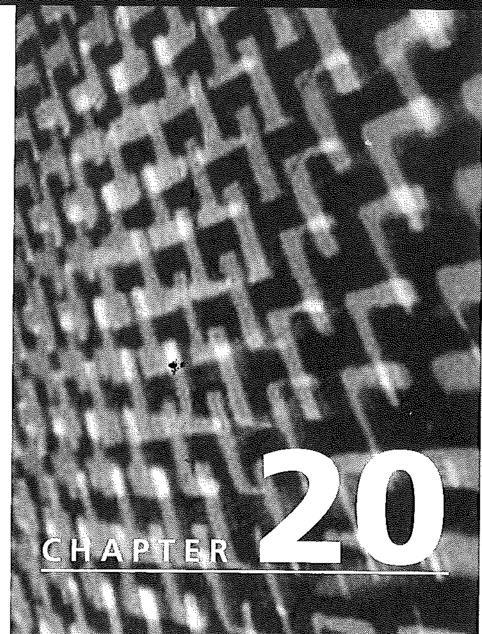


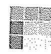
STRING MATCHING AND DOCUMENT PROCESSING

Finding some or all occurrences of a given pattern string in a given text is an important and commonly encountered problem. For example, most word processing software packages have built-in search-and-replace functions and spell checkers, both of which depend on finding the occurrences of words in texts. On the Internet, string matching is used for locating Web pages containing a given query string. String matching and approximate string matching is also a key technique in bioinformatics, which entails searching gene sequences for patterns of interest.

In this chapter, we present three standard string-matching algorithms, due to Knuth, Morris, and Pratt (the KMP algorithm); Boyer and Moore (the BM algorithm); and Karp and Rabin (the KR algorithm). The KMP and BM algorithms preprocess the pattern string so that information gained during a search for a match of the pattern can be used to shift the pattern more than the one position shifted by a naive algorithm when a mismatch occurs. The KR algorithm shifts the pattern by only one position at a time, but it performs an efficient (constant-time) check at each new position.

Often it is useful to find an approximate match in a text to a given pattern string. An important measure of approximation is known as the edit distance between two strings, which is, roughly speaking, the minimum number of single-character alterations that will transform one string into another. We present a dynamic programming solution to computing the edit distance between strings.

We finish the chapter with a discussion of tries and suffix trees. When the text is fixed, preprocessing the text as opposed to the pattern string leads to efficient string-matching algorithms. This preprocessing is based on constructing a trie and a related suffix tree corresponding to the text. Tries can be used to create inverted indexes to strings in a large collection of data files, such as Web pages on the Internet.



20.1 The Naive Algorithm

The string-matching problem can be formally described as follows. An *alphabet* is a set of characters or symbols $A = \{a_1, a_2, \ldots, a_k\}$. A *string* $S = S[0{:}n-1]$ of length n on A is a sequence of n characters (repetitions allowed) from A. Such a string $S = s_0s_1, \ldots, s_{n-1}$ can be viewed as an array of characters $S[0{:}n-1]$ from A, so that the $(i+1)^{st}$ character s_i in the string is denoted by $S[i]$, $i = 0, \ldots, n-1$. More generally, we denote the substring consisting of symbols in consecutive positions i through j of S by $S[i{:}j]$, $0 \le i \le j \le n-1$. The *null string*, denoted by ε, is the string that contains no symbols. We let A^* denote the set of all finite strings (including the null string ε) on A. The length of a string S, denoted $|S|$, is the number of characters in S. For $a \in A$, we let a^i denote the string of length i consisting of the single symbol a repeated i times.

Given a *pattern string* $P = P[0{:}m-1]$ of length m and *text string* $T = T[0{:}n-1]$ of length n, where $m \le n$, the string-matching problem is to determine whether P occurs in T. In our string-matching algorithms, we assume that we are looking for the first occurrence (if any) of the pattern string in the text string. The algorithms can be readily modified to return all occurrences of the pattern string.

A naive algorithm for finding the first occurrence of P in T is to position P at the start of T and simply shift the pattern P along T, one position at a time, until either a match is found or the string T is exhausted (that is, position $n - m + 1$ in T is reached without finding a match).

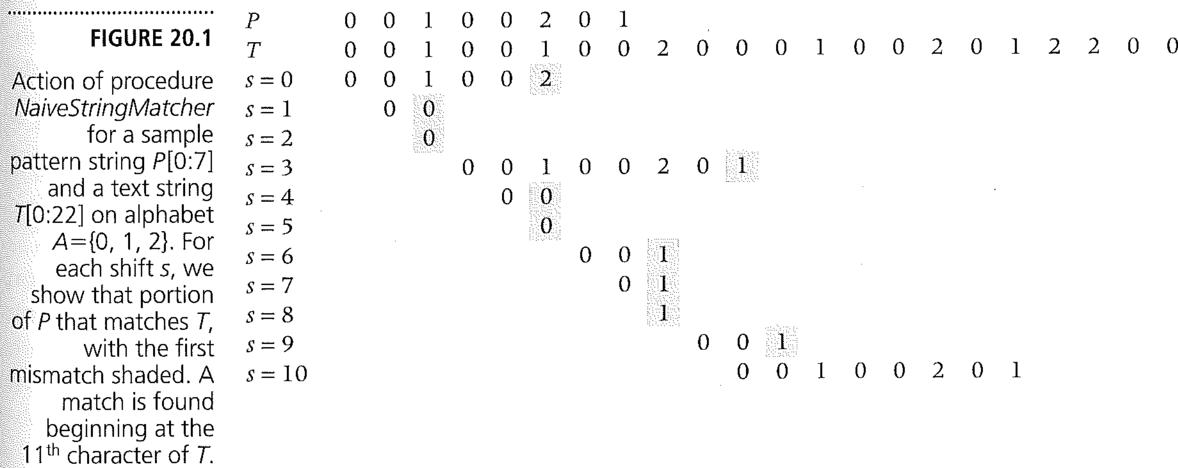
function *NaiveStringMatcher*($P[0{:}m-1]$, $T[0{:}n-1]$)
Input: $P[0{:}m-1]$ (a pattern string of length m)
$T[0{:}n-1]$ (a text string of length n)
Output: returns the position in T of the first occurrence of P, or -1 if P does not occur in T

```
    for s ← 0 to n − m do
        if T[s : s + m − 1] = P then
            return(s)
        endif
    endfor
    return(−1)
end NaiveStringMatcher
```

When measuring the complexity of procedure *NaiveStringMatcher*, it is natural to choose comparison of symbols as our basic operation. We can test whether $T[s{:}\, s + m - 1] = P$ by using a simple linear scan. Clearly, this scan requires a single comparison in the best case ($P[0] \neq T[s]$) and m comparisons in the worst case ($P[i] = T[s + i], i = 0, \ldots, m - 2$). Because the **for** loop of procedure *NaiveStringMatcher* is iterated $n - m + 1$ times, *NaiveStringMatcher* never performs more than $m(n - m + 1)$ comparisons. Moreover, $m(n - m + 1)$ comparisons are performed, for example, when $P[0{:}m - 1]$ and $T[0{:}n - 1]$ are the strings $0^{m-1}1$ and 0^n, respectively, over the alphabet $A = \{0, 1\}$. Thus, *NaiveStringMatcher* has worst-case complexity

$$W(m, n) = m(n - m + 1) \in O(nm).$$

The action of *NaiveStringMatcher* is illustrated for a sample pattern and text string in Figure 20.1.



