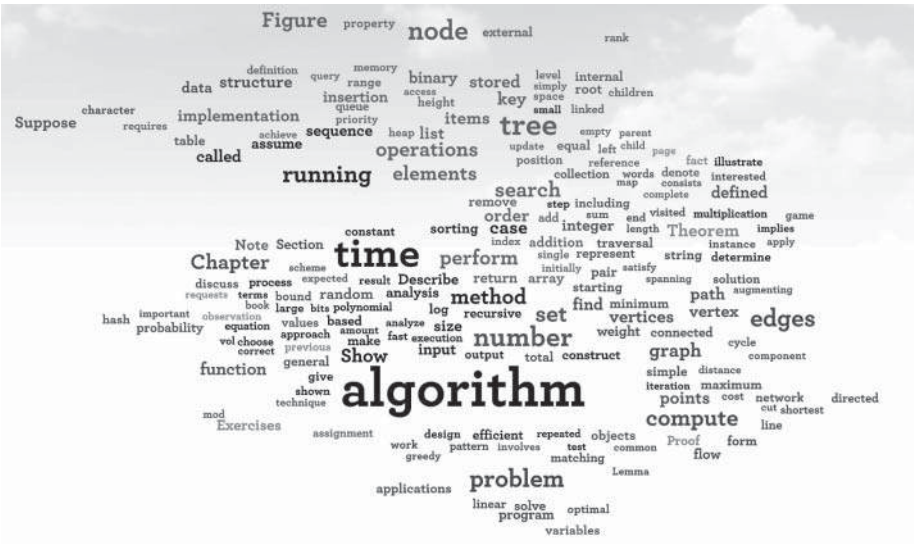
# Chapter 23 String Algorithms




Algorithms Book Word Cloud, 2014. Word cloud produced by frequency ranking the words in this book using wordcloud.cs.arizona.edu. Used with permission.

## Contents


**23.1 String Operations . . . . . 653**
**23.2 The Boyer-Moore Algorithm . . . . . 656**
**23.3 The Knuth-Morris-Pratt Algorithm . . . . . 660**
**23.4 Hash-Based Lexicon Matching . . . . . 664**
**23.5 Tries . . . . . 669**
**23.6 Exercises . . . . . 680**

Document processing is one of the main applications of computers. We use computers to edit documents, to search documents, to transport documents over the Internet, and to display documents on printers and computer screens. Web searching is a significant and important document processing application, and many of the key computations in all of this document processing involve character strings and string pattern matching. For example, the Internet document formats HTML and XML are primarily text formats, with added links to multimedia content. Making sense of the many terabytes of information on the Internet requires a considerable amount of text processing.

In this chapter, we study several fundamental text processing algorithms for quickly performing important operations on strings of characters. We pay particular attention to algorithms for string searching and pattern matching, since these can often be computational bottlenecks in many document processing applications. We also study some fundamental algorithmic issues involved in text processing, as well.

Not surprisingly, text processing algorithms operate primarily on inputs that use character strings as the underlying data type of the objects involved. The terminology and notation for strings, as used in this chapter, is fairly intuitive, and it turns out that representing a string as an array of characters is simple and efficient. Here, we typically think of the input size, $n$, as referring to the length of the strings used as inputs. So we don't spend a lot of attention discussing data structures for string representations. For instance, we can think of a string simply as an array of characters, with many string operations amounting to simple query operations on such arrays. Nevertheless, string processing often involves an interesting method for string pattern matching, and we study pattern matching algorithms in this chapter.

We also study the trie data structure, which is a tree-based structure that allows for fast searching in a collection of strings. One of the special cases of this data structure is the suffix trie, which allows for a number of interesting queries to be performed on strings.

We should also mention that there are several string problems discussed in previous chapters. For instance, we discuss an important text processing problem in Section 10.3—namely, the problem of compressing a document of text so that it fits more efficiently in storage or can be transmitted more efficiently over a network. In addition, in Section 12.5, we deal with how we can measure the similarity between two documents, based on the use of dynamic programming to solve the longest common subsequence problem. All of these problems are topics that arise often in Internet computations, such as web crawlers, search engines, document distribution, and information retrieval. We discuss, for instance, how the trie data structure can be used to implement a supporting data structure for a search engine.

# 23.1 String Operations

Text documents are ubiquitous in modern computing, as they are used to communicate and publish information. From the perspective of algorithm design, such documents can be viewed as simple character strings. That is, they can be abstracted as a sequence of the characters that make up their content. Performing interesting searching and processing operations on such data, therefore, requires that we have efficient methods for dealing with character strings.

At the heart of algorithms for processing text are methods for dealing with character strings. Character strings can come from a wide variety of sources, including scientific, linguistic, and Internet applications. Indeed, the following are examples of such strings:

$$\begin{aligned} P &= \texttt{"CGTAAACTGCTTTAATCAAACGC"} \\ R &= \texttt{"U.S. Lands an Astronaut on Mars!"} \\ S &= \texttt{"http://www.wiley.com/college/goodrich/"}. \end{aligned}$$

The first string, $P$, comes from DNA applications, the last string, $S$, is the URL for the website that accompanies this book, and the middle string, $R$, is a fictional news headline. In this section, we present some of the useful operations that are supported by string representations for processing strings.

## Substrings

Several of the typical string processing operations involve breaking large strings into smaller strings. In order to be able to speak about the pieces that result from such operations, we use the term ***substring*** of an $m$-character string $P$ to refer to a string of the form $P[i]P[i+1]P[i+2]\cdots P[j]$, for some $0 \leq i \leq j \leq m-1$, that is, the string formed by the characters in $P$ from index $i$ to index $j$, inclusive. Technically, this means that a string is actually a substring of itself (taking $i = 0$ and $j = m-1$), so if we want to rule this out as a possibility, we must restrict the definition to ***proper*** substrings, which require that either $i > 0$ or $j < m-1$. To simplify the notation for referring to substrings, let us use $P[i..j]$ to denote the substring of $P$ from index $i$ to index $j$, inclusive. That is,

$$P[i..j] = P[i]P[i+1]\cdots P[j].$$

We use the convention that if $i > j$, then $P[i..j]$ is equal to the ***null string***, which has length 0. In addition, in order to distinguish some special kinds of substrings, let us refer to any substring of the form $P[0..i]$, for $0 \leq i \leq m-1$, as a ***prefix*** of $P$, and any substring of the form $P[i..m-1]$, for $0 \leq i \leq m-1$, as a ***suffix*** of $P$. For example, if we again take $P$ to be the string of DNA given above, then `"CGTAA"` is a prefix of $P$, `"CGC"` is a suffix of $P$, and `"TTAATC"` is a (proper) substring of $P$. Note that the null string is a prefix and a suffix of any other string.

## The Pattern Matching Problem

In the classic ***pattern matching*** problem on strings, we are given a ***text*** string $T$ of length $n$ and a ***pattern*** string $P$ of length $m$, and want to find whether $P$ is a substring of $T$. The notion of a "match" is that there is a substring of $T$ starting at some index $i$ that matches $P$, character by character, so that

$$T[i] = P[0],\ T[i+1] = P[1],\ \ldots,\ T[i+m-1] = P[m-1].$$

That is,

$$P = T[i..i+m-1].$$

Thus, the output from a pattern matching algorithm is either an indication that the pattern $P$ does not exist in $T$ or the starting index in $T$ of a substring matching $P$.

To allow for fairly general notions of a character string, we typically do not restrict the characters in $T$ and $P$ to come explicitly from a well-known character set, like the ASCII or Unicode character sets. Instead, we typically use the general symbol $\Sigma$ to denote the character set, or ***alphabet***, from which the characters of $T$ and $P$ can come. This alphabet $\Sigma$ can, of course, be a subset of the ASCII or Unicode character sets, but it could also be more general and is even allowed to be infinite. Nevertheless, since most document processing algorithms are used in applications where the underlying character set is finite, we usually assume that the size of the alphabet $\Sigma$, denoted with $|\Sigma|$, is a fixed constant.

**Example 23.1:** *Suppose we are given the text string*

$$T = \texttt{"abacaabaccabacabaabb"}$$

*and the pattern string*

$$P = \texttt{"abacab"}.$$

*Then $P$ is a substring of $T$. Namely, $P = T[10..15]$.*

## Brute-Force Pattern Matching

The ***brute-force*** approach pattern is a technique for algorithm design when we have something we wish to search for or when we wish to optimize some function and we can afford to spend a considerable amount of time optimizing it. In applying this technique in a general situation, we typically enumerate all possible configurations of the inputs involved and pick the best of all these enumerated configurations.

In applying this technique to the ***pattern matching*** algorithm, we simply test all the possible placements of $P$ relative to $T$. This approach, shown in Algorithm 23.1, is quite simple.

The brute-force pattern matching algorithm could not be simpler. It consists of two nested loops, with the outer loop indexing through all possible starting indices of the pattern in the text, and the inner loop indexing through each character of the

**Algorithm** BruteForceMatch$(T, P)$:
***Input:*** Strings $T$ (text) with $n$ characters and $P$ (pattern) with $m$ characters
***Output:*** Starting index of the first substring of $T$ matching $P$, or an indication that $P$ is not a substring of $T$

```
for i ← 0 to n − m // for each candidate index in T do
    j ← 0
    while (j < m and T[i + j] = P[j]) do
        j ← j + 1
    if j = m then
        return i
return "There is no substring of T matching P."
```

**Algorithm 23.1:** Brute-force pattern matching.

pattern, comparing it to its potentially corresponding character in the text. Thus, the correctness of the brute-force pattern matching algorithm follows immediately.

The running time of brute-force pattern matching in the worst case is not good, however, because, for each candidate index in $T$, we can perform up to $m$ character comparisons to discover that $P$ does not match $T$ at the current index. Referring to Algorithm 23.1, we see that the outer for-loop is executed at most $n - m + 1$ times, and the inner loop is executed at most $m$ times. Thus, the running time of the brute-force method is $O((n - m + 1)m)$, which is $O(nm)$. Note that, when $m = n/2$, this algorithm has quadratic running time $O(n^2)$.

In Figure 23.2 we illustrate the execution of the brute-force pattern matching algorithm on the strings $T$ and $P$ from Example 23.1.

