-time algorithm for the exact matching problem is due to Knuth, Morris, and Pratt [278]. Although it is rarely the method of choice, and is often much inferior in practice to the Boyer–Moore method (and others), it can be simply explained, and its linear time bound is (fairly) easily proved. The algorithm also forms the basis of the well-known Aho–Corasick algorithm, which efficiently finds all occurrences in a text of any pattern from a *set* of patterns.[3]

2.3.1. The Knuth-Morris-Pratt shift idea

For a given alignment of P with T, suppose the naive algorithm matches the first i characters of P against their counterparts in T and then mismatches on the next comparison. The naive algorithm would shift P by just *one* place and begin comparing again from the left end of P. But a larger shift may often be possible. For example, if $P = abcxabcde$ and, in the present alignment of P with T, the mismatch occurs in position 8 of P, then it is easily deduced (and we will prove below) that P can be shifted by four places without passing over any occurrences of P in T. Notice that this can be deduced without even knowing what string T is or exactly how P is aligned with T. Only the location of the mismatch in P must be known. The Knuth-Morris-Pratt algorithm is based on this kind of reasoning to make larger shifts than the naive algorithm makes. We now formalize this idea.

[3] We will present several solutions to that set problem including the Aho–Corasick method in Section 3.4. For those reasons, and for its historical role in the field, we fully develop the Knuth-Morris-Pratt method here.

Definition For each position i in pattern P, define $sp_i(P)$ to be the *length* of the longest proper *suffix* of $P[1..i]$ that matches a prefix of P.

Stated differently, $sp_i(P)$ is the length of the longest proper substring of $P[1..i]$ that ends at i and that matches a prefix of P. When the string is clear by context we will use sp_i in place of the full notation.

For example, if $P = abcaeabcabd$, then $sp_2 = sp_3 = 0$, $sp_4 = 1$, $sp_8 = 3$, and $sp_{10} = 2$. Note that by definition, $sp_1 = 0$ for any string.

An optimized version of the Knuth-Morris-Pratt algorithm uses the following values.

Definition For each position i in pattern P, define $sp'_i(P)$ to be the length of the longest proper suffix of $P[1..i]$ that matches a prefix of P, *with the added condition that characters $P(i+1)$ and $P(sp'_i + 1)$ are unequal.*

Clearly, $sp'_i(P) \le sp_i(P)$ for all positions i and any string P. As an example, if $P = bbccaebbcabd$, then $sp_8 = 2$ because string bb occurs both as a proper prefix of $P[1..8]$ and as a suffix of $P[1..8]$. However, both copies of the string are followed by the same character c, and so $sp'_8 < 2$. In fact, $sp'_8 = 1$ since the single character b occurs as both the first and last character of $P[1..8]$ and is followed by character b in position 2 and by character c in position 9.

The Knuth-Morris-Pratt shift rule

We will describe the algorithm in terms of the sp' values, and leave it to the reader to modify the algorithm if only the weaker sp values are used.[4] The Knuth-Morris-Pratt algorithm aligns P with T and then compares the aligned characters from *left to right*, as the naive algorithm does.

For any alignment of P and T, if the first mismatch (comparing from left to right) occurs in position $i+1$ of P and position k of T, then shift P to the right (relative to T) so that $P[1..sp'_i]$ aligns with $T[k - sp'_i..k-1]$. In other words, shift P exactly $i + 1 - (sp'_i + 1) = i - sp'_i$ places to the right, so that character $sp'_i + 1$ of P will align with character k of T. In the case that an occurrence of P has been found (no mismatch), shift P by $n - sp'_n$ places.

The shift rule guarantees that the prefix $P[1..sp'_i]$ of the shifted P matches its opposing substring in T. The next comparison is then made between characters $T(k)$ and $P[sp'_i + 1]$. The use of the stronger shift rule based on sp'_i guarantees that the same mismatch will not occur again in the new alignment, but it does not guarantee that $T(k) = P[sp'_i + 1]$.

In the above example, where $P = abcxabcde$ and $sp'_7 = 3$, if character 8 of P mismatches then P will be shifted by $7 - 3 = 4$ places. This is true even without knowing T or how P is positioned with T.

The advantage of the shift rule is twofold. First, it often shifts P by more than just a single character. Second, after a shift, the left-most sp'_i characters of P are guaranteed to match their counterparts in T. Thus, to determine whether the newly shifted P matches its counterpart in T, the algorithm can start comparing P and T at position $sp'_i + 1$ of P (and position k of T). For example, suppose $P = abcxabcde$ as above, $T = xyabcxabcxadcdqfeg$, and the left end of P is aligned with character 3 of T. Then P and T will match for 7 characters but mismatch on character 8 of P, and P will be shifted

[4] The reader should be alerted that traditionally the Knuth-Morris-Pratt algorithm has been described in terms of *failure functions*, which are related to the sp_i values. Failure functions will be explicitly defined in Section 2.3.3.