P	0	0	1	0	0	2	0	1														
T	0	0	1	0	0	1	0	0	2	0	0	0	1	0	0	2	0	1	2	2	0	0
$s = 0$	0	0	1	0	0	2																
$s = 1$		0	0																			
$s = 2$			0																			
$s = 3$				0	0	1	0	0	2	0	1											
$s = 4$					0	0																
$s = 5$						0																
$s = 6$							0	0	1													
$s = 7$								0	1													
$s = 8$									1													
$s = 9$										0	0	1										
$s = 10$											0	0	1	0	0	2	0	1				

FIGURE 20.1
Action of procedure *NaiveStringMatcher* for a sample pattern string $P[0{:}7]$ and a text string $T[0{:}22]$ on alphabet $A = \{0, 1, 2\}$. For each shift s, we show that portion of P that matches T, with the first mismatch shaded. A match is found beginning at the 11th character of T.

20.2 The Knuth-Morris-Pratt Algorithm

There is an obvious inefficiency in *NaiveStringMatcher*: At a given point we might have matched a good part of the pattern P with the text T until we found a mismatch, but we don't exploit this in any way. The KMP string-matching algorithm is based on a strategy of using information from partial matchings of P to not only skip over portions of the text that cannot contain a match but also to avoid checking characters in T that we already know match a prefix of P. The KMP algorithm achieves $O(n)$ worst-case complexity by preprocessing the string P to obtain information that can exploit partial matchings.

To illustrate, consider the pattern string P = "00100201" and text string T = "0010010020001002012200". Placing P at the beginning of T, note that a mismatch occurs at index position 5. The naive algorithm would then shift by one to $T[1]$ and simply start all over again checking at the beginning of P. This completely ignores that we have already determined that $P[0:4] = 00100 = T[0:4]$. Indeed, by simply looking at $P[0:4]$, we see that the first position in T where a match could possibly occur is at $i = 3$, because any shift of P by less than three will cause mismatches between the relevant prefix of $P[0:3] = T[0:3]$ and suffix of $P[1:4] = T[1:4]$ determined by the shift. Indeed, if we shift by one, we would be comparing the prefix 0010 of $P[0:3]$ with the suffix 0100 of $P[1:4]$. Similarly, if we shift by two, we would be comparing the prefix 001 of $P[0:3]$ with the suffix 100 of $P[1:4]$. Thus, we need to shift by three before we match the prefix 00 of $P[0:3]$ with the suffix 00 of $P[1:4]$. Hence, from the mismatch that occurs at position $i = 5$, the next starting position where a match can occur is at position $5 - 2 = 3$. Moreover, we do not need to check the first two characters in P because they already match the first two characters of T at this new position for P. Note that next position where a match can occur was obtained by subtracting the length of the largest prefix of $P[0:3]$ that was also a suffix of $P[1:4]$ from the position where the mismatch occurred.

More generally, suppose we have detected a mismatch at position i in T, where $T[i] \neq P[j]$, but we know that the previous j characters of T match with $P[0:j-1]$. Also, suppose d_j is the length of the longest prefix of $P[0:j-2]$ that also occurs as a suffix of $P[1:j-1]$. Then the next position where a match can occur is at position $i - d_j$. Moreover, to see whether we actually have a match starting at position $i - d_j$, we can avoid checking the characters that we already know agree with those in T—that is, the characters in the substring $T[s - d_j:i - 1]$. Hence, we need only check the characters in the substring $T[i:i + m - d_j - 1]$ with those in the substring $P[d_j: m - 1]$ to see if a match occurs (see Figures 20.2 and 20.3).

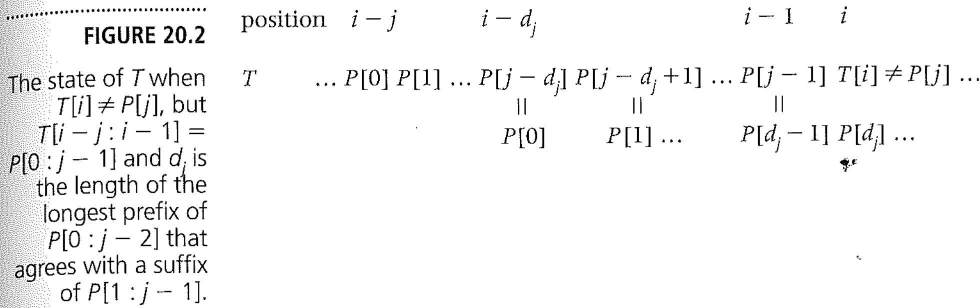


FIGURE 20.2

The state of T when $T[i] \neq P[j]$, but $T[i-j:i-1] = P[0:j-1]$ and d_j is the length of the longest prefix of $P[0:j-2]$ that agrees with a suffix of $P[1:j-1]$.

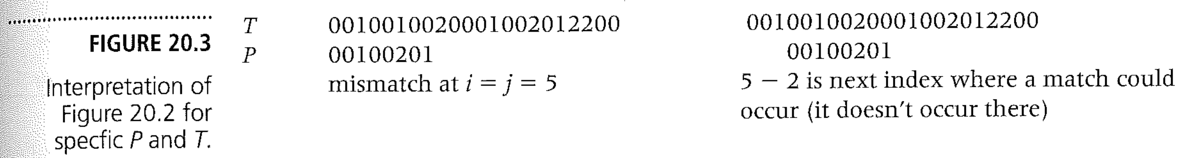


FIGURE 20.3

Interpretation of Figure 20.2 for specfic P and T.

By preprocessing the string P, we can compute the array $Next[0{:}m-1]$, where $Next[j]$ is the length of longest prefix of $P[0:j-2]$ that agrees with a suffix of $P[1:j-1]$, $j = 2, \ldots, m$. We set $Next[0] = Next[1] = 0$. For example, for the string P = "00100201", we have the corresponding array $Next[0{:}7] = [0, 0, 1, 0, 1, 2, 0, 1]$. Then for this P and the T discussed earlier, Figure 20.3 shows the situation described in Figure 20.2 for $i = 5$, $j = 5$, and $d_j = 2$.