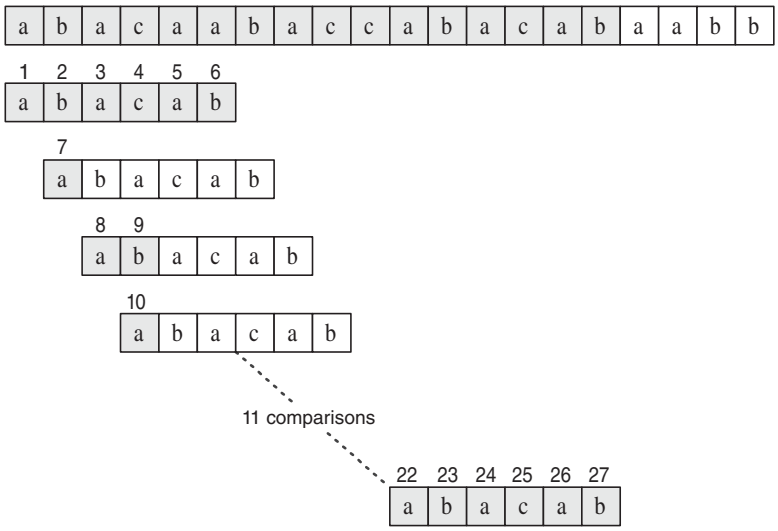



**Figure 23.2:** Example run of the brute-force pattern matching algorithm. The algorithm performs 27 character comparisons, indicated above with numerical labels.

## 23.2 The Boyer-Moore Algorithm

At first, we might feel that it is always necessary to examine every character in $T$ in order to locate a pattern $P$ as a substring. But this is not always the case, for the ***Boyer-Moore*** (***BM***) pattern matching algorithm, which we study in this section, can sometimes avoid comparisons between $P$ and a sizable fraction of the characters in $T$. The only caveat is that whereas the brute-force algorithm can work even with a potentially unbounded alphabet, the BM algorithm assumes the alphabet is of fixed, finite size. It works most quickly when the alphabet is moderately sized and the pattern is relatively long.

In this section, we describe a simplified version of the original BM algorithm. The main idea is to improve the running time of the brute-force algorithm by adding two potentially time-saving heuristics:

***Looking-Glass Heuristic*:** When testing a possible placement of $P$ against $T$, begin the comparisons from the end of $P$ and move backward to the front of $P$.

***Character-Jump Heuristic*:** During the testing of a possible placement of $P$ against $T$, a mismatch of text character $T[i] = c$ with the corresponding pattern character $P[j]$ is handled as follows. If $c$ is not contained anywhere in $P$, then shift $P$ completely past $T[i]$ (for it cannot match any character in $P$). Otherwise, shift $P$ until an occurrence of character $c$ in $P$ gets aligned with $T[i]$.

We will formalize these heuristics shortly, but at an intuitive level, they work as an integrated team. The looking-glass heuristic sets up the other heuristic to allow us to avoid comparisons between $P$ and whole groups of characters in $T$. In this case at least, we can get to the destination faster by going backward, for if we encounter a mismatch during the consideration of $P$ at a certain location in $T$, then we are likely to avoid lots of needless comparisons by significantly shifting $P$ relative to $T$ using the character-jump heuristic. The character-jump heuristic pays off big if it can be applied early in the testing of a potential placement of $P$ against $T$.

Therefore, let us define how the character-jump heuristics can be integrated into a string pattern matching algorithm. To implement this heuristic, we define a function $\mathsf{last}(c)$ that takes a character $c$ from the alphabet and specifies how far we may shift the pattern $P$ if a character equal to $c$ is found in the text that does not match the pattern. In particular, we define $\mathsf{last}(c)$ as follows:

- If $c$ is in $P$, $\mathsf{last}(c)$ is the index of the last (right-most) occurrence of $c$ in $P$. Otherwise, we conventionally define $\mathsf{last}(c) = -1$.

If characters can be used as indices in arrays, then the $\mathsf{last}$ function can be easily implemented as a lookup table. We leave the method for computing this table

efficiently as a simple exercise (R-23.6). The last function will give us all the information we need to perform the character-jump heuristic. In Algorithm 23.3, we show the BM pattern matching method. The jump step is illustrated in Figure 23.4.

**Algorithm** BMMatch$(T, P)$:
***Input:*** Strings $T$ (text) with $n$ characters and $P$ (pattern) with $m$ characters
***Output:*** Starting index of the first substring of $T$ matching $P$, or an indication that $P$ is not a substring of $T$

```
compute function last
i ← m − 1
j ← m − 1
repeat
    if P[j] = T[i] then
        if j = 0 then
            return i        // a match!
        else
            i ← i − 1
            j ← j − 1
    else
        i ← i + m − min(j, 1 + last(T[i]))        // jump step
        j ← m − 1
until i > n − 1
return "There is no substring of T matching P."
```

**Algorithm 23.3:** The Boyer-Moore pattern matching algorithm.

In Figure 23.5, we illustrate the execution of the Boyer-Moore pattern matching algorithm on a similar input string as in Example 23.1.

## Worst-Case Analysis of the Boyer-Moore Algorithm

The correctness of the BM pattern matching algorithm follows from the fact that each time the method makes a shift, it is guaranteed not to "skip" over any possible matches. For last$(c)$ is the location of the ***last*** occurrence of $c$ in $P$.

The worst-case running time of the BM algorithm is $O(nm + |\Sigma|)$. Namely, the computation of the last function takes time $O(m + |\Sigma|)$ and the actual search for the pattern takes $O(nm)$ time in the worst case, the same as the brute-force algorithm. An example of a text-pattern pair that achieves the worst case is

$$\begin{aligned} T &= \overbrace{aaaaaa\cdots a}^{n} \\ P &= b\overbrace{aa\cdots a}^{m-1}. \end{aligned}$$

The worst-case performance, however, is unlikely to be achieved for English text.

## Improved Analysis of the Boyer-Moore Algorithm

Indeed, the BM algorithm is often able to skip over large portions of the text. (See Figure 23.6.) There is experimental evidence that on English text, the average number of comparisons done per text character is approximately $0.24$ for a five-character pattern string. The payoff is not as great for binary strings or for very short patterns, however, in which case the KMP algorithm, discussed in Section 23.3, or, for very short patterns, the brute-force algorithm, may be better.

We have actually presented a simplified version of the Boyer-Moore (BM) algorithm. The original BM algorithm achieves running time $O(n + m + |\Sigma|)$ by using an alternative shift heuristic to the partially matched text string, whenever it shifts the pattern more than the character-jump heuristic. This alternative shift heuristic is based on applying the main idea from the Knuth-Morris-Pratt pattern matching algorithm, which we discuss in the next section.

## Worst-Case Improvement for String Pattern Matching

In studying the worst-case performance of the brute-force and BM pattern matching algorithms on specific instances of the problem, such as that given in Example 23.1, we should notice a major inefficiency. Specifically, we may perform many comparisons while testing a potential placement of the pattern against the text, yet if we

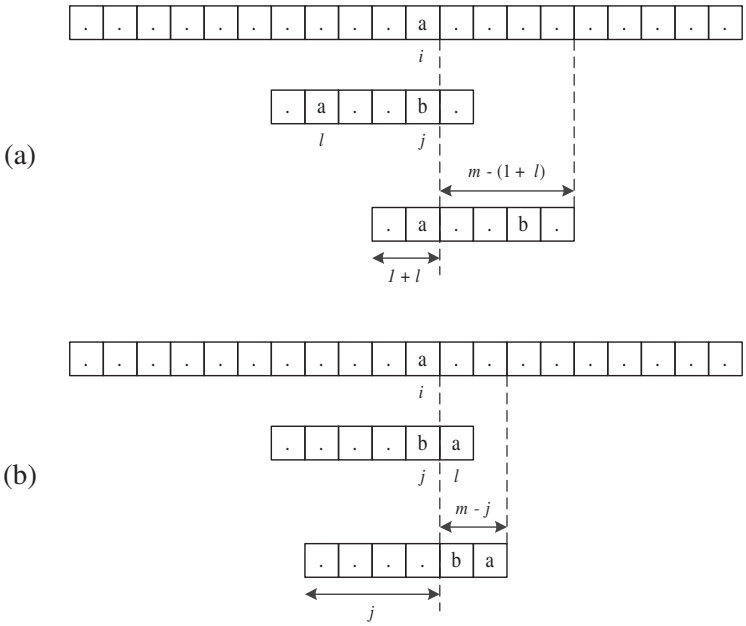
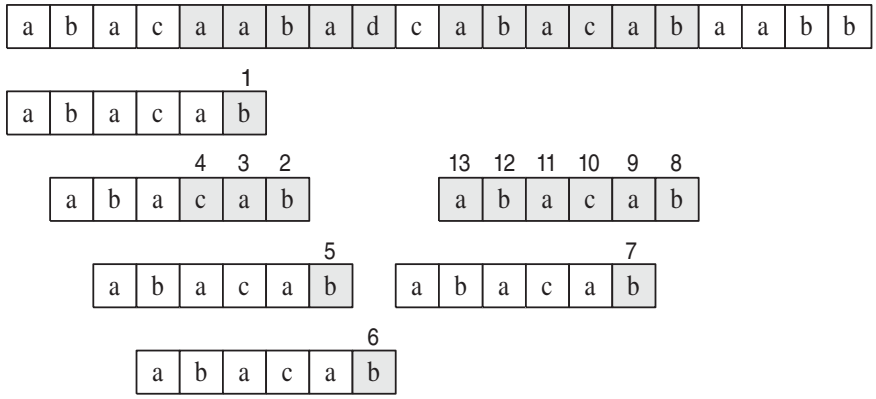



**Figure 23.4:** Illustration of the jump step in the BM algorithm, where $l$ denotes $\mathsf{last}(T[i])$. We distinguish two cases: (a) $1 + l \leq j$, where we shift the pattern by $j - l$ units; (b) $j < 1 + l$, where we shift the pattern by one unit.




The $\mathsf{last}(c)$ function:

| $c$ | a | b | c | d |
|---|---|---|---|---|
| $\mathsf{last}(c)$ | 4 | 5 | 3 | $-1$ |

**Figure 23.5:** An illustration of the BM pattern matching algorithm. The algorithm performs 13 character comparisons, which are indicated with numerical labels.

