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

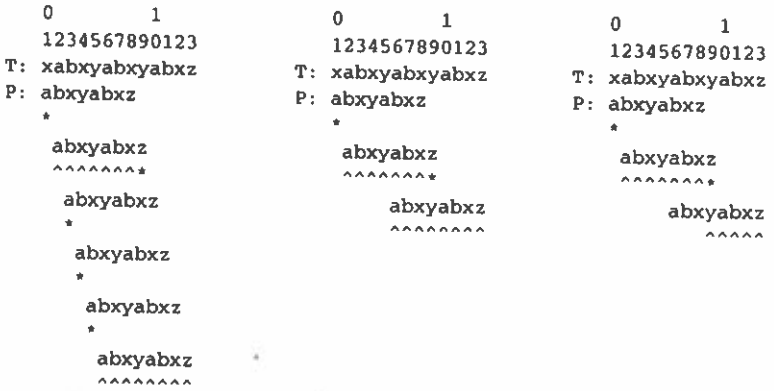

Figure 1.1: The first scenario illustrates pure naive matching, and the next two illustrate smarter shifts. A caret beneath a character indicates a match and a star indicates a mismatch made by the algorithm.

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

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

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

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

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

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

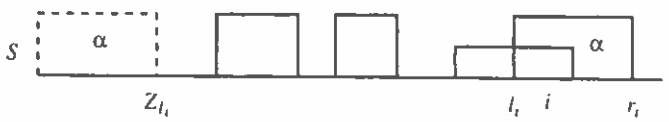

Figure 1.2: Each solid box represents a substring of S that matches a prefix of S and that starts between positions 2 and i. Each box is called a Z-*box*. We use r_i to denote the *right-most* end of any Z-box that begins at or to the left of position i and α to denote the substring in the Z-box ending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

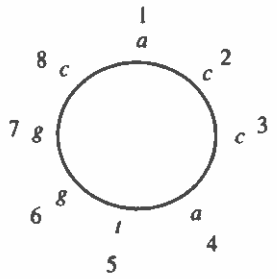

Figure 1.6: A circular string β. The linear string $\bar{\beta}$ derived from it is *accatggc*.

problem is the following. Let $\bar{\beta}$ be the *linear* string obtained from β starting at character 1 and ending at character n. Then α is a substring of circular string β if and only if α is a substring of some circular rotation of $\bar{\beta}$.

A digression on circular strings in DNA

The above two problems are mostly exercises in using the existence of a linear-time exact matching algorithm, and we don't know any critical biological problems that they address. However, we want to point out that *circular DNA* is common and important. Bacterial and mitochondrial DNA is typically circular, both in its genomic DNA and in additional small double-stranded circular DNA molecules called *plasmids*, and even some true eukaryotes (higher organisms whose cells contain a nucleus) such as yeast contain plasmid DNA in addition to their nuclear DNA. Consequently, tools for handling circular strings may someday be of use in those organisms. Viral DNA is not always circular, but even when it is linear some virus genomes exhibit circular properties. For example, in some viral populations the linear order of the DNA in one individual will be a circular rotation of the order in another individual [450]. Nucleotide mutations, in addition to rotations, occur rapidly in viruses, and a plausible problem is to determine if the DNA of two individual viruses have mutated away from each other only by a circular rotation, rather than additional mutations.

It is very interesting to note that the problems addressed in the exercises are actually "solved" in nature. Consider the special case of Exercise 2 when string α has length n. Then the problem becomes: Is α a circular rotation of $\bar{\beta}$? This problem is solved in linear time as in Exercise 1. Precisely this matching problem arises and is "solved" in *E. coli* replication under the certain experimental conditions described in [475]. In that experiment, an enzyme (RecA) and ATP molecules (for energy) are added to *E. coli* containing a single strand of one of its plasmids, called string β, and a double-stranded linear DNA molecule, one strand of which is called string α. If α is a circular rotation of $\bar{\beta}$ then the strand opposite to α (which has the DNA sequence complementary to α) hybridizes with β creating a proper double-stranded plasmid, leaving α as a single strand. This transfer of DNA may be a step in the replication of the plasmid. Thus the problem of determining whether α is a circular rotation of $\bar{\beta}$ is solved by this natural system.

Other experiments in [475] can be described as *substring* matching problems relating to circular and linear DNA in *E. coli*. Interestingly, these natural systems solve their matching problems faster than can be explained by kinetic analysis, and the molecular mechanisms used for such rapid matching remain undetermined. These experiments demonstrate the role of enzyme RecA in *E. coli* replication, but do not suggest immediate important computational problems. They do, however, provide indirect motivation for developing computational tools for handling circular strings as well as linear strings. Several other uses of circular strings will be discussed in Sections 7.13 and 16.17 of the book.

3. **Suffix-prefix matching.** Give an algorithm that takes in two strings α and β, of lengths n

and m, and finds the longest suffix of α that exactly matches a prefix of β. The algorithm should run in $O(n+m)$ time.

4. **Tandem arrays.** A substring α contained in string S is called a *tandem array* of β (called the base) if α consists of more than one consecutive copy of β. For example, if $S =$ *xyzabcabcabcabcpq*, then $\alpha =$ *abcabcabcabc* is a tandem array of $\beta =$ *abc*. Note that S also contains a tandem array of *abcabc* (i.e., a tandem array with a longer base). A *maximal* tandem array is a tandem array that cannot be extended either left or right. Given the base β, a tandem array of β in S can be described by two numbers (s, k), giving its starting location in S and the number of times β is repeated. A tandem array is an example of a repeated substring (see Section 7.11.1).

   Suppose S has length n. Give an example to show that two maximal tandem arrays of a given base β can overlap.

   Now give an $O(n)$-time algorithm that takes S and β as input, finds every maximal tandem array of β, and outputs the pair (s, k) for each occurrence. Since maximal tandem arrays of a given base can overlap, a naive algorithm would establish only an $O(n^2)$-time bound.

5. If the Z algorithm finds that $Z_2 = q > 0$, all the values $Z_3, \ldots, Z_{q+1}, Z_{q+2}$ can then be obtained immediately without additional character comparisons and without executing the main body of Algorithm Z. Flesh out and justify the details of this claim.

6. In Case 2b of the Z algorithm, when $Z_{k'} \geq |\beta|$, the algorithm does explicit comparisons until it finds a mismatch. This is a reasonable way to organize the algorithm, but in fact Case 2b can be refined so as to eliminate an unneeded character comparison. Argue that when $Z_{k'} > |\beta|$ then $Z_k = |\beta|$ and hence no character comparisons are needed. Therefore, explicit character comparisons are needed only in the case that $Z_{k'} = |\beta|$.

7. If Case 2b of the Z algorithm is split into two cases, one for $Z_{k'} > |\beta|$ and one for $Z_{k'} = |\beta|$, would this result in an overall speedup of the algorithm? You must consider all operations, not just character comparisons.

8. Baker [43] introduced the following matching problem and applied it to a problem of software maintenance: "The application is to track down duplication in a large software system. We want to find not only exact matches between sections of code, but parameterized matches, where a parameterized match between two sections of code means that one section can be transformed into the other by replacing the parameter names (e.g., identifiers and constants) of one section by the parameter names of the other via a one-to-one function".

   Now we present the formal definition. Let Σ and Π be two alphabets containing no symbols in common. Each symbol in Σ is called a *token* and each symbol in Π is called a *parameter*. A string can consist of any combinations of tokens and parameters from Σ and Π. For example, if Σ is the upper case English alphabet and Π is the lower case alphabet then *XYabCaCXZddW* is a legal string over Σ and Π. Two strings S_1 and S_2 are said to *p-match* if and only if

   a. Each token in S_1 (or S_2) is opposite a matching token in S_2 (or S_1).

   b. Each parameter in S_1 (or S_2) is opposite a parameter in S_2 (or S_1).

   c. For any parameter x, if one occurrence of x in S_1 (S_2) is opposite a parameter y in S_2 (S_1), then every occurrence of x in S_1 (S_2) must be opposite an occurrence of y in S_2 (S_1). In other words, the alignment of parameters in S_1 and S_2 defines a one-one correspondence between parameter names in S_1 and parameter names in S_2.

   For example, $S_1 =$ *XYabCaCXZddbW* p-matches $S_2 =$ *XYdxCdCXZccxW*. Notice that parameter a in S_1 maps to parameter d in S_2, while parameter d in S_1 maps to c in S_2. This does not violate the definition of p-matching.

   In Baker's application, a token represents a part of the program that cannot be changed,

whereas a parameter represents a program's variable, which can be renamed as long as all occurrences of the variable are renamed consistently. Thus if S_1 and S_2 p-match, then the variable names in S_1 could be changed to the corresponding variable names in S_2, making the two programs identical. If these two programs were part of a larger program, then they could both be replaced by a call to a single subroutine.

The most basic p-match problem is: Given a text T and a pattern P, each a string over Σ and Π, find all substrings of T that p-match P. Of course, one would like to find all those occurrences in $O(|P|+|T|)$ time. Let function Z_i^p for a string S be the length of the longest string starting at position i in S that p-matches a prefix of $S[1..i]$. Show how to modify algorithm Z to compute all the Z_i^p values in $O(|S|)$ time (the implementation details are slightly more involved than for function Z_i, but not too difficult). Then show how to use the modified algorithm Z to find all substrings of T that p-match P, in $O(|P|+|T|)$ time.