The following key fact summarizes our discussion and is the key to the efficiency of the KMP string matching algorithm.

> **Key Fact**
>
> **Suppose in our scan of T looking for a match with P that we have a mismatch at position s in T, where $T[i] \neq P[j]$, but $P[0{:}j-1] = T[i-j:i-1]$. Setting $d_j = Next[j]$, the next position where a match of P can occur is at position $i - d_j$. Moreover, we need only check the substring $T[i : i+m-d_j-1]$ with the substring $P[d_j : m-1]$ to see if a match occurs there.**

In the pseudocode *KMPStringMatcher*, we look for matches using a variable i that scans the text T from left to right one position at a time, and a second variable j that scans the pattern P in a slightly oscillatory manner, as dictated by the key fact (see Figure 20.4). The variable i never backs up, so that when a match occurs, it is actually at position $i - m + 1$ (in other words, s is the position of the last character in P corresponding to the matching). The pseudocode is elegant but somewhat subtle, because when a mismatch $P[j] \neq T[i]$ occurs, there is no

need to explicitly place the pattern P at the next position $i - Next[j]$; we merely need to check $T[i : i + m - Next[j] - 1]$ against $P[Next[j] : m - 1]$ to see if a match of P occurs at $i - Next[j]$. We perform this check by continuing the scan of T by i and replacing j by $Next[j]$ before continuing the scan of P by j.

```
function KMPStringMatcher(P[0:m − 1], T[0:n − 1])
Input:    P[0:m − 1] (a pattern string of length m)
          T[0:n − 1] (a text string of length n)
Output:   returns the position in T of the first occurrence of P, or −1
          if P does not occur in T
    i ← 0                              //i runs through text string T
    j ← 0                              //j runs through pattern string P in manner
                                         dictated by key fact
    CreateNext(P[0:m − 1], Next[0:m − 1])
    while i < n do
        if P[j] = T[i] then
            if j = m − 1 then          //match found at position i − m + 1
                return(i − m + 1)
            endif
            i ← i + 1                  //continue scan of T
            j ← j + 1                  //continue scan of P
        else                           //P[j] ≠ T[i]
            j ← Next[j]                //continue looking for a match of P which
                                         now could begin at position i − Next[j]
                                         in T
            if j = 0 then
                if T[i] ≠ P[0] then    //no match at position i
                    i ← i + 1
                endif
            endif
        endif
    endwhile
    return(−1)
end KMPStringMatcher
```

In Figure 20.4, we illustrate the action of *KMPStringMatcher* for the pattern string P and text string T discussed earlier, by tracing the values of s and j for each iteration of the **while** loop. While *NaiveStringMatcher* used 37 comparisons to find a match, *KMPStringMatcher* only used 21 (not counting the comparisons made by *CreateNext* in preprocessing P).

iteration						!						!		!							
i	0	1	2	3	4	5	5	6	7	8	9	10	10	11	11	12	13	14	15	16	17
j	0	1	2	3	4	5	2	3	4	5	6	7	1	2	1	2	3	4	5	6	7

(a)

```
T    0 0 1 0 0 1 0 0 2 0 0 0 1 0 0 2 0 1 2 2 0 0
P    0 0 1 0 0 2 0 1
          0 0 1 0 0 2 0 1
                      0 0 1 0 0 2 0 1
                        0 0 1 0 0 2 0 1  match
```

(b)

FIGURE 20.4 (a) A trace of the values of i and j for each iteration of the **while** loop in *KMPStringMatcher* for P = "00100201" and T = "0010010020001002012200". Positions marked with ! are where $T[i] \neq P[j]$ and j is reassigned with the value $Next[j]$. (b) The implicit shifting of P until a match is found.

KMPStringMatcher has linear complexity because of the following key fact.

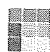
> **Key Fact**
>
> **The while loop in *KMPStringMatcher* is executed at most 2*n* times.**

To verify the key fact, note that the loop executes n times when i is incremented within the loop. If i is not incremented in the loop, then the pattern P is implicitly shifted to position $i - Next[j]$, which is at least one more than the last implicit placement of P. Therefore, this implicit shift can happen at most n times, which verifies the key fact.

It remains to design the algorithm *CreateNext*. Again, there is a naive algorithm *NaiveCreateNext* that computes each value of $Next[i]$ from scratch, without using any of the information gleaned from computing $Next[k]$, $k < i$. It is easy to see that the worst case of *NaiveCreateNext* is in $\Omega(m^2)$. However, similar to the design of *KMPStringMatcher*, using information gleaned about $Next[i - 1]$ leads to a more efficient way to compute $Next[i]$ and yields an $O(m)$ algorithm. We leave the complexity analysis and correctness of *CreateNext* to the exercises.

function *CreateNext*($P[0{:}m-1]$, $Next[0{:}m-1]$)
Input: $P[0{:}m-1]$ (a pattern string of length m)
Output: $Next[0{:}m-1]$ ($Next[i]$ is length of longest prefix of $P[0{:}i-2]$ that is a suffix of $P[1{:}i-1]$, $i = 0, \ldots, m-1$)

```
    Next[0] ← Next[1] ← 0
    i ← 2
    j ← 0
    while i < m do
        if P[j] = P[i − 1] then
            Next[i] ← j + 1
            i ← i + 1
            j ← j + 1
        else
            if j > 0 then
                j ← Next[j − 1]
            else
                Next[i] ← 0
                i ← i + 1
            endif
        endif
    endwhile
end CreateNext
```



20.3 The Boyer-Moore String-Matching Algorithm

Similar to the KMP algorithm, the BM algorithm uses preprocessing of the pattern string to facilitate shifting the pattern string, but it is based on a right-to-left scan of the pattern string instead of the left-to-right scan made by the KMP algorithm. We present a simplified version of the BM algorithm that compares the rightmost character of the pattern with the character in the text corresponding to the current shift of the pattern and uses this comparison to determine the next pattern shift (if any). The full version of the BM algorithm is developed in the exercises.

In the BM algorithm, the pattern $P[0{:}m - 1]$ is first placed at the beginning of the text, and we check for a match by scanning the pattern from right-to-left. If we find a mismatch, then we have two cases to consider, depending on the character x of the text in index position $m - 1$ that is compared against the last character of P. If x does not occur in the first $m - 1$ positions of P, then clearly we can shift P by its entire length m to continue our search for a match. If x does occur in the first $m - 1$ positions of P, then we shift P so that the rightmost occurrence of x in $P[0{:}m - 1]$ is now at index position $m - 1$ in the text, and we repeat the process of scanning P from right-to-left at this new position. Again, if we find a mismatch, we shift the pattern again based on the text character that was aligned at the rightmost character of P. Also, in this simplified version, we ignore any information gleaned about partial matchings in our previous placement of P (this information *is* used in the full version of the BM algorithm).