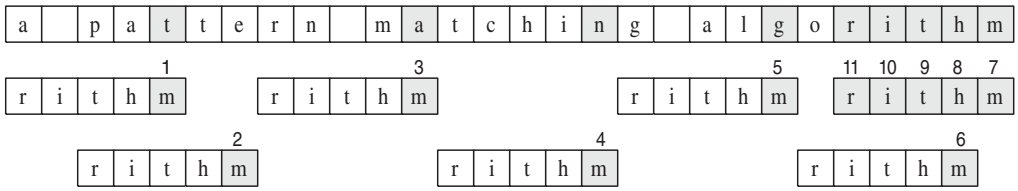



**Figure 23.6:** Execution of the Boyer-Moore algorithm on an English text and pattern, where a significant speedup is achieved. Note that not all text characters are examined.

discover a pattern character that does not match in the text, then we throw away all the information gained by these comparisons and start over again from scratch with the next incremental placement of the pattern.

The Knuth-Morris-Pratt (or "KMP") algorithm, which we discuss next, avoids this waste of information and, in so doing it achieves a running time of $O(n + m)$, which is optimal in the worst case. That is, in the worst case any pattern matching algorithm will have to examine all the characters of the text and all the characters of the pattern a constant number of times.

## 23.3 The Knuth-Morris-Pratt Algorithm

The main idea of the KMP algorithm is to preprocess the pattern string $P$ so as to compute a ***failure function*** $f$ that indicates the proper shift of $P$ so that, to the largest extent possible, we can reuse previously performed comparisons. Specifically, the failure function $f(j)$ is defined as the length of the longest prefix of $P$ that is a suffix of $P[1..j]$ (note that we did ***not*** put $P[0..j]$ here). We also use the convention that $f(0) = 0$. Later, we will discuss how to compute the failure function efficiently. The importance of this failure function is that it "encodes" repeated substrings inside the pattern itself.

**Example 23.2:** *Consider the pattern string $P =$ "abacab" from Example 23.1. The KMP failure function $f(j)$ for the string $P$ is as shown in the following table:*

| $j$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $P[j]$ | a | b | a | c | a | b |
| $f(j)$ | 0 | 0 | 1 | 0 | 1 | 2 |

The KMP pattern matching algorithm, shown in Algorithm 23.7, incrementally processes the text string $T$ comparing it to the pattern string $P$.

**Algorithm** KMPMatch$(T, P)$:
***Input:*** Strings $T$ (text) with $n$ characters and $P$ (pattern) with $m$ characters
***Output:*** Starting index of the first substring of $T$ matching $P$, or an indication that $P$ is not a substring of $T$

```
f ← KMPFailureFunction(P)        // construct the failure function f for P
i ← 0
j ← 0
while i < n do
    if P[j] = T[i] then
        if j = m − 1 then
            return i − m + 1      // a match!
        i ← i + 1
        j ← j + 1
    else if j > 0 // no match, but we have advanced in P then
        j ← f(j − 1)              // j indexes just after prefix of P that must match
    else
        i ← i + 1
return "There is no substring of T matching P."
```

**Algorithm 23.7:** The KMP pattern matching algorithm.

### Intuition Behind the KMP Algorithm

During the execution of the KMP algorithm, each time there is a match, we increment the current indices. On the other hand, if there is a mismatch and we have previously made progress in $P$, then we consult the failure function to determine the new index in $P$ where we need to continue checking $P$ against $T$. Otherwise (there was a mismatch and we are at the beginning of $P$), we simply increment the index for $T$ (and keep the index variable for $P$ at its beginning). We repeat this process until we find a match of $P$ in $T$ or the index for $T$ reaches $n$, the length of $T$ (indicating that we did not find the pattern $P$ in $T$).

The main part of the KMP algorithm is the while-loop, which performs a comparison between a character in $T$ and a character in $P$ each iteration. Depending upon the outcome of this comparison, the algorithm either moves on to the next characters in $T$ and $P$, consults the failure function for a new candidate character in $P$, or starts over with the next index in $T$. The correctness of this algorithm follows from the definition of the failure function. The skipped comparisons are actually unnecessary, for the failure function guarantees that all the ignored comparisons are redundant—they would involve comparing characters we already know match.

In Figure 23.8, we illustrate the execution of the KMP pattern matching algorithm on the same input strings as in Example 23.1. Note the use of the failure function to avoid redoing one of the comparisons between a character of the pattern and a character of the text. Also note that the algorithm performs fewer overall comparisons than the brute-force algorithm run on the same strings (Figure 23.2).

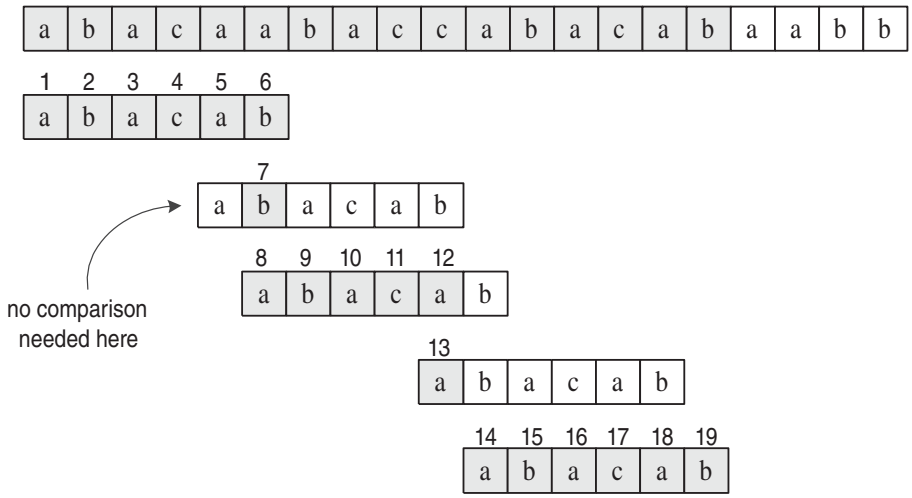



**Figure 23.8:** An illustration of the KMP pattern matching algorithm. The failure function $f$ for this pattern is given in Example 23.2. The algorithm performs 19 character comparisons, which are indicated with numerical labels.

### Analysis of the KMP Algorithm

Excluding the computation of the failure function, the running time of the KMP algorithm is clearly proportional to the number of iterations of the while-loop. For the sake of the analysis, let us define $k = i - j$. Intuitively, $k$ is the total amount by which the pattern $P$ has been shifted with respect to the text $T$. Note that throughout the execution of the algorithm, we have $k \leq n$. One of the following three cases occurs at each iteration of the loop.

- If $T[i] = P[j]$, then $i$ increases by 1, and $k$ does not change, since $j$ also increases by 1.
- If $T[i] \neq P[j]$ and $j > 0$, then $i$ does not change and $k$ increases by at least 1, since in this case $k$ changes from $i-j$ to $i-f(j-1)$, which is an addition of $j - f(j-1)$, which is positive because $f(j-1) < j$.
- If $T[i] \neq P[j]$ and $j = 0$, then $i$ increases by 1 and $k$ increases by 1, since $j$ does not change.

Thus, at each iteration of the loop, either $i$ or $k$ increases by at least 1 (possibly both); hence, the total number of iterations of the while-loop in the KMP pattern matching algorithm is at most $2n$. Of course, achieving this bound assumes that we have already computed the failure function for $P$.

### Constructing the KMP Failure Function

To construct the failure function used in the KMP pattern matching algorithm, we use the method shown in Algorithm 23.9. This algorithm is another example of a "bootstrapping" process quite similar to that used in the KMPMatch algorithm. We compare the pattern to itself as in the KMP algorithm. Each time we have two characters that match, we set $f(i) = j + 1$. Note that since we have $i > j$ throughout the execution of the algorithm, $f(j-1)$ is always defined when we need to use it.

Algorithm KMPFailureFunction runs in $O(m)$ time. Its analysis is analogous to that of algorithm KMPMatch. Thus, we have the following:

**Theorem 23.3:** *The Knuth-Morris-Pratt algorithm performs pattern matching on a text string of length $n$ and a pattern string of length $m$ in $O(n+m)$ time.*

The running time analysis of the KMP algorithm may seem a little surprising at first, for it states that, in time proportional to that needed just to read the strings $T$ and $P$ separately, we can find the first occurrence of $P$ in $T$. Also, it should be noted that the running time of the KMP algorithm does not depend on the size of the alphabet.

**Algorithm** KMPFailureFunction($P$):
***Input:*** String $P$ (pattern) with $m$ characters
***Output:*** The failure function $f$ for $P$, which maps $j$ to the length of the longest prefix of $P$ that is a suffix of $P[1..j]$

```
i ← 1
j ← 0
f(0) ← 0
while i < m do
    if P[j] = P[i] then
        // we have matched j + 1 characters
        f(i) ← j + 1
        i ← i + 1
        j ← j + 1
    else if j > 0 then
        // j indexes just after a prefix of P that must match
        j ← f(j − 1)
    else
        // we have no match here
        f(i) ← 0
        i ← i + 1
```

**Algorithm 23.9:** Computation of the failure function used in the KMP pattern matching algorithm. Note how the algorithm uses the previous values of the failure function to efficiently compute new values.

The intuition behind the worst-case efficiency of the KMP algorithm comes from our being able to get the most out of each comparison that we do, and by our not performing comparisons we know to be redundant. The KMP algorithm is best suited for strings from small-size alphabets, such as DNA sequences.

## Limitations for Repeated Queries

The BM and KMP pattern matching algorithms presented above speed up the search of a pattern in a text by preprocessing the pattern (to compute the failure function in the KMP algorithm or the last function in the BM algorithm). In some applications, however, we would like to take a complementary approach, where we would consider a string searching algorithms that preprocess the text to support multiple queries. This approach is suitable for applications where a series of queries is performed on a fixed text, so that the initial cost of preprocessing the text is compensated by a speedup in each subsequent query (for example, a website that offers pattern matching in Shakespeare's *Hamlet* or a search engine that offers web pages on the *Hamlet* topic).

## 23.4 Hash-Based Lexicon Matching

In this section, we discuss an approach to string pattern matching that is due to Karp and Rabin and is based on hashing. The advantage of this approach is that it lets us efficiently solve a generalization of the string pattern matching problem, which we call ***lexicon matching***. In this problem, we are given a set, $L = \{P_1, P_2, \ldots, P_l\}$, of $l$ different pattern strings, and a text string, $T$, and we would like to find all the places in $T$ where a pattern, $P_i$, is a substring. We call the set, $L$, the ***lexicon*** of strings we would like to find in $T$.