In [43] and [239], more involved versions of the p-match problem are solved by more complex methods.

The following three problems can be solved without the *Z* algorithm or other fancy tools. They only require thought.

9. You are given two strings of n characters each and an additional parameter k. In each string there are $n-k+1$ substrings of length k, and so there are $\Theta(n^2)$ pairs of substrings, where one substring is from one string and one is from the other. For a pair of substrings, we define the *match-count* as the number of opposing characters that match when the two substrings of length k are aligned. The problem is to compute the match-count for each of the $\Theta(n^2)$ pairs of substrings from the two strings. Clearly, the problem can be solved with $O(kn^2)$ operations (character comparisons plus arithmetic operations). But by better organizing the computations, the time can be reduced to $O(n^2)$ operations. (From Paul Horton.)

10. A DNA molecule can be thought of as a string over an alphabet of four characters $\{a, t, c, g\}$ (nucleotides), while a protein can be thought of as a string over an alphabet of twenty characters (amino acids). A gene, which is physically embedded in a DNA molecule, typically encodes the amino acid sequence for a particular protein. This is done as follows. Starting at a particular point in the DNA string, every three consecutive DNA characters encode a single amino acid character in the protein string. That is, three DNA nucleotides specify one amino acid. Such a coding triple is called a *codon*, and the full association of codons to amino acids is called the *genetic code*. For example, the codon *ttt* codes for the amino acid Phenylalanine (abbreviated in the single character amino acid alphabet as *F*), and the codon *gtt* codes for the amino acid Valine (abbreviated as *V*). Since there are $4^3 = 64$ possible triples but only twenty amino acids, there is a possibility that two or more triples form codons for the same amino acid and that some triples do not form codons. In fact, this is the case. For example, the amino acid Leucine is coded for by six different codons.

    Problem: Suppose one is given a DNA string of n nucleotides, but you don't know the correct "reading frame". That is, you don't know if the correct decomposition of the string into codons begins with the first, second, or third nucleotide of the string. Each such "frameshift" potentially translates into a different amino acid string. (There are actually known genes where each of the three reading frames not only specifies a string in the amino acid alphabet, but each specifies a functional, yet different, protein.) The task is to produce, for each of the three reading frames, the associated amino acid string. For example, consider the string *atggacgga*. The first reading frame has three complete codons, *atg*, *gac*, and *gga*, which in the genetic code specify the amino acids *Met*, *Asp*, and *Gly*. The second reading frame has two complete codons, *tgg* and *acg*, coding for amino acids *Trp* and *Thr*. The third reading frame has two complete codons, *gga* and *cgg*, coding for amino acids *Gly* and *Arg*.

    The goal is to produce the three translations, using the fewest number of character exami-

nations of the DNA string and the fewest number of indexing steps (when using the codons to look up amino acids in a table holding the genetic code). Clearly, the three translations can be done with $3n$ examinations of characters in the DNA and $3n$ indexing steps in the genetic code table. Find a method that does the three translations in at most n character examinations and n indexing steps.

Hint: If you are acquainted with this terminology, the notion of a finite-state transducer may be helpful, although it is not necessary.

11. Let T be a text string of length m and let $\mathcal{S}$ be a *multiset* of n characters. The problem is to find all substrings in T of length n that are formed by the characters of $\mathcal{S}$. For example, let $\mathcal{S} = \{a, a, b, c\}$ and $T = abahgcabah$. Then *caba* is a substring of T formed from the characters of $\mathcal{S}$.

Give a solution to this problem that runs in $O(m)$ time. The method should also be able to state, for each position i, the length of the longest substring in T starting at i that can be formed from $\mathcal{S}$.

Fantasy protein sequencing. The above problem may become useful in sequencing protein from a particular organism after a large amount of the genome of that organism has been sequenced. This is most easily explained in prokaryotes, where the DNA is not interrupted by introns. In prokaryotes, the amino acid sequence for a given protein is encoded in a contiguous segment of DNA – one DNA codon for each amino acid in the protein. So assume we have the protein molecule but do not know its sequence or the location of the gene that codes for the protein. Presently, chemically determining the amino acid sequence of a protein is very slow, expensive, and somewhat unreliable. However, finding the multiset of amino acids that make up the protein is relatively easy. Now suppose that the whole DNA sequence for the genome of the organism is known. One can use that long DNA sequence to determine the amino acid sequence of a protein of interest. First, translate each codon in the DNA sequence into the amino acid alphabet (this may have to be done three times to get the proper frame) to form the string T; then chemically determine the multiset $\mathcal{S}$ of amino acids in the protein; then find all substrings in T of length $|\mathcal{S}|$ that are formed from the amino acids in $\mathcal{S}$. Any such substrings are candidates for the amino acid sequence of the protein, although it is unlikely that there will be more than one candidate. The match also locates the gene for the protein in the long DNA string.

12. Consider the two-dimensional variant of the preceding problem. The input consists of two-dimensional text (say a filled-in crossword puzzle) and a multiset of characters. The problem is to find a *connected* two-dimensional substructure in the text that matches all the characters in the multiset. How can this be done? A simpler problem is to restrict the structure to be rectangular.

13. As mentioned in Exercise 10, there are organisms (some viruses for example) containing intervals of DNA encoding not just a single protein, but three viable proteins, each read in a different reading frame. So, if each protein contains n amino acids, then the DNA string encoding those three proteins is only $n + 2$ nucleotides (characters) long. That is a very compact encoding.

(Challenging problem?) Give an algorithm for the following problem: The input is a protein string S_1 (over the amino acid alphabet) of length n and another protein string of length $m > n$. Determine if there is a string specifying a DNA encoding for S_2 that contains a substring specifying a DNA encoding of S_1. Allow the encoding of S_1 to begin at any point in the DNA string for S_2 (i.e., in any reading frame of that string). The problem is difficult because of the degeneracy of the genetic code and the ability to use any reading frame.

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*

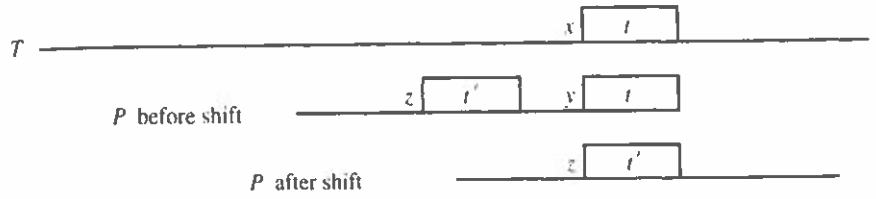

Figure 2.1: Good suffix shift rule, where character x of T mismatches with character y of P. Characters y and z of P are guaranteed to be distinct by the good suffix rule, so z has a chance of matching x.

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 ∈ Σ.
{Search stage}
    k := n;
    while k ≤ 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.

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, $sp'_i(P) \leq sp_i(P)$ 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.

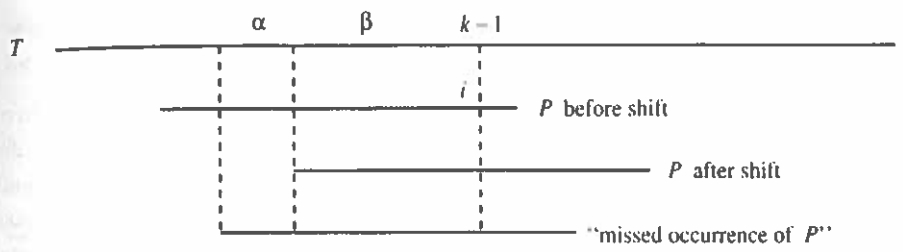

Figure 2.2: Assumed missed occurrence used in correctness proof for Knuth-Morris-Pratt.

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) ≤ m
    do begin
        While P(p) = T(c) and p ≤ 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

texts, the Boyer–Moore algorithm runs faster in practice when given longer patterns. Thus, on an English text of about 300,000 characters, it took about five times as long to search for the word "Inter" as it did to search for "Interactively".

Give a hand-waving explanation for this. Consider now the case that the pattern length increases without bound. At what point would you expect the search times to stop decreasing? Would you expect search times to start increasing at some point?