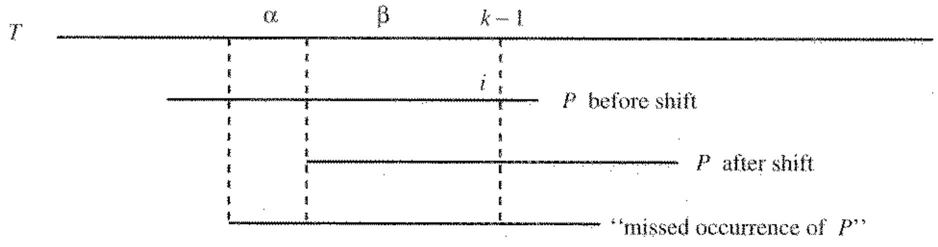

Figure 2.2: Assumed missed occurrence used in correctness proof for Knuth-Morris-Pratt.

by 4 places as shown below:

```
1         2
123456789012345678
xyabcxabcxadcdqfeg
  abcxabcde
      abcxabcde
```

As guaranteed, the first 3 characters of the shifted P match their counterparts in T (and their counterparts in the unshifted P).

Summarizing, we have

Theorem 2.3.1. *After a mismatch at position $i+1$ of P and a shift of $i - sp'_i$ places to the right, the left-most sp'_i characters of P are guaranteed to match their counterparts in T.*

Theorem 2.3.1 partially establishes the correctness of the Knuth-Morris-Pratt algorithm, but to fully prove correctness we have to show that the shift rule never shifts too far. That is, using the shift rule no occurrence of P will ever be overlooked.

Theorem 2.3.2. *For any alignment of P with T, if characters 1 through i of P match the opposing characters of T but character $i+1$ mismatches $T(k)$, then P can be shifted by $i - sp'_i$ places to the right without passing any occurrence of P in T.*

PROOF Suppose not, so that there is an occurrence of P starting strictly to the left of the shifted P (see Figure 2.2), and let α and β be the substrings shown in the figure. In particular, β is the prefix of P of length sp'_i, shown relative to the shifted position of P. The unshifted P matches T up through position i of P and position $k-1$ of T, and all characters in the (assumed) missed occurrence of P match their counterparts in T. Both of these matched regions contain the substrings α and β, so the unshifted P and the assumed occurrence of P match on the entire substring $\alpha\beta$. Hence $\alpha\beta$ is a suffix of $P[1..i]$ that matches a proper prefix of P. Now let $l = |\alpha\beta| + 1$ so that position l in the "missed occurrence" of P is opposite position k in T. Character $P(l)$ cannot be equal to $P(i+1)$ since $P(l)$ is assumed to match $T(k)$ and $P(i+1)$ does not match $T(k)$. Thus $\alpha\beta$ is a proper suffix of $P[1..i]$ that matches a prefix of P, and the next character is unequal to $P(i+1)$. But $|\alpha| > 0$ due to the assumption that an occurrence of P starts strictly before the shifted P, so $|\alpha\beta| > |\beta| = sp'_i$, contradicting the definition of sp'_i. Hence the theorem is proved. □

Theorem 2.3.2 says that the Knuth-Morris-Pratt shift rule does not miss any occurrence of P in T, and so the Knuth-Morris-Pratt algorithm will correctly find all occurrences of P in T. The time analysis is equally simple.

Theorem 2.3.3. *In the Knuth-Morris-Pratt method, the number of character comparisons is at most 2m.*

PROOF Divide the algorithm into compare/shift phases, where a single phase consists of the comparisons done between successive shifts. After any shift, the comparisons in the phase go left to right and start either with the last character of T compared in the previous phase or with the character to its right. Since P is never shifted left, in any phase at most one comparison involves a character of T that was previously compared. Thus, the total number of character comparisons is bounded by $m + s$, where s is the number of shifts done in the algorithm. But $s < m$ since after m shifts the right end of P is certainly to the right of the right end of T, so the number of comparisons done is bounded by $2m$. □

2.3.2. Preprocessing for Knuth-Morris-Pratt

The key to the speed up of the Knuth-Morris-Pratt algorithm over the naive algorithm is the use of sp' (or sp) values. It is easy to see how to compute all the sp' and sp values from the Z values obtained during the fundamental preprocessing of P. We verify this below.

