7

Strings

Many important computational problems can be formulated as a search for the occurrences of a given set of objects, called **patterns,** in a given subject. Specific application areas where such problems arise include text editing, computer vision, speech recognition, and molecular biology. In this chapter, we deal with strings and their exact occurrences in other strings. Powerful algorithmic tools, based on mathematical properties of strings, are well known and can be used to handle efficiently many processing tasks on strings.

Most of the algorithmic techniques for exact **string matching** seem to revolve around the *periodic properties* of strings. Such properties are introduced first, followed by the development of an optimal logarithmic-time parallel string-matching algorithm. The main paradigm used consists of *pattern analysis*, followed by *text analysis*. The output generated at the end of the pattern-analysis phase is a table containing information related to the periodic structure of the pattern. The text-analysis phase uses this table to identify all the occurrences of the pattern in the text. We also introduce a data structure, called the *suffix tree*, that can be used to support many string manipulations. An efficient parallel algorithm to set up such a data structure is presented, as are several of its applications.

7.1 Preliminary Facts About Strings

Let Σ be an **alphabet** consisting of a finite number of *symbols*. A **string** Y over Σ is a finite sequence of elements from Σ. The **length** of Y, denoted by $|Y|$, is the total number of elements in the sequence defining Y. If X and Y are strings, the **concatenation** of X and Y is the string XY; that is, it is the sequence defining X followed by the sequence defining Y.

In describing our algorithms on strings, we assume that a string Y is represented by an array such that $Y(i)$ is the ith element of Y, where $1 \leq i \leq |Y|$.

Let Y be a string of length m. For any pair of indices i and j such that $1 \leq i \leq j \leq m$, the subarray $Y(i:j)$ defines a **substring** of Y. Hence, a substring of Y is any contiguous block of characters in Y. A substring defined by $Y(1:i)$, for some i such that $1 \leq i \leq m$, is called a **prefix** of Y, whereas a substring of the form $Y(j:m)$, for some j such that $1 \leq j \leq m$, is called a **suffix** of Y.

A string X **occurs** in Y at position i if X is equal to the substring of Y of length $|X|$ starting at location i. More formally, $X(j) = Y(i+j-1)$, for all indices j such that $1 \leq j \leq |X|$. We also say that X **matches** Y at position i.

EXAMPLE 7.1:

Let $\Sigma = \{a, b, c\}$, and let Y be the string $Y = aabcabccaa$. Then, $X = abc$ is a substring of Y that occurs at positions 2 and 5. The prefix $Y(1:5)$ is the substring $aabca$, and the suffix $Y(5:10)$ is the substring $abccaa$. □

7.1.1 PERIODICITIES IN STRINGS

The periodic properties of strings provide the basis for certain algorithmic tools for the efficient manipulation of strings. In this section, we introduce the notion of the period of a string, and we study a few related properties.

Let Y be a string of length m. A substring X of a string Y will be called **a period** of Y if $Y = X^k X'$, where X^k is the string consisting of X concatenated with itself k times, k being a positive integer, and X' is a prefix of X. Hence, Y consists of several consecutive copies of X, followed by a prefix of X. It is easy to check that X is a period of Y if and only if Y is a prefix of XY.

The definition of a period has the following important implication. Suppose a copy of Y is placed above Y but is shifted i positions from the left end of Y, for some i such that $1 \leq i \leq m-1$ (see Fig. 7.1). Then, if the overlapping portions are identical, Y has a period consisting of the prefix $Y(1:i)$. In other words, if, for some i, the two substrings $Y(i+1:m)$ and $Y(1:m-i)$ are equal, then $Y(1:i)$ is a period of Y. Note that Y is always a period of itself.

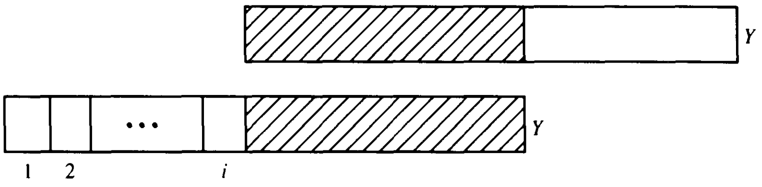

FIGURE 7.1
Two copies of the same string Y, shifted by i positions. The prefix $Y(1 : i)$ is a period if the overlapping portions are identical.