For example, $L$ could be a set consisting of trademarked words for a certain company, Example.com, and $T$ could be the text of a book published by a former employee. A lawyer for Example.com might like to search for all the instances in $T$ where a trademarked word for Example.com is used. Alternatively, $L$ could be a set consisting of nontrivial sentences from published articles about Shakespeare and $T$ could be a term paper submitted by a certain student, William Fakespeare, in a Shakespeare course. The instructor for this course might wish to know whether William plagiarized any of the sentences from $L$ in writing his term paper, $T$. Both of these examples are possible applications of an efficient algorithm for the lexicon matching problem.

Let $h(X)$ be a hash function that takes a character string, $X$, and maps it to an integer. (See Chapter 6.) Say that such a function is a ***uniform hash*** function for $L$ if, for any pattern $P_k$ in $L$, the number of other patterns, $P_j$, with $j \neq k$, such that $h(P_i) = h(P_j)$, is $O(1)$. Intuitively, we refer to $h$ as a "uniform" hash function in this case, because it spreads the hash values for the strings in $L$ uniformly in the range of $h$. Note, however, that we are not necessarily requiring that we store the patterns in $L$ in a hash table—each hash value for a pattern in $L$ is instead a kind of "fingerprint" for that pattern.

Let us assume, for the sake of simplicity, that all the patterns in the lexicon $L$ are of the same length, $m$. (We explore a more general case in Exercise C-23.13.) The hash-based lexicon matching algorithm for $L$ and $T$ consists of two phases. In the first phase, we compute the hash value of each pattern in the lexicon, $L$, and insert this value, together with the index of the corresponding pattern, in a set, $H$. In the second phase, we step through the text, $T$, and compute the hash value of the length-$m$ substring of $T$ starting at that point. If we find a match with a hash value of a pattern in the lexicon $L$, then we do a full character-by-character comparison of the corresponding pattern and this substring of $T$ to see if we have a confirmed substring match. (See Figure 23.10.)

We give a pseudocode description of the Karp-Rabin hash-based lexicon matching algorithm in Algorithm 23.11.

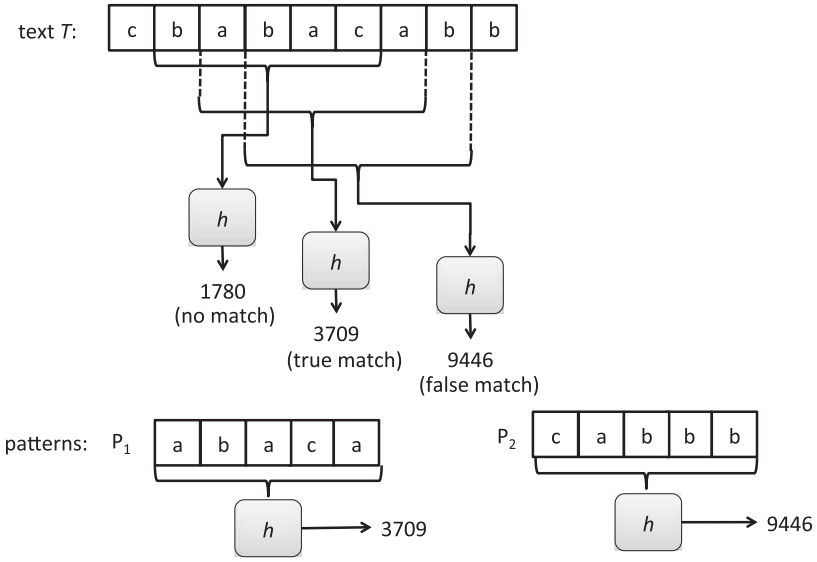



**Figure 23.10:** How the Karp-Rabin hash-based lexicon matching algorithm works. A hash value, $h(X)$, is computed for each length-$m$ substring, $X$, of the text. If it matches the hash value of a pattern, it is then confirmed as either a true match or a false match.

**Algorithm** HashMatch$(L, T)$:

***Input:*** A set, $L = \{P_1, \ldots, P_l\}$, of $l$ pattern strings, each of length $m$, and a text string, $T$, of length $n$

***Output:*** Each pair, $(i, k)$, such that a pattern, $P_k$, in $L$, appears as a substring of $T$ starting at index $i$

```
Let H be an initially empty set of key-value pairs (with hash-value keys)
Let A be an initially empty list of integer pairs (for found matches)
for k ← 1 to l do
    Add the key-value pair, (h(P_k), k), to H
for i ← 0 to n − m do
    f ← h(T[i..i + m − 1])
    for each key-value pair, (f, k), with key f in H do
        // check P_k against T[i..i + m − 1]
        j ← 0
        while j < m and T[i + j] = P_k[j] do
            j ← j + 1
        if j = m then
            Add (i, k) to A        // a match at index i for P_k
return A
```

**Algorithm 23.11:** The Karp-Rabin hash-based lexicon matching algorithm.

### Analysis of the Karp-Rabin Algorithm

Let us analyze the Karp-Rabin hash-based lexicon matching algorithm, assuming that computing $h(X)$ takes $O(m)$ time for any string, $X$, of length $m$. Since there are $l$ pattern strings in $L$, each of length $m$, computing the hash values for the patterns in $L$ and inserting all the hash-indexed pairs into $H$, for the patterns in $L$, takes $O(lm)$ expected time, if we implement $H$ as a hash table, or $O(lm \log l)$ time if we implement $H$ using a balanced binary search tree. Similarly, each execution of Step 6, to compute the hash value of a length-$m$ substring of $T$, takes $O(m)$ time; hence, computing all the hash values for the length-$m$ substrings in $T$ takes $O(nm)$ time. If we implement $H$ as a hash table, then doing each lookup for such a computed hash value takes $O(1)$ expected time, and if $H$ is implemented with a balanced binary search tree, then this lookup takes $O(\log l)$ time. Finally, the total time for performing all the while-loops is $O(lnm)$ in the worst case, since there are $l$ patterns, each of length $m$. Thus, the total running time of this algorithm is $O(lnm + (n + l) \log l) = O(lnm + l \log l)$ in the worst case. Of course, this worst-case time is no better than using the brute-force algorithm to search for every pattern in $L$ separately.

This worst-case bound is based on the pessimistic assumption that the hash values for the patterns in $L$ might all be identical, however. Instead, if we assume that the hash function, $h$, is a uniform hash function for $L$, then the number of collisions for any hash value, $h$, is $O(1)$. Under this assumption, the Karp-Rabin hash-based lexicon matching algorithm runs in $O(lm+nm)$ expected time, if $H$ is implemented with a hash table, since the number of iterations of the **for-each** loop is $O(1)$ in each invocation in this case.

## 23.4.1 An Optimization for Rolling Hash Functions

Suppose the patterns in $L$ and the text $T$ are sufficiently diverse so that the probability that any pattern in $L$ appears as a substring at a given location in $T$ is at most $1/m$. This assumption is reasonable for real-world applications, for example, such as in the trademarked-words or plagiarism examples given above. Under this diversity assumption, the total time needed to perform all the tests for possible matches in the text $T$ is expected to be $O(n(1/m)m) = O(n)$, which is optimal, since it takes $O(n)$ time just to input $T$ itself.

Unfortunately, even under this diversity assumption, the running time for the above Karp-Rabin hash-based lexicon matching algorithm is still $\Omega(nm)$, if it takes $\Omega(m)$ time to compute the hash value of each length-$m$ substring of the text, $T$ (in Step 6 of Algorithm 23.11). We can eliminate this bottleneck, however, if we use a ***rolling-hash function***, $h$, for which there exists a constant-time shift-hash function, $\mathsf{shiftHash}(h(X[i..i + m - 1]), X, i)$, which takes a hash value, $h(X[i..i + m - 1])$, for the length-$m$ substring, $X[i..i + m - 1]$, starting at index $i$, of a string, $X$, and computes the hash value, $h(X[i + 1..i + m])$, of the length-$m$ substring,

$X[i+1..i+m]$, starting at index $i+1$ of $X$. (See Figure 23.12.)

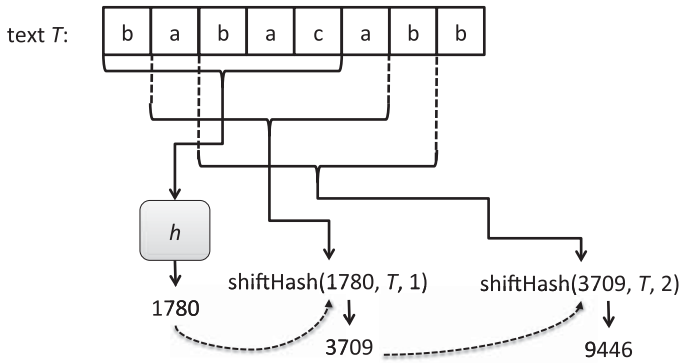



**Figure 23.12:** How a rolling-hash function works in the Karp-Rabin algorithm.

For example, consider polynomial-based hashing (which, as discussed in Section 6.2.2, works well for character strings) applied to length-$m$ character strings. In this case, we have a seed value, $a > 0$, and a prime number, $p > cm$, where $c$ is the number of characters in our alphabet, and we define the hash, $h(X[i..i+m-1])$, of a length-$m$ substring, $X[i..i+m-1]$, of a string, $x$, as follows:

$$X[i]a^{m-1} + X[i+1]a^{m-2} + \cdots + X[i+m-2]a + X[i+m-1],$$

where we view each character in $X$ as an integer in the range $[0, c-1]$ and all arithmetic is done modulo $p$. We can compute this hash function using $O(m)$ arithmetic operations, by Horner's rule, as

$$X[i+m-1] + a(X[i-m-2] + \cdots + a(X[i+1] + aX[i]) \cdots),$$

where all arithmetic is done modulo $p$.

In this case, we can compute the function, $\mathsf{shiftHash}(h(X[i..i+m-1],\ X,\ i)$, as

$$a\left(h(X[i..i+m-1]) - X[i]a^{m-1}\right) + X[i+m],$$

which equals

$$X[i+1]a^{m-1} + X[i+2]a^{m-2} + \cdots + X[i+m-1]a + X[i+m],$$

with all arithmetic done modulo $p$. Also, note that we can compute this shift-hash function using $O(1)$ modular arithmetic operations if we have precomputed the value of $a^{m-1} \bmod p$ and we have the hash value $h(X[i..i+m-1])$.