Definition Position $j > 1$ *maps to* i if $i = j + Z_j(P) - 1$. That is, j maps to i if i is the right end of a Z-box starting at j.

Theorem 2.3.4. *For any $i > 1$, $sp'_i(P) = Z_j = i - j + 1$, where $j > 1$ is the smallest position that maps to i. If there is no such j then $sp'_i(P) = 0$. For any $i > 1$, $sp_i(P) = i - j + 1$, where j is the smallest position in the range $1 < j \leq i$ that maps to i or beyond. If there is no such j, then $sp_i(P) = 0$.*

PROOF If $sp'_i(P)$ is greater than zero, then there is a proper suffix α of $P[1..i]$ that matches a prefix of P, such that $P[i + 1]$ does not match $P[|\alpha| + 1]$. Therefore, letting j denote the start of α, $Z_j = |\alpha| = sp'_i(P)$ and j maps to i. Hence, if there is no j in the range $1 < j \leq i$ that maps to i, then $sp'_i(P)$ must be zero.

Now suppose $sp'_i(P) > 0$ and let j be as defined above. We claim that j is the smallest position in the range 2 to i that maps to i. Suppose not, and let j^* be a position in the range $1 < j^* < j$ that maps to i. Then $P[j^*..i]$ would be a proper suffix of $P[1..i]$ that matches a prefix (call it β) of P. Moreover, by the definition of mapping, $P(i + 1) \neq P(|\beta|)$, so $sp'_i(P) \geq |\beta| > |\alpha|$, contradicting the assumption that $sp'_i = \alpha$.

The proofs of the claims for $sp_i(P)$ are similar and are left as exercises. □

Given Theorem 2.3.4, all the sp' and sp values can be computed in linear time using the Z_i values as follows:

Z-based Knuth-Morris-Pratt

```
for $i := 1$ to $n$ do
    $sp'_i := 0$;
for $j := n$ downto 2 do
begin
    $i := j + Z_j(P) - 1$;
    $sp'_i := Z_j$;
end;
```

The sp values are obtained by adding the following:

```
$sp_n(P) := sp'_n(P)$;
for $i := n - 1$ downto 2 do
    $sp_i(P) := \max[sp_{i+1}(P) - 1, sp'_i(P)]$
```

2.3.3. A full implementation of Knuth-Morris-Pratt

We have described the Knuth-Morris-Pratt algorithm in terms of shifting P, but we never accounted for time needed to implement shifts. The reason is that shifting is only conceptual and P is never explicitly shifted. Rather, as in the case of Boyer–Moore, pointers to P and T are incremented. We use pointer p to point into P and one pointer c (for "current" character) to point into T.

Definition For each position i from 1 to $n + 1$, define the failure function $F'(i)$ to be $sp'_{i-1} + 1$ (and define $F(i) = sp_{i-1} + 1$), where sp'_0 and sp_0 are defined to be zero.

We will only use the (stronger) failure function $F'(i)$ in this discussion but will refer to $F(i)$ later.

After a mismatch in position $i + 1 > 1$ of P, the Knuth-Morris-Pratt algorithm "shifts" P so that the next comparison is between the character in position c of T and the character in position $sp'_i + 1$ of P. But $sp'_i + 1 = F'(i + 1)$, so a general "shift" can be implemented in constant time by just setting p to $F'(i + 1)$. Two special cases remain. When the mismatch occurs in position 1 of P, then p is set to $F'(1) = 1$ and c is incremented by one. When an occurrence of P is found, then P is shifted right by $n - sp'_n$ places. This is implemented by setting $F'(n + 1)$ to $sp'_n + 1$.

Putting all the pieces together gives the full Knuth-Morris-Pratt algorithm.

Knuth-Morris-Pratt algorithm

```
begin
Preprocess $P$ to find $F'(k) = sp'_{k-1} + 1$ for $k$ from 1 to $n + 1$.
    $c := 1$;
    $p := 1$;
    While $c + (n - p) \leq m$
    do begin
        While $P(p) = T(c)$ and $p \leq n$
        do begin
            $p := p + 1$;
            $c := c + 1$;
        end;
    if $p = n + 1$ then
        report an occurrence of $P$ starting at position $c - n$ of $T$.
    if $p := 1$ then $c := c + 1$
    $p := F'(p)$;
    end;
end.
```