The shifts associated with the two cases can be computed easily by preprocessing the pattern string P. In the following pseudocode for *CreateShift*, for convenience, we assume that the array *Shift* is indexed by the alphabet A from which the characters for the pattern P and text T are drawn.

```
procedure CreateShift(P[0:m − 1], Shift[0: |A| − 1])
Input:    P[0:m − 1] (a pattern string)
Output:   Shift[0: |A| − 1] (the array of character-based shifts)
    for i ← 0 to |A| − 1 do                //initialize Shift to all m's.
        Shift[i] = m
    endfor
    for i ← 0 to m − 2 do                  //compute shifts based on rightmost
                                             occurrence of P[i] in P[0:m − 1]
        Shift[P[i]] = m − i − 1
    endfor
end CreateShift
```

For example, suppose $P[0:8]$ is the string "character". Then the values for the shifts of the characters e, t, c, a, r, h are 1, 2, 3, 4, 5, 7, respectively. The shift value for all other characters is 9. In Figure 20.5, we illustrate how the simplified BM algorithm uses these shifts to find a match of the pattern "character" in the text "BMmatcher_shift_character_example".

The worst-case performance of the simplfied BM algorithm is the same as that of the naive algorithm, $\Theta(nm)$ (see Exercise 20.8). However, it can be shown that its average behavior is linear in n and often works as well as the full version of the BM algorithm. The full version of the BM algorithm works identically to the simplified version when there is a mismatch between the rightmost character of the pattern and the text character corresponding to this rightmost character in the current shift of the pattern. If these two characters agree, however, the full version acts differently by exploiting the information gained by the matching of a suffix of P with the corresponding characters in the text for the given placement of P (see the discussion preceding Exercise 20.9).

```
       !          !            !
B M m a t c h e r _ s h i f t _ c h a r a c t e r _ e x a m p l e
c h a r a c t e r                                Shift(r) = 5
          c h a r a c t e r                      Shift(r) = 9
                        c h a r a c t e r        Shift(r) = 2
                            c h a r a c t e r    match
```

FIGURE 20.5
Action of simplified BM algorithm, with positions where mismatches first occur in the right-to-left scan of the pattern indicated by !.

20.4 The Karp-Rabin String-Matching Algorithm

In this section, we assume without loss of generality that our strings are chosen from the k-ary alphabet $A = \{0,1, \ldots, k - 1\}$. Each character of A can be thought of as a digit in radix-k notation, and each string $S \in A^*$ can be identified with the base k representation of an integer $\overline{S}$. For example, when $k = 10$, the string of numeric characters "6832355" can be identified with the integer 6832355. Given a pattern string $P[0{:}m - 1]$, we can compute the corresponding integer using m multiplications and m additions by employing Horner's rule.

$$\overline{P} = P[m-1] + k(P[m-2] + k(P[m-2] + k(P[m-3] + \cdots + k(P[1] + kP[0])\ldots)) \qquad \textbf{(20.4.1)}$$

Given a text string $T[0{:}n - 1]$ and an integer s, we find it convenient to denote the substring $T[s, s + m - 1]$ by T_s. A string-matching algorithm is obtained by using Horner's rule to successively compute $\overline{T_0}, \overline{T_1}, \overline{T_2}, \ldots$, where the computation continues until $\overline{P} = \overline{T_s}$ for some s (a match) or until we reach the end of the text T. Of course, this is no better than the naive string-matching algorithm. However, the following key fact is the basis of a linear algorithm.

Key Fact

Given the integers $\overline{T_{s-1}}$ and k^{m-1}, we can compute the integer $\overline{T_s}$ in constant time.

The key fact follows from the following recurrence relation:

$$\overline{T_s} = k(\overline{T_{s-1}} - k^{m-1}T[s-1]) + T[s+m-1] \quad s = 1, \ldots, n - m. \qquad \textbf{(20.4.2)}$$

For example, if $k = 10$, $m = 7$, $\overline{T_{s-1}} = 7937245$, and $\overline{T_s} = 9372458$, then recurrence relation (20.4.2) becomes

$$\overline{T_s} = 10[7937245 - (1000000 \times 7)] + 8 = 9372458.$$

The constant $c = k^{m-1}$ in (20.4.2) can be computed in time $O(\log m)$ using the binary method for computing powers. Once c is computed, it does not need to be recomputed when Formula (20.4.2) is applied again. Thus, assuming that the arithmetic operations in (20.4.2) take constant time, each application of (20.4.2) takes constant time. Hence, the $n - m + 2$ integers $\overline{P}$ and $\overline{T_s}$, $s = 0, 1, \ldots, n - m$, can be computed in total time $O(n)$.

The problem with the preceding approach is that the integers $\overline{P}$ and $\overline{T}_s$, $s = 0, 1, \ldots, n - m$, may be too large to work with efficiently, and the assumption that Formula (20.4.2) can be performed in constant time becomes unreasonable. To get around this difficulty, we reduce these integers modulo q for some randomly chosen integer q. To avoid multiple-precision arithmetic, q is often chosen to be a random prime number such that kq fits within one computer word.

We now let

$$\overline{P}^{(q)} = \overline{P} \bmod q,$$
$$\overline{T}_s^{(q)} = \overline{T}_s \bmod q. \qquad \textbf{(20.4.3)}$$