We give a pseudocode description of the rolling-hash implementation of the Karp-Rabin lexicon matching algorithm in Algorithm 23.13.

**Algorithm** RollingHashMatch$(L, T)$:
***Input:*** A set, $L = \{P_1, \ldots, P_l\}$, of $l$ pattern strings, each of length $m$, and a text string, $T$, of length $n$
***Output:*** Each pair, $(i, k)$, such that a pattern, $P_k$, in $L$, appears as a substring of $T$ starting at index $i$

```
1: Let H be an initially empty set of key-value pairs (with hash-value keys)
2: Let A be an initially empty list of integer pairs (for found matches)
3: for k ← 1 to l do
4:     Add the key-value pair, (h(P_k), k), to H
5: for i ← 0 to n − m do
6:     if i = 0 then  // initial hash
7:         f ← h(T[0..m − 1])
8:     else
9:         f ← shiftHash(f, T, i)
10:    for each key-value pair, (f, k), with key f in H do
11:        // check P_k against T[i..i + m − 1]
12:        j ← 0
13:        while j < m and T[i + j] = P_k[j] do
14:            j ← j + 1
15:        if j = m then
16:            Add (i, k) to A        // a match at index i for P_k
17: return A
```

**Algorithm 23.13:** A rolling-hash implementation of the Karp-Rabin lexicon matching algorithm.

## Analysis of Rolling-Hash Lexicon Matching

Let us analyze this rolling-hash lexicon matching algorithm, assuming that the patterns in $L$ and $T$ are sufficiently diverse and the rolling-hash function, $h$, is chosen so that the probability that a hash value for a length-$m$ substring of $T$ has a match with any pattern in $L$ is at most $1/m$. For example, in the polynomial hash function discussed above, if the prime number, $p$, is chosen to be larger than $cm$, where $c$ is the number of characters in the alphabet, then the probability that two randomly chosen length-$m$ character strings have the same hash value is at most $1/cm$. Under this diversity and hash-function assumption, then, the expected running time for the Karp-Rabin lexicon matching algorithm is $O(lm + n)$, assuming the set $H$ is implemented with a hash table. This performance is due to the fact that the expected number of $O(m)$-time comparisons between a pattern $P_k$ and a length-$m$ substring of $T$ is $O(n/m)$. Note that the expected running time of $O(lm + n)$ is also optimal, since it takes $O(lm + n)$ time just to read in the input strings for this problem.

# 23.5 Tries

A ***trie*** (pronounced "try") is a tree-based data structure for storing strings in order to support fast pattern matching. The main application for tries is in information retrieval. Indeed, the name "trie" comes from the word "re*trie*val." In an information retrieval application, such as a search for a certain DNA sequence in a genomic database, we are given a collection $S$ of strings, all defined using the same alphabet.

The primary query operations that tries support are pattern matching and ***prefix matching***. The latter operation involves being given a string $X$, and looking for all the strings in $S$ that contain $X$ as a prefix.

## 23.5.1 Standard Tries

Let $S$ be a set of $s$ strings from alphabet $\Sigma$, such that no string in $S$ is a prefix of another string. A ***standard trie*** for $S$ is an ordered tree $T$ with the following properties (see Figure 23.14):

- Each node of $T$, except the root, is labeled with a character of $\Sigma$.
- The ordering of the children of an internal node of $T$ is determined by a canonical ordering of the alphabet $\Sigma$.
- $T$ has $s$ external nodes, each associated with a string of $S$, such that the concatenation of the labels of the nodes on the path from the root to an external node $v$ of $T$ yields the string of $S$ associated with $v$.

Thus, a trie $T$ represents the strings of $S$ with paths from the root to the external nodes of $T$. Note the importance of assuming that no string in $S$ is a prefix of another string. This ensures that each string of $S$ is uniquely associated with an external node of $T$. We can always satisfy this assumption by adding a special character that is not in the original alphabet $\Sigma$ at the end of each string.

An internal node in a standard trie $T$ can have anywhere between 1 and $d$ children, where $d$ is the size of the alphabet. There is an edge going from the root $r$ to one of its children for each character that is first in some string in the collection $S$. In addition, a path from the root of $T$ to an internal node $v$ at depth $i$ corresponds to an $i$-character prefix $X[0..i-1]$ of a string $X$ of $S$. In fact, for each character $c$ that can follow the prefix $X[0..i-1]$ in a string of the set $S$, there is a child of $v$ labeled with character $c$. In this way, a trie concisely stores the common prefixes that exist among a set of strings.

If there are only two characters in the alphabet, then the trie is essentially a binary tree, although some internal nodes may have only one child (that is, it may be an improper binary tree). In general, if there are $d$ characters in the alphabet, then the trie will be a multi-way tree where each internal node has between 1 and

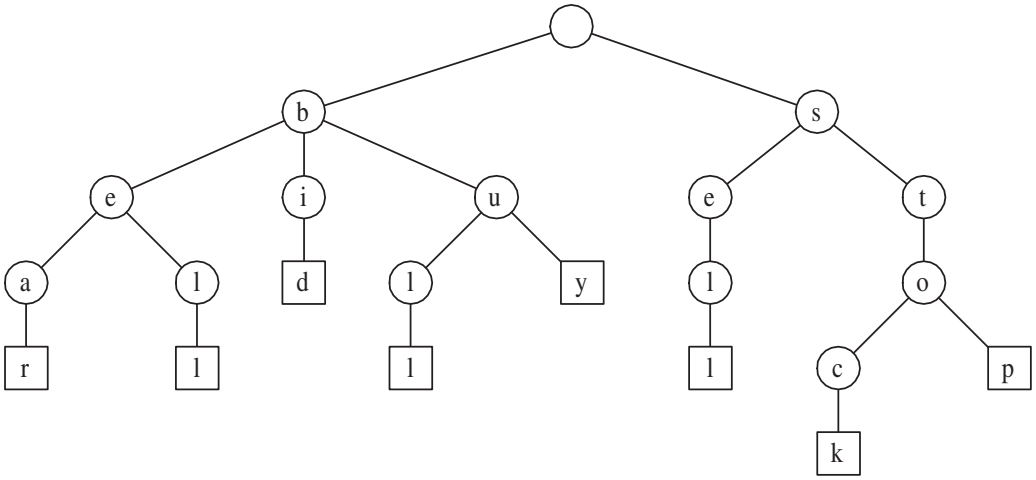



**Figure 23.14:** Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

$d$ children. In addition, there are likely to be several internal nodes in a standard trie that have fewer than $d$ children. For example, the trie shown in Figure 23.14 has several internal nodes with only one child. We can implement a trie with a tree storing characters at its nodes.

**Theorem 23.4:** *A standard trie storing a collection $S$ of $s$ strings of total length $n$ from an alphabet of size $d$ has the following properties:*

- *Every internal node of $T$ has at most $d$ children*
- *$T$ has $s$ external nodes*
- *The height of $T$ is equal to the length of the longest string in $S$*
- *The number of nodes of $T$ is $O(n)$.*

The worst case for the number of nodes of a trie occurs when no two strings share a common nonempty prefix—that is, except for the root, all internal nodes have one child.

A trie $T$ for a set $S$ of strings can be used to implement a dictionary whose keys are the strings of $S$. Namely, we perform a search in $T$ for a string $X$ by tracing down from the root the path indicated by the characters in $X$. If this path can be traced and terminates at an external node, then we know $X$ is in the dictionary. For example, in the trie in Figure 23.14, tracing the path for "bull" ends up at an external node. If the path cannot be traced or the path can be traced but terminates at an internal node, then $X$ is not in the dictionary. In the example in Figure 23.14, the path for "bet" cannot be traced and the path for "be" ends at an internal node. Neither such word is in the dictionary. Note that in this implementation of a dictionary, single characters are compared instead of the entire string (key).

### Analysis

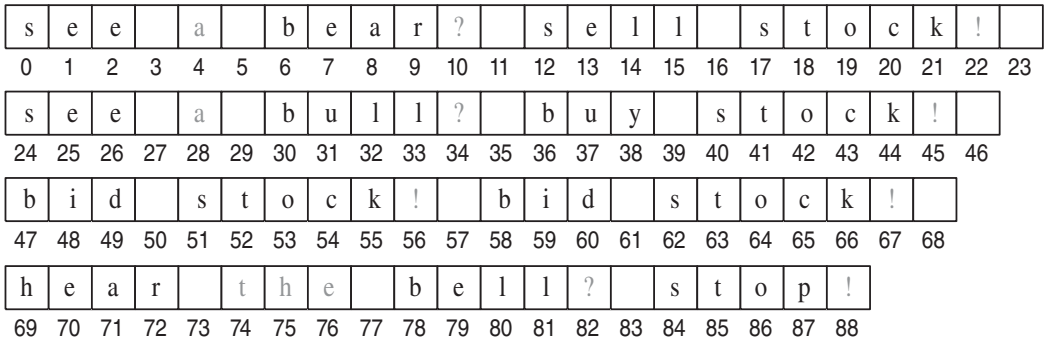
It is easy to see that the running time of the search for a string of size $m$ is $O(dm)$, where $d$ is the size of the alphabet. Indeed, we visit at most $m + 1$ nodes of $T$ and we spend $O(d)$ time at each node. For some alphabets, we may be able to improve the time spent at a node to be $O(1)$ or $O(\log d)$ by using a dictionary of characters implemented in a hash table or lookup table. However, since $d$ is a constant in most applications, we can stick with the simple approach that takes $O(d)$ time per node visited.