2.4. Real-time string matching

In the search stage of the Knuth-Morris-Pratt algorithm, P is aligned against a substring of T and the two strings are compared left to right until either all of P is exhausted (in which case an occurrence of P in T has been found) or until a mismatch occurs at some positions $i+1$ of P and k of T. In the latter case, if $sp'_i > 0$, then P is shifted right by $i - sp'_i$ positions, guaranteeing that the prefix $P[1..sp'_i]$ of the shifted pattern matches its opposing substring in T. No explicit comparison of those substrings is needed, and the next comparison is between characters $T(k)$ and $P(sp'_i + 1)$. Although the shift based on sp'_i guarantees that $P(i+1)$ differs from $P(sp'_i + 1)$, it does not guarantee that $T(k) = P(sp'_i + 1)$. Hence $T(k)$ might be compared several times (perhaps $\Omega(|P|)$ times) with differing characters in P. For that reason, the Knuth-Morris-Pratt method is not a *real-time* method.

To be real time, a method must do at most a *constant* amount of work between the time it first examines any position in T and the time it last examines that position. In the Knuth-Morris-Pratt method, if a position of T is involved in a match, it is never examined again (this is easy to verify) but, as indicated above, this is not true when the position is involved in a mismatch. Note that the definition of real time only concerns the search stage of the algorithm. Preprocessing of P need not be real time. Note also that if the search stage is real time it certainly is also linear time.

The utility of a real-time matcher is two fold. First, in certain applications, such as when the characters of the text are being sent to a small memory machine, one might need to guarantee that each character can be fully processed before the next one is due to arrive. If the processing time for each character is constant, independent of the length of the string, then such a guarantee may be possible. Second, in this particular real-time matcher, the shifts of P may be longer but never shorter than in the original Knuth-Morris-Pratt algorithm. Hence, the real-time matcher may run faster in certain problem instances.

Admittedly, arguments in favor of real-time matching algorithms over linear-time methods are somewhat tortured, and the real-time matching is more a theoretical issue than a practical one. Still, it seems worthwhile to spend a little time discussing real-time matching.

2.4.1. Converting Knuth-Morris-Pratt to a real-time method

We will use the Z values obtained during fundamental preprocessing of P to convert the Knuth-Morris-Pratt method into a real-time method. The required preprocessing of P is quite similar to the preprocessing done in Section 2.3.2 for the Knuth-Morris-Pratt algorithm. For historical reasons, the resulting real-time method is generally referred to as a *deterministic finite-state string matcher* and is often represented with a finite state machine diagram. We will not use this terminology here and instead represent the method in pseudo code.

Definition Let x denote a character of the alphabet. For each position i in pattern P, define $sp'_{(i,x)}(P)$ to be the length of the longest proper suffix of $P[1..i]$ that matches a prefix of P, *with the added condition that character* $P(sp'_i + 1)$ *is* x.

Knowing the $sp'_{(i,x)}$ values for each character x in the alphabet allows a shift rule that converts the Knuth-Morris-Pratt method into a real-time algorithm. Suppose P is compared against a substring of T and a mismatch occurs at characters $T(k) = x$ and $P(i+1)$. Then P should be shifted right by $i - sp'_{(i,x)}$ places. This shift guarantees that the prefix $P[1..sp'_{(i,x)}]$ matches the opposing substring in T and that $T(k)$ matches the next character in P. Hence, the comparison between $T(k)$ and $P(sp'_{(i,x)} + 1)$ can be skipped. The next needed comparison is between characters $P(sp'_{(i,x)} + 2)$ and $T(k+1)$. With this shift rule, the method becomes real time because it still never reexamines a position in T involved in a match (a feature inherited from the Knuth-Morris-Pratt algorithm), and it now also never reexamines a position involved in a mismatch. So, the search stage of this algorithm never examines a character in T more than once. It follows that the search is done in real time. Below we show how to find all the $sp'_{(i,x)}$ values in linear time. Together, this gives an algorithm that does linear preprocessing of P and real-time search of T.

It is easy to establish that the algorithm finds all occurrences of P in T, and we leave that as an exercise.