4. Evaluate empirically the utility of the extended bad character rule compared to the original bad character rule. Perform the evaluation in combination with different choices for the two good-suffix rules. How much more is the average shift using the extended rule? Does the extra shift pay for the extra computation needed to implement it?
5. Evaluate empirically, using different assumptions about the sizes of P and T, the number of occurrences of P in T, and the size of the alphabet, the following idea for speeding up the Boyer–Moore method. Suppose that a phase ends with a mismatch and that the good suffix rule shifts P farther than the extended bad character rule. Let x and y denote the mismatching characters in T and P respectively, and let z denote the character in the shifted P below x. By the suffix rule, z will not be y, but there is no guarantee that it will be x. So rather than starting comparisons from the right of the shifted P, as the Boyer–Moore method would do, why not first compare x and z? If they are equal then a right-to-left comparison is begun from the right end of P, but if they are unequal then we apply the extended bad character rule from z in P. This will shift P again. At that point we must begin a right-to-left comparison of P against T.
6. The idea of the bad character rule in the Boyer–Moore algorithm can be generalized so that instead of examining characters in P from right to left, the algorithm compares characters in P in the order of how unlikely they are to be in T (most *unlikely* first). That is, it looks first at those characters in P that are least likely to be in T. Upon mismatching, the bad character rule or extended bad character rule is used as before. Evaluate the utility of this approach, either empirically on real data or by analysis assuming random strings.
7. Construct an example where fewer comparisons are made when the bad character rule is used alone, instead of combining it with the good suffix rule.
8. Evaluate empirically the effectiveness of the strong good suffix shift for Boyer–Moore versus the weak shift rule.
9. Give a proof of Theorem 2.2.4. Then show how to accumulate all the $l'(i)$ values in linear time.
10. If we use the weak good suffix rule in Boyer–Moore that shifts the closest copy of t under the matched suffix t, but doesn't require the next character to be different, then the preprocessing for Boyer–Moore can be based directly on sp_i values rather than on Z values. Explain this.
11. Prove that the Knuth-Morris-Pratt shift rules (either based on sp or sp') do not miss any occurrences of P in T.
12. It is possible to incorporate the bad character shift rule from the Boyer–Moore method to the Knuth-Morris-Pratt method or to the naive matching method itself. Show how to do that. Then evaluate how effective that rule is and explain why it is more effective when used in the Boyer–Moore algorithm.
13. Recall the definition of l_i on page 8. It is natural to conjecture that $sp_i = i - l_i$ for any index i, where $i \geq l_i$. Show by example that this conjecture is incorrect.
14. Prove the claims in Theorem 2.3.4 concerning $sp'_i(P)$.
15. Is it true that given only the sp values for a given string P, the sp' values are completely determined? Are the sp values determined from the sp' values alone?

Using *sp* values to compute *Z* values

In Section 2.3.2, we showed that one can compute all the *sp* values knowing only the *Z* values for string *S* (i.e., not knowing *S* itself). In the next five exercises we establish the converse, creating a linear-time algorithm to compute all the *Z* values from *sp* values alone. The first exercise suggests a natural method to accomplish this, and the following exercise exposes a hole in that method. The final three exercises develop a correct linear-time algorithm, detailed in [202]. We say that sp_i *maps to* k if $k = i - sp_i + 1$.

16. Suppose there is a position i such that sp_i maps to k, and let i be the largest such position. Prove that $Z_k = i - k + 1 = sp_i$ and that $r_k = i$.

17. Given the answer to the previous exercise, it is natural to conjecture that Z_k always equals sp_i, where i is the largest position such that sp_i maps to k. Show that this is not true. Given an example using at least three distinct characters.

    Stated another way, give an example to show that Z_k can be greater than zero even when there is *no* position i such that sp_i maps to k.

18. Recall that r_{k-1} is known at the start of iteration k of the Z algorithm (when Z_k is computed), but r_k is known only at the end of iteration k. Suppose, however, that r_k is known (somehow) at the start of iteration k. Show how the Z algorithm can then be modified to compute Z_k using *no* character comparisons. Hence this modified algorithm need not even know the string S.

19. Prove that if Z_k is greater than zero, then r_k equals the largest position i such that $k \geq i - sp_i$. Conclude that r_k can be deduced from the sp values for every position k where Z_k is not zero.

20. Combine the answers to the previous two exercises to create a linear-time algorithm that computes all the Z values for a string S given only the sp values for S and not the string S itself.

    Explain in what way the method is a "simulation" of the Z algorithm.

21. It may seem that $l'(i)$ (needed for Boyer–Moore) should be sp_n for any i. Show why this is not true.

22. In Section 1.5 we showed that all the occurrences of P in T could be found in linear time by computing the Z values on the string $S = P\$T$. Explain how the method would change if we use $S = PT$, that is, we do not use a separator symbol between P and T. Now show how to find all occurrences of P in T in linear time using $S = PT$, but with sp values in place of Z values. (This is not as simple as it might at first appear.)

23. In Boyer–Moore and Boyer–Moore–like algorithms, the search moves right to left in the pattern, although the pattern moves left to right relative to the text. That makes it more difficult to explain the methods and to combine the preprocessing for Boyer–Moore with the preprocessing for Knuth-Morris-Pratt. However, a small change to Boyer–Moore would allow an easier exposition and more uniform preprocessing. First, place the pattern at the *right* end of the text, and conduct each search *left to right* in the pattern, shifting the pattern *left* after a mismatch. Work out the details of this approach, and show how it allows a more uniform exposition of the preprocessing needed for it and for Knuth-Morris-Pratt. Argue that on average this approach has the same behavior as the original Boyer–Moore method.

24. Below is working Pascal code (in Turbo Pascal) implementing Richard Cole's preprocessing, for the strong good suffix rule. It is different than the approach based on fundamental preprocessing and is closer to the original method in [278]. Examine the code to extract the algorithm behind the program. Then explain the idea of the algorithm, prove correctness of the algorithm, and analyze its running time. The point of the exercise is that it is difficult to convey an algorithmic idea using a program.

```
program gsmatch(input,output);
{This is an implementation of Richard Cole's
preprocessing for the strong good suffix rule}
type
tstring = string[200];
indexarray = array[1..100] of integer;

const
zero = 0;

var
p:tstring;
bmshift,matchshift:indexarray;
m,i:integer;

procedure readstring(var p:tstring; var m:integer);

begin
read(p);

m:=Length(p);
writeln('the length of the string is ', m);

end;

procedure gsshift(p:tstring; var
gs_shift:indexarray;m:integer);

var
i,j,j_old,k:integer;
kmp_shift:indexarray;
go_on:boolean;

begin {1}
  for j:= 1 to m do
    gs_shift[j] := m;
  kmp_shift[m]:=1;

{stage 1}
  j:=m;

  for k:=m-1 downto 1 do
    begin {2}
       go_on:=true;
       while (p[j] <> p[k]) and go_on do
          begin {3}
          if (gs_shift[j] > j-k) then gs_shift[j] := j-k;
          if (j < m) then j:= j+kmp_shift[j+1]
          else go_on:=false;
          end; {3}
```

```
      if (p[k] = p[j]) then
         begin {3}
         kmp_shift[k]:=j-k;
        j:=j-1;
         end {3}
      else
      kmp_shift[k]:=j-k+1;
   end; {2}

(stage 2)
 j:=j+1;
 j_old:=1;

 while (j <= m) do
   begin {2}
     for i:=j_old to j-1 do
     if (gs_shift[i] > j-1) then gs_shift[i]:=j-1;

     j_old:=j;
     j:=j+kmp_shift[j];
   end;  2}
end; {1}

begin {main}

writeln('input a string on a single line');

readstring(p,m);
gsshift(p,matchshift,m);
writeln('the value in cell i is the number of positions to shift');
writeln('after a mismatch occurring in position i of the pattern');

for i:= 1 to m do
write(matchshift[i]:3);
writeln;
end. {main}
```

25. Prove that the shift rule used by the real-time string matcher does not miss any occurrences of P in T.

26. Prove Theorem 2.4.1.

27. In this chapter, we showed how to use Z values to compute both the sp'_i and sp_i values used in Knuth-Morris-Pratt and the $sp'_{i,x}$ values needed for its real-time extension. Instead of using Z values for the $sp'_{i,x}$ values, show how to obtain these values from the sp_i and/or sp'_i values in linear $[O(n|\Sigma|)]$ time, where n is the length of P and $|\Sigma|$ is the length of the alphabet.

28. Although we don't know how to simply convert the Boyer–Moore algorithm to be a real-time method the way Knuth-Morris-Pratt was converted, we can make similar changes to the strong shift rule to make the Boyer–Moore shift more effective. That is, when a mismatch occurs between $P(i)$ and $T(h)$ we can look for the right-most copy in P of $P[i+1..n]$ (other than $P[i+1..n]$ itself) such that the preceding character is $T(h)$. Show how to modify

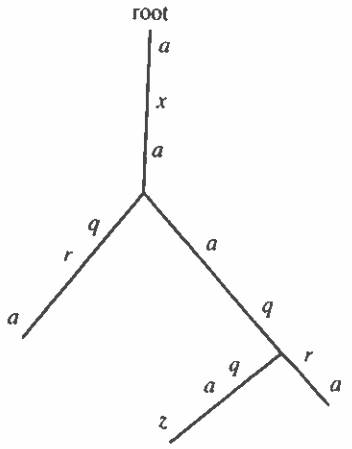

Figure 2.3: The pattern $P = aqra$ labels two subpaths of paths starting at the root. Those paths start at the root, but the subpaths containing *aqra* do not. There is also another subpath in the tree labeled *aqra* (it starts above the character *z*), but it violates the requirement that it be a subpath of a path starting at the root. Note that an edge label is displayed from the top of the edge down towards the bottom of the edge. Thus in the figure, there is an edge labeled "qra", not "arq".

the Boyer–Moore preprocessing so that the needed information is collected in linear time, assuming a fixed size alphabet.