The values $\overline{T}_s^{(q)}$ and $\overline{P}^{(q)}$ can be computed in time $O(n)$ using exactly the same algorithm described earlier for computing $\overline{T}_s$ and $\overline{P}$, except that all arithmetic operations are performed modulo q. Clearly, if $\overline{T}_s^{(q)} \neq \overline{P}^{(q)}$ then $T_s \neq P$. However, if $\overline{T}_s^{(q)} = \overline{P}^{(q)}$, we are not guaranteed that $P = T_s$. When a shift s has the property that $\overline{T}_s^{(q)} = \overline{P}^{(q)}$, but $T_s \neq P$ we have a *spurious match*. However, for sufficiently large q, the probability of a spurious match can be expected to be small. We check whether a match is spurious by explicitly checking whether $T_s = P$, and continuing our search for a match if $T_s \neq P$.

```
function KarpRabinStringMatcher(P[0:m − 1], T[0:n − 1], k, q)
Input:   P[0:m − 1] (a pattern string of length m)
         T[0:n − 1] (a text string of length n)
         k (A is the k-ary alphabet {0, 1, . . . , k − 1})
         q (a random prime number q such that kq fits in one computer word)
Output:  returns the position in T of the first occurrence of P, or −1 if P does not occur
         in T
   c ← k^(m − 1) mod q
   P̄^(q) ← 0
   T̄_0^(q) ← 0
   for i ← 0 to m − 1 do                        //apply Horner's rule to compute
                                                  P̄^(q) and T̄_0^(q)
      P̄^(q) ← (k*P̄^(q) + P[i]) mod q
      T̄_0^(q) ← (k*T̄_0^(q) + T[i]) mod q
   endfor
   for s ← 0 to n − m do
      if s > 0 then
         T̄_s^(q) ← (k*(T̄_(s−1)^(q) − T[s − 1]*c) + T[s + m−1]) mod q
      endif
```

if $\overline{T_s}^{(q)} = \overline{P}^{(q)}$ then
    if $T_s = P$ then   //match is not spurious
        return(s)
    endif
endif
endfor
return(0)
end *KarpRabinStringMatcher*

Figure 20.6 illustrates the action of function *KarpRabinStringMatcher* for a sample pattern string $P[0{:}6]$ = "6832355", text string $T[0{:}20]$ = "895732102583235544031", and prime modulus $q = 11$. To illustrate the calculations of $\overline{T_s}^{(11)}$ in Figure 20.6, we show how *KarpRabinStringMatcher* computes $\overline{T_3}^{(11)}$ from $\overline{T_2}^{(11)}$. Using Formula (20.4.2) and doing all arithmetic modulo $q = 11$, we have

$$\begin{aligned} 7321026 &\equiv 10(5732102 - 1000000 \times 5) + 6 \pmod{11} \\ &\equiv 10(2 - 1 \times 5) + 6 \pmod{11} \\ &\equiv 9 \pmod{11}. \end{aligned}$$

Because *KarpRabinStringMatcher* terminates after finding the first nonspurious match, the worst-case performance occurs for an input pair (P, T), where the pattern string P occurs precisely at the end ($s = n - m$) of the text string T. With q chosen at random, we can expect different behavior for different choices of q, so that we now consider the expected number $\tau_{\exp}(P, T)$ of string comparisons made by the algorithm. For $s \neq n - m$, we make the assumption that $\overline{T_s}^{(q)}$ takes on a particular value $i \in \{0, 1, \ldots, q-1\}$ with equal probability $1/q$. Because a spurious match occurs only when $\overline{T_s}^{(q)} = \overline{P}^{(q)}$, $s = 0, \ldots, n - m - 1$, it follows that

FIGURE 20.6 Action of *KarpRabinString-Matcher* for a sample pattern string $P[0{:}6]$, text string $T[0{:}20]$, and prime modulus $q = 11$.

$T[0{:}20]$ 895732102683235544031
$P[0{:}6]$ 6832355

s							T_s										$\overline{T_s}^{(11)}$
0	8	9	5	7	3	2	1										10
1		9	5	7	3	2	1	0									9
2			5	7	3	2	1	0	2								2 Spurious match
3				7	3	2	1	0	2	6							9
4					3	2	1	0	2	6	8						6
5						2	1	0	2	6	8	3					0
6							1	0	2	6	8	3	2				4
7								0	2	6	8	3	2	3			0
8									2	6	8	3	2	3	5		5
9										6	8	3	2	3	5	5	2 Match: return (9)

a spurious match occurs at shift s with probability $1/q$. Let r denote the expected number of spurious matches. A test for a spurious match involves m comparisons in the worst case, and $r + 1$ such tests are performed (including the test at shift $s = n - m$); thus, the expected performance $\tau_{exp}(P, T)$ of *KarpRabinStringMatcher* for the input (P, T) is

$$\tau_{exp}(P, T) = (r + 1)m + (n - m + 1). \qquad \textbf{(20.4.4)}$$

The value r is the expectation of the binomial distribution with $n - m$ trials (shifts), where success (a spurious match) occurs with probability $1/q$. Thus, from Formula (E.3.9) of Appendix E, we have

$$r = \frac{n - m}{q}. \qquad \textbf{(20.4.5)}$$

Substituting Formula (20.4.5) into Formula (20.4.4), we obtain

$$\tau_{exp}(P, T) = \left(\frac{n - m}{q} + 1\right)m + (n - m + 1). \qquad \textbf{(20.4.6)}$$

If we assume that q is bounded above by a fixed constant, then *KarpRabinStringMatcher* achieves a worst-case complexity in $\Theta(nm)$, which is no better than the naive algorithm. However, in practice, it is reasonable to assume that q is much larger than m, in which case *KarpRabinStringMatcher* has complexity in $O(n)$. The Karp-Rabin string-matching algorithm has the additional feature that it is readily adapted to the problem of finding $m \times m$ patterns in $n \times n$ texts (see Exercise 20.16).

20.5 Approximate String Matching

In practice, there are often misspellings when creating a text, and it is useful when searching for a pattern string P in a text to find words that are approximately the same as P. In this section, we formulate a solution to this problem using dynamic programming. We have already discussed a solution to a similar problem in Chapter 9—namely, the problem of finding the longest common subsequence of two strings.

We first consider the problem of determining whether a pattern string $P[0{:}m - 1]$ is a k-approximation of a text string $T[0{:}n - 1]$. Later, we look at the problem of finding occurrences of substrings of T for which P is a k-approximation. The pattern string P is a *k-approximate matching* of the text string T if T can be converted to P using at most k operations involving one of the following

1. Changing a character of T (substitution)
2. Adding a character to T (insertion)
3. Removing a character of T (deletion)

For example, when P is the string "algorithm", one of the following might occur:

1. elgorithm → algorithm (*substitution* of e with a)
2. algorthm → algorithm (*insertion* of letter i)
3. lalgorithm → algorithm (*deletion* of letter l)

In this example, each string T differs from P by at most one character. Unfortunately, in practice, more serious mistakes are made, and the difference involves multiple characters. We define the *edit distance $D(P, T)$* between P and T to be the minimum number of operations of substitution, deletion, and insertion needed to convert T to P. For example, the strings "algorithm" and "logarithm" have edit distance 3.