### Application to Word Matching

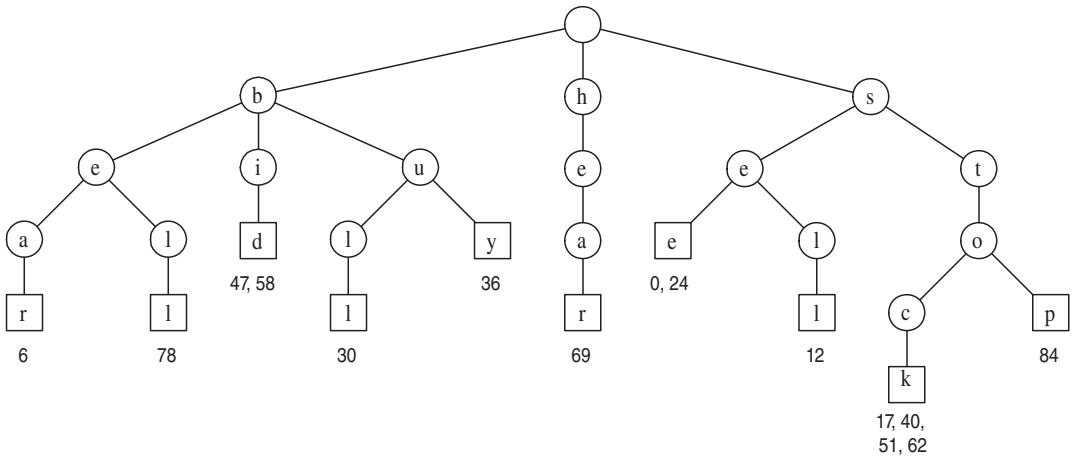
From the above discussion, it follows that we can use a trie to perform a special type of pattern matching, called ***word matching***, where we want to determine whether a given pattern matches one of the words of the text exactly. (See Figure 23.15.) Word matching differs from standard pattern matching since the pattern cannot match an arbitrary substring of the text, but only one of its words. Using a trie, word matching for a pattern of length $m$ takes $O(dm)$ time, where $d$ is the size of the alphabet, independent of the size of the text. If the alphabet has constant size (as is the case for text in natural languages and DNA strings), a query takes $O(m)$ time, proportional to the size of the pattern. A simple extension of this scheme supports prefix matching queries. However, arbitrary occurrences of the pattern in the text (for example, the pattern is a proper suffix of a word or spans two words) cannot be efficiently performed.

### Standard Trie Construction

To construct a standard trie for a set $S$ of strings, we can use an incremental algorithm that inserts the strings one at a time. Recall the assumption that no string of $S$ is a prefix of another string. To insert a string $X$ into the current trie $T$, we first try to trace the path associated with $X$ in $T$. Since $X$ is not already in $T$ and no string in $S$ is a prefix of another string, we will stop tracing the path at an ***internal*** node $v$ of $T$ before reaching the end of $X$. We then create a new chain of node descendants of $v$ to store the remaining characters of $X$. The time to insert $X$ is $O(dm)$, where $m$ is the length of $X$ and $d$ is the size of the alphabet. Thus, constructing the entire trie for set $S$ takes $O(dn)$ time, where $n$ is the total length of the strings of $S$.




(a)




(b)

**Figure 23.15:** Word matching and prefix matching with a standard trie: (a) an example text that is to be searched; (b) a standard trie for the words in the text (with articles and prepositions, which are also known as ***stop words***, excluded). We show external nodes augmented with indications of the corresponding word positions.

There is a potential space inefficiency in the standard trie that has prompted the development of the ***compressed trie***, which is also known (for historical reasons) as the ***Patricia trie***. Namely, there are potentially a lot of nodes in the standard trie that have only one child, and the existence of such nodes is a waste, for it implies that the total number of nodes in the tree could be more than the number of words in the corresponding text.

We discuss the compressed trie data structure in the next subsection.

## 23.5.2 Compressed Tries

A ***compressed trie*** is similar to a standard trie but it ensures that each internal node in the trie has at least two children. It enforces this rule by compressing chains of single-child nodes into individual edges. (See Figure 23.16.) Let $T$ be a standard trie. We say that an internal node $v$ of $T$ is ***redundant*** if $v$ has one child and is not the root. For example, the trie of Figure 23.14 has eight redundant nodes. Let us also say that a chain of $k \geq 2$ edges,

$$(v_0, v_1)(v_1, v_2) \cdots (v_{k-1}, v_k),$$

is ***redundant*** if

- $v_i$ is redundant for $i = 1, \ldots, k-1$
- $v_0$ and $v_k$ are not redundant.

We can transform $T$ into a compressed trie by replacing each redundant chain $(v_0, v_1) \cdots (v_{k-1}, v_k)$ of $k \geq 2$ edges into a single edge $(v_0, v_k)$, relabeling $v_k$ with the concatenation of the labels of nodes $v_1, \ldots, v_k$.

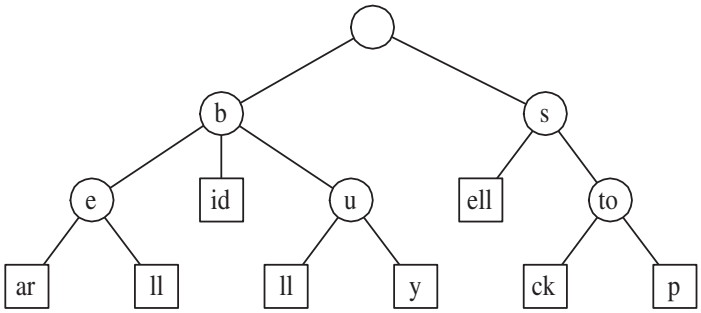



**Figure 23.16:** Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}. Compare this with the standard trie shown in Figure 23.14.

Thus, nodes in a compressed trie are labeled with strings, which are substrings of strings in the collection, rather than with individual characters. The advantage of a compressed trie over a standard trie is that the number of nodes of the compressed trie is proportional to the number of strings and not to their total length, as shown in the following theorem (compare with Theorem 23.4).

**Theorem 23.5:** *A compressed trie storing a collection $S$ of $s$ strings from an alphabet of size $d$ has the following properties:*

- *Every internal node of $T$ has at least two children and at most $d$ children*
- *$T$ has $s$ external nodes*
- *The number of nodes of $T$ is $O(s)$.*

The attentive reader may wonder whether the compression of paths provides any significant advantage, since it is offset by a corresponding expansion of the node labels. Indeed, a compressed trie is truly advantageous only when it is used as an ***auxiliary*** index structure over a collection of strings already stored in a primary structure, and is not required to actually store all the characters of the strings in the collection. Given this auxiliary structure, however, the compressed trie is indeed quite efficient.

Suppose, for example, that the collection $S$ of strings is an array of strings $S[0]$, $S[1]$, ..., $S[s-1]$. Instead of storing the label $X$ of a node explicitly, we represent it implicitly by a triplet of integers $(i, j, k)$, such that $X = S[i][j..k]$; that is, $X$ is the substring of $S[i]$ consisting of the characters from the $j$th to the $k$th included. (See the example in Figure 23.17. Also compare with the standard trie of Figure 23.15.)

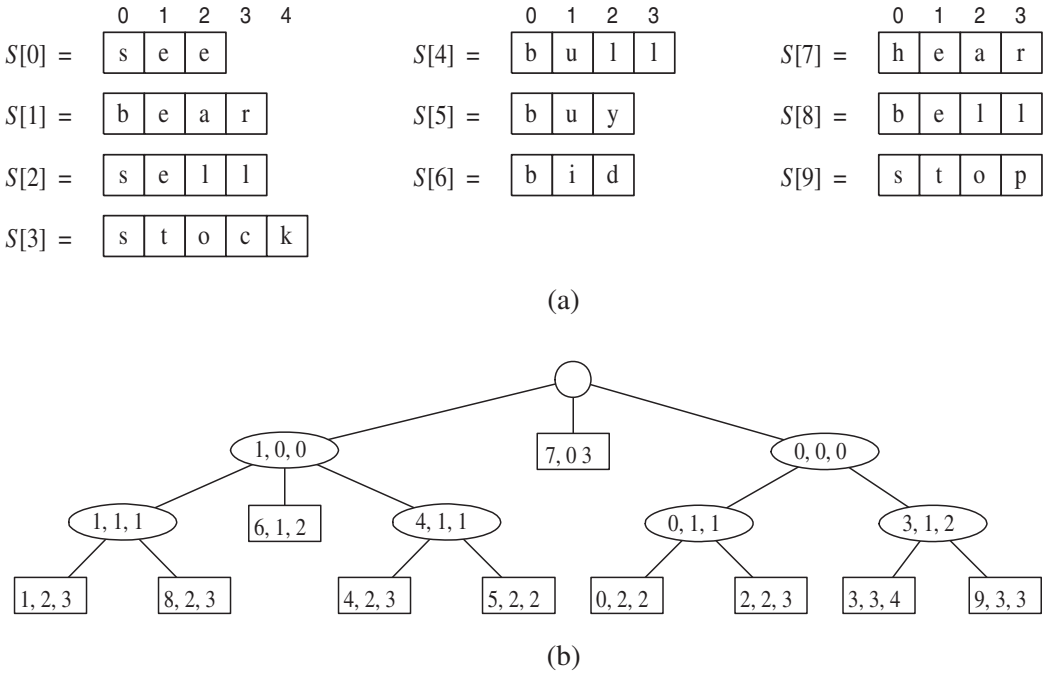



**Figure 23.17:** (a) Collection $S$ of strings stored in an array. (b) Compact representation of the compressed trie for $S$.

This additional compression scheme allows us to reduce the total space for the trie itself from $O(n)$ for the standard trie to $O(s)$ for the compressed trie, where $n$ is the total length of the strings in $S$ and $s$ is the number of strings in $S$. We must still store the different strings in $S$, of course, but we nevertheless reduce the space for the trie. In the next section, we present an application where the collection of strings can also be stored compactly.

### 23.5.3 Suffix Tries

One of the primary applications for tries is for the case when the strings in the collection $S$ are all the suffixes of a string $X$. Such a trie is called the ***suffix trie*** (also known as a ***suffix tree*** or ***position tree***) of string $X$. For example, Figure 23.18a shows the suffix trie for the eight suffixes of string `"minimize"`.

For a suffix trie, the compact representation presented in the previous section can be further simplified. Namely, we can construct the trie so that the label of each vertex is a pair $(i, j)$ indicating the string $X[i..j]$. (See Figure 23.18b.) To satisfy the rule that no suffix of $X$ is a prefix of another suffix, we can add a special character, denoted with \$, that is not in the original alphabet $\Sigma$ at the end of $X$ (and thus to every suffix). That is, if string $X$ has length $n$, we build a trie for the set of $n$ strings $X[i..n-1]\$$, for $i = 0, \ldots, n-1$.

