Exact matching: what's the problem?

Given a string P called the *pattern* and a longer string T called the *text*, the **exact matching** problem is to find all occurrences, if any, of pattern P in text T.

For example, if $P = aba$ and $T = bbabaxababay$ then P occurs in T starting at locations 3, 7, and 9. Note that two occurrences of P may overlap, as illustrated by the occurrences of P at locations 7 and 9.

Importance of the exact matching problem

The practical importance of the exact matching problem should be obvious to anyone who uses a computer. The problem arises in widely varying applications, too numerous to even list completely. Some of the more common applications are in word processors; in utilities such as *grep* on Unix; in textual information retrieval programs such as Medline, Lexis, or Nexis; in library catalog searching programs that have replaced physical card catalogs in most large libraries; in internet browsers and crawlers, which sift through massive amounts of text available on the internet for material containing specific keywords;[1] in internet news readers that can search the articles for topics of interest; in the giant digital libraries that are being planned for the near future; in electronic journals that are already being "published" on-line; in telephone directory assistance; in on-line encyclopedias and other educational CD-ROM applications; in on-line dictionaries and thesauri, especially those with cross-referencing features (the *Oxford English Dictionary* project has created an electronic on-line version of the *OED* containing 50 million words); and in numerous specialized databases. In molecular biology there are several hundred specialized databases holding raw DNA, RNA, and amino acid strings, or processed patterns (called motifs) derived from the raw string data. Some of these databases will be discussed in Chapter 15.

Although the practical importance of the exact matching problem is not in doubt, one might ask whether the problem is still of any research or educational interest. Hasn't exact matching been so well solved that it can be put in a black box and taken for granted? Right now, for example, I am editing a ninety-page file using an "ancient" shareware word processor and a PC clone (486), and every exact match command that I've issued executes faster than I can blink. That's rather depressing for someone writing a book containing a large section on exact matching algorithms. So is there anything left to do on this problem?

The answer is that for typical word-processing applications there probably is little left to do. The exact matching problem is solved for those applications (although other more sophisticated string tools might be useful in word processors). But the story changes radically

[1] I just visited the Alta Vista web page maintained by the Digital Electronics Corporation. The Alta Vista database contains over 21 billion words collected from over 10 million web sites. A search for all web sites that mention "Mark Twain" took a couple of seconds and reported that twenty thousand sites satisfy the query. For another example see [392].

for other applications. Users of Melvyl, the on-line catalog of the University of California library system, often experience long, frustrating delays even for fairly simple matching requests. Even *grepping* through a large directory can demonstrate that exact matching is not yet trivial. Recently we used GCG (a very popular interface to search DNA and protein databanks) to search Genbank (the major U.S. DNA database) for a thirty-character string, which is a small string in typical uses of Genbank. The search took over four hours (on a local machine using a local copy of the database) to find that the string was not there.[2] And Genbank today is only a fraction of the size it will be when the various genome programs go into full production mode, cranking out massive quantities of sequenced DNA. Certainly there are faster, common database searching programs (for example, BLAST), and there are faster machines one can use (for example, an e-mail server is available for exact and inexact database matching running on a 4,000 processor MasPar computer). But the point is that the exact matching problem is not so effectively and universally solved that it needs no further attention. It will remain a problem of interest as the size of the databases grow and also because exact matching will continue to be a *subtask* needed for more complex searches that will be devised. Many of these will be illustrated in this book.

But perhaps the most important reason to study *exact* matching in detail is to understand the various ideas developed for it. Even assuming that the exact matching problem itself is sufficiently solved, the entire field of string algorithms remains vital and open, and the education one gets from studying exact matching may be crucial for solving less understood problems. That education takes three forms: specific algorithms, general algorithmic styles, and analysis and proof techniques. All three are covered in this book, but style and proof technique get the major emphasis.

Overview of Part I

In Chapter 1 we present naive solutions to the exact matching problem and develop the fundamental tools needed to obtain more efficient methods. Although the classical solutions to the problem will not be presented until Chapter 2, we will show at the end of Chapter 1 that the use of fundamental tools alone gives a simple linear-time algorithm for exact matching. Chapter 2 develops several classical methods for exact matching, using the fundamental tools developed in Chapter 1. Chapter 3 looks more deeply at those methods and extensions of them. Chapter 4 moves in a very different direction, exploring methods for exact matching based on arithmetic-like operations rather than character comparisons.

Although exact matching is the focus of Part I, some aspects of inexact matching and the use of wild cards are also discussed. The exact matching problem will be discussed again in Part II, where it (and extensions) will be solved using suffix trees.

Basic string definitions

We will introduce most definitions at the point where they are first used, but several definitions are so fundamental that we introduce them now.