logarithm → alogarithm → algarithm → algorithm

Let $D[i,j]$ denote the edit distance between the substring $P[0:i-1]$ consisting of the first i characters of the pattern string P and $T[0:j-1]$ consisting of the first j characters of the text string T. If $P[i] = T[j]$, then $D[i,j] = D[i-1,j-1]$. Otherwise, consider an optimal intermixed sequence involving the three operations substitution, insertion, and deletion that converts $T[0:j-1]$ into $P[0:i-1]$. The number of such operations is the edit distance between these two substrings. Note that in transforming T to P, inserting a character into T is equivalent to deleting a character from P. For convenience, we will perform the equivalent operation of deleting characters from P rather than adding characters to T. We can assume without loss of generality that the sequence of operations involving the first $i-1$ characters of P and the first $j-1$ characters of T are operated on first. To obtain a recurrence relation for $D[i,j]$, we examine the last operation. If the last operation is substitution of $T[j]$ with $P[i]$ in T, then $D[i,j] = D[i-1,j-1] + 1$. If the last operation is the deletion of $P[i]$ from P, then $D[i,j] = D[i-1,j] + 1$. Finally, if the last operation is deletion of $T[j]$ from T, then $D[i,j] = D[i,j-1] + 1$ The edit distance is realized by computing the minimum of these three possibilities. Observing that the edit distance between a string of size i and the null string is i, we obtain the following recurrence relation for the edit distance:

$$D[i,j] = \begin{cases} D[i-1,j-1], & \text{if } P[i] = T[j], \\ \min\{D[i-1,j-1]+1,\ D[i-1,j]+1,\ D[i,j-1]+1\}, & \text{otherwise.} \end{cases}$$

init. cond. $D[0,i] = D[i,0] = i$.

(20.5.1)

The design of a dynamic programming algorithm based on this recurrence and its analysis is similar to that given for the longest common subsequence problem discussed in Chapter 9, and we leave it to the exercises. We also leave as an exercise designing an algorithm for finding the first occurrence or all occurrences of a substring of the text string T that is a k-approximation of the pattern string P.

20.6 Tries and Suffix Trees

By preprocessing the pattern string, the KMP and BM algorithms achieved improvement over the naive algorithm. Another approach that can be applied when the text is fixed is to preprocess the strings in the text using a data structure such as a tree. In this section, we discuss two important tree-based data structures, tries and suffix trees, for preprocessing the text to allow for very efficient pattern matching and information retrieval.

20.6.1 Standard Tries

Consider a collection C of strings from an alphabet A of size k, where no string in S is a prefix of any other string. We can then construct a tree T whose nodes are labeled with symbols from A, such that the strings in C correspond precisely to the paths in T from the root R to a leaf node as follows. We construct T such that the labels of the children of each node are unique and occur in increasing order as the children are scanned from left to right. Starting with the tree T consisting of a single root node R, we inductively incorporate a new string $S[0{:}p-1]$ from C into T as follows. If no child of the root R is labeled $S[0]$, then we simply add a new branch at the root consisting of a path of length p whose node at level $i+1$ is labeled $S[i]$, $i = 0, \ldots, p-1$. Otherwise, we follow a path from the root by first following the edge from the root to the unique child of the root labeled $S[0]$, then following the edge from that node to its child labeled $S[1]$, and so forth until we reach a node v at level i labeled $S[i-1]$ having no child (at level $i+1$) labeled $S[i]$. We then add a new branch at v consisting of a path of length $p-i-1$, such that node in the path at level j (in the tree) is labeled $S[j-1]$, $j = i+1, \ldots, p-1$. A tree T constructed in this way is called a *standard trie* for the string collection C. Figure 20.7 shows a standard trie for the sample string collection C = {"internet", "interview", "internally", "algorithm", "all", "web", "world"}.

The leaf nodes of the trie T can be used to store information about the string S corresponding to the leaf, such as the location in the text of P, the number of occurrences of P in the text, and so forth. The term *trie* comes from the word re*trie*val, because a trie can be used to retrieve information about P. In addition to pattern matching, tries can be used for word matching, where the pattern is matched only to substrings of the text corresponding to words. This is useful for creating a forward index of words in a web document.

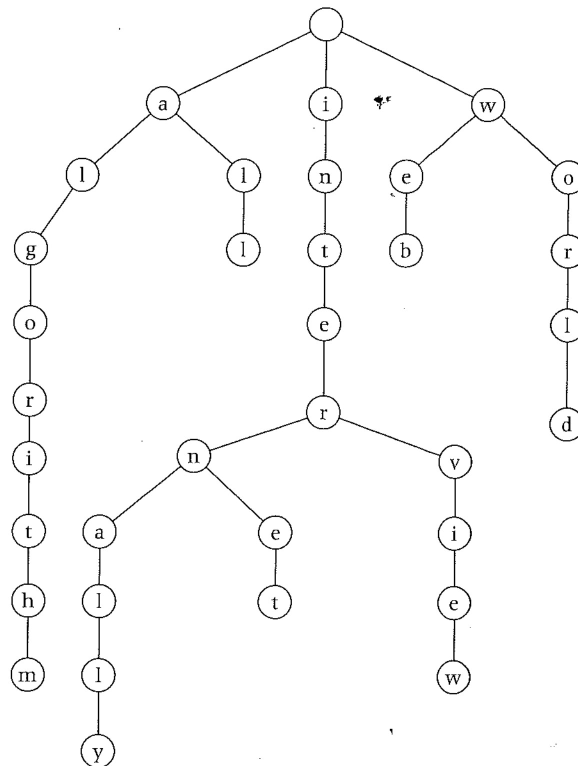


FIGURE 20.7

Standard trie for the collection of strings C = {"internet", "interview", "internally", "algorithm", "all", "web", "world"}.

It is immediate that a standard trie T has the following three properties: (1) Each nonleaf node has at most k children, where k is the size of the alphabet A; (2) the number of leaf nodes equals the number of strings in S; and (3) the depth of T equals the length of the longest string in S. The following proposition about the space requirements for storing T is easily verified (see Exercise 20.22).

Proposition 20.6.1 Let T be a standard trie for a collection C of strings, and let s denote the total length over all the strings in C. Then the number of nodes $N(T)$ of T satisfies

$$N(T) \in O(s).$$

We can efficiently test whether a given pattern string $P[0{:}m - 1]$ belongs to C by scanning the string P and successively following the child in the trie labeled with the current symbol that has been scanned until either no child of the current node has a label equal to the symbol or a leaf node labeled with the last symbol of P has been reached. Because we have made the assumption that no string in C is a prefix of any other string, it follows that the pattern string

$P[0{:}m - 1]$ belongs to C if and only if a leaf node labeled with the last symbol in P is reached. Because each node has at most k children, this procedure has complexity $O(km)$. Thus, if the alphabet has constant size, the complexity of searching for P is linear in its length.