29. Suppose we are given a tree where each edge is labeled with one or more characters, and we are given a pattern P. The label of a subpath in the tree is the concatenation of the labels on the edges in the subpath. The problem is to find all subpaths of paths starting at the root that are labeled with pattern P. Note that although the subpath must be part of a path directed from the root, the subpath itself need not start at the root (see Figure 2.3). Give an algorithm for this problem that runs in time proportional to the total number of characters on the edges of the tree plus the length of P.

3

Exact Matching: A Deeper Look at Classical Methods

3.1. A Boyer–Moore variant with a "simple" linear time bound

Apostolico and Giancarlo [26] suggested a variant of the Boyer–Moore algorithm that allows a fairly simple proof of linear worst-case running time. With this variant, no character of T will ever be compared after it is first matched with any character of P. It is then immediate that the number of comparisons is at most $2m$: Every comparison is either a match or a mismatch; there can only be m mismatches since each one results in a nonzero shift of P; and there can only be m matches since no character of T is compared again after it matches a character of P. We will also show that (in addition to the time for comparisons) the time taken for all the other work in this method is linear in m.

Given the hist(ry of very difficult and partial analyses of the Boyer–Moore algorithm, it is quite amazing that a close variant of the algorithm allows a simple linear time bound. We present here a further improvement of the Apostolico–Giancarlo idea, resulting in an algorithm that simulates *exactly* the shifts of the Boyer–Moore algorithm. The method therefore has all the rapid shifting advantages of the Boyer–Moore method as well as a simple linear worst-case time analysis.

3.1.1. Key ideas

Our version of the Apostolico–Giancarlo algorithm simulates the Boyer–Moore algorithm, finding exactly the same mismatches that Boyer–Moore would find and making exactly the same shifts. However, it infers and avoids many of the explicit matches that Boyer–Moore makes.

We take the following high-level view of the Boyer–Moore algorithm. We divide the algorithm into *compare/shift phases* numbered 1 through $q \leq m$. In a compare/shift phase, the right end of P is aligned with a character of T, and P is compared right to left with selected characters of T until either all of P is matched or until a mismatch occurs. Then, P is shifted right by some amount as given in the Boyer–Moore shift rules.

Recall from Section 2.2.4, where preprocessing for Boyer–Moore was discussed, that $N_i(P)$ is the length of the longest suffix of $P[1..i]$ that matches a suffix of P. In Section 2.2.4 we showed how to compute N_i for every i in $O(n)$ time, where n is the length of P. We assume here that vector N has been obtained during the preprocessing of P.

Two modifications of the Boyer–Moore algorithm are required. First, during the search for P in T (after the preprocessing), we maintain an m length vector M in which at most one entry is updated in every phase. Consider a phase where the right end of P is aligned with position j of T and suppose that P and T match for l places (from right to left) but no farther. Then, set $M(j)$ to a value $k \leq l$ (the rules for selecting k are detailed below). $M(j)$ records the fact that a suffix of P of length k (at least) occurs in T and ends exactly

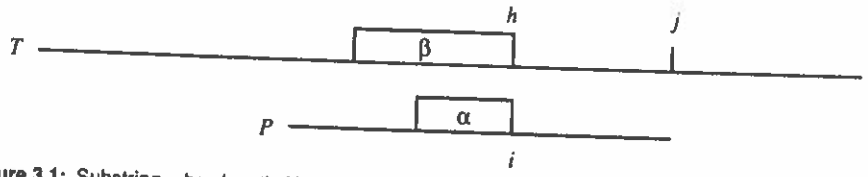

Figure 3.1: Substring α has length N_i and substring β has length $M(h) > N_i$. The two strings must match from their right ends for N_i characters, but mismatch at the next character.

at position j. As the algorithm proceeds, a value for $M(j)$ is set for every position j in T that is aligned with the right end of P; $M(j)$ is undefined for all other positions in T.

The second modification exploits the vectors N and M to speed up the Boyer–Moore algorithm by inferring certain matches and mismatches. To get the idea, suppose the Boyer–Moore algorithm is about to compare characters $P(i)$ and $T(h)$, and suppose it knows that $M(h) > N_i$ (see Figure 3.1). That means that an N_i-length substring of P ends at position i and matches a suffix of P, while an $M(h)$-length substring of T ends at position h and matches a suffix of P. So the N_i-length suffixes of those two substrings must match, and we can conclude that the next N_i comparisons (from $P(i)$ and $T(h)$ moving leftward) in the Boyer–Moore algorithm would be matches. Further, if $N_i = i$, then an occurrence of P in T has been found, and if $N_i < i$, then we can be sure that the next comparison (after the N_i matches) would be a mismatch. Hence in simulating Boyer–Moore, if $M(h) > N_i$ we can avoid at least N_i explicit comparisons. Of course, it is not always the case that $M(h) > N_i$, but all the cases are similar and are detailed below.

3.1.2. One phase in detail

As in the original Boyer–Moore algorithm, when the right end of P is aligned with a position j in T, P is compared with T from right to left. When a mismatch is found or inferred, or when an occurrence of P is found, P is shifted according to the original Boyer–Moore shift rules (either the strong or weak version) and the compare/shift phase ends. Here we will only give the details for a single phase. The phase begins with h set to j and i set to n.

Phase algorithm

1. If $M(h)$ is undefined or $M(h) = N_i = 0$, then compare $T(h)$ and $P(i)$ as follows:

   If $T(h) = P(i)$ and $i = 1$, then report an occurrence of P ending at position j of T, set $M(j) = n$, and shift as in the Boyer–Moore algorithm (ending this phase).

   If $T(h) = P(i)$ and $i > 1$, then set h to $h - 1$ and i to $i - 1$ and repeat the phase algorithm.

   If $T(h) \neq P(i)$, then set $M(j) = j - h$ and shift P according to the Boyer–Moore rules based on a mismatch occurring in position i of P (this ends the phase).

2. If $M(h) < N_i$, then P matches its counterparts in T from position n down to position $i - M(h) + 1$ of P. By the definition of $M(h)$, P might match more of T to the left, so set i to $i - M(h)$, set h to $h - M(h)$, and repeat the phase algorithm.

3. If $M(h) \geq N_i$ and $N_i = i > 0$, then declare that an occurrence of P has been found in T ending at position j. $M(j)$ must be set to a value less than or equal to n. Set $M(j)$ to $j - h$, and shift according to the Boyer–Moore rules based on finding an occurrence of P ending at j (this ends the phase).

4. If $M(h) > N_i$ and $N_i < i$, then P matches T from the right end of P down to character $i - N_i + 1$ of P, but the next pair of characters mismatch [i.e., $P(i - N_i) \neq T(h - N_i)$]. Hence P matches T for $j - h + N_i$ characters and mismatches at position $i - N_i$ of P. $M(j)$ must be set to a value less than or equal to $j - h + N_i$. Set $M(j)$ to $j - h$. Shift P by the Boyer–Moore rules based on a mismatch at position $i - N_i$ of P (this ends the phase).
5. If $M(h) = N_i$ and $0 < N_i < i$, then P and T must match for at least $M(h)$ characters to the left, but the left end of P has not yet been reached, so set i to $i - M(h)$ and set h to $h - M(h)$ and repeat the phase algorithm.

3.1.3. Correctness and linear-time analysis

Theorem 3.1.1. *Using M and N as above, the Apostolico–Giancarlo variant of the Boyer–Moore algorithm correctly finds all occurrences of P in T.*

PROOF We prove correctness by showing that the algorithm simulates the original Boyer–Moore algorithm. That is, for any given alignment of P with T, the algorithm is correct when it declares a match down to a given position and a mismatch at the next position. The rest of the simulation is correct since the shift rules are the same as in the Boyer–Moore algorithm.

Assume inductively that $M(h)$ values are valid up to some position in T. That is, wherever $M(h)$ is defined, there is an $M(h)$-length substring in T ending at position h in T that matches a suffix of P. The first such value, $M(n)$, is valid because it is found by aligning P at the left of T and making explicit comparisons, repeating rule 1 of the phase algorithm until a mismatch occurs or an occurrence of P is found. Now consider a phase where the right end of P is aligned with position j of T. The phase simulates the workings of Boyer–Moore except that in cases 2, 3, 4, and 5 certain explicit comparisons are skipped and in case 4 a mismatch is inferred, rather than observed. But whenever comparisons are skipped, they are certain to be matches by the definition of N and M and the assumption that the M values are valid thus far. Thus it is correct to skip these comparisons. In case 4, a mismatch at position $i - N_i$ of P is correctly inferred because N_i is the maximum length of any substring ending at i that matches a suffix of P, whereas $M(h)$ is less than or equal to the maximum length of any substring ending at h that matches a suffix of P. Hence this phase correctly simulates Boyer–Moore and finds exactly the same mismatch (or an occurrence of P in T) that Boyer–Moore would find. The value given to $M(j)$ is valid since in all cases it is less than or equal to the length of the suffix of P shown to match its counterpart in the substring $T[1..i]$. □

The following definitions and lemma will be helpful in bounding the work done by the algorithm.

Definition If j is a position where $M(j)$ is greater than zero then the interval $[j - M(j) + 1..j]$ is called a *covered interval* defined by j.