Definition A *string* S is an ordered list of characters written contiguously from left to right. For any string S, $S[i..j]$ is the (contiguous) *substring* of S that starts at position

[2] We later repeated the test using the Boyer–Moore algorithm on our own raw copy of Genbank. The search took less than ten minutes, most of which was devoted to movement of text between the disk and the computer, with less than one minute used by the actual text search.

i and ends at position j of S. In particular, $S[1..i]$ is the *prefix* of string S that ends at position i, and $S[i..|S|]$ is the *suffix* of string S that begins at position i.

For example, *california* is a string, *lifo* is a substring, *cal* is a prefix, and *ornia* is a suffix.

Definition A *proper* prefix, suffix, or substring of S is, respectively, a prefix, suffix, or substring that is not the entire string S.

Definition For any string S, $S(i)$ denotes the ith character of S.

We will usually use the symbol S to refer to an arbitrary fixed string that has no additional assumed features or roles. However, when a string is known to play the role of a pattern or the role of a text, we will refer to the string as P or T respectively. We will use lower case Greek characters (α, β, γ, δ) to refer to variable strings and use lower case roman characters to refer to single variable characters.

Definition When comparing two characters, we say that the characters *match* if they are equal; otherwise we say they *mismatch*.

Terminology confusion

The words "string" and "word" are often used synonymously in the computer science literature, but for clarity in this book we will never use "word" when "string" is meant. (However, we do use "word" when its colloquial English meaning is intended.)

More confusing, the words "string" and "sequence" are often used synonymously, particularly in the biological literature. This can be the source of much confusion because "*sub*strings" and "*sub*sequences" are very different objects and because algorithms for substring problems are usually very different than algorithms for the analogous subsequence problems. The characters in a substring of S must occur *contiguously* in S, whereas characters in a subsequence might be interspersed with characters not in the subsequence. Worse, in the biological literature one often sees the word "sequence" used in place of "subsequence". Therefore, for clarity, in this book we will always maintain a distinction between "subsequence" and "substring" and never use "sequence" for "subsequence". We will generally use "string" when pure computer science issues are discussed and use "sequence" or "string" interchangeably in the context of biological applications. Of course, we will also use "sequence" when its standard mathematical meaning is intended.

The first two parts of this book primarily concern problems on strings and substrings. Problems on subsequences are considered in Parts III and IV.

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

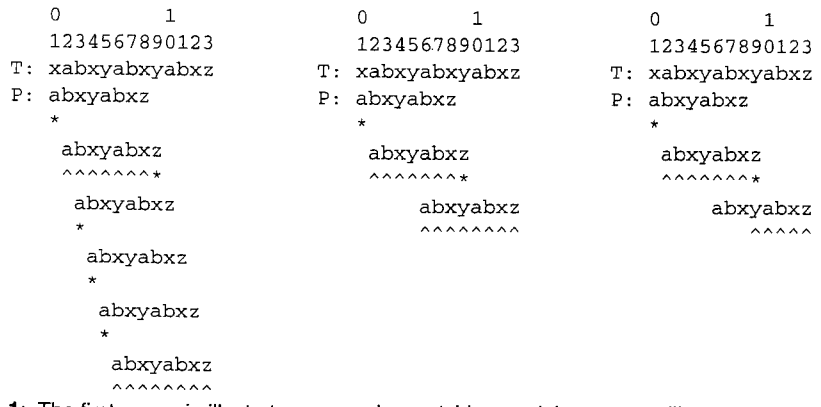

Figure 1.1: The first scenario illustrates pure naive matching, and the next two illustrate smarter shifts. A caret beneath a character indicates a match and a star indicates a mismatch made by the algorithm.

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

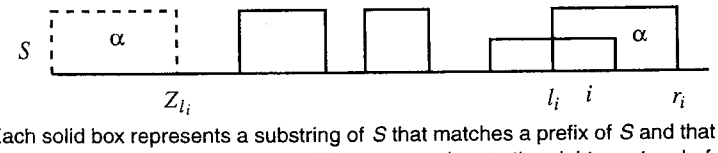

Figure 1.2: Each solid box represents a substring of S that matches a prefix of S and that starts between positions 2 and i. Each box is called a *Z-box*. We use r_i to denote the *right-most* end of any Z-box that begins at or to the left of position i and α to denote the substring in the Z-box ending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

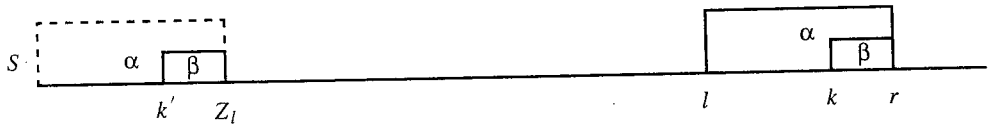

Figure 1.3: String $S[k..r]$ is labeled β and also occurs starting at position k' of S.

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 1 to $n + m + 1$. Because "$" does not appear in P or T, $Z_i \leq n$ for every i. 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 set. 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