20.6.2 Compressed Tries

The $O(s)$ space requirement of a standard trie T can be reduced if there are nodes in T that have only one child. Consider any such node v, and let c denote its only child. Let x and y denote the labels of v and c, respectively, and let u denote the parent of v. Then the path generated by a string S from C that contains v must also contain c. Thus, without affecting our ability to match S, we can compress the trie by removing v, making c a child of u, and replacing the label y of c with the string xy. This operation can be repeated for other nodes having only one child, except that x and y may themselves be strings instead of just single symbols. After repeatly performing this compression operation until all internal nodes have at least two children, we obtain a tree labeled with strings, which we call the *compressed trie* for S. A compressed trie is also called a PATRICIA (practical algorithm to retrieve information coded in alphanumeric) tree . Note that the compressed trie for S can also be obtained by replacing every path $uu_1 \ldots u_k v$ from a node u to a node v in T whose internal nodes $u_1, u_2, \ldots, u_k$ are bivalent (have exactly one child), but whose end nodes u and v are not, with the edge uv and replacing the label y of v with the string $x_1 x_2 \ldots x_k y$, where x_i denotes the label of u_i, $i = 1, \ldots, k$. The compressed trie for the trie given in Figure 20.7 is shown in Figure 20.8.

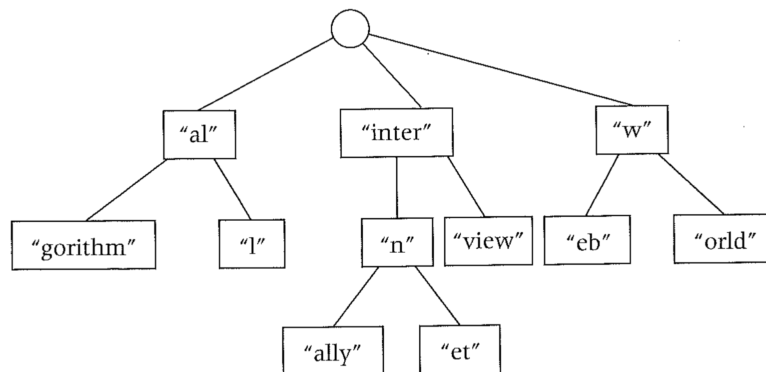


FIGURE 20.8
Compressed trie for the same string set C = {"internet", "interview", "internally", "algorithm", "all", "web", "world"} of Figure 20.7

The following proposition about the number of nodes of a compressed trie T is easily verified.

Proposition 20.6.2 Let T be a compressed trie for a collection C of c strings. Then the number of nodes $N(T)$ of T satisfies

$$N(T) \in O(c).$$

Comparing this result with Proposition 20.6.1, we see that compressing the standard trie has reduced the space requirements from $O(s)$ to $O(c)$. This becomes significant when the strings in C are long. A compressed trie can be created directly from the set of strings C without first constructing a standard trie for C and then compressing it. We leave as an exercise designing an algorithm for constructing a compressed trie directly from C. Using a slight modification of the technique used for a standard trie, we can search a compressed trie to efficiently test whether a given pattern string belongs to C (see Exercise 20.24).

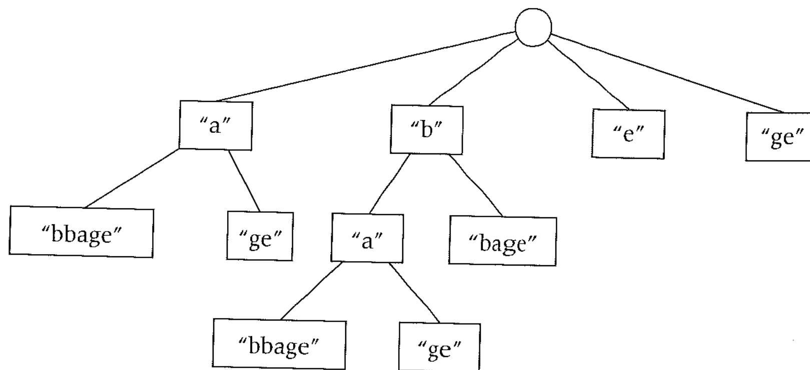
20.6.3 Suffix Trees

A *suffix tree* (also called a *suffix trie*) with respect to a given text string T is a compressed trie for the string collection C consisting of all suffixes of T. This definition requires that no suffix be a prefix of any other suffix. For strings in which this occurs, we simply add a special symbol to the end of every suffix in C. Suffix trees are useful in practice because they can be used to determine whether a pattern string P is a substring of a given text string T. The suffix tree for the string $T =$ "babbage" is shown in Figure 20.9(a). Because the string label on each node corresponds to a substring $T[i:j]$ of T, it can be represented more compactly using just the pair (i, j). Figure 20.9(b) shows the more compact representation of the node labels for the suffix tree in part (a).

Given a pattern $P[0:m-1]$ in string P, it is easy to design an $O(km)$ algorithm that traces a path in the suffix tree corresponding to test T to determine whether P occurs as a substring of T. We leave the design of such an algorithm as an exercise.

FIGURE 20.9

(a) Suffix tree for string $T =$ "babbage".



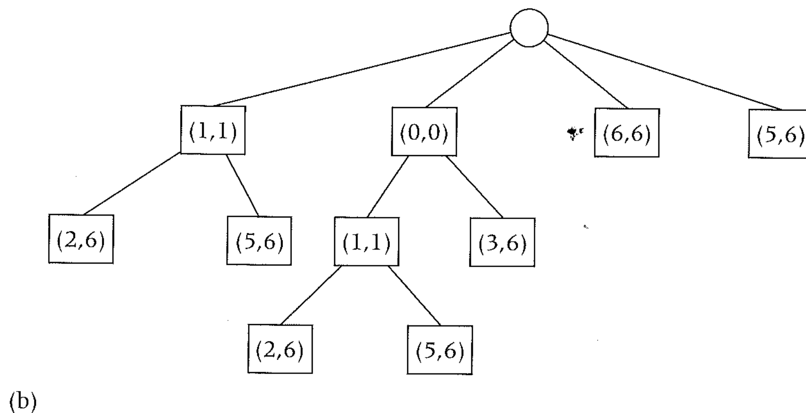


FIGURE 20.9 Continued. (b) more compact representation of the node labels for the suffix tree in part (a).

20.7 Closing Remarks

String and pattern matching have been areas of interest for a long time, and string algorithms have recently received increased attention because of their role in Web searching as well as in computational biology. In this chapter, we have introduced several of the most important string-matching algorithms, but the subject is vast. The interested reader should consult the references for more extended treatments of this important topic and its applications.

References and Suggestions for Further Reading

Books on string matching:

Aoe, J. *Computer Algorithms: String Pattern Matching Strategies*. Wiley-IEEE Computer Society Press, 1994.