2.4.2. Preprocessing for real-time string matching

Theorem 2.4.1. *For $P[i+1] \neq x$, $sp'_{(i,x)}(P) = i - j + 1$, where j is the smallest position such that j maps to i and $P(Z_j + 1) = x$. If there is no such j then $sp'_{(i,x)}(P) = 0$.*

The proof of this theorem is almost identical to the proof of Theorem 2.3.4 (page 26) and is left to the reader. Assuming (as usual) that the alphabet is finite, the following minor modification of the preprocessing given earlier for Knuth-Morris-Pratt (Section 2.3.2) yields the needed $sp'_{(i,x)}$ values in linear time:

Z-based real-time matching

```
for $i := 1$ to $n$ do
    $sp'_{(i,x)} := 0$ for every character $x$;
for $j := n$ downto 2 do
    begin
        $i := j + Z_j(P) - 1$;
        $x := P(Z_j + 1)$;
        $sp'_{(i,x)} := Z_j$;
    end;
```

Note that the linear time (and space) bound for this method require that the alphabet Σ be finite. This allows us to do $|\Sigma|$ comparisons in constant time. If the size of the alphabet is explicitly included in the time and space bounds, then the preprocessing time and space needed for the algorithm is $O(|\Sigma|n)$.

2.5. Exercises

1. In "typical" applications of exact matching, such as when searching for an English word in a book, the simple bad character rule seems to be as effective as the extended bad character rule. Give a "hand-waving" explanation for this.
2. When searching for a single word or a small phrase in a large English text, brute force (the naive algorithm) is reported [184] to run faster than most other methods. Give a hand-waving explanation for this. In general terms, how would you expect this observation to hold up with smaller alphabets (say in DNA with an alphabet size of four), as the size of the pattern grows, and when the text has many long sections of similar but not exact substrings?
3. "Common sense" and the $\Theta(nm)$ worst-case time bound of the Boyer–Moore algorithm (using only the bad character rule) both would suggest that empirical running times increase with increasing pattern length (assuming a fixed text). But when searching in actual English

3.3. The original preprocessing for Knuth-Morris-Pratt

3.3.1. The method does not use fundamental preprocessing

In Section 1.3 we showed how to compute all the sp_i values from Z_i values obtained during fundamental preprocessing of P. The use of Z_i values was conceptually simple and allowed a uniform treatment of various preprocessing problems. However, the classical preprocessing method given in Knuth-Morris-Pratt [278] is not based on fundamental preprocessing. The approach taken there is very well known and is used or extended in several additional methods (such as the Aho–Corasick method that is discussed next). For those reasons, a serious student of string algorithms should also understand the classical algorithm for Knuth-Morris-Pratt preprocessing.

The preprocessing algorithm computes $sp_i(P)$ for each position i from $i = 2$ to $i = n$ (sp_1 is zero). To explain the method, we focus on how to compute sp_{k+1} assuming that sp_i is known for each $i \leq k$. The situation is shown in Figure 3.9, where string α is the prefix of P of length sp_k. That is, α is the longest string that occurs both as a proper prefix of P and as a substring of P ending at position k. For clarity, let α' refer to the copy of α that ends at position k.

Let x denote character $k+1$ of P, and let $\beta = \bar{\beta}x$ denote the prefix of P of length sp_{k+1} (i.e., the prefix that the algorithm will next try to compute). Finding sp_{k+1} is equivalent to finding string $\bar{\beta}$. And clearly,

*) $\bar{\beta}$ is the longest proper prefix of $P[1..k]$ that matches a suffix of $P[1..k]$ and that is followed by character x in position $|\bar{\beta}| + 1$ of P. See Figure 3.10.

Our goal is to find sp_{k+1}, or equivalently, to find $\bar{\beta}$.

3.3.2. The easy case

Suppose the character just after α is x (i.e., $P(sp_k + 1) = x$). Then, string αx is a prefix of P and also a proper suffix of $P[1..k+1]$, and thus $sp_{k+1} \geq |\alpha x| = sp_k + 1$. Can we then end our search for sp_{k+1} concluding that sp_{k+1} equals $sp_k + 1$, or is it possible for sp_{k+1} to be strictly greater than $sp_k + 1$? The next lemma settles this.

Lemma 3.3.1. *For any k, $sp_{k+1} \leq sp_k + 1$. Further, $sp_{k+1} = sp_k + 1$ if and only if the character after α is x. That is, $sp_{k+1} = sp_k + 1$ if and only if $P(sp_k + 1) = P(k+1)$.*

PROOF Let $\beta = \bar{\beta}x$ denote the prefix of P of length sp_{k+1}. That is, $\beta = \bar{\beta}x$ is the longest proper suffix of $P[1..k+1]$ that is a prefix of P. If sp_{k+1} is strictly greater than

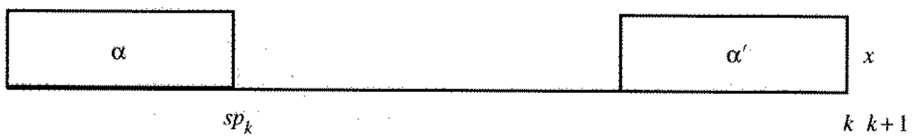

Figure 3.9: The situation after finding sp_k.