Definition Let $j' < j$ and suppose covered intervals are defined for both j and j'. We say that the covered intervals for j and j' *cross* if $j - M(j) + 1 \leq j'$ and $j' - M(j') + 1 < j - M(j) + 1$ (see Figure 3.2).

Lemma 3.1.1. *No covered intervals computed by the algorithm ever cross each other. Moreover, if the algorithm examines a position h of T in a covered interval, then h is at the right end of that interval.*

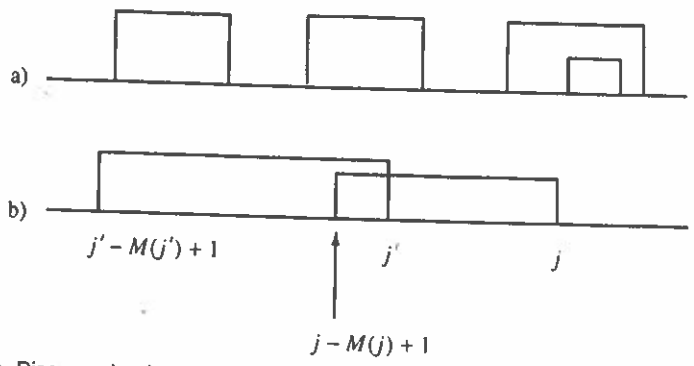

Figure 3.2: a. Diagram showing covered intervals that do not cross, although one interval can contain another. b. Two covered intervals that do cross.

PROOF The proof is by induction on the number of intervals created. Certainly the claim is true until the first interval is created, and that interval does not cross itself. Now assume that no intervals cross and consider the phase where the right end of P is aligned with position j of T.

Since $h = j$ at the start of the phase, and j is to the right of any interval, h begins outside any interval. We consider how h could first be set to a position inside an interval, other than the right end of the interval. Rule 1 is never executed when h is at the right end of an interval (since then $M(h)$ is defined and greater than zero), and after any execution of Rule 1, either the phase ends or h is decremented by one place. So an execution of Case 1 cannot cause h to move beyond the right-most character of a covered interval. This is also true for Cases 3 and 4 since the phase ends after either of those cases. So if h is ever moved into an interval in a position other than its right end, that move must follow an execution of Case 2 or 5. An execution of Case 2 or 5 moves h from the right end of some interval $I = [k..h]$ to position $k - 1$, one place to the left of I. Now suppose that $k - 1$ is in some interval I' but is not at its right end, and that this is the first time in the phase that h (presently $k - 1$) is in an interval in a position other than its right end. That means that the right end of I cannot be to the left of the right end of I' (for then position $k - 1$ would have been strictly inside I'), and the right ends of I and I' cannot be equal (since $M(h)$ has at most one value for any h). But these conditions imply that I and I' cross, which is assumed to be untrue. Hence, if no intervals cross at the start of the phase, then in that phase only the right end of any covered interval is examined.

A new covered interval gets created in the phase only after the execution of Case 1, 3, or 4. In any of these cases, the interval $[h + 1..j]$ is created after the algorithm examines position h. In Case 1, h is not in any interval, and in Cases 3 and 4, h is the right end of an interval, so in all cases $h + 1$ is either not in a covered interval or is at the left end of an interval. Since j is to the right of any interval, and $h + 1$ is either not in an interval or is the left end of one, the new interval $[h + 1..j]$ does not cross any existing interval. The previously existing intervals have not changed, so there are no crossing intervals at the end of the phase, and the induction is complete. □

Theorem 3.1.2. *The modified Apostolico–Giancarlo algorithm does at most 2m character comparisons and at most $O(m)$ additional work.*

PROOF Every phase ends if a comparison finds a mismatch and every phase, except the last, is followed by a nonzero shift of P. Thus the algorithm can find at most m mismatches. To bound the matches, observe that characters are explicitly compared only in Case 1, and

if the comparison involving $T(h)$ is a match then, at the end of the phase, $M(j)$ is set at least as large as $j - h + 1$. That means that all characters in T that matched a character of P during that phase are contained in the covered interval $[j - M(j) + 1..j]$. Now the algorithm only examines the right end of an interval, and if h is the right end of an interval then $M(h)$ is defined and greater than 0, so the algorithm never compares a character of T in a covered interval. Consequently, no character of T will ever be compared again after it is first in a match. Hence the algorithm finds at most m matches, and the total number of character comparisons is bounded by $2m$.

To bound the amount of additional work, we focus on the number of accesses of M during execution of the five cases since the amount of additional work is proportional to the number of such accesses. A character comparison is done whenever Case 1 applies. Whenever Case 3 or 4 applies, P is immediately shifted. Hence Cases 1, 3, and 4 can apply at most $O(m)$ times since there are at most $O(m)$ shifts and compares. However, it is possible that Case 2 or Case 5 can apply without an immediate shift or immediate character comparison. That is, Case 2 or 5 could apply repeatedly before a comparison or shift is done. For example, Case 5 would apply twice in a row (without a shift or character comparison) if $N_i = M(h) > 0$ and $N_{i-N_i} = M(h - M(h))$. But whenever Case 2 or 5 applies, then $j > h$ and $M(j)$ will certainly get set to $j - h + 1$ or more at the end of that phase. So position h will be in the strict interior of the covered interval defined by j. Therefore, h will never be examined again, and $M(h)$ will never be accessed again. The effect is that Cases 2 and 5 can apply at most once for any position in T, so the number of accesses made when these cases apply is also $O(m)$. □

3.2. Cole's linear worst-case bound for Boyer–Moore

Here we finally present a linear worst-case time analysis of the *original* Boyer–Moore algorithm. We consider first the use of the (strong) good suffix rule by itself. Later we will show how the analysis is affected by the addition of the bad character rule. Recall that the good suffix rule is the following:

> 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 *such that the character to the left of t' differs from the mismatched character in P*. Shift P to the right so that substring t' in P is below substring t in T (recall 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 pattern matches a suffix of the occurrence of P in T. If no such shift is possible, then shift P by n places.

We will show that by using the 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. Later we will extend the Boyer–Moore method to take care of the case that P does occur in T.

As in our analysis of the Apostolico–Giancarlo algorithm, we divide the Boyer–Moore algorithm into *compare/shift* phases numbered 1 through $q \leq m$. In compare/shift phase i, a suffix of P is matched right to left with characters of T until either all of P is matched or until a mismatch occurs. In the latter case, the substring of T consisting of the matched

characters is denoted t_i, and the mismatch occurs just to the left of t_i. The pattern is then shifted right by an amount determined by the good suffix rule.

3.2.1. Cole's proof when the pattern does not occur in the text

Definition Let s_i denote the amount by which P is shifted right at the end of phase i.

Assume that P does not occur in T, so the compare part of every phase ends with a mismatch. In each compare/shift phase, we divide the comparisons into those that compare a character of T that has previously been compared (in a previous phase) and those comparisons that compare a character of T for the first time in the execution of the algorithm. Let g_i be the number of comparisons in phase i of the first type (comparisons involving a previously examined character of T), and let g_i' be the number of comparisons in phase i of the second type. Then, over the entire algorithm the number of comparisons is $\sum_{i=1}^{q}(g_i + g_i')$, and our goal is to show that this sum is $O(m)$.

Certainly, $\sum_{i=1}^{q} g_i' \leq m$ since a character can be compared for the first time only once. We will show that for any phase i, $s_i \geq g_i/3$. Then since $\sum_{i=1}^{q} s_i \leq m$ (because the total length of all the shifts is at most m) it will follow that $\sum_{i=1}^{q} g_i \leq 3m$. Hence the total number of comparisons done by the algorithm is $\sum_{i=1}^{q}(g_i + g_i') \leq 4m$.

An initial lemma

We start with the following definition and a lemma that is valuable in its own right.

Definition For any string β, β^i denotes the string obtained by concatenating together i copies of β.

Lemma 3.2.1. *Let γ and δ be two nonempty strings such that $\gamma\delta = \delta\gamma$. Then $\delta = \rho^i$ and $\gamma = \rho^j$ for some string ρ and positive integers i and j.*

This lemma says that if a string is the same before and after a circular shift (so that it can be written both as $\gamma\delta$ and $\delta\gamma$, for some strings γ and δ) then γ and δ can both be written as concatenations of some single string ρ.

For example, let $\delta = abab$ and $\gamma = ababab$, so $\delta\gamma = abababbabab = \gamma\delta$. Then $\rho = ab$, $\delta = \rho^2$, and $\gamma = \rho^3$.