#### Saving Space

Using a suffix trie allows us to save space over a standard trie by using several space compression techniques, including those used for the compressed trie. The advantage of the compact representation of tries now becomes apparent for suffix tries. Since the total length of the suffixes of a string $X$ of length $n$ is

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2},$$

storing all the suffixes of $X$ explicitly would take $O(n^2)$ space. Even so, the suffix trie represents these strings implicitly in $O(n)$ space, as formally stated in the following theorem.

**Theorem 23.6:** *The compact representation of a suffix trie $T$ for a string $X$ of length $n$ uses $O(n)$ space.*

#### Construction

We can construct the suffix trie for a string of length $n$ with an incremental algorithm like the one given in Section 23.5.1. This construction takes $O(dn^2)$ time because the total length of the suffixes is quadratic in $n$. However, the (compact) suffix trie for a string of length $n$ can be constructed in $O(n)$ time with a specialized algorithm, different from the one for general tries. This linear-time construction algorithm is fairly complex, however, and is not reported here. Still, we can take advantage of the existence of this fast construction algorithm when we want to use a suffix trie to solve other problems.

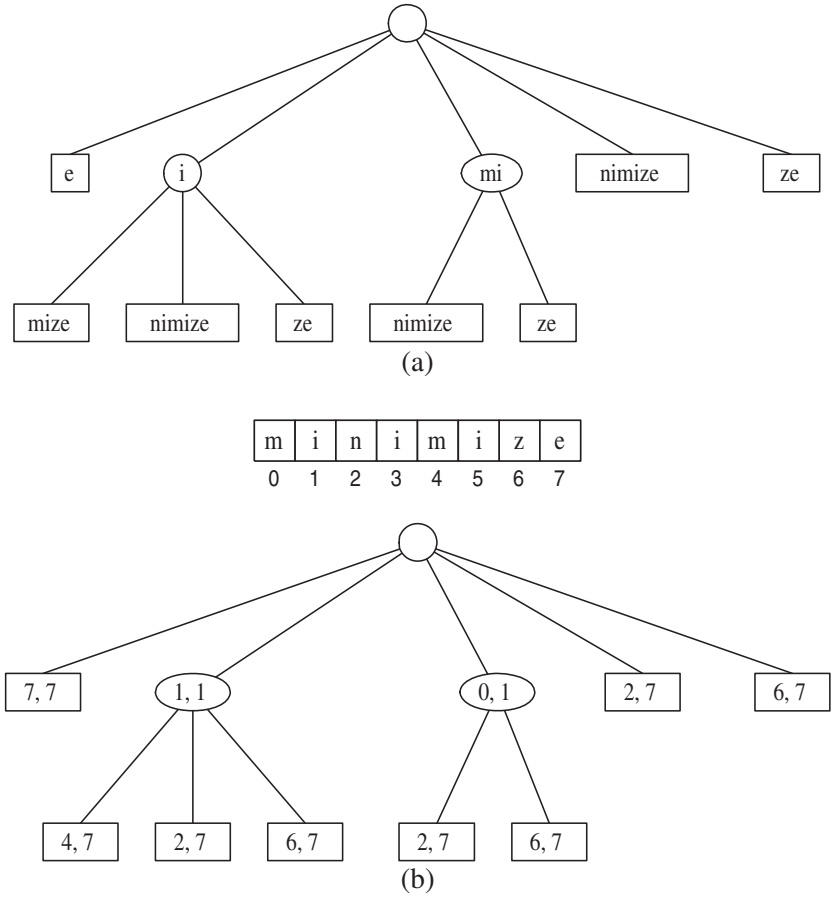



**Figure 23.18:** (a) Suffix trie $T$ for the string $X =$ `"minimize"`. (b) Compact representation of $T$, where pair $(i, j)$ denotes $X[i..j]$.

## Using a Suffix Trie

The suffix trie $T$ for a string $X$ can be used to efficiently perform pattern matching queries on text $X$. Namely, we can determine whether a pattern $P$ is a substring of $X$ by trying to trace a path associated with $P$ in $T$. $P$ is a substring of $X$ if and only if such a path can be traced. The details of the pattern matching algorithm are given in Algorithm 23.19, which assumes the following additional property on the labels of the nodes in the compact representation of the suffix trie:

> If node $v$ has label $(i, j)$ and $Y$ is the string of length $y$ associated with the path from the root to $v$ (included), then $X[j - y + 1..j] = Y$.

This property ensures that we can easily compute the start index of the pattern in the text when a match occurs.

**Algorithm** suffixTrieMatch$(T, P)$:
*Input:* Compact suffix trie $T$ for a text $X$ and pattern $P$
*Output:* Starting index of a substring of $X$ matching $P$ or an indication that $P$ is not a substring of $X$

```
p ← P.length() // length of suffix of the pattern to be matched
j ← 0 // start of suffix of the pattern to be matched
v ← T.root()
repeat
    f ← true // flag indicating that no child was successfully processed
    for each child w of v do
        i ← start(w)
        if P[j] = T[i] then
            // process child w
            x ← end(w) − i + 1
            if p ≤ x then
                // suffix is shorter than or of the same length of the node label
                if P[j..j + p − 1] = X[i..i + p − 1] then
                    return i − j // match
                else
                    return "P is not a substring of X"
            else
                // suffix is longer than the node label
                if P[j..j + x − 1] = X[i..i + x − 1] then
                    p ← p − x // update suffix length
                    j ← j + x // update suffix start index
                    v ← w
                    f ← false
                    break out of the for loop
until f or T.isExternal(v)
return "P is not a substring of X"
```

**Algorithm 23.19:** Pattern matching with a suffix trie. We denote the label of a node $v$ with $(\mathsf{start}(v), \mathsf{end}(v))$, that is, the pair of indices specifying the substring of the text associated with $v$.

## Suffix Trie Properties

The correctness of algorithm suffixTrieMatch follows from the fact that we search down the trie $T$, matching characters of the pattern $P$ one at a time until one of the following events occurs:

- We completely match the pattern $P$.
- We get a mismatch (caught by the termination of the for-loop without a breakout).
- We are left with characters of $P$ still to be matched after processing an external node.

Let $m$ be the size of pattern $P$ and $d$ be the size of the alphabet. In order to determine the running time of algorithm suffixTrieMatch, we make the following observations:

- We process at most $m + 1$ nodes of the trie.
- Each node processed has at most $d$ children.
- At each node $v$ processed, we perform at most one character comparison for each child $w$ of $v$ to determine which child of $v$ needs to be processed next (which may possibly be improved by using a fast dictionary to index the children of $v$).
- We perform at most $m$ character comparisons overall in the processed nodes.
- We spend $O(1)$ time for each character comparison.

## Analysis

We conclude that algorithm suffixTrieMatch performs pattern matching queries in $O(dm)$ time (and would possibly run even faster if we used a dictionary to index children of nodes in the suffix trie). Note that the running time does not depend on the size of the text $X$. Also, the running time is linear in the size of the pattern, that is, it is $O(m)$, for a constant-size alphabet. Hence, suffix tries are suited for repetitive pattern matching applications, where a series of pattern matching queries is performed on a fixed text.

We summarize the results of this section in the following theorem.

**Theorem 23.7:** *Let $X$ be a text string with $n$ characters from an alphabet of size $d$. We can perform pattern matching queries on $X$ in $O(dm)$ time, where $m$ is the length of the pattern, with the suffix trie of $X$, which uses $O(n)$ space and can be constructed in $O(dn)$ time.*

We explore another application of tries in the next subsection.

### 23.5.4 Search Engines

The World Wide Web contains a huge collection of text documents (web pages). Information about these pages is gathered by a program called a ***web crawler***, which then stores this information in a special dictionary database. A web ***search engine*** allows users to retrieve relevant information from this database, thereby identifying relevant pages on the web containing given keywords. In this section, we present a simplified model of a search engine.

#### Inverted Files

The core information stored by a search engine is a dictionary, called an ***inverted index*** or ***inverted file***, storing key-value pairs $(w, L)$, where $w$ is a word and $L$ is a collection of references to pages containing word $w$. The keys (words) in this dictionary are called ***index terms*** and should be a set of vocabulary entries and proper nouns as large as possible. The elements in this dictionary are called ***occurrence lists*** and should cover as many web pages as possible.

We can efficiently implement an inverted index with a data structure consisting of the following:

- An array storing the occurrence lists of the terms (in no particular order)
- A compressed trie for the set of index terms, where each external node stores the index of the occurrence list of the associated term.

The reason for storing the occurrence lists outside the trie is to keep the size of the trie data structure sufficiently small to fit in internal memory. Instead, because of their large total size, the occurrence lists have to be stored on disk.

With our data structure, a query for a single keyword is similar to a word matching query (see Section 23.5.1). Namely, we find the keyword in the trie and we return the associated occurrence list.

When multiple keywords are given and the desired output is the pages containing ***all*** the given keywords, we retrieve the occurrence list of each keyword using the trie and return their intersection. To facilitate the intersection computation, each occurrence list should be implemented with a sequence sorted by address or with a dictionary, which allows for a simple intersection algorithm similar to sorted sequence merging (Section 8.1).

In addition to the basic task of returning a list of pages containing given keywords, search engines provide an important additional service by ***ranking*** the pages returned by relevance. Devising fast and accurate ranking algorithms for search engines is a major challenge for computer researchers and electronic commerce companies.

## 23.6 Exercises

### Reinforcement

**R-23.1** How many nonempty prefixes of the string $P =$"aaabbaaa" are also suffixes of $P$?

**R-23.2** Draw a figure illustrating the comparisons done by the brute-force pattern matching algorithm for the case when the text is "aaabaadaabaaa" and the pattern is "aabaaa".

**R-23.3** Repeat the previous problem for the BM pattern matching algorithm, not counting the comparisons made to compute the last function.

**R-23.4** Repeat the previous problem for the KMP pattern matching algorithm, not counting the comparisons made to compute the failure function.

**R-23.5** Compute a table representing the last function used in the BM pattern matching algorithm for the pattern string

"the quick brown fox jumped over a lazy cat"

assuming the following alphabet (which starts with the space character):

$\Sigma = \{$ ,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z$\}$.

**R-23.6** Assuming that the characters in alphabet $\Sigma$ can be enumerated and can index arrays, give an $O(m + |\Sigma|)$ time method for constructing the last function from an $m$-length pattern string $P$.