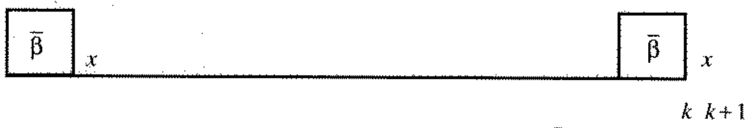

Figure 3.10: sp_{k+1} is found by finding $\bar{\beta}$.

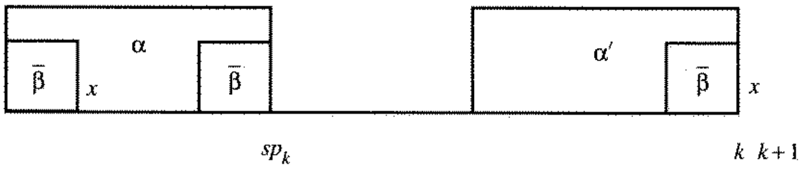

Figure 3.11: $\bar{\beta}$ must be a suffix of α.

$sp_k + 1 = |\alpha| + 1$, then $\bar{\beta}$ would be a prefix of P that is longer than α. But $\bar{\beta}$ is also a proper suffix of $P[1..k]$ (because $\bar{\beta}x$ is a proper suffix of $P[1..k+1]$). Those two facts would contradict the definition of sp_k (and the selection of α). Hence $sp_{k+1} \le sp_k + 1$.

Now clearly, $sp_{k+1} = sp_k + 1$ if the character to the right of α is x, since αx would then be a prefix of P that also occurs as a proper suffix of $P[1..k+1]$. Conversely, if $sp_{k+1} = sp_k + 1$ then the character after α must be x. □

Lemma 3.3.1 identifies the largest "candidate" value for sp_{k+1} and suggests how to initially look for that value (and for string β). We should first check the character $P(sp_k+1)$, just to the right of α. If it equals $P(sp_k + 1)$ then we conclude that $\bar{\beta}$ equals α, β is αx, and sp_{k+1} equals $sp_k + 1$. But what do we do if the two characters are not equal?

3.3.3. The general case

When character $P(k+1) \ne P(sp_k + 1)$, then $sp_{k+1} < sp_k + 1$ (by Lemma 3.3.1), so $sp_{k+1} \le sp_k$. It follows that β must be a prefix of α, and $\bar{\beta}$ must be a *proper* prefix of α. Now substring $\beta = \bar{\beta}x$ ends at position $k+1$ and is of length at most sp_k, whereas α' is a substring ending at position k and is of length sp_k. So $\bar{\beta}$ is a suffix of α', as shown in Figure 3.11. But since α' is a copy of α, $\bar{\beta}$ is also a suffix of α.

In summary, when $P(k+1) \ne P(sp_k + 1)$, $\bar{\beta}$ occurs as a suffix of α and also as a proper prefix of α followed by character x. So when $P(k+1) \ne P(sp_k+1)$, $\bar{\beta}$ is the longest proper prefix of α that matches a suffix of α and that is followed by character x in position $|\bar{\beta}| + 1$ of P. See Figure 3.11.

However, since $\alpha = P[1..sp_k]$, we can state this as

**) $\bar{\beta}$ is the longest proper prefix of $P[1..sp_k]$ that matches a suffix of $P[1..k]$ and that is followed by character x in position $|\bar{\beta}| + 1$ of P.

The general reduction

Statements $*$ and $**$ differ only by the substitution of $P[1..sp_k]$ for $P[1..k]$ and are otherwise exactly the same. Thus, when $P(sp_k + 1) \ne P(k+1)$, the problem of finding $\bar{\beta}$ reduces to another instance of the original problem but on a smaller string ($P[1..sp_k]$ in place of $P[1..k]$). We should therefore proceed as before. That is, to search for $\bar{\beta}$ the algorithm should find the longest proper prefix of $P[1..sp_k]$ that matches a suffix of $P[1..sp_k]$ and then check whether the character to the right of that prefix is x. By the definition of sp_k, the required prefix ends at character sp_{sp_k}. So if character $P(sp_{sp_k}+1) = x$ then we have found $\bar{\beta}$, or else we recurse again, restricting our search to ever smaller prefixes of P. Eventually, either a valid prefix is found, or the beginning of P is reached. In the latter case, $sp_{k+1} = 1$ if $P(1) = P(k+1)$; otherwise $sp_{k+1} = 0$.