EXAMPLE 7.2:

Let $Y = ababababa$. Then, the prefix $U = abab$ is a period of Y, since $Y = U^2a$. String Y also has the following periods: *ab*, *ababab*, *abababab*. □

The period of a string Y is the shortest period of Y. Let p be the length (also called size) of the period of Y. We will also refer to p as the period of Y. Hence, p is the smallest integer between 1 and m such that $Y(i) = Y(i + p)$, for all i satisfying $1 \leq i \leq m - p$.

The string Y is called **periodic** if its period p satisfies $p \leq m/2$.

EXAMPLE 7.3:

Consider the following two strings: $Y = abcaabcab$ and $Z = abcabcab$. String Y is not periodic, since its shortest period is *abcaabc*; string Z is periodic, and has the period $p = 3$. □

An important fact about the periods of a string is stated in the next lemma. Before giving the lemma we review a simple version of Euclid's algorithm for computing the greatest common divisor δ of two positive integers p and q.

Remark 7.1: To compute the greatest common divisor δ of p and q, we set $\delta: = p = q$ if $p = q$. Otherwise, we assume without loss of generality that $p > q$. Then, it is easy to verify that $\delta = gcd(p - q, q)$. Therefore, computing δ is now reduced to finding the greatest common divisor of the two integers $p' = p - q$ and $q' = q$. Clearly, $p' + q' < p + q$. Hence, the process can be continued until the two remaining integers are equal. We then set δ to their common value. □

Lemma 7.1 (Periodicity Lemma): *If a string Y has two periods of sizes p and q, and $|Y| \geq p + q$, then Y has a period of size $gcd(p, q)$, where $gcd(p, q)$ is the greatest common divisor of p and q.*

Proof of Lemma 7.1: If $p = q$, the proof follows trivially. Let $p > q$ and let $|Y| = m$.

Claim: Y has a period of size $p - q$.

Proof of Claim: Since Y has a period of size p, we have that $Y(i) = Y(i + p)$, for $1 \le i \le m - p$. This fact implies $Y(i - q) = Y(i + p - q)$, for $q + 1 \le i \le m - (p - q)$. But Y also has a period of size q. Hence, $Y(i) = Y(i - q) = Y(i + p - q)$, for $q + 1 \le i \le m - (p - q)$. On the other hand, $Y(i) = Y(i + p) = Y(i + p - q)$, for $1 \le i \le q$. Note that we have used here the fact that $m \ge p + q$. Hence, $Y(i) = Y(i + p - q)$, for $1 \le i \le m - (p - q)$, and Y has a period of size $p - q$.

The rest of the proof follows from Euclid's algorithm for computing the *gcd* of p and q (Remark 7.1). □

Corollary 7.1: *Let p be the period of string Y of length m. If Y has any period of size q such that $q \le m - p$, then q is a multiple of p.* □

Remark 7.2: Corollary 7.1 applies if $p + q \le m$. On the other hand, we should emphasize that Y may have a period q greater than $m - p$ that is *not* a multiple of p. The string $Y = aabcaabcaa$ of period $p = 4$ has also a period of size 9. □

The following lemma is crucial to the parallel string-matching algorithm presented in the next section.

Lemma 7.2: *Let Y be a string of length m whose period is p, and let Z be an arbitrary string of length $n \ge m$. Then, the following two statements hold.*

1. *If Y occurs in Z at positions i and j, then $|i - j| \ge p$.*
2. *If Y occurs in Z at positions i and $i + d$, where $d \le m - p$, then d must be a multiple of p. If $0 < d \le \frac{m}{2}$, then Y must also occur at positions $i + kp$, where k is an integer such that $kp \le d$.*

Proof: Suppose that Y occurs in Z at positions i and j—say, $i < j \le i + m$. Then, $Y(1 : j - i)$ is a period of Y. Hence, $j - i \ge p$, where p is the period of Y. This result establishes this first statement of the lemma.