**R-23.7** Compute a table representing the KMP failure function for the pattern string "cgtacgttcgtac".

**R-23.8** Draw a standard trie for the following set of strings:

$\{$abab, baba, ccccc, bbaaaa, caa, bbaacc, cbcc, cbca$\}$.

**R-23.9** Draw a compressed trie for the set of strings given in Exercise R-23.8.

**R-23.10** Draw the compact representation of the suffix trie for the string

"minimize minime".

**R-23.11** What is the longest prefix of the string "cgtacgttcgtacg" that is also a suffix of this string?

**R-23.12** Give an example of an input instance for lexicon matching problem, with just a single pattern in the lexicon, $L$, that forces the Karp-Rabin algorithm given in Algorithm 23.11 to run in $\Omega(nm)$ time.

**R-23.13** Explain why tabulation hashing, which is discussed in Section 6.2.3, is not a good candidate for use in a rolling-hash lexicon matching algorithm.

**R-23.14** Describe how to compute shiftHash$(h(X[i..i+m-1]), X, i)$ for the hash function, $h(X[i..i+m-1]) = X[i] + \cdots + X[i+m-1]$, where each character is viewed as an integer in the range $[0, c-1]$, with $c$ being the size of the alphabet, and all arithmetic is done modulo a prime, $p$.

## Creativity

**C-23.1** Give an example of a text $T$ of length $n$ and a pattern $P$ of length $m$ that force the brute-force pattern matching algorithm to have a running time that is $\Omega(nm)$.

**C-23.2** Give a justification of why the KMPFailureFunction method (Algorithm 23.9) runs in $O(m)$ time on a pattern of length $m$.

**C-23.3** Show how to modify the KMP string pattern matching algorithm so as to find ***every*** occurrence of a pattern string $P$ that appears as a substring in $T$, while still running in $O(n + m)$ time. (Be sure to catch even those matches that overlap.)

**C-23.4** Let $T$ be a text of length $n$, and let $P$ be a pattern of length $m$. Describe an $O(n + m)$-time method for finding the longest prefix of $P$ that is a substring of $T$.

**C-23.5** Say that a pattern $P$ of length $m$ is a ***circular substring*** of a text $T$ of length $n$ if there is an index $0 \leq i < m$, such that $P = T[n - m + i..n - 1] + T[0..i - 1]$, that is, if $P$ is a substring of $T$ or $P$ is equal to the concatenation of a suffix of $T$ and a prefix of $T$. Give an $O(n + m)$-time algorithm for determining whether $P$ is a circular substring of $T$.

**C-23.6** The KMP pattern matching algorithm can be modified to run faster on binary strings by redefining the failure function as

$$f(j) = \text{the largest } k < j \text{ such that } P[0..k-2]\overline{P[k-1]} \text{ is a suffix of } P[1..j],$$

where $overline P[k]$ denotes the complement of the $k$th bit of $P$. Describe how to modify the KMP algorithm to be able to take advantage of this new failure function and also give a method for computing this failure function. Show that this method makes at most $n$ comparisons between the text and the pattern (as opposed to the $2n$ comparisons needed by the standard KMP algorithm given in Section 23.3).

**C-23.7** Modify the simplified BM algorithm presented in this chapter using ideas from the KMP algorithm so that it runs in $O(n + m)$ time.

**C-23.8** Consider the substring pattern matching problem for a length-$m$ pattern, $P$, and a length-$n$ text, $T$, where one of the characters in $P$ is a symbol, "?," which is not in the alphabet for the text. This symbol, "?," is a ***wild-card*** character, which matches with any character of the alphabet for the text. The pattern, $P$, contains exactly one "?" symbol. Show how to modify the Karp-Rabin matching algorithm for this single-pattern instance of the lexicon matching problem so that the expected running time of the resulting algorithm is $O(n + m)$.

**C-23.9** Show how to perform prefix matching queries using a suffix trie.

**C-23.10** Give an efficient algorithm for deleting a string from a standard trie and analyze its running time.

**C-23.11** Give an efficient algorithm for deleting a string from a compressed trie and analyze its running time.

**C-23.12** Describe an algorithm for constructing the compact representation of a suffix trie and analyze its running time.

**C-23.13** Describe a generalized version of the Karp-Rabin lexicon matching algorithm for the case when their are $k$ different possible pattern sizes. Characterize the running time of this algorithm in terms of $n$, $k$, and the total size, $N$, of all the patterns in the lexicon.

## Applications

**A-23.1** When a web crawler is exploring the Internet looking for content to index for a search engine, the crawler needs some way of detecting when it is visiting a copy of a website it has encountered before. Describe a way for a web crawler to store its web pages efficiently so that it can detect in $O(n)$ time whether a web page of length $n$ has been previously encountered and, if not, add it to the collection of previously encountered web pages in $O(1)$ additional time.

**A-23.2** Search engines need a fast way to detect and ignore stop words, that is, words, such as prepositions, pronouns, and articles, that are very common and carry no meaningful information content. Describe an efficient method for storing and searching a set of stop words in a way that supports stop-word identification in constant time for all constant-length stop words.

**A-23.3** DNA strings are sometimes spliced into other DNA strings as a product of recombinant DNA processes. But DNA strings can be read in what would be either the forward or backward direction for a standard character string. Thus, it is useful to be able to identify prefixes and their reversals. Let $T$ be a DNA text string of length $n$. Describe an $O(n)$-time method for finding the longest prefix of $T$ that is a substring of the reversal of $T$.

**A-23.4** Linguists are interested in studying the way in which words are constructed, with common prefixes and suffixes giving important clues to the meanings of words they are contained in. Thus, a useful tool for a linguist would be to be able to identify all the words in a given collection, $W$, of words, that have the same prefix, $p$, or suffix, $s$. Indeed, it is useful even to just know the number of such words in $W$. Describe how to build a data structure for $W$ that can quickly answer, for any prefix, $p$, or suffix, $s$, the number of words in $W$ that have the prefix $p$ or suffix $s$. What is the performance of your method?

**A-23.5** One way to mask a message, $M$, using a version of ***steganography***, is to insert random characters into $M$ at pseudo-random locations so as to expand $M$ into a larger string, $C$. For instance, the message,

`ILOVEMOM`,

could be expanded into

`AMIJLONDPVGEMRPIOM`.

It is an example of hiding the string, $M$, in plain sight, since the characters in $M$ and $C$ are not encrypted. As long as someone knows where the random

characters where inserted, he or she can recover $M$ from $C$. The challenge for law enforcement, therefore, is to prove when someone is using this technique, that is, to determine whether a string $C$ contains a message $M$ in this way. Thus, describe an $O(n)$-time method for detecting if a string, $M$, is a subsequence of a string, $C$, of length $n$.

**A-23.6** Consider the previous exercise, but now suppose that the masking process first pads many random characters to form a prefix and suffix of $M$ before adding a few random characters at various locations in the middle of $M$. Thus, a likely message, $M$, is one that is much shorter than the host string, $C$. Describe an $O(n^2)$-time algorithm for determining if $M$ is a subsequence of $C$, and, if so, returns the shortest substring of $C$ having $M$ as a subsequence.

**A-23.7** Suppose Bill is graduate of Slacker University, and he took a little "shortcut" when he was asked to build a software system that could take a pattern, $P$, of length, $m$, and text, $T$, of length, $n$, with both defined over the same alphabet of size $d$ for some constant $d > 1$, and determine whether $P$ is a substring of $T$. Namely, Bill's software simply returns the answer "no" whenever it is asked to determine whether a pattern $P$ of length $m$ is contained in a text $T$ of length $n$. When confronted by his Boss about this software, Bill replied that his software system is almost always correct, that is, Bill claims that his software fails with probability that is $o(1)$ as a function of $m$ and $n$. Give an asymptotic characterization of the probability that Bill's simple algorithm incorrectly determines whether $P$ is a substring in $T$, assuming that all possible pattern strings of length $m$ are equally likely. Is Bill right about his software?

**A-23.8** Suppose the trustee for the estate of famous photographer, Ansel Adams, was interested in finding examples of people posting Ansel Adams photographs on their personal websites without including attributions to him. Suppose further that some of these people have tried to conceal their possible copyright infringement by cropping the image down to be a smaller size. Thus, given a candidate image, $P$, taken from someone's personal website, and his own photograph, $T$, Mr. Adams is interested in determining whether it is possible to create the image $P$ by cropping the image $T$. Assuming that $P$ and $T$ are square images, this problem can be easily modeled as a two-dimensional version of the pattern matching problem. In ***two-dimensional pattern matching***, a pattern, $P$, is given as an $m \times m$ array of characters, and the text, $T$, is given as an $n \times n$ array of characters, and we are interested in finding a location, $(i, j)$, such that $P$ is a submatrix of $T$ when shifted to the location, $(i, j)$. That is,

$$P[k, l] = T[i + k, j + l],$$

for $k = 0, \ldots, m - 1$ and $l = 0, \ldots, m - 1$. In the context of the problem of interest for the trustee of the estate of Ansel Adams, each pixel in an image could be viewed as a character (corresponding, for example, to an 8-bit intensity value in a black-and-white image). Describe an efficient algorithm for solving this two-dimensional pattern matching problem. What are the worst-case and expected-case running times for your algorithm?

# Chapter Notes

The KMP algorithm is described by Knuth, Morris, and Pratt in their journal article [132]. Boyer and Moore published their algorithm in the same year [35]. In their article, Knuth *et al.* [132] also prove that the BM algorithm runs in linear time. More recently, Cole [46] shows that the BM algorithm makes at most $3n$ character comparisons in the worst case, and this bound is tight. Hashing-based pattern matching is presented by Karp and Rabin [123]. Some of the algorithms presented in this chapter are also discussed in the book chapter by Aho [7], albeit in a more theoretical framework. The reader interested in further study of string pattern matching algorithms is referred to the book by Stephen [200] and the book chapters by Aho [7] and Crochemore and Lecroq [52].

The trie was invented by Morrison [161] and is discussed extensively in the classic book *Sorting and Searching* by Knuth [131]. The name "Patricia" is short for "Practical Algorithm to Retrieve Information Coded in Alphanumeric" [161]. McCreight [148] shows how to construct suffix tries in linear time. An introduction to the field of information retrieval, which includes a discussion of search engines for the World Wide Web, is provided in the book by Baeza-Yates and Ribeiro-Neto [20].