The complete preprocessing algorithm

Putting all the pieces together gives the following algorithm for finding $\bar{\beta}$ and sp_{k+1}:

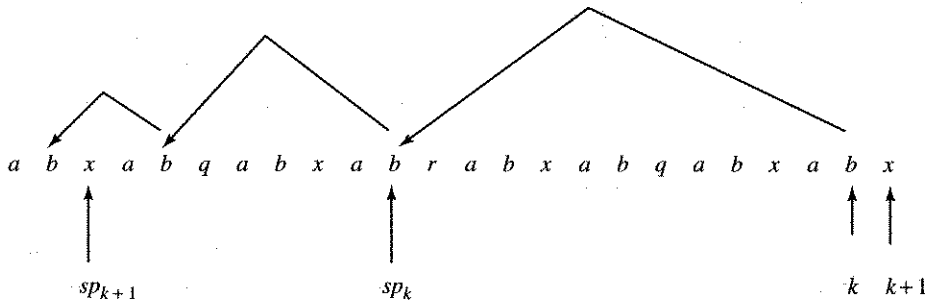

Figure 3.12: "Bouncing ball" cartoon of original Knuth-Morris-Pratt preprocessing. The arrows show the successive assignments to the variable *v*.

How to find sp_{k+1}

```
    $x := P(k+1)$;
    $v := sp_k$;
    While $P(v+1) \neq x$ and $v \neq 0$ do
        $v := sp_v$;
    end;
    If $P(v+1) = x$ then
        $sp_{k+1} := v+1$
    else
        $sp_{k+1} := 0$;
```

See the example in Figure 3.12.

The entire set of *sp* values are found as follows:

Algorithm SP(P)

```
$sp_1 = 0$
For $k := 1$ to $n-1$ do
begin
    $x := P(k+1)$;
    $v := sp_k$;
    While $P(v+1) \neq x$ and $v \neq 0$ do
        $v := sp_v$;
    end;
    If $P(v+1) = x$ then
        $sp_{k+1} := v+1$
    else
        $sp_{k+1} := 0$;
end;
```

Theorem 3.3.1. *Algorithm SP finds all the $sp_i(P)$ values in $O(n)$ time, where n is the length of P.*

PROOF Note first that the algorithm consists of two nested loops, a *for* loop and a *while* loop. The *for* loop executes exactly $n-1$ times, incrementing the value of k each time. The *while* loop executes a variable number of times each time it is entered.

The work of the algorithm is proportional to the number of times the value of v is assigned. We consider the places where the value of v is assigned and focus on how the value of v changes over the execution of the algorithm. The value of v is assigned once each time the *for* statement is reached; it is assigned a variable number of times inside the *while* loop, each time this loop is reached. Hence the number of times v is assigned is $n - 1$ plus the number of times it is assigned inside the *while* loop. How many times that can be is the key question.

Each assignment of v inside the *while* loop must decrease the value of v, and each of the $n - 1$ times v is assigned at the *for* statement, its value either increases by one or it remains unchanged (at zero). The value of v is initially zero, so the total amount that the value of v can increase (at the *for* statement) over the entire algorithm is at most $n - 1$. But since the value of v starts at zero and is never negative, the total amount that the value of v can *decrease* over the entire algorithm must also be bounded by $n - 1$, the total amount it can increase. Hence v can be assigned in the *while* loop at most $n - 1$ times, and hence the total number of times that the value of v can be assigned is at most $2(n - 1) = O(n)$, and the theorem is proved. □

3.3.4. How to compute the optimized shift values

The (stronger) sp'_i values can be easily computed from the sp_i values in $O(n)$ time using the algorithm below. For the purposes of the algorithm, character $P(n + 1)$, which does not exist, is defined to be different from any character in P.

Algorithm SP′(P)

```
$sp'_1 = 0$;
For $i := 2$ to $n$ do
begin
    $v := sp_i$;
    If $P(v + 1) \neq P(i + 1)$ then
        $sp'_i := v$
    else
        $sp'_i := sp'_v$;
end;
```

Theorem 3.3.2. *Algorithm $SP'(P)$ correctly computes all the sp'_i values in $O(n)$ time.*

PROOF The proof is by induction on the value of i. Since $sp_1 = 0$ and $sp'_i \leq sp_i$ for all i, then $sp'_1 = 0$, and the algorithm is correct for $i = 1$. Now suppose that the value of sp'_i set by the algorithm is correct for all $i < k$ and consider $i = k$. If $P[sp_k + 1] \neq P[k + 1]$ then clearly sp'_k is equal to sp_k, since the sp_k length prefix of $P[1..k]$ satisfies all the needed requirements. Hence in this case, the algorithm correctly sets sp'_k.