Concerning the second statement, note that, if Y occurs at positions i and $i + d$, then $Y(1 : d)$ is a period of Y, and hence d must be a multiple of p, since $d \le m - p$ (by Corollary 7.1). If $1 \le d \le \frac{m}{2}$, then clearly Y is periodic and Y will appear in all positions $i + kp$ such that $kp \le d$, where k is an integer. □

Corollary 7.2: *Let Y be a string whose period is p, and let Z be an arbitrary string of length n. Then, Y can occur in Z at most n/p times.* □

7.1.2 THE WITNESS ARRAY

We now introduce a function on strings that will play a fundamental role in our parallel string-matching algorithm.

Let Y be a string of length m and period p. Let $\pi(Y) = \min\left(p, \lceil \frac{m}{2} \rceil\right)$. That is to say, $\pi(Y)$ is equal to the period p if Y is periodic; otherwise, $\pi(Y)$ is equal to $\lceil \frac{m}{2} \rceil$. A **witness function** ϕ_{wit} is defined as follows: (1) $\phi_{wit}(1) = 0$, and, (2) for i between 2 and $\pi(Y)$, $\phi_{wit}(i) = k$, where k is some index for which $Y(k) \neq Y(i + k - 1)$.

Intuitively, imagine that a copy of Y is placed on top of Y such that the first and the ith positions are aligned. Then, the overlapping portions of the two copies cannot be identical, because otherwise $Y(1 : i - 1)$ would be a period of Y (recall that $i \leq \pi(Y) \leq p$, the period of Y). Hence, there exists at least one position for which the corresponding symbols are different. The index k is one such position.

A witness function will be represented by an array $WITNESS(1 : r)$ in the obvious way, where $r = \pi(Y)$.

EXAMPLE 7.4:

Consider the two strings $Y = abcaabcab$ and $Z = abcabcab$, introduced in Example 7.3. The following are two possible *WITNESS* arrays corresponding to Y and Z:

$Y : WITNESS = (0, 3, 2, 2, 5)$
$Z : WITNESS = (0, 3, 2)$ □

Given a string Z of length $n \geq m$, consider the problem of determining all the positions where the string Y can occur in Z. The array *WITNESS* corresponding to Y provides a powerful tool to disqualify many positions of Z as possible candidates for a matching.

Let i and j be two arbitrary positions of Z such that $|i - j| < \pi(Y)$. We know, from Lemma 7.2, that Y cannot occur at positions i and j simultaneously. Let us place two copies of Y on top of Z, one starting at position i, the other starting at position j (see Fig. 7.2). Then, $WITNESS(j - i + 1)$ provides a position at which the two copies of Y differ. Hence, a simple test comparing the symbols at these locations and the symbol at the corresponding location of Z will eliminate at least one of the positions i and j for a possible occurrence of Y in Z.

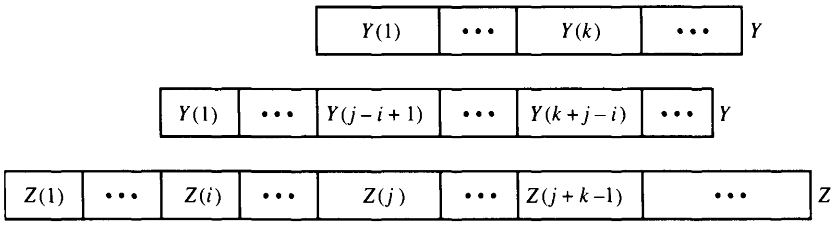

FIGURE 7.2
Elimination of one of the indices i or j as possible candidates for a matching by using $WITNESS(j - i + 1) = k$. Since $Y(k) \neq Y(k + j - i)$, comparing $Z(j + k - 1)$ with $Y(k)$ and $Y(k + j - i)$ will eliminate at least one of the indices i and j as possible locations where Y can occur in Z.

The algorithm is given more formally next.