PROOF The proof is by induction on $|\delta| + |\gamma|$. For the basis, if $|\delta| + |\gamma| = 2$, it must be that $\delta = \gamma = \rho$ and $i = j = 1$. Now consider larger lengths. If $|\delta| = |\gamma|$, then again $\delta = \gamma = \rho$ and $i = j = 1$. So suppose $|\delta| < |\gamma|$. Since $\delta\gamma = \gamma\delta$ and $|\delta| < |\gamma|$, δ must be a prefix of γ, so $\gamma = \delta\delta'$ for some string δ'. Substituting this into $\delta\gamma = \gamma\delta$ gives $\delta\delta\delta' = \delta\delta'\delta$. Deleting the left copy of δ from both sides gives $\delta\delta' = \delta'\delta$. However, $|\delta| + |\delta'| = |\gamma| < |\delta| + |\gamma|$, and so by induction, $\delta = \rho^i$ and $\delta' = \rho^j$. Thus, $\gamma = \delta\delta' = \rho^k$, where $k = i + j$. □

Definition A string α is *semiperiodic* with *period* β if α consists of a nonempty suffix of a string β (possibly the entire β) followed by one or more copies of β. String α is called *periodic* with *period* β if α consists of two or more *complete* copies of β. We say that string α is *periodic* if it is periodic with some period β.

For example, *bcabcabc* is semiperiodic with period *abc*, but it is not periodic. String *abcabc* is periodic with period *abc*. Note that a periodic string is by definition also semiperiodic. Note also that a string cannot have itself as a period although a period may itself

be periodic. For example, *abababab* is periodic with period *abab* and also with shorter period *ab*. An alternate definition of a semiperiodic string is sometimes useful.

Definition A string α is *prefix semiperiodic* with period γ if α consists of one or more copies of string γ followed by a nonempty prefix (possibly the entire γ) of string γ.

We use the term "prefix semiperiodic" to distinguish this definition from the definition given for "semiperiodic", but the following lemma (whose proof is simple and is left as an exercise) shows that these two definitions are really alternate reflections of the same structure.

Lemma 3.2.2. *A string α is semiperiodic with period β if and only if it is prefix semiperiodic with the same length period as β.*

For example, the string *abaabaabaabaabaab* is semiperiodic with period *aab* and is prefix semiperiodic with period *aba*.

The following useful lemma is easy to verify, and its proof is typical of the style of thinking used in dealing with overlapping matches.

Lemma 3.2.3. *Suppose pattern P occurs in text T starting at positions p and $p' > p$, where $p' - p \leq \lfloor n/2 \rfloor$. Then P is semiperiodic with period $p' - p$.*

The following lemma, called the *GCD Lemma*, is a very powerful statement about periods of strings. We won't need the lemma in our discussion of Cole's proof, but it is natural to state it here. We will prove it and use it in Section 16.17.5.

Lemma 3.2.4. *Suppose string α is semiperiodic with both a period of length p and a period of length q, and $|\alpha| \geq p + q$. Then α is semiperiodic with a period whose length is the greatest common divisor of p and q.*

Return to Cole's proof

Recall that the key thing to prove is that $s_i \geq g_i/3$ in every phase i. As noted earlier, it then follows easily that the total number of comparisons is bounded by $4m$.

Consider the ith compare/shift phase, where substring t_i of T matches a suffix of P and then P is shifted s_i places to the right. If $s_i \geq (|t_i| + 1)/3$, then $s_i \geq g_i/3$ even if all characters of T that were compared in phase i had been previously compared. Therefore, it is easy to handle phases wnere the shift is "relatively" large compared to the total number of characters examined during the phase. Accordingly, for the next several lemmas we consider the case when the shift is relatively small (i.e., $s_i < (|t_i| + 1)/3$ or, equivalently, $|t_i| + 1 > 3s_i$).

We need some notation at this point. Let α be the suffix of P of length s_i, and let β be the smallest substring such that $\alpha = \beta^l$ for some integer l (it may be that $\beta = \alpha$ and $l = 1$). Let $\bar{P} = P[n - |t_i|..n]$ be the suffix of P of length $|t_i| + 1$, that is, that portion of P (including the mismatch) that was examined in phase i. See Figure 3.3.

Lemma 3.2.5. *If $|t_i| + 1 > 3s_i$, then both t_i and $\bar{P}$ are semiperiodic with period α and hence with period β.*

PROOF Starting from the right end of $\bar{P}$, mark off substrings of length s_i until less than s_i characters remain on the left (see Figure 3.4). There will be at least three full substrings since $|\bar{P}| = |t_i| + 1 > 3s_i$. Phase i ends by shifting P right by s_i positions. Consider how $\bar{P}$ aligns with T before and after that shift (see Figure 3.5). By definition of s_i and α, α is the part of the shifted $\bar{P}$ to the right of the original $\bar{P}$. By the good suffix rule, the portion

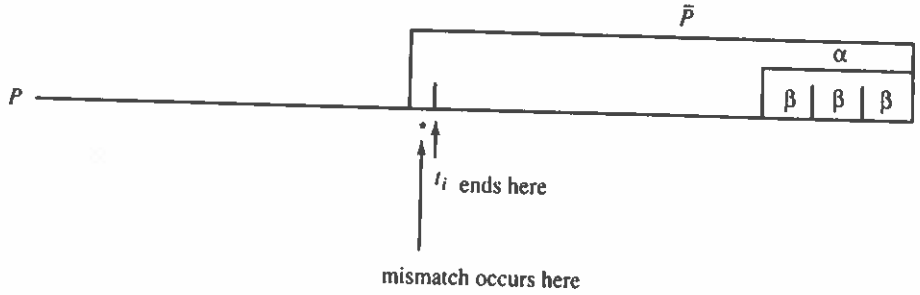

Figure 3.3: String α has length s_i; string $\bar{P}$ has length $|t_i| + 1$.

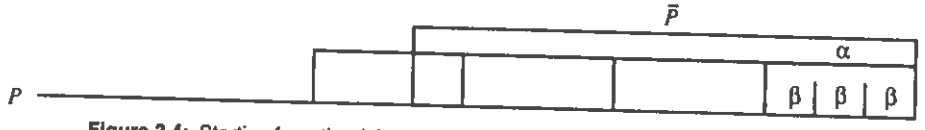

Figure 3.4: Starting from the right, substrings of length $|\alpha| = s_i$ are marked off in $\bar{P}$.

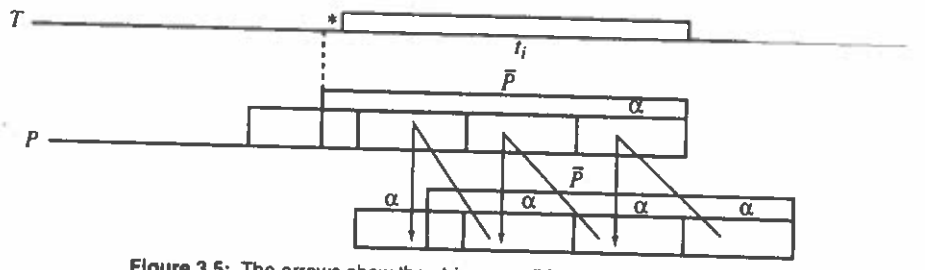

Figure 3.5: The arrows show the string equalities described in the proof.

of the shifted $\bar{P}$ below t_i must match the portion of the unshifted $\bar{P}$ below t_i, so the second marked-off substring from the right end of the shifted $\bar{P}$ must be the same as the first substring of the unshifted $\bar{P}$. Hence they must both be copies of string α. But the second substring is the same in both copies of $\bar{P}$, so continuing this reasoning we see that all the s_i-length marked substrings are copies of α and the left-most substring is a suffix of α (if it is not a complete copy of α). Hence $\bar{P}$ is semiperiodic with period α. The right-most $|t_i|$ characters of $\bar{P}$ match t_i, and so t_i is also semiperiodic with period α. Then since $\alpha = \beta^l$, $\bar{P}$ and t_i must also be semiperiodic with period β. □

Recall that we want to bound g_i, the number of characters compared in the ith phase that have been previously compared in earlier phases. All but one of the characters compared in phase i are contained in t_i, and a character in t_i could have previously been examined only during a phase where P overlaps t_i. So to bound g_i, we closely examine in what ways P could have overlapped t_i during earlier phases.

Lemma 3.2.6. *If $|t_i| + 1 > 3s_i$, then in any phase $h < i$, the right end of P could not have been aligned opposite the right end of any full copy of β in substring t_i of T.*

PROOF By Lemma 3.2.5, t_i is semiperiodic with period β. Figure 3.6 shows string t_i as a concatenation of copies of string β. In phase h, the right end of P cannot be aligned with the right end of t_i since that is the alignment of P and T in phase $i > h$, and P must have moved right between phases h and i. So, suppose, for contradiction, that in phase h the right end of P is aligned with the right end of some other full copy of β in t_i. For

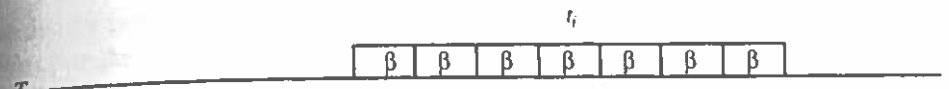

Figure 3.6: Substring t_i in T is semiperiodic with period β.