If $P(sp_k + 1) = P(k + 1)$, then $sp'_k < sp_k$ and, since $P[1..sp_k]$ is a suffix $P[1..k]$, sp'_k can be expressed as the length of the longest proper prefix of $P[1..sp_k]$ that also occurs as a suffix of $P[1..sp_k]$ with the condition that $P(k + 1) \neq P(sp'_k + 1)$. But since $P(k + 1) = P(sp_k + 1)$, that condition can be rewritten as $P(sp_k + 1) \neq P(sp'_k + 1)$. By the induction hypothesis, that value has already been correctly computed as sp'_{sp_k}. So when $P(sp_k + 1) = P(k + 1)$ the algorithm correctly sets sp'_k to sp'_{sp_k}.

Because the algorithm only does constant work per position, the total time for the algorithm is $O(n)$. □

It is interesting to compare the classical method for computing sp and sp' and the method based on fundamental preprocessing (i.e., on Z values). In the classical method the (weaker) sp values are computed first and then the more desirable sp' values are derived

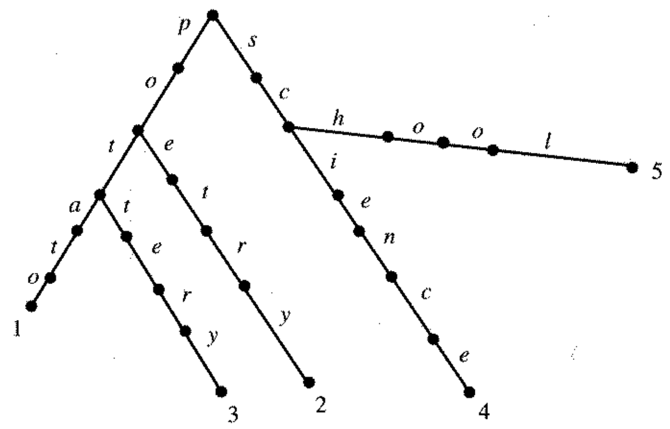

Figure 3.13: Keyword tree $\mathcal{K}$ with five patterns.

from them, whereas the order is just the opposite in the method based on fundamental preprocessing.

3.4. Exact matching with a set of patterns

An immediate and important generalization of the exact matching problem is to find all occurrences in text T of any pattern in a *set* of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$. This generalization is called the *exact set matching* problem. Let n now denote the total length of all the patterns in $\mathcal{P}$ and m be, as before, the length of T. Then, the exact set matching problem can be solved in time $O(n + zm)$ by separately using any linear-time method for each of the z patterns.

Perhaps surprisingly, the exact set matching problem can be solved faster than, $O(n + zm)$. It can be solved in $O(n + m + k)$ time, where k is the number of occurrences in T of the patterns from $\mathcal{P}$. The first method to achieve this bound is due to Aho and Corasick [9].[2] In this section, we develop the Aho–Corasick method; some of the proofs are left to the reader. An equally efficient, but more robust, method for the exact set matching problem is based on suffix trees and is discussed in Section 7.2.

Definition The *keyword tree* for set $\mathcal{P}$ is a rooted directed tree $\mathcal{K}$ satisfying three conditions: 1. each edge is labeled with exactly one character; 2. any two edges out of the same node have distinct labels; and 3. every pattern P_i in $\mathcal{P}$ maps to some node v of $\mathcal{K}$ such that the characters on the path from the root of $\mathcal{K}$ to v exactly spell out P_i, and every leaf of $\mathcal{K}$ is mapped to by some pattern in $\mathcal{P}$.

For example, Figure 3.13 shows the keyword tree for the set of patterns {*potato, poetry, pottery, science, school*}.

Clearly, every node in the keyword tree corresponds to a prefix of one of the patterns in $\mathcal{P}$, and every prefix of a pattern maps to a distinct node in the tree.

Assuming a fixed-size alphabet, it is easy to construct the keyword tree for $\mathcal{P}$ in $O(n)$ time. Define $\mathcal{K}_i$ to be the (partial) keyword tree that encodes patterns $P_1, \ldots, P_i$ of $\mathcal{P}$.

[2] There is a more recent exposition of the Aho–Corasick method in [8], where the algorithm is used just as an "acceptor", deciding whether or not there is an occurrence in T of at least one pattern from $\mathcal{P}$. Because we will want to explicitly find all occurrences, that version of the algorithm is too limited to use here.