ALGORITHM 7.1

(Duel(i, j))

Input: *(1) A string Z of length n; (2) a WITNESS array of another string Y of length $m \leq n$; and (3) two indices i and j, where $1 \leq i < j \leq n - m$, such that $j - i < \pi(Y) = \min\left(p, \lceil \frac{m}{2} \rceil\right)$ and p is the period of Y.*

Output: *One of i or j; string Y cannot occur in Z at the eliminated position.*

```
begin
  1. Set $k := WITNESS(j - i + 1)$
  2. if $Z(j + k - 1) \neq Y(k)$ then return($i$)
                                     else return($j$)
end
```

EXAMPLE 7.5:

Let $Y = abcabcab$ with the following *WITNESS* array: $WITNESS = (0, 3, 2)$, and let $Z = abcaabcabaa$. Consider the two positions $i = 5$ and $j = 7$ of Z. Since $WITNESS(7 - 5 + 1) = 2$, the duel algorithm checks $Y(2) = b$ against $Z(8) = a$. Since $Y(2) \neq Z(8)$, the index $i = 5$ is returned. □

Lemma 7.3: *Let Y and Z be two given strings, and let $i < j$ be two distinct indices of Z such that $j - i < \pi(Y)$. Then, no matching of Y in Z can occur at the position eliminated by duel(i, j). Moreover, this procedure takes $O(1)$ sequential time.*

Proof: Since $2 \le j - i + 1 \le p$, we have that $WITNESS(j - i + 1) = k \neq 0$. This fact implies that $Y(k) \neq Y(k + j - i)$. Consider location $j + k - 1$ of Z. We cannot simultaneously have the two equalities $Z(j + k - 1) = Y(k)$ and $Z(j + k - 1) = Y(k + j - i)$ (see Fig. 7.2). Clearly, if $Y(k) \neq Z(j + k - 1)$, then Y cannot occur at location j in Z. Hence, index i is returned. If $Z(j + k - 1) = Y(k)$, then j is returned, since $Z(j + k - 1) \neq Y(k + j - i)$, and hence Y cannot occur at location i of Z. In either case, no matching occurs at the eliminated index. Note that the index returned does not necessarily represent a matching at that location.

The duel algorithm can be implemented trivially in $O(1)$ sequential time; hence, the lemma follows. □

7.1.3 *THE PERIODS OF THE PREFIXES

In this section, we address the following string problem. Let Z and Y be strings of lengths n and m, respectively, such that $n \ge m$. Let i be an arbitrary position of Z such that Y does not occur in Z at postion i, where $1 \le i \le n - m + 1$. Let j be the smallest index such that $Y(j) \neq Z(i + j - 1)$. Such an index j always exists because Y does not occur in Z at position i (see Fig. 7.3). Our problem is to determine the closest next position (after i) that should be considered for the possible occurrence of Y in Z.

At first thought, it may seem that the location $i + 1$ is the next possible candidate position. However, based on the information we have, it may be possible to eliminate many of the consecutive positions $i + 1, i + 2, \ldots, n - m + 1$ from consideration. We next justify this statement more rigorously.

Assume that the index j is greater than 1. Then, we know that the two substrings $Y(1 : j - 1)$ and $Z(i : i + j - 2)$ are identical, since j is the smallest index such that $Y(j) \neq S(i + j - 1)$ (see Fig. 7.3). Hence, the substring $Y(1 : j - 1)$ has occurred at location i in Z. By Lemma 7.2, the next possible occurrence of $Y(1 : j - 1)$ in Z is at location $i + D(j)$, where $D(j)$ is the period of $Y(1 : j - 1)$. It follows that Y cannot occur in Z at any of the locations $i + 1, i + 2, \ldots, i + D(j) - 1$. Therefore, the next possible location of Z where Y can possibly occur is $i + D(j)$.

The case when $j = 1$ provides no useful information; hence, the next candidate position is $i + 1$, which can be expressed as $i + D(j)$ as in the case where $j > 1$.