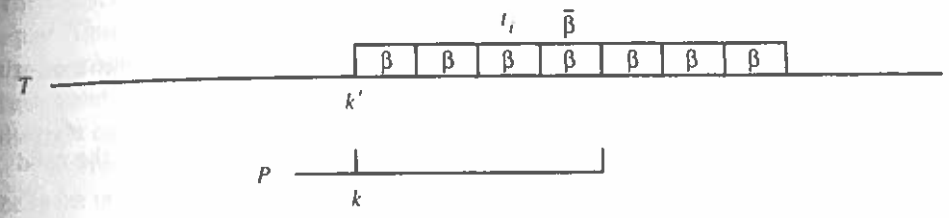

Figure 3.7: The case when the right end of P is aligned with a right end of $\bar{\beta}$ in phase h. Here $q = 3$. A mismatch must occur between $T(k')$ and $P(k)$.

concreteness, call that copy $\bar{\beta}$ and say that its right end is $q|\beta|$ places to the left of the right of t_i, where $q \geq 1$ (see Figure 3.7). We will first deduce how phase h must have ended, and then we'll use that to prove the lemma.

Let k' be the position in T just to the left of t_i (so $T(k')$ is involved in the mismatch ending phase i), and let k be the position in P opposite $T(k')$ in phase h. We claim that, in phase h, the comparison of P and T will find matches until the left end of t_i but then mismatch when comparing $T(k')$ and $P(k)$. The reason is the following: Strings $\bar{P}$ and t_i are semiperiodic with period β, and in phase h the right end of P is aligned with the right end of some β. So in phase h, P and T will certainly match until the left end of string t_i. Now $\bar{P}$ is semiperiodic with β, and in phase h, the right end of P is exactly $q|\beta|$ places to the left of the right end of t_i. Therefore, $\bar{P}(1) = \bar{P}(1 + |\beta|) = \cdots = \bar{P}(1 + q|\beta|) = P(k)$. But in phase i the mismatch occurs when comparing $T(k')$ with $\bar{P}(1)$, so $P(k) = \bar{P}(1) \neq T(k')$. Hence, if in phase h the right end of P is aligned with the right end of a β, then phase h must have ended with a mismatch between $T(k')$ and $P(k)$. This fact will be used below to prove the lemma.[1]

Now we consider the possible shifts of P done in phase h. We will show that every possible shift leads to a contradiction, so no shifts are possible and the assumed alignment of P and T in phase h is not possible, proving the lemma.

Since $h < i$, the right end of P will not be shifted in phase h past the right end of t_i; consequently, after the phase h shift a character of $\bar{P}$ is opposite character $T(k')$ (the character of T that will mismatch in phase i). Consider where the right end of P is after the phase h shift. There are two cases to consider: 1. Either the right end of P is opposite the right end of another full copy of β (in t_i) or 2. The right end of P is in the interior of a full copy of β.

Case 1 If the phase h shift aligns the right end of P with the right end of a full copy of β, then the character opposite $T(k')$ would be $P(k - r|\beta|)$ for some r. But since $\bar{P}$ is

[1] Later we will analyze the Boyer–Moore algorithm when P is in T. For that purpose we note here that when phase h is assumed to end by finding an occurrence of P, then the proof of Lemma 3.2.6 is complete at this point, having established a contradiction. That is, on the assumption that the right end of P is aligned with the right end of a β in phase h, we proved that phase h ends with a mismatch, which would contradict the assumption that h ends by finding an occurrence of P in T. So even if phase h ends by finding an occurrence of P, the right end of P could not be aligned with the right end of a β block in phase h.

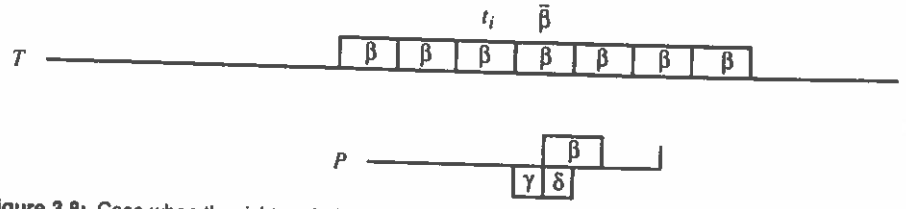

Figure 3.8: Case when the right end of P is aligned with a character in the interior of a β. Then t_i would have a smaller period than β, contradicting the definition of β.

semiperiodic with period β, $P(k)$ must be equal to $P(k - r|\beta|)$, contradicting the good suffix rule.

Case 2 Suppose the phase h shift aligns P so that its right end aligns with some character in the interior of a full copy of β. That means that, in this alignment, the right end of some β string in P is opposite a character in the interior of $\bar{\beta}$. Moreover, by the good suffix rule, the characters in the shifted P below $\bar{\beta}$ agree with $\bar{\beta}$ (see Figure 3.8). Let $\gamma\delta$ be the string in the shifted P positioned opposite $\bar{\beta}$ in t_i, where γ is the string through the end of β and δ is the remainder. Since $\bar{\beta} = \beta$, γ is a suffix of β, δ is a prefix of β, and $|\gamma| + |\delta| = |\bar{\beta}| = |\beta|$; thus $\gamma\delta = \delta\gamma$. By Lemma 3.2.1, however, $\beta = \rho^t$ for $t > 1$, which contradicts the assumption that β is the smallest string such that $\alpha = \beta^l$ for some l.

Starting with the assumption that in phase h the right end of P is aligned with the right end of a full copy of β, we reached the conclusion that no shift in phase h is possible. Hence the assumption is wrong and the lemma is proved. □

Lemma 3.2.7. *If $|t_i| + 1 > 3s_i$, then in phase $h < i$, P can match t_i in T for at most $|\beta| - 1$ characters.*

PROOF Since P is not aligned with the end of any β in phase h, if P matches t_i in T for β or more characters then the right-most β characters of P would match a string consisting of a suffix (γ) of β followed by a prefix (δ) of β. So we would again have $\beta = \gamma\delta = \delta\gamma$, and by Lemma 3.2.1, this again would lead to a contradiction to the selection of β. □

Note again that this lemma holds even if phase h is assumed to find an occurrence of P. That is, nowhere in the proof is it assumed that phase h ends with a mismatch, only that phase i does. This observation will be used later.

Lemma 3.2.8. *If $|t_i| + 1 > 3s_i$, then in phase $h < i$ if the right end of P is aligned with a character in t_i, it can only be aligned with one of the left-most $|\beta| - 1$ characters of t_i or one of the right-most $|\beta|$ characters of t_i.*

PROOF Suppose in phase h that the right end of P is aligned with a character of t_i other than one of the left-most $|\beta| - 1$ characters or the right-most $|\beta|$ characters. For concreteness, say that the right end of P is aligned with a character in copy β' of string β. Since β' is not the left-most copy of β, the right end of P is at least $|\beta|$ characters to the right of the left end of t_i, and so by Lemma 3.2.7 a mismatch would occur in phase h before the left end of t_i is reached. Say that mismatch occurs at position k'' of T. After that mismatch, P is shifted right by some amount determined by the good suffix rule. By Lemma 3.2.6, the phase-h shift cannot move the right end of P to the right end of β', and we will show that the shift will also not move the end of P past the right end of β'.

Recall that the good suffix rule shifts P (when possible) by the smallest amount so that all the characters of T that matched in phase h again match with the shifted P and

so that the two characters of P aligned with $T(k'')$ before and after the shift are unequal. We claim these conditions hold when the right end of P is aligned with the right end of β'. Consider that alignment. Since $\bar{P}$ is semiperiodic with period β, that alignment of P and T would match at least until the left end of t_i and so would match at position k'' of T. Therefore, the two characters of P aligned with $T(k'')$ before and after the shift cannot be equal. Thus if the end of P were aligned with the end of β' then all the characters of T that matched in phase h would again match, and the characters of P aligned with $T(k'')$ before and after the shift would be different. Hence the good suffix rule would not shift the right end of P past the right of the end of β'.

Therefore, if the right end of P is aligned in the interior of β' in phase h, it must also be in the interior of β' in phase $h+1$. But h was arbitrary, so the phase-$h+1$ shift would also not move the right end of P past β'. So if the right end of P is in the interior of β' in phase h, it remains there forever. This is impossible since in phase $i > h$ the right end of P is aligned with the right end of t_i, which is to the right of β'. Hence the right end of P is not in the interior of β', and the Lemma is proved. □

Note again that Lemma 3.2.8 holds even if phase h is assumed to end by finding an occurrence of P in T. That is, the proof only needs the assumption that phase i ends with a mismatch, not that phase h does. In fact, when phase h finds an occurrence of P in T, then the proof of the lemma only needs the reasoning contained in the first two paragraphs of the above proof.