Crochemore, M. *Text Algorithms*. New York: Oxford University Press, 1994.

Gusfield, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge: Cambridge University Press, 1997.

Navarro, G., and Raffinot, M. *Flexible Pattern Matching in Strings*. Cambridge: Cambridge University Press, 2002.

Stephen, G. A. *String Searching Algorithms*. London: World Scientific Publishing, 1994.

Chen, D., and Cheng, X., eds. *Pattern Recognition and String Matching*. Dordrecht, The Netherlands: Kluwer Academic Publishers, 2002. A collection of 28 articles contributed by experts on pattern recognition and string matching.

Survey articles on string matching:

Baeza-Yates, R. A. "Algorithms for String Matching: A Survey." *ACM SIGIR Forum* 23 (1989): 34–58.

Navarro, G. "A Guided Tour to Approximate String Matching." *ACM Computing Surveys* 33, no. 1 (2001): 31–88.

EXERCISES

Section 20.2 The Knuth-Morris-Pratt Algorithm

20.1. Suppose $P[0{:}m-1]$ and $T[0{:}n-1]$ are the strings $0^{m-1}1$ and 0^n that were the worst-case strings of length m and n for *NaiveStringMatcher*.

a. Show that $P[0{:}m-1]$ and $T[0{:}n-1]$ are best-case strings of length m and n for *KMPStringMatcher* for the case where P does not occur in T.

b. Find worst-case strings $P[0{:}m-1]$ and $T[0{:}n-1]$, respectively, for *KMPStringMatcher*, and thereby determine $W(m,n)$ for *KMPStringMatcher.*

20.2 Compute the array *Next*[0:10] for the pattern string P = "abracadabra".

20.3 Trace the action of *KMPStringMatcher* as in Figure 20.4 for the pattern string P = "cincinnati" and the text string T = "cincinatti_is_cincinnati_misspelled".

20.4 Verify the correctness of the algorithm *CreateNext*.

20.5 Show that *CreateNext* has $O(m)$ complexity.

20.6 Write programs implementing *NaiveStringMatcher* and *KMPStringMatcher*, and run them for various inputs for comparison.

Section 20.3 The Boyer-Moore String-Matching Algorithm

20.7 Design and analyze pseudocode for the simplified BM algorithm.

20.8 Show that the worst case of the simplified BM algorithm is as bad as the naive algorithm.

Exercises 20.9 through 20.14 involve the full version of the BM algorithm. As mentioned, when a mismatch occurs in the last position of the pattern string, the full version works the same way as the simplified version using the value *Shift*[*c*] to shift the pattern based on the mismatched text character *c*. The difference arises when we have matched the last $k > 0$ characters of the pattern string P, called a *good suffix* (of length k), before a mismatch occurs with a character c from the text. We then shift the pattern by the larger of the following two shifts, called the *bad character shift* s_1 and the *good suffix shift* s_2, respectively. The bad character shift s_1 is simply defined as $s_1 = \max\{Shift[c] - k, 1\}$. We can shift P by s_1 and not miss any matches, for reasons similar to those used when comparing against the last character in P.

The good suffix shift s_2 is also based on reasoning similar to that used to create the array *Shift*, but where we consider suffixes of *P* instead of a single character. More precisely, we look for the rightmost repeated occurrence of the good suffix of length k (if any) to shift this occurrence by the amount s_2 required to bring it to the end of the pattern. Of course, the character preceding this repeated occurrence (if any) must be different from the character preceding the good suffix; otherwise, a mismatch will occur again. Even when such a repeated occurrence of the good suffix does not happen, we might not be able to shift the pattern by its entire length m because we might miss a match that might occur when a suffix of length j of a good suffix of length k, $0 < j < k$, matches a prefix of length j of P. In the latter case, we shift by the amount required to bring the prefix to the end of the pattern P. For example, if $P = 011101001$, then the good suffix shifts for $k = 1, \dots, 8$, which are given by 5, 3, 7, 7, 7, 7, 7, 7, respectively.

20.9 Design and analyze pseudocode for an algorithm that creates the good-suffix shift values for a given input pattern $P[0{:}m - 1]$.

20.10 Compute the bad-character shifts and the good-suffix shifts for the following patterns:

a. $P =$ "01212121"

b. $P =$ "001200100"

20.11 Trace the action of the full version of the BM algorithm for the pattern $P =$ "amalgam" and text $T =$ "ada_gamely_amasses_amalgam_information".

20.12 Design and analyze pseudocode for the full version of the BM algorithm.

20.13 Write programs implementing the simplified and full versions of the BM algorithm, and compare their performance for various inputs.

20.14 Compare the performance of the programs written in the previous exercise to programs implementing *NaiveStringMatcher* and *KMPStringMatcher* (see Exercise 20.6) for various inputs.

Section 20.4 The Karp-Rabin String-Matching Algorithm

20.15 Trace the action of *KarpRabinStringMatcher* for the alphanumeric strings $P =$ "108" and $T =$ "002458108235" for the following values of q:

a. $q = 7$

b. $q = 11$

20.16 Adapt the Karp-Rabin string matching algorithm to the problem of finding $m \times m$ patterns $P[0{:}m-1, 0{:}m-1]$ in $n \times n$ texts $T[0{:}n-1, 0{:}n-1]$.

20.17 Verify recurrence relation (20.4.2).

20.18 Write a program implementing *KarpRabinStringMatcher*, and test its performance for various input strings and randomly generated modulus q.

Section 20.5 Approximate String Matching

20.19 a. Design and give pseudocode for a dynamic programming algorithm for approximate string matching based on recurrence relation (20.5.1).

b. Analyze the approximate string algorithm that you gave in part (a).

20.20 a. Show how the k-approximate string-matching algorithm can be modified to find the first *substring* of the text string T that is a k-approximation of the pattern string P.

b. Repeat part (a) for the problem of finding *all* occurrence of substrings of T that are k-approximations of P.

20.21 Show the $n \times m$ matrix $D[0{:}7, 0{:}10]$ that results from solving the recurrence relation (20.5.1) for the pattern string $P[0{:}7]$ = "patricia" and text string $P[0{:}8]$ = "patriarch".

Section 20.6 Tries and Suffix Trees

20.22 Prove Proposition 20.6.1.

20.23 Design an algorithm for constructing a compressed trie directly from a collection C of strings (without first constructing a standard trie and compressing).

20.24 Design an algorithm for searching a compressed trie to efficiently test whether a given pattern string belongs to the associated collection C.

20.25 Given a pattern $P[0{:}m-1]$ in string P, design and analyze an algorithm that traces a path in a suffix tree for text string T to determine whether P occurs as a substring of T.