EXAMPLE 7.6:

Consider the strings $Y = abcabcab$ and $Z = abcaabcabaa$. Let us try to match Y at location $i = 1$ of Z. Location 5 is the first position of a mismatch; hence, $j = 5$. The period of the prefix $Y(1 : 4) = abca$ is equal to 3; thus, $D(5) = 3$.

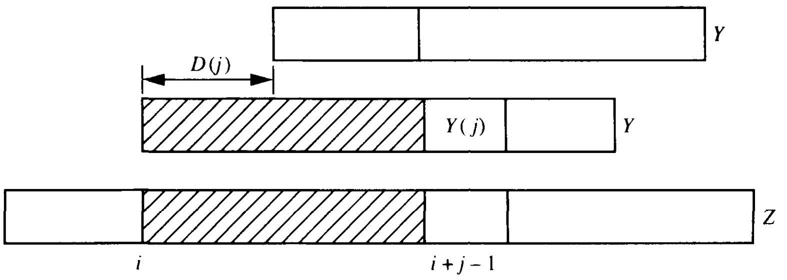

FIGURE 7.3
Determination of the next closest position where Y can possibly occur in Z. The index j is the smallest index such that $Y(j) \neq Z(i + j - 1)$; hence the two substrings $Y(1 : j - 1)$ (lower copy) and $Z(i : i + j - 2)$ are identical (hatched portions). Let $D(j)$ be the period of the substring $Y(1 : j - 1)$. The next possible location where $Y(1 : j - 1)$ can occur is $i + D(j)$; therefore, this location is the next location that should be considered for a possible occurrence of Y in Z.

This result implies that the next possible occurrence of Y in Z is at location $i + D(j) = 4$; hence, there is no need to consider positions 2 and 3 for a possible match. □

The function D introduced previously plays a fundamental role in the linear-time sequential algorithm for string matching described in Section 7.3.4.

7.2 String Matching

Given a *text* string and a *pattern* string, the **string-matching problem** is to find all the occurrences of the pattern in the string. More precisely, let the text and the pattern be represented by the arrays $T(1 : n)$ and $P(1 : m)$, respectively, such that $n \geq m$. We wish to determine a Boolean array $MATCH$ such that, for each $1 \leq i \leq n - m + 1$, $MATCH(i) = 1$ if and only if $P(1 : m)$ occurs in T at position i; that is, $P(j) = T(i + j - 1)$, for $1 \leq j \leq m$.

String matching is a fundamental problem in text editing. There are two classic $O(n + m)$ sequential algorithms for string matching: the Knuth–Morris–Pratt (KMP) algorithm, and the Boyre–Moore (BM) algorithm. A simple version of the KMP algorithm will be briefly outlined in Section 7.3.4. The reader can consult the references cited at the end of this chapter for more details concerning these two algorithms and several extensions. The techniques used in these two algorithms do not lead to an efficient parallel approach for string matching. In Sections 7.3 and 7.4, we develop different techniques that will yield an $O(\log m)$ time parallel string-matching algorithm using $O(n + m)$ operations.

There is a well-known paradigm to handle various pattern-matching problems. It consists of the following two main steps:

1. **Pattern analysis:** This step involves the processing of the pattern only. Some information regarding the structure of the pattern is extracted, and is stored in one or more tables.
2. **Text analysis:** This phase involves the processing of the text using the pattern and the information generated during the pattern-analysis phase.

Our method for handling the string-matching problem will also follow this paradigm. The pattern-analysis phase will consist of generating the array *WITNESS*, as introduced in the previous section; the second phase will generate the output array *MATCH* efficiently, by using the array *WITNESS*. We start by describing the latter phase.

7.3 Text Analysis

Our input to the text-analysis phase consists of the text array $T(1:n)$, the pattern array $P(1:m)$, and the *WITNESS* array of the pattern $WITNESS(1:r)$, where $r = \pi(P) = \min\left(p, \lceil \frac{m}{2} \rceil\right)$ and p is the period of the pattern P. Recall that $WITNESS(i) = k$ implies that $P(k) \neq P(i + k - 1)$, for $2 \leq i \leq r$ and $1 \leq k \leq m - i + 1$. We shall describe the algorithm to find all the occurrences of the pattern P in the text T.