Theorem 3.2.1. *Assuming P does not occur in T, $s_i \geq g_i/3$ in every phase i.*

PROOF This is trivially true if $s_i \geq (|t_i|+1)/3$, so assume $|t_i|+1 > 3s_i$. By Lemma 3.2.8, in any phase $h < i$, the right end of P is opposite either one of the left-most $|\beta|-1$ characters of t_i or one of the right-most $|\beta|$ characters of t_i (excluding the extreme right character). By Lemma 3.2.7, at most $|\beta|$ comparisons are made in phase $h < i$. Hence the only characters compared in phase i that could possibly have been compared before phase i are the left-most $|\beta|-1$ characters of t_i, the right-most $2|\beta|$ characters of t_i, or the character just to the left of t_i. So $g_i \leq 3|\beta| = 3s_i$ when $|t_i|+1 > 3s_i$. In both cases then, $s_i \geq g_i/3$. □

Theorem 3.2.2. [108] *Assuming that P does not occur in T, the worst-case number of comparisons made by the Boyer–Moore algorithm is at most $4m$.*

PROOF As noted before, $\sum_{i=1}^{q} g_i' \leq m$ and $\sum_{i=1}^{q} s_i \leq m$, so the total number of comparisons done by the algorithm is $\sum_{i=1}^{q}(g_i + g_i') \leq (\sum_i 3s_i) + m \leq 4m$. □

3.2.2. The case when the pattern does occur in the text

Consider P consisting of n copies of a single character and T consisting of m copies of the same character. Then P occurs in T starting at every position in T except the last $n-1$ positions, and the number of comparisons done by the Boyer–Moore algorithm is $\Theta(mn)$. The $O(m)$ time bound proved in the previous section breaks down because it was derived by showing that $g_i \leq 3s_i$, and that required the assumption that phase i ends with a mismatch. So when P does occur in T (and phases do not necessarily end with mismatches), we must modify the Boyer–Moore algorithm in order to recover the linear running time. Galil [168] gave the first such modification. Below we present a version of his idea.

The approach comes from the following observation: Suppose in phase i that the right end of P is positioned with character k of T, and that P is compared with T down

to character s of T. (We don't specify whether the phase ends by finding a mismatch or by finding an occurrence of P in T.) If the phase-i shift moves P so that its left end is to the right of character s of T, then in phase $i + 1$ a *prefix* of P definitely matches the characters of T up to $T(k)$. Thus, in phase $i + 1$, if the right-to-left comparisons get down to position k of T, the algorithm can conclude that an occurrence of P has been found even without explicitly comparing characters to the left of $T(k + 1)$. It is easy to implement this modification to the algorithm, and we assume in the rest of this section that the Boyer–Moore algorithm includes this rule, which we call the *Galil rule*.

Theorem 3.2.3. *Using the Galil rule, the Boyer–Moore algorithm never does more than $O(m)$ comparisons, no matter how many occurrences or P there are in T.*

PROOF Partition the phases into those that do find an occurrence of P and those that do not. Let Q be the set of phases of the first type and let d_i be the number of comparisons done in phase i if $i \in Q$. Then $\sum_{i \in Q} d_i + \sum_{i \notin Q}(|t_i| + 1)$ is a bound on the total number of comparisons done in the algorithm.

The quantity $\sum_{i \notin Q}(|t_i| + 1)$ is again $O(m)$. To see this, recall that the lemmas of the previous section, which proved that $g_i \leq 3s_i$, only needed the assumption that phase i ends with a mismatch and that $h < i$. In particular, the analysis of how P of phase $h < i$ is aligned with P of phase i did not need the assumption that phase h ends with a mismatch. Those proofs cover both the case that h ends with a mismatch and that h ends by finding an occurrence of P. Hence it again holds that $g_i \leq 3s_i$ if phase i ends with a mismatch, even though earlier phases might end with a match.

For phases in Q, we again ignore the case that $s_i \geq (n + 1)/3 \geq (d_i + 1)/3$, since the total number of comparisons done in such phases must be bounded by $\sum 3s_i \leq 3m$. So suppose phase i ends by finding an occurrence of P in T and then shifts by less than $n/3$. By a proof essentially the same as for Lemma 3.2.5 it follows that P is semiperiodic; let β denote the shortest period of P. Hence the shift in phase i moves P right by exactly $|\beta|$ positions, and using the Galil rule in the Boyer–Moore algorithm, no character of T compared in phase $i + 1$ will have ever been compared previously. Repeating this reasoning, if phase $i + 1$ ends by finding an occurrence of P then P will again shift by exactly $|\beta|$ places and no comparisons in phase $i + 2$ will examine a character of T compared in any earlier phase. This cycle of shifting P by exactly $|\beta|$ positions and then identifying another occurrence of P by examining only $|\beta|$ new characters of T may be repeated many times. Such a succession of overlapping occurrences of P then consists of a concatenation of copies of β (each copy of P starts exactly $|\beta|$ places to the right of the previous occurrence) and is called a *run*. Using the Galil rule, it follows immediately that in any single run the number of comparisons used to identify the occurrences of P contained in that run is exactly the length of the run. Therefore, over the entire algorithm the number of comparisons used to find those occurrences is $O(m)$. If no additional comparisons were possible with characters in a run, then the analysis would be complete. However, additional examinations are possible and we have to account for them.

A run ends in some phase $k > i$ when a mismatch is found (or when the algorithm terminates). It is possible that characters of T in the run could be examined again in phases after k. A phase that reexamines characters of the run either ends with a mismatch or ends by finding an occurrence of P that overlaps the earlier run but is not part of it. However,

all comparisons in phases that end with a mismatch have already been accounted for (in the accounting for phases not in Q) and are ignored here.

Let $k' > k > i$ be a phase in which an occurrence of P is found overlapping the earlier run but is not part of that run. As an example of such an overlap, suppose $P = axaaxa$ and T contains the substring $axaaxaaxaaxaxaaxaaxa$. Then a run begins at the start of the substring and ends with its twelfth character, and an overlapping occurrence of P (not part of the run) begins with that character. Even with the Galil rule, characters in the run will be examined again in phase k', and since phase k' does not end with a mismatch those comparisons must still be counted.

In phase k', if the left end of the new occurrence of P in T starts at a left end of a copy of β in the run, then contiguous copies of β continue past the right end of the run. But then no mismatch would have been possible in phase k since the pattern in phase k is aligned exactly $|\beta|$ places to the right of its position in phase $k-1$ (where an occurrence of P was found). So in phase k', the left end of the new P in T must start with an interior character of some copy of β. But then if P overlaps with the run by more than $|\beta|$ characters, Lemma 3.2.1 implies that β is periodic, contradicting the selection of β. So P can overlap the run only by part of the run's left-most copy of β. Further, since phase k' ends by finding an occurrence of P, the pattern is shifted right by $s_{k'} = |\beta|$ positions. Thus any phase that finds an occurrence of P overlapping an earlier run next shifts P by a number of positions larger than the length of the overlap (and hence the number of comparisons). It follows then that over the entire algorithm the total number of such additional comparisons in overlapping regions is $O(m)$.

All comparisons are accounted for and hence $\sum_{i \in Q} d_i = O(m)$, finishing the proof of the lemma. □

3.2.3. Adding in the bad character rule

Recall that in computing a shift after a mismatch, the Boyer–Moore algorithm uses the largest shift given by either the (extended) bad character rule or the (strong) good suffix rule. It seems intuitive that if the time bound is $O(m)$ when only the good suffix rule is used, it should still be $O(m)$ when both rules are used. However, certain "interference" is plausible, and so the intuition requires a proof.

Theorem 3.2.4. *When both shift rules are used together, the worst-case running time of the modified Boyer–Moore algorithm remains $O(m)$.*

PROOF In the analysis using only the suffix rule we focused on the comparisons done in an arbitrary phase i. In phase i the right end of P was aligned with some character of T. However, we never made any assumptions about how P came to be positioned there. Rather, given an arbitrary placement of P in a phase ending with a mismatch, we deduced bounds on how many characters compared in that phase could have been compared in earlier phases. Hence all of the lemmas and analyses remain correct if P is arbitrarily picked up and moved some distance to the right at any time during the algorithm. The (extended) bad character rule only moves P to the right, so all lemmas and analyses showing the $O(m)$ bound remain correct even with its use. □

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.

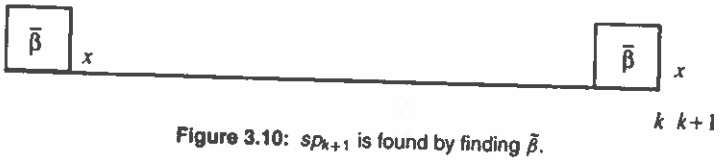

Figure 3.10: sp_{k+1} is found by finding $\bar{\beta}$.

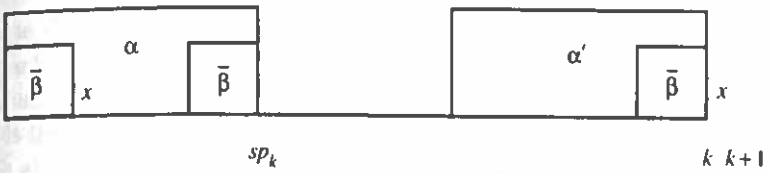

Figure 3.11: $\bar{\beta}$ must be a suffix of α.

$sp_k + 1 = |\alpha| + 1$, 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 βx 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 $sp_{k+1} \leq sp_k + 1$.

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 $P(k + 1) \neq P(sp_k + 1)$, then $sp_{k+1} < sp_k + 1$ (by Lemma 3.3.1), so $sp_{k+1} \leq sp_k$. 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 $P(k + 1) \neq P(sp_k + 1)$, $\bar{\beta}$ occurs as a suffix of α and also as a proper prefix of α followed by character x. So when $P(k + 1) \neq P(sp_k + 1)$, $\bar{\beta}$ 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 $P(sp_k + 1) \neq P(k + 1)$, 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}: