STRING MATCHING

Finding all occurrences of a pattern in a text is a problem that arises frequently in text-editing programs. Typically, the text is a document being edited, and the pattern searched for is a particular word supplied by the user. Efficient algorithms for this problem can greatly aid the responsiveness of the text-editing program. String-matching algorithms are also used, for example, to search for particular patterns in DNA sequences.

We formalize the ***string-matching problem*** as follows. We assume that the text is an array $T[1 \,.\,.\, n]$ of length n and that the pattern is an array $P[1 \,.\,.\, m]$ of length m. We further assume that the elements of P and T are characters drawn from a finite alphabet Σ. For example, we may have $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b, \ldots, z\}$. The character arrays P and T are often called ***strings*** of characters.

We say that pattern P ***occurs with shift s*** in text T (or, equivalently, that pattern P ***occurs beginning at position s + 1*** in text T) if $0 \le s \le n - m$ and $T[s + 1 \,.\,.\, s + m] = P[1 \,.\,.\, m]$ (that is, if $T[s + j] = P[j]$, for $1 \le j \le m$). If P occurs with shift s in T, then we call s a ***valid shift***; otherwise, we call s an ***invalid shift***. The string-matching problem is the problem of finding all valid shifts with which a given pattern P occurs in a given text T. Figure 34.1 illustrates these definitions.

This chapter is organized as follows. In Section 34.1 we review the naive brute-force algorithm for the string-matching problem, which has worst-case running time $O((n - m + 1)m)$. Section 34.2 presents an interesting string-matching algorithm, due to Rabin and Karp. This algorithm also has worst-case running time $O((n - m + 1)m)$, but it works much better on average and in practice. It also generalizes nicely to other pattern-matching problems. Section 34.3 then describes a string-matching algorithm that begins by constructing a finite automaton specifically designed to search for occurrences of the given pattern P in a text. This algorithm runs in time $O(n + m |\Sigma|)$. The similar but much cleverer Knuth-Morris-Pratt (or KMP) algorithm is presented in Section 34.4; the KMP algorithm runs in time $O(n + m)$. Finally, Section 34.5 describes an algorithm due to Boyer and Moore that is often the best practical choice, although its worst-case running time (like that of the Rabin-Karp algorithm) is no better than that of the naive string-matching algorithm.

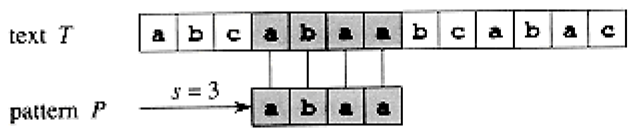

Figure 34.1 The string-matching problem. The goal is to find all occurrences of the pattern P = abaa in the text T = abcabaabcabac. The pattern occurs

only once in the text, at shift s = 3. The shift s = 3 is said to be a valid shift. Each character of the pattern is connected by a vertical line to the matching character in the text, and all matched characters are shown shaded.

Notation and terminology

We shall let Σ^* (read "sigma-star") denote the set of all finite-length strings formed using characters from the alphabet Σ. In this chapter, we consider only finite-length strings. The zero-length ***empty string***, denoted ε, also belongs to Σ^*. The length of a string x is denoted $|x|$. The ***concatenation*** of two strings x and y, denoted xy, has length $|x| + |y|$ and consists of the characters from x followed by the characters from y.

We say that a string w is a ***prefix*** of a string x, denoted $w \sqsubset x$, if $x = wy$ for some string $y \in \Sigma^*$. Note that if $w \sqsubset x$, then $|w| \le |x|$. Similarly, we say that a string w is a ***suffix*** of a string x, denoted $w \sqsupset x$, if $x = yw$ for some $y \in \Sigma^*$. It follows from $w \sqsupset x$ that $|w| \le |x|$. The empty string ε is both a suffix and a prefix of every string. For example, we have `ab` $\sqsubset$ `abcca` and `cca` $\sqsubset$ `abcca`. It is useful to note that for any strings x and y and any character a, we have $x \sqsupset y$ if and only if $xa \sqsupset ya$. Also note that $\sqsubset$ and $\sqsupset$ are transitive relations. The following lemma will be useful later.

Lemma 34.1

Suppose that x, y, and z are strings such that $x \sqsupset z$ and $y \sqsupset z$. If $|x| \le |y|$, then $x \sqsupset y$. If $|x| \ge |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

Proof See Figure 34.2 for a graphical proof.

For brevity of notation, we shall denote the k-character prefix $P[1 \,.\,.\, k]$ of the pattern $P[1 \,.\,.\, m]$ by P_k. Thus, $P_0 = \varepsilon$ and $P_m = P = P[1 \,.\,.\, m]$. Similarly, we denote the k-character prefix of the text T as T_k. Using this notation, we can state the string-matching problem as that of finding all shifts s in the range $0 \le s \le n - m$ such that $P \sqsupset T_{s+m}$.

In our pseudocode, we allow two equal-length strings to be compared for equality as a primitive operation. If the strings are compared from left to right and the comparison stops when a mismatch is discovered, we assume that the time taken by such a test is a linear function of the number of matching characters discovered.

To be precise, the test "$x = y$" is assumed to take time $\Theta(t + 1)$, where t is the length of the longest string z such that $z \sqsubset x$ and $z \sqsubset y$.

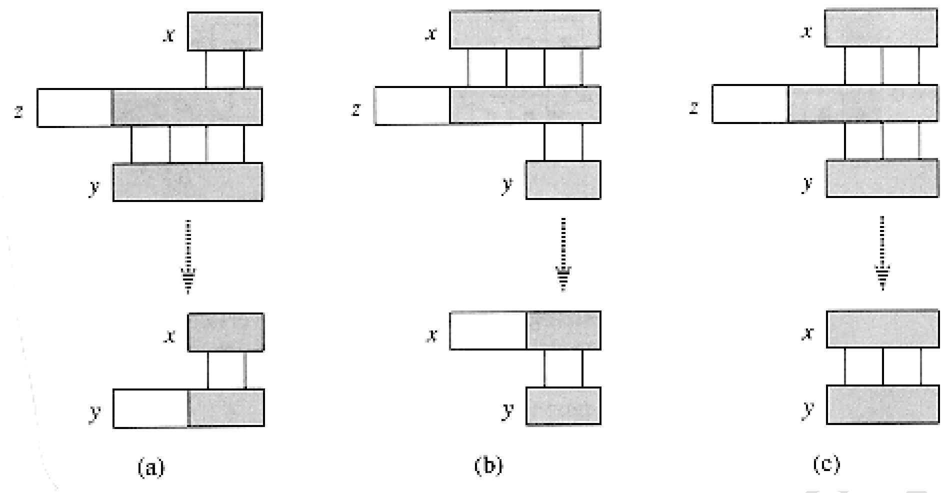

Figure 34.2 A graphical proof of Lemma 34.1. We suppose that $x \sqsupset z$ and $y \sqsupset z$. The three parts of the figure illustrate the three cases of the lemma. Vertical lines connect matching regions (shown shaded) of the strings. (a) If |x| ≤ |y|, then $x \sqsupset y$. (b) If |x| ≥ |y|, then $y \sqsupset x$. (c) If |x| = |y|, then x = y.