The brute-force algorithm would try to match $P(1:m)$ against $T(i:i + m - 1)$ for each position i, where $1 \leq i \leq n - m + 1$. Clearly, $\Theta(m)$ operations are required for each such position, leading to an algorithm that uses a total of $O(nm)$ operations. This algorithm can be implemented in $O(1)$ time on the common CRCW PRAM, since the m comparisons required for each position i can be done concurrently in one parallel step, and the value $MATCH(i)$ is nothing but the Boolean AND of the m outcomes.

In spite of its speed, the brute-force algorithm requires an excessive number of operations compared with the linear number of operations used by several of the known sequential algorithms. We develop an alternative strategy that reduces the total number of operations to $O(n)$.

7.3.1 THE NONPERIODIC CASE

We start by considering the case when P is nonperiodic. The corresponding *WITNESS* array is of size $\lceil \frac{m}{2} \rceil$. In addition, we know from Lemma 7.2 that, given any segment of the text T of length $\le \frac{m}{2}$, P can occur at most once in that segment. This observation suggests the following approach.

Decompose T into blocks T_i, each T_i consisting of $\lceil \frac{m}{2} \rceil$ consecutive characters. Each such block can contain at most one position where a matching can occur. Suppose that we have eliminated all but one possible match point in each T_i. Then, the brute-force algorithm can be applied to all the possible match points concurrently in $O(1)$ time, using a total of $O\left(\frac{2n}{m} \times m\right) = O(n)$ operations. We now proceed to show how to disqualify all the positions but one, in each block T_i.

Initially, all the positions of a block T_i are potential candidates. Let j and k be two arbitrary such positions. The function *duel*(j, k) introduced in Algorithm 7.1 will eliminate one of the positions j and k in $O(1)$ sequential time using the array *WITNESS*. Therefore, the *duel* function can be used to eliminate all but one position as follows. We construct a balanced binary tree on the elements of T_i such that the value stored at each leaf is the index of the corresponding element, and the value stored at an internal node v is the value returned when the *duel* function is applied to the values stored in the children nodes. The index stored at the root is the only possible candidate for a matching. This process takes $O(\log m)$ time, using $O(m)$ operations.

Our text-analysis algorithm for the case when the pattern in nonperiodic is stated more formally next.

ALGORITHM 7.2

(Text Analysis, Nonperiodic Pattern)

Input: *Three arrays* $T(1:n)$, $P(1:m)$, *and* $WITNESS\left(1:\frac{m}{2}\right)$ *corresponding to a text, a pattern and a witness function of the pattern, respectively. It is assumed that P is nonperiodic, and m is even.*

Output: *The array MATCH, indicating all the positions where the pattern occurs in the text.*

```
begin
  1. Partition T into ⌈2n/m⌉ blocks T_i, each block containing no
     more than m/2 consecutive characters.
  2. For each block T_i, eliminate all but one position as a possible
     candidate for a matching by using a balanced binary tree, where
     each internal node u contains the index returned by the function
     duel(i, j), and i and j are the indices stored in the children of u.
  3. For each candidate position, verify whether P occurs at that
     position in T by using the brute-force algorithm.
end
```

EXAMPLE 7.7:

Let $T = babaababaaba$ and $P = abaab$. Clearly, P is nonperiodic and $\pi(P) = \lceil 5/2 \rceil = 3$. Suppose we are given the array $WITNESS = (0, 1, 2)$. Algorithm 7.2 starts by decomposing T into four blocks, each block containing three consecutive characters. Let $T_1 = bab, T_2 = aab, T_3 = aba$, and $T_4 = aba$. All these blocks are then handled concurrently. After the first round of duels, we obtain $duel(1, 2) = 2, duel(4, 5) = 5, duel(7, 8) = 7$ and $duel(10, 11) = 10$. The outcomes of these duels are based on $WITNESS(2) = 1$. After the second round of duels, we get $duel(2,3) = 2, duel(5,6) = 5, duel(7,9) = 7$ and $duel(10, 12) = 10$. Note that the outcomes of $duel(7, 9)$ and $duel(10, 12)$ are based on $WITNESS(3) = 2$ (a comparison between $P(2) = b$ and the nonexistent character $T(13)$ is assumed to yield a true value to the inequality $P(2) \neq T(13)$). Therefore we are left with the potential candidates 2, 5, 7 and 10—one per block. We can easily check that P occurs at locations 2 and 7. □

We now state the following lemma concerning Algorithm 7.2, whose proof is left to Exercise 7.8.

Lemma 7.4: *Algorithm 7.2 correctly identifies all the positions where the nonperiodic pattern can occur in the text. It can be implemented to run in $O(\log m)$ time, using $O(n)$ operations.* □

PRAM Model: Since the information stored in the *WITNESS* array will be used for all the blocks T_i, a concurrent-read capability is required by Algorithm 7.2. No concurrent write is used. Hence, Algorithm 7.2 runs on the CREW PRAM model. □

Remark 7.3: We developed, in Section 2.6, an optimal $O(\log \log n)$ time algorithm to compute the maximum of n numbers. It turns out that the *duel* function behaves like the maximum function, thereby allowing us to use the strategy of Section 2.6 for designing an $O(\log \log m)$ time optimal algorithm to eliminate all but one candidate in each block of size $m/2$. However, the resulting algorithm will require the common CRCW PRAM. The details are left to Exercise 7.9. □

7.3.2 THE PERIODIC CASE

We now consider the case when the pattern is periodic—say, $P(1:m) = u^k v$, where v is a proper prefix of u (which could be empty) and $|u| = p$ is the period of P. Our *WITNESS* array is then of length p.

Consider the prefix $P(1:2p-1)$ as a new pattern whose occurrences in T are to be found. We can easily adjust the array *WITNESS* so that it corresponds to $P(1:2p-1)$ by using the following lemma.

Lemma 7.5: *Given a pattern $P(1:m)$ that has period $p < m/2$, the array $WITNESS(1:p)$ can be assumed to satisfy $WITNESS(j) \leq p$, for any index j.*

Proof: Let $k = WITNESS(j)$. Define $k' = k \bmod p$, if k is not a multiple of p; otherwise, define $k' = p$. Since $P(1:m)$ is periodic of period p, $P(k) = P(k')$ and $P(j+k-1) = P(j+k'-1)$. Hence, we can set $WITNESS(j) := k' \leq p$. □

Since $P(1:2p-1)$ is nonperiodic, we can use Algorithm 7.2 to determine all the occurrences of $P(1:2p-1)$ in the text T in $O(\log p)$ time using $O(n)$ operations. As a result, we know each position i of T at which $P(1:2p-1)$ occurs. Each such position i can be easily checked for an occurrence of u^2v. Now the problem of generating $MATCH(i)$ comes down to checking whether u^2v occurs at positions $i, i+p, \cdots, i+(k-2)p$, where k is the integer defined before by $P(1:m) = u^kv$.

The text analysis for the case when the pattern is periodic is given next.

ALGORITHM 7.3

(Text Analysis, Periodic Case)

Input: *Three arrays $T(1:n)$, $P(1:m)$, and $WITNESS(1:p)$ representing respectively a text, a pattern, and a WITNESS array corresponding to $P(1:2p-1)$. We also know that the pattern is periodic and has period p.*

Output: *The array MATCH identifying all the positions in the text where the pattern occurs.*

```
begin
  1. Use Algorithm 7.2 to determine the occurrences of the prefix
     $P(1:2p-1)$ in $T$.
  2. For each occurrence of $P(1:2p-1)$ in $T$—say, at position
     $i$—find whether $u^2v$ occurs at $i$, where $u = P(1:p)$, $v = P(kp+1:m)$,
     and $k = \lfloor m/p \rfloor$. If it does, set $M(i) := 1$. For all the other
     positions $i$ such that $1 \leq i \leq n$, set $M(i) := 0$.
  3. For each $i$ such that $1 \leq i \leq p$, let $S_i$ be the subarray of $M$
     consisting of the bits $(M(i), M(i+p), M(i+2p), \ldots)$. For each
     position $l$ of $S_i$ that contains a 1, set $C(i,l) := 1$ if there are at least
     $k-1$ consecutive 1's starting at this position. For all the remaining
     positions $l$ of $S_i$, set $C(i,l) := 0$.
  4. For each $1 \leq j \leq n-m+1$, let $j = i + lp$, where $1 \leq i \leq p$
     and $l \geq 0$. Then, set $MATCH(j) := C(i, l+1)$.
end
```

EXAMPLE 7.8:

Let $T = bababababababaabab = (ba)^6(ab)^2$, and let $P = abababa = (ab)^3a$. Hence, the pattern P has the period ab of length 2. Since $P(1:3) = aba$, step 1 of Algorithm 7.3 identifies the locations 2, 4, 6, 8, 10, and 13, where $P(1:3)$ occurs in T. The only occurrences that can be extended to $(ab)^2a$ are in locations 2, 4, 6, and 8. Hence, the array M is given by $M = (0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$. During the execution of step 3, we generate the two subarrays $S_1 = (0, 0, 0, 0, 0, 0, 0, 0)$ and $S_2 = (1, 1, 1, 1, 0, 0, 0, 0)$. We then look for the occurrence of two consecutive 1's in each of S_1 and S_2. We therefore obtain $C(1, l) = 0$, for all values of $1 \leq l \leq 8$, and $C(2, 1) = C(2, 2) = C(2, 3) = 1$, and $C(2, l) = 0$ for $4 \leq l \leq 8$. Step 4 sets $MATCH(2){:} = C(2, 1) = 1$, $MATCH(4){:} = C(2, 2) = 1$, $MATCH(6){:} = C(2, 3) = 1$, and $MATCH(i){:} = 0$ otherwise. Therefore, P occurs in T at positions 2, 4, and 6. □

Lemma 7.6: *Algorithm 7.3 determines all the occurrences of the periodic pattern $P(1 : m)$ in the text $T(1 : n)$ in $O(\log m)$ time, using a total of $O(n)$ operations.*

Proof: The correctness proof follows from the discussion before the statement of the algorithm.

The complexity bounds can be estimated as follows. By Lemma 7.4, step 1 takes $O(\log m)$ time, using a total of $O(n)$ operations. As for step 2, testing whether the occurrence of $P(1 : 2p - 1)$ can be extended to u^2v can be done in $O(\log m)$ time, using $O(p)$ operations, for each such position by the brute-force algorithm. Since $P(1 : 2p - 1)$ can occur at most $O(\frac{n}{p})$ positions in T (Corollary 7.2), the total number of operations for step 2 is $O(n)$.

Setting up the S_i arrays required by step 3 can be done in $O(1)$ time, using a total of $O(n)$ operations. To compute $C(i, l)$, we do the following. We partition each S_i into buckets, each bucket containing $k - 1$ consecutive bits. We compute the suffix sums of each maximal sequence of 1's within each pair of consecutive buckets. The boundary of such a maximal sequence of 1's is either a 0 or a boundary of the combined pair of consecutive buckets. We can use the segmented prefix-sums algorithm (see Exercise 2.5) to compute the suffix sums in $O(\log k) = O(\log m)$ time, using a linear number of operations. Then, $C(i, l) = 1$ if and only if the suffix sum corresponding to position l in S_i is greater than or equal to $k - 1$. Hence, step 3 takes $O(\log m)$ time, using a linear number of operations.

Step 4 is straightforward; it takes $O(1)$ time, using $O(n)$ operations. Therefore, the running time of the whole algorithm is $O(\log m)$, and the total number of operations is $O(n)$. □

PRAM Model: Step 1 of Algorithm 7.3 requires the CREW PRAM model. The remaining steps can be executed on the CREW PRAM within the stated bounds. Note that steps 2, 3, and 4 can be implemented in $O(1)$ time optimally on the common CRCW PRAM. Such an implementation is left to Exercise 7.11. □