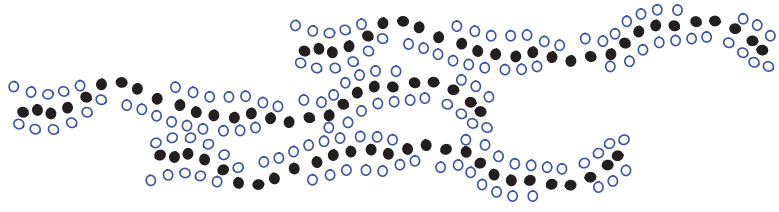
Chapter 12 Strings and Dynamic Programming



Contents

12.1 String Operations

Document processing is rapidly becoming one of the dominant functions of computers. Computers are used to edit documents, to search documents, to transport documents over the Internet, and to display documents on printers and computer screens. For example, the Internet document formats HTML and XML are primarily text formats, with added tags for multimedia content. Making sense of the many terabytes of information on the Internet requires a considerable amount of text processing.

In addition to having interesting applications, text processing algorithms also highlight some important algorithmic design patterns. In particular, the pattern matching problem gives rise to the ***brute-force method***, which is often inefficient but has wide applicability. For text compression, we can apply the ***greedy method***, which often allows us to approximate solutions to hard problems, and for some problems (such as in text compression) actually gives rise to optimal algorithms. Finally, in discussing text similarity, we introduce the ***dynamic programming*** design pattern, which can be applied in some special instances to solve a problem in polynomial time that appears at first to require exponential time to solve.

Text Processing

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:

$$\begin{aligned} P &= \text{“}\texttt{CGTAAACTGCTTTAATCAAACGC}\text{”} \\ S &= \text{“}\texttt{http://www.wiley.com}\text{”}. \end{aligned}$$

The first string, P, comes from DNA applications, and the second string, S, is the Internet address (URL) for the publisher of this book.

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.

To allow for fairly general notions of a character string, we typically do not restrict the characters in T and P to explicitly come from a well-known character set, like the ASCII or Unicode character sets. Instead, we typically use the symbol Σ to denote the character set, or ***alphabet***, from which characters can come. 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 Σ, denoted with $|\Sigma|$, is a fixed constant.

12.1.1 The STL String Class

Recall from Chapter 1 that C++ supports two types of strings. A C-style string is just an array of type **char** terminated by a null character `'\0'`. By themselves, C-style strings do not support complex string operations. The C++ Standard Template Library (STL) provides a complete string class. This class supports a bewildering number of string operations. We list just a few of them. In the following, let S denote the STL string object on which the operation is being performed, and let Q denote another STL string or a C-style string.

size(): Return the number of characters, n, of S.

empty(): Return true if the string is empty and false otherwise.

operator[i]: Return the character at index i of S, without performing array bounds checking.

at(i): Return the character at index i of S. An out_of_range exception is thrown if i is out of bounds.

insert(i, Q): Insert string Q prior to index i in S and return a reference to the result.

append(Q): Append string Q to the end of S and return a reference to the result.

erase(i, m): Remove m characters starting at index i and return a reference to the result.

substr(i,m): Return the substring of S of length m starting at index i.

find(Q): If Q is a substring of S, return the index of the beginning of the first occurrence of Q in S, else return n, the length of S.

c_str(): Return a C-style string containing the contents of S.

By default, a string is initialized to the empty string. A string may be initialized from another STL string or from a C-style string. It is not possible, however, to initialize an STL string from a single character. STL strings also support functions that return both forward and backward iterators. All operations that are defined in terms of integer indices have counterparts that are based on iterators.

The STL string class also supports assignment of one string to another. It provides relational operators, such as ==, <, >=, which are performed lexicographically. Strings can be concatenated using +, and we may append one string to another using +=. Strings can be input using >> and output using <<. The function getline(in,S) reads an entire line of input from the input stream *in* and assigns it to the string S.

The STL string class is actually a special case of a more general templated class, called basic_string<T>, which supports all the string operations but allows its elements to be of an arbitrary type, T, not just **char**. The STL string is just a short way of saying basic_string<char>. A "string of integers" could be defined as basic_string<int>.

Example 12.1: *Consider the following series of operations, which are performed on the string $S =$ "abcdefghijklmnop":*

Operation	*Output*
S.size()	16
S.at(5)	'f'
S[5]	'f'
S + "qrs"	"abcdefghijklmnopqrs"
S == "abcdefghijklmnop"	**true**
S.find("ghi")	6
S.substr(4,6)	"efghij"
S.erase(4,6)	"abcdklmnop"
S.insert(1,"xxx")	"axxxbcdklmnop"
S += "xy"	"axxxbcdklmnopxy"
S.append("z")	"axxxbcdklmnopxyz"

With the exception of the find(Q) function, which we discuss in Section 12.3, all the above functions are easily implemented simply by representing the string as an array of characters.

12.2 Dynamic Programming

In this section, we discuss the ***dynamic programming*** algorithm-design technique. This technique is similar to the divide-and-conquer technique (Section 11.1.1), in that it can be applied to a wide variety of different problems. There are few algorithmic techniques that can take problems that seem to require exponential time and produce polynomial-time algorithms to solve them. Dynamic programming is one such technique. In addition, the algorithms that result from applications of the dynamic programming technique are usually quite simple—often needing little more than a few lines of code to describe some nested loops for filling in a table.

12.2.1 Matrix Chain-Product

Rather than starting out with an explanation of the general components of the dynamic programming technique, we begin by giving a classic, concrete example. Suppose we are given a collection of n two-dimensional arrays (matrices) for which we wish to compute the product

$$A = A_0 \cdot A_1 \cdot A_2 \cdots A_{n-1},$$

where A_i is a $d_i \times d_{i+1}$ matrix, for $i = 0, 1, 2, \ldots, n-1$. In the standard matrix multiplication algorithm (which is the one we use), to multiply a $d \times e$-matrix B times an $e \times f$-matrix C, we compute the product, A, as

$$A[i][j] = \sum_{k=0}^{e-1} B[i][k] \cdot C[k][j].$$

This definition implies that matrix multiplication is associative, that is, it implies that $B \cdot (C \cdot D) = (B \cdot C) \cdot D$. Thus, we can parenthesize the expression for A any way we wish and we still end up with the same answer. We do not necessarily perform the same number of primitive (that is, scalar) multiplications in each parenthesization, however, as is illustrated in the following example.

Example 12.2: *Let B be a 2×10-matrix, let C be a 10×50-matrix, and let D be a 50×20-matrix. Computing $B \cdot (C \cdot D)$ requires $2 \cdot 10 \cdot 20 + 10 \cdot 50 \cdot 20 = 10,400$ multiplications, whereas computing $(B \cdot C) \cdot D$ requires $2 \cdot 10 \cdot 50 + 2 \cdot 50 \cdot 20 = 3000$ multiplications.*

The ***matrix chain-product*** problem is to determine the parenthesization of the expression defining the product A that minimizes the total number of scalar multiplications performed. As the example above illustrates, the differences between different solutions can be dramatic, so finding a good solution can result in significant speedups.

Defining Subproblems

Of course, one way to solve the matrix chain-product problem is to simply enumerate all the possible ways of parenthesizing the expression for A and determine the number of multiplications performed by each one. Unfortunately, the set of all different parenthesizations of the expression for A is equal in number to the set of all different binary trees that have n external nodes. This number is exponential in n. Thus, this straightforward ("brute force") algorithm runs in exponential time, for there are an exponential number of ways to parenthesize an associative arithmetic expression.

We can improve the performance achieved by the brute-force algorithm significantly, however, by making a few observations about the nature of the matrix chain-product problem. The first observation is that the problem can be split into ***subproblems***. In this case, we can define a number of different subproblems, each of which computes the best parenthesization for some subexpression $A_i \cdot A_{i+1} \cdots A_j$. As a concise notation, we use $N_{i,j}$ to denote the minimum number of multiplications needed to compute this subexpression. Thus, the original matrix chain-product problem can be characterized as that of computing the value of $N_{0,n-1}$. This observation is important, but we need one more in order to apply the dynamic programming technique.

Characterizing Optimal Solutions

The other important observation we can make about the matrix chain-product problem is that it is possible to characterize an optimal solution to a particular subproblem in terms of optimal solutions to its subproblems. We call this property the ***subproblem optimality*** condition.

In the case of the matrix chain-product problem, we observe that, no matter how we parenthesize a subexpression, there has to be some final matrix multiplication that we perform. That is, a full parenthesization of a subexpression $A_i \cdot A_{i+1} \cdots A_j$ has to be of the form $(A_i \cdots A_k) \cdot (A_{k+1} \cdots A_j)$, for some $k \in \{i, i+1, \ldots, j-1\}$. Moreover, for whichever k is the correct one, the products $(A_i \cdots A_k)$ and $(A_{k+1} \cdots A_j)$ must also be solved optimally. If this were not so, then there would be a global optimal that had one of these subproblems solved suboptimally. But this is impossible, since we could then reduce the total number of multiplications by replacing the current subproblem solution by an optimal solution for the subproblem. This observation implies a way of explicitly defining the optimization problem for $N_{i,j}$ in terms of other optimal subproblem solutions. Namely, we can compute $N_{i,j}$ by considering each place k where we could put the final multiplication and taking the minimum over all such choices.

Designing a Dynamic Programming Algorithm

We can therefore characterize the optimal subproblem solution, $N_{i,j}$, as

$$N_{i,j} = \min_{i \le k < j} \{N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\},$$

where $N_{i,i} = 0$, since no work is needed for a single matrix. That is, $N_{i,j}$ is the minimum, taken over all possible places to perform the final multiplication, of the number of multiplications needed to compute each subexpression plus the number of multiplications needed to perform the final matrix multiplication.

Notice that there is a ***sharing of subproblems*** going on that prevents us from dividing the problem into completely independent subproblems (as we would need to do to apply the divide-and-conquer technique). We can, nevertheless, use the equation for $N_{i,j}$ to derive an efficient algorithm by computing $N_{i,j}$ values in a bottom-up fashion, and storing intermediate solutions in a table of $N_{i,j}$ values. We can begin simply enough by assigning $N_{i,i} = 0$ for $i = 0, 1, \ldots, n-1$. We can then apply the general equation for $N_{i,j}$ to compute $N_{i,i+1}$ values, since they depend only on $N_{i,i}$ and $N_{i+1,i+1}$ values that are available. Given the $N_{i,i+1}$ values, we can then compute the $N_{i,i+2}$ values, and so on. Therefore, we can build $N_{i,j}$ values up from previously computed values until we can finally compute the value of $N_{0,n-1}$, which is the number that we are searching for. The details of this ***dynamic programming*** solution are given in Code Fragment 12.1.

Algorithm MatrixChain$(d_0, \ldots, d_n)$:
Input: Sequence $d_0, \ldots, d_n$ of integers
Output: For $i, j = 0, \ldots, n-1$, the minimum number of multiplications $N_{i,j}$ needed to compute the product $A_i \cdot A_{i+1} \cdots A_j$, where A_k is a $d_k \times d_{k+1}$ matrix

```
for i ← 0 to n − 1 do
    N_{i,i} ← 0
for b ← 1 to n − 1 do
    for i ← 0 to n − b − 1 do
        j ← i + b
        N_{i,j} ← +∞
        for k ← i to j − 1 do
            N_{i,j} ← min{N_{i,j}, N_{i,k} + N_{k+1,j} + d_i d_{k+1} d_{j+1}}.
```

Code Fragment 12.1: Dynamic programming algorithm for the matrix chain-product problem.

Thus, we can compute $N_{0,n-1}$ with an algorithm that consists primarily of three nested for-loops. The outside loop is executed n times. The loop inside is executed at most n times. And the inner-most loop is also executed at most n times. Therefore, the total running time of this algorithm is $O(n^3)$.

12.2.2 DNA and Text Sequence Alignment

A common text processing problem, which arises in genetics and software engineering, is to test the similarity between two text strings. In a genetics application, the two strings could correspond to two strands of DNA, that we want to compare. Likewise, in a software engineering application, the two strings could come from two versions of source code for the same program. We might want to compare the two versions to determine what changes have been made from one version to the next. Indeed, determining the similarity between two strings is so common that the Unix and Linux operating systems have a built-in program, `diff`, for comparing text files.

Given a string $X = x_0x_1x_2\cdots x_{n-1}$, a ***subsequence*** of X is any string that is of the form $x_{i_1}x_{i_2}\cdots x_{i_k}$, where $i_j < i_{j+1}$; that is, it is a sequence of characters that are not necessarily contiguous but are nevertheless taken in order from X. For example, the string *AAAG* is a subsequence of the string *CGATAATTGAGA*.

The DNA and text similarity problem we address here is the ***longest common subsequence*** (LCS) problem. In this problem, we are given two character strings, $X = x_0x_1x_2\cdots x_{n-1}$ and $Y = y_0y_1y_2\cdots y_{m-1}$, over some alphabet (such as the alphabet $\{A,C,G,T\}$ common in computational genetics) and are asked to find a longest string S that is a subsequence of both X and Y. One way to solve the longest common subsequence problem is to enumerate all subsequences of X and take the largest one that is also a subsequence of Y. Since each character of X is either in or not in a subsequence, there are potentially 2^n different subsequences of X, each of which requires $O(m)$ time to determine whether it is a subsequence of Y. Thus, this brute-force approach yields an exponential-time algorithm that runs in $O(2^n m)$ time, which is very inefficient. Fortunately, the LCS problem is efficiently solvable using ***dynamic programming***.

The Components of a Dynamic Programming Solution

As mentioned above, the dynamic programming technique is used primarily for ***optimization*** problems, where we wish to find the "best" way of doing something. We can apply the dynamic programming technique in such situations if the problem has certain properties.

Simple Subproblems: There has to be some way of repeatedly breaking the global-optimization problem into subproblems. Moreover, there should be a simple way of defining subproblems with just a few indices, like i, j, k, and so on.

Subproblem Optimization: An optimal solution to the global problem must be a composition of optimal subproblem solutions.

Subproblem Overlap: Optimal solutions to unrelated subproblems can contain subproblems in common.

Applying Dynamic Programming to the LCS Problem

Recall that in the LCS problem, we are given two character strings, X and Y, of length n and m, respectively, and are asked to find a longest string S that is a subsequence of both X and Y. Since X and Y are character strings, we have a natural set of indices with which to define subproblems—indices into the strings X and Y. Let us define a subproblem, therefore, as that of computing the value $L[i,j]$, which we will use to denote the length of a longest string that is a subsequence of both $X[0..i] = x_0x_1x_2\ldots x_i$ and $Y[0..j] = y_0y_1y_2\ldots y_j$. This definition allows us to rewrite $L[i,j]$ in terms of optimal subproblem solutions. This definition depends on which of two cases we are in. (See Figure 12.1.)

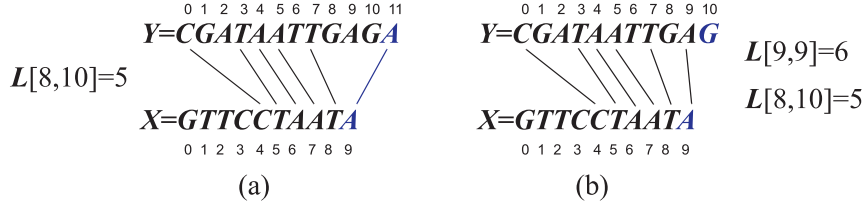


Figure 12.1: The two cases in the longest common subsequence algorithm: (a) $x_i = y_j$; (b) $x_i \neq y_j$. Note that the algorithm stores only the $L[i,j]$ values, not the matches.

- $x_i = y_j$. In this case, we have a match between the last character of $X[0..i]$ and the last character of $Y[0..j]$. We claim that this character belongs to a longest common subsequence of $X[0..i]$ and $Y[0..j]$. To justify this claim, let us suppose it is not true. There has to be some longest common subsequence $x_{i_1}x_{i_2}\ldots x_{i_k} = y_{j_1}y_{j_2}\ldots y_{j_k}$. If $x_{i_k} = x_i$ or $y_{j_k} = y_j$, then we get the same sequence by setting $i_k = i$ and $j_k = j$. Alternately, if $x_{j_k} \neq x_i$, then we can get an even longer common subsequence by adding x_i to the end. Thus, a longest common subsequence of $X[0..i]$ and $Y[0..j]$ ends with x_i. Therefore, we set

$$L[i,j] = L[i-1,j-1]+1 \quad \text{if } x_i = y_j.$$

- $x_i \neq y_j$. In this case, we cannot have a common subsequence that includes both x_i and y_j. That is, we can have a common subsequence end with x_i or one that ends with y_j (or possibly neither), but certainly not both. Therefore, we set

$$L[i,j] = \max\{L[i-1,j], L[i,j-1]\} \quad \text{if } x_i \neq y_j.$$

In order to make both of these equations make sense in the boundary cases when $i = 0$ or $j = 0$, we assign $L[i,-1] = 0$ for $i = -1,0,1,\ldots,n-1$ and $L[-1,j] = 0$ for $j = -1,0,1,\ldots,m-1$.

The LCS Algorithm

The definition of $L[i,j]$ satisfies subproblem optimization, since we cannot have a longest common subsequence without also having longest common subsequences for the subproblems. Also, it uses subproblem overlap, because a subproblem solution $L[i,j]$ can be used in several other problems (namely, the problems $L[i+1,j]$, $L[i,j+1]$, and $L[i+1,j+1]$). Turning this definition of $L[i,j]$ into an algorithm is actually quite straightforward. We initialize an $(n+1)\times(m+1)$ array, L, for the boundary cases when $i=0$ or $j=0$. Namely, we initialize $L[i,-1]=0$ for $i=-1,0,1,\ldots,n-1$ and $L[-1,j]=0$ for $j=-1,0,1,\ldots,m-1$. Then, we iteratively build up values in L until we have $L[n-1,m-1]$, the length of a longest common subsequence of X and Y. We give a pseudo-code description of this algorithm in Code Fragment 12.2.

Algorithm LCS(X,Y):
Input: Strings X and Y with n and m elements, respectively
Output: For $i=0,\ldots,n-1$, $j=0,\ldots,m-1$, the length $L[i,j]$ of a longest string that is a subsequence of both the string $X[0..i]=x_0x_1x_2\cdots x_i$ and the string $Y[0..j]=y_0y_1y_2\cdots y_j$

```
for i ← −1 to n − 1 do
  L[i, −1] ← 0
for j ← 0 to m − 1 do
  L[−1, j] ← 0
for i ← 0 to n − 1 do
  for j ← 0 to m − 1 do
    if x_i = y_j then
      L[i, j] ← L[i − 1, j − 1] + 1
    else
      L[i, j] ← max{L[i − 1, j], L[i, j − 1]}
return array L
```

Code Fragment 12.2: Dynamic programming algorithm for the LCS problem.

The running time of the algorithm of Code Fragment 12.2 is easy to analyze, because it is dominated by two nested **for** loops, with the outer one iterating n times and the inner one iterating m times. Since the if-statement and assignment inside the loop each requires $O(1)$ primitive operations, this algorithm runs in $O(nm)$ time. Thus, the dynamic programming technique can be applied to the longest common subsequence problem to improve significantly over the exponential-time brute-force solution to the LCS problem.

Algorithm LCS (Code Fragment 12.2) computes the length of the longest common subsequence (stored in $L[n-1,m-1]$), but not the subsequence itself. As shown in the following proposition, a simple postprocessing step can extract the longest common subsequence from the array L returned by the algorithm.

Proposition 12.3: *Given a string X of n characters and a string Y of m characters, we can find the longest common subsequence of X and Y in $O(nm)$ time.*

Justification: Algorithm LCS computes $L[n-1,m-1]$, the ***length*** of a longest common subsequence, in $O(nm)$ time. Given the table of $L[i,j]$ values, constructing a longest common subsequence is straightforward. One method is to start from $L[n,m]$ and work back through the table, reconstructing a longest common subsequence from back to front. At any position $L[i,j]$, we can determine whether $x_i = y_j$. If this is true, then we can take x_i as the next character of the subsequence (noting that x_i is ***before*** the previous character we found, if any), moving next to $L[i-1,j-1]$. If $x_i \neq y_j$, then we can move to the larger of $L[i,j-1]$ and $L[i-1,j]$. (See Figure 12.2.) We stop when we reach a boundary cell (with $i=-1$ or $j=-1$). This method constructs a longest common subsequence in $O(n+m)$ additional time. ■

L	-1	0	1	2	3	4	5	6	7	8	9	10	11
-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	1	1	1
1	0	0	1	1	2	2	2	2	2	2	2	2	2
2	0	0	1	1	2	2	2	3	3	3	3	3	3
3	0	1	1	1	2	2	2	3	3	3	3	3	3
4	0	**1**	1	1	2	2	2	3	3	3	3	3	3
5	0	1	1	1	**2**	2	2	3	4	4	4	4	4
6	0	1	1	2	2	**3**	3	3	4	4	5	5	5
7	0	1	1	2	2	3	**4**	4	4	4	5	5	6
8	0	1	1	2	3	3	4	5	**5**	5	5	5	6
9	0	1	1	2	3	4	4	5	5	5	6	6	**6**

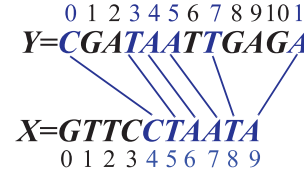


Figure 12.2: The algorithm for constructing a longest common subsequence from the array L.

12.3 Pattern Matching Algorithms

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 $T[i] = P[0]$, $T[i+1] = P[1]$, ..., $T[i+m-1] = P[m-1]$. That is, $P = T[i..i+m-1]$. Thus, the output from a pattern matching algorithm could either be some indication that the pattern P does not exist in T or an integer indicating the starting index in T of a substring matching P. This is exactly the computation performed by the find function of the STL string interface. Alternatively, one may want to find all the indices where a substring of T matching P begins.

In this section, we present three pattern matching algorithms (with increasing levels of difficulty).

12.3.1 Brute Force

The ***brute-force*** algorithmic design pattern is a powerful technique for algorithm design when we have something we wish to search for or when we wish to optimize some function. 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 design the ***brute-force pattern matching*** algorithm, we derive what is probably the first algorithm that we might think of for solving the pattern matching problem—we simply test all the possible placements of P relative to T. This algorithm, shown in Code Fragment 12.3, is quite simple.

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."
```

Code Fragment 12.3: Brute-force pattern matching.

Performance

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

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 Code Fragment 12.3, 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 simplified as $O(nm)$. Note that when $m = n/2$, this algorithm has quadratic running time $O(n^2)$.

Example 12.4: *Suppose we are given the text string*

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

and the pattern string

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

In Figure 12.3, we illustrate the execution of the brute-force pattern matching algorithm on T and P.

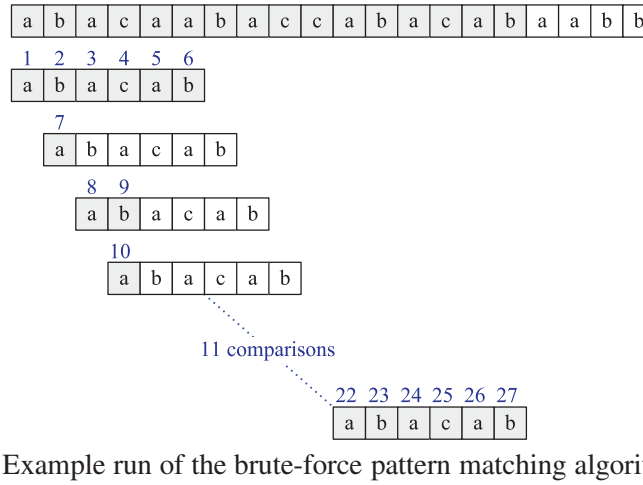


Figure 12.3: Example run of the brute-force pattern matching algorithm. The algorithm performs 27 character comparisons, indicated above with numerical labels.

12.3.2 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. 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 the fastest when the alphabet is moderately sized and the pattern is relatively long. Thus, the BM algorithm is ideal for searching words in documents. In this section, we describe a simplified version of the original algorithm by Boyer and Moore.

The main idea of the BM algorithm is to improve the running time of the brute-force algorithm by adding two potentially time-saving heuristics. Roughly stated, these heuristics are as follows:

***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 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 backwards, 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.

Let us therefore get down to the business of defining 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 characterizes 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:

- 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 last function can be easily implemented as a lookup table. We leave the method for computing this table in $O(m+|\Sigma|)$ time, given P, as a simple exercise (Exercise R-12.8). This last function gives us all the information we need to perform the character-jump heuristic.

In Code Fragment 12.4, we show the BM pattern matching algorithm.

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."
```

Code Fragment 12.4: The Boyer-Moore pattern matching algorithm.

The jump step is illustrated in Figure 12.4.

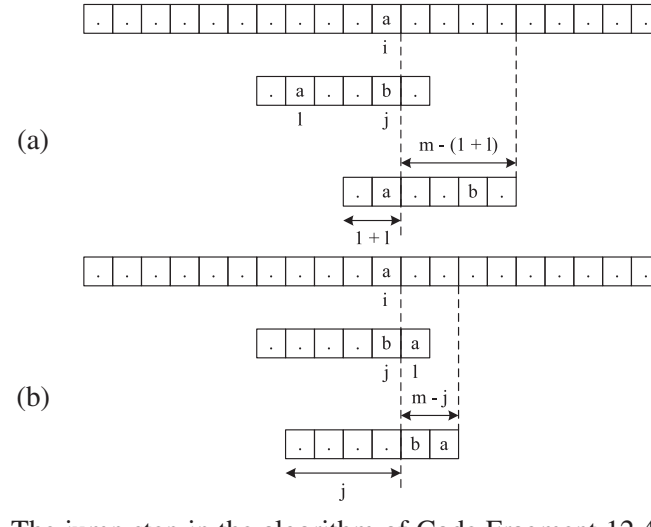
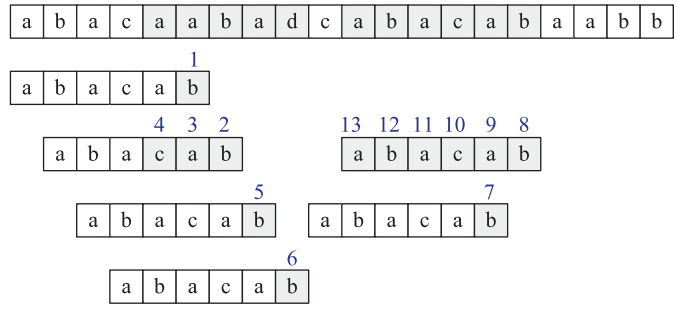


Figure 12.4: The jump step in the algorithm of Code Fragment 12.4, where we let $l = \mathsf{last}(T[i])$. We distinguish two cases: (a) $1 + l \leq j$, where we shift the pattern by $j - l$ units; (b) $j < 1 + l$, where we shift the pattern by one unit.

In Figure 12.5, we illustrate the execution of the Boyer-Moore pattern matching algorithm on an input string similar to Example 12.4.



The last(c) function:

c	a	b	c	d
last(c)	4	5	3	-1

Figure 12.5: The BM pattern matching algorithm. The algorithm performs 13 character comparisons, which are indicated with numerical labels.

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. This is because last(c) indicates 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 $O(m+|\Sigma|)$ time 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

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

The worst-case performance, however, is unlikely to be achieved for English text because, in this case, the BM algorithm is often able to skip large portions of text. (See Figure 12.6.) Experimental evidence on English text shows that the average number of comparisons done per character is 0.24 for a five-character pattern string.

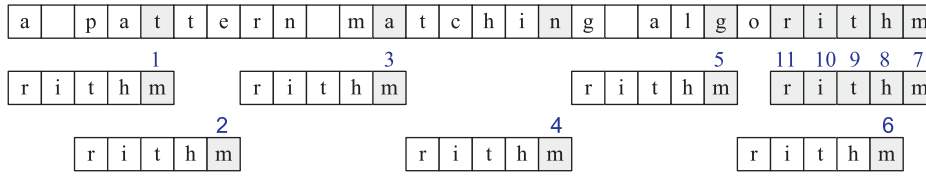


Figure 12.6: An example of a Boyer-Moore execution on English text.

A C++ implementation of the BM pattern matching algorithm, based on an STL vector, is shown in Code Fragment 12.5.

```
/** Simplified version of the Boyer-Moore algorithm. Returns the index of
 *  the leftmost substring of the text matching the pattern, or -1 if none.
 */
int BMmatch(const string& text, const string& pattern) {
  std::vector<int> last = buildLastFunction(pattern);
  int n = text.size();
  int m = pattern.size();
  int i = m - 1;
  if (i > n - 1)                          // pattern longer than text?
    return -1;                            // ...then no match
  int j = m - 1;
  do {
    if (pattern[j] == text[i])
      if (j == 0) return i;               // found a match
      else {                              // looking-glass heuristic
        i--; j--;                         // proceed right-to-left
      }
    else {                                // character-jump heuristic
      i = i + m - std::min(j, 1 + last[text[i]]);
      j = m - 1;
    }
  } while (i <= n - 1);
  return -1;                              // no match
}
                                          // construct function last
std::vector<int> buildLastFunction(const string& pattern) {
  const int N_ASCII = 128;                // number of ASCII characters
  int i;
  std::vector<int> last(N_ASCII);         // assume ASCII character set
  for (i = 0; i < N_ASCII; i++)           // initialize array
    last[i] = -1;
  for (i = 0; i < pattern.size(); i++) {
    last[pattern[i]] = i;                 // (implicit cast to ASCII code)
  }
  return last;
}
```

Code Fragment 12.5: C++ implementation of the Boyer-Moore (BM) pattern matching algorithm. The algorithm is expressed by two static functions: Method BMmatch performs the matching and calls the auxiliary function buildLastFunction to compute the last function, expressed by an array indexed by the ASCII code of the character. Method BMmatch indicates the absence of a match by returning the conventional value -1.

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 next.

12.3.3 The Knuth-Morris-Pratt Algorithm

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 12.4, 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 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 discussed in this section, 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 at least once.

The Failure Function

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 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 12.5: *Consider the pattern string $P =$ `"abacab"` from Example 12.4. The Knuth-Morris-Pratt (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 Code Fragment 12.6, incrementally processes the text string T comparing it to the pattern string P. 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).

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."
```

Code Fragment 12.6: The KMP pattern matching algorithm.

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. Any comparisons that are skipped are actually unnecessary, for the failure function guarantees that all the ignored comparisons are redundant—they would involve comparing the same matching characters over again.

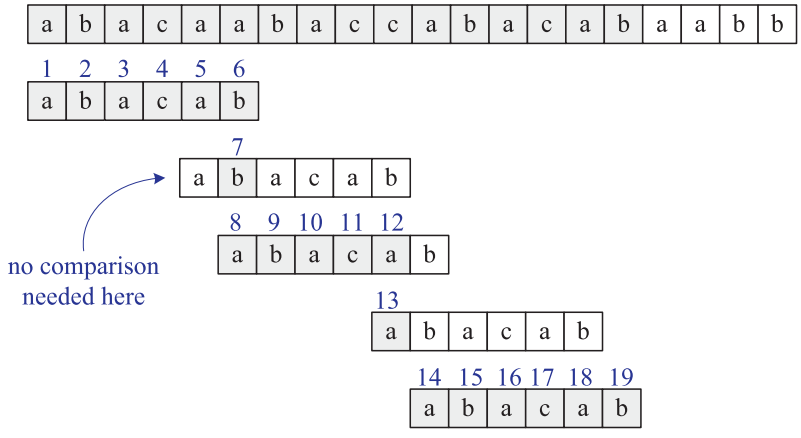


Figure 12.7: The KMP pattern matching algorithm. The failure function f for this pattern is given in Example 12.5. The algorithm performs 19 character comparisons, which are indicated with numerical labels.

In Figure 12.7, we illustrate the execution of the KMP pattern matching algorithm on the same input strings as in Example 12.4. 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 12.3).

Performance

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 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, we use the method shown in Code Fragment 12.7, which is a "bootstrapping" process quite similar to 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(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
```

Code Fragment 12.7: 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.

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

Proposition 12.6: *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.*

A C++ implementation of the KMP pattern matching algorithm, based on an STL vector, is shown in Code Fragment 12.8.

```
                                                    // KMP algorithm
int KMPmatch(const string& text, const string& pattern) {
  int n = text.size();
  int m = pattern.size();
  std::vector<int> fail = computeFailFunction(pattern);
  int i = 0;                                        // text index
  int j = 0;                                        // pattern index
  while (i < n) {
    if (pattern[j] == text[i]) {
      if (j == m - 1)
        return i - m + 1;                           // found a match
      i++; j++;
    }
    else if (j > 0) j = fail[j - 1];
    else i++;
  }
  return -1;                                        // no match
}

std::vector<int> computeFailFunction(const string& pattern) {
  std::vector<int> fail(pattern.size());
  fail[0] = 0;
  int m = pattern.size();
  int j = 0;
  int i = 1;
  while (i < m) {
    if (pattern[j] == pattern[i]) {                 // j + 1 characters match
      fail[i] = j + 1;
      i++; j++;
    }
    else if (j > 0)                                 // j follows a matching prefix
      j = fail[j - 1];
    else {                                          // no match
      fail[i] = 0;
      i++;
    }
  }
  return fail;
}
```

Code Fragment 12.8: C++ implementation of the KMP pattern matching algorithm. The algorithm is expressed by two static functions. Function KMPmatch performs the matching and calls the auxiliary function computeFailFunction to compute the failure function, expressed by an array. Method KMPmatch indicates the absence of a match by returning the conventional value -1.

12.4 Text Compression and the Greedy Method

In this section, we consider an important text processing task, ***text compression***. In this problem, we are given a string X defined over some alphabet, such as the ASCII or Unicode character sets, and we want to efficiently encode X into a small binary string Y (using only the characters 0 and 1). Text compression is useful in any situation where we are communicating over a low-bandwidth channel, such as a modem line or infrared connection, and we wish to minimize the time needed to transmit our text. Likewise, text compression is also useful for storing collections of large documents more efficiently, in order to allow for a fixed-capacity storage device to contain as many documents as possible.

The method for text compression explored in this section is the ***Huffman code***. Standard encoding schemes, such as the ASCII and Unicode systems, use fixed-length binary strings to encode characters (with 7 bits in the ASCII system and 16 in the Unicode system). A Huffman code, on the other hand, uses a variable-length encoding optimized for the string X. The optimization is based on the use of character ***frequencies***, where we have, for each character c, a count $f(c)$ of the number of times c appears in the string X. The Huffman code saves space over a fixed-length encoding by using short code-word strings to encode high-frequency characters and long code-word strings to encode low-frequency characters.

To encode the string X, we convert each character in X from its fixed-length code word to its variable-length code word, and we concatenate all these code words in order to produce the encoding Y for X. In order to avoid ambiguities, we insist that no code word in our encoding is a prefix of another code word in our encoding. Such a code is called a ***prefix code***, and it simplifies the decoding of Y in order to get back X. (See Figure 12.8.) Even with this restriction, the savings produced by a variable-length prefix code can be significant, particularly if there is a wide variance in character frequencies (as is the case for natural language text in almost every spoken language).

Huffman's algorithm for producing an optimal variable-length prefix code for X is based on the construction of a binary tree T that represents the code. Each node in T, except the root, represents a bit in a code word, with each left child representing a "0" and each right child representing a "1." Each external node v is associated with a specific character, and the code word for that character is defined by the sequence of bits associated with the nodes in the path from the root of T to v. (See Figure 12.8.) Each external node v has a ***frequency***, $f(v)$, which is simply the frequency in X of the character associated with v. In addition, we give each internal node v in T a frequency, $f(v)$, that is the sum of the frequencies of all the external nodes in the subtree rooted at v.

(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

(b)

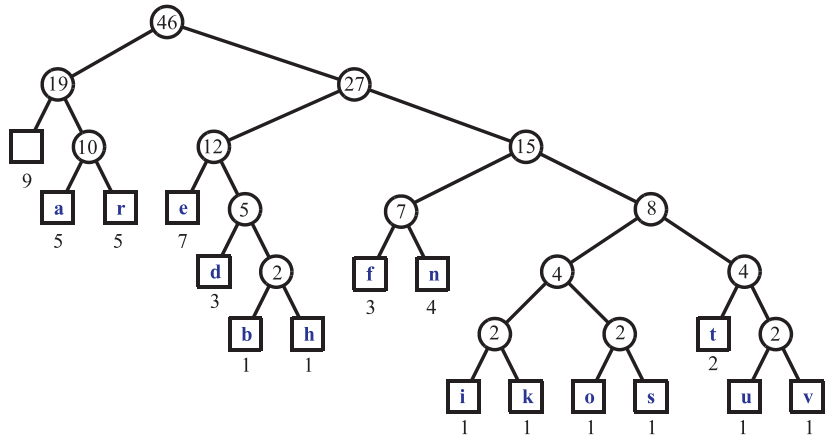


Figure 12.8: An example Huffman code for the input string X = `"a fast runner need never be afraid of the dark"`: (a) frequency of each character of X; (b) Huffman tree T for string X. The code for a character c is obtained by tracing the path from the root of T to the external node where c is stored, and associating a left child with 0 and a right child with 1. For example, the code for "a" is 010, and the code for "f" is 1100.

12.4.1 The Huffman-Coding Algorithm

The Huffman-coding algorithm begins with each of the d distinct characters of the string X to encode being the root node of a single-node binary tree. The algorithm proceeds in a series of rounds. In each round, the algorithm takes the two binary trees with the smallest frequencies and merges them into a single binary tree. It repeats this process until only one tree is left. (See Code Fragment 12.9.)

Each iteration of the **while** loop in Huffman's algorithm can be implemented in $O(\log d)$ time using a priority queue represented with a heap. In addition, each iteration takes two nodes out of Q and adds one in, a process that is repeated $d-1$ times before exactly one node is left in Q. Thus, this algorithm runs in $O(n+d\log d)$ time. Although a full justification of this algorithm's correctness is beyond our scope, we note that its intuition comes from a simple idea—any optimal code can be converted into an optimal code in which the code words for the two lowest-frequency characters, a and b, differ only in their last bit. Repeating the argument for a string with a and b replaced by a character c, gives the following.

Proposition 12.7: *Huffman's algorithm constructs an optimal prefix code for a string of length n with d distinct characters in $O(n+d\log d)$ time.*

Algorithm Huffman(X):
Input: String X of length n with d distinct characters
Output: Coding tree for X

Compute the frequency $f(c)$ of each character c of X.
Initialize a priority queue Q.
for each character c in X **do**
  Create a single-node binary tree T storing c.
  Insert T into Q with key $f(c)$.
while Q.size() > 1 **do**
  $f_1 \leftarrow Q$.min()
  $T_1 \leftarrow Q$.removeMin()
  $f_2 \leftarrow Q$.min()
  $T_2 \leftarrow Q$.removeMin()
  Create a new binary tree T with left subtree T_1 and right subtree T_2.
  Insert T into Q with key $f_1 + f_2$.
return tree Q.removeMin()

Code Fragment 12.9: Huffman-coding algorithm.

12.4.2 The Greedy Method

Huffman's algorithm for building an optimal encoding is an example application of an algorithmic design pattern called the ***greedy method***. This design pattern is applied to optimization problems, where we are trying to construct some structure while minimizing or maximizing some property of that structure.

The general formula for the greedy method pattern is almost as simple as that for the brute-force method. In order to solve a given optimization problem using the greedy method, we proceed by a sequence of choices. The sequence starts from some well-understood starting condition, and computes the cost for that initial condition. The pattern then asks that we iteratively make additional choices by identifying the decision that achieves the best cost improvement from all of the choices that are currently possible. This approach does not always lead to an optimal solution.

But there are several problems that it does work for, and such problems are said to possess the ***greedy-choice*** property. This is the property that a global optimal condition can be reached by a series of locally optimal choices (that is, choices that are each the current best from among the possibilities available at the time), starting from a well-defined starting condition. The problem of computing an optimal variable-length prefix code is just one example of a problem that possesses the greedy-choice property.

12.5 Tries

The pattern matching algorithms presented in the previous section speed up the search 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 this section, we take a complementary approach, namely, we present string searching algorithms that preprocess the text. 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 Web site that offers pattern matching in Shakespeare's *Hamlet* or a search engine that offers Web pages on the *Hamlet* topic).

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.

12.5.1 Standard Tries

Let S be a set of s strings from alphabet Σ 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 12.9):

- Each node of T, except the root, is labeled with a character of Σ.
- The ordering of the children of an internal node of T is determined by a canonical ordering of the alphabet Σ.
- 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 Σ 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

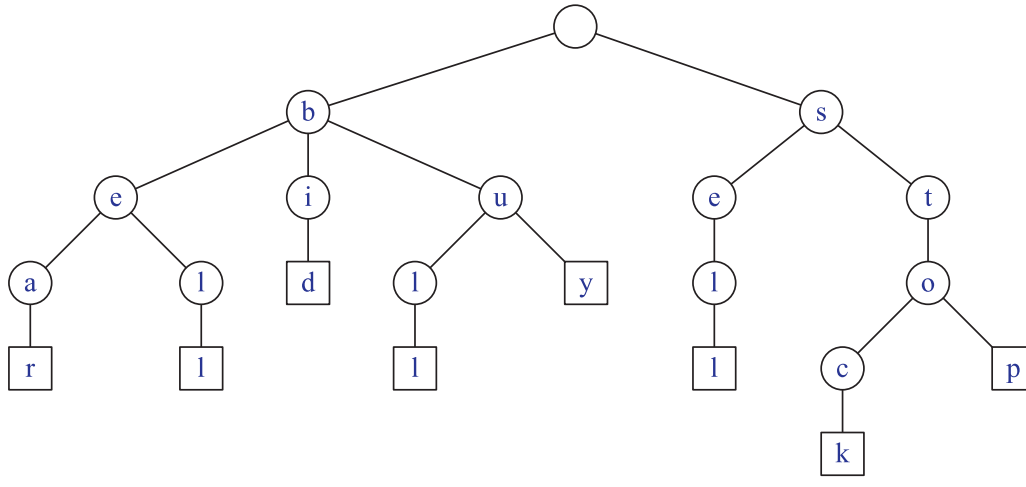


Figure 12.9: Standard trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}.

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, with some internal nodes possibly having 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 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 12.9 has several internal nodes with only one child. We can implement a trie with a tree storing characters at its nodes.

The following proposition provides some important structural properties of a standard trie.

Proposition 12.8: *A standard trie storing a collection S of s strings of total length n from an alphabet of size d has the following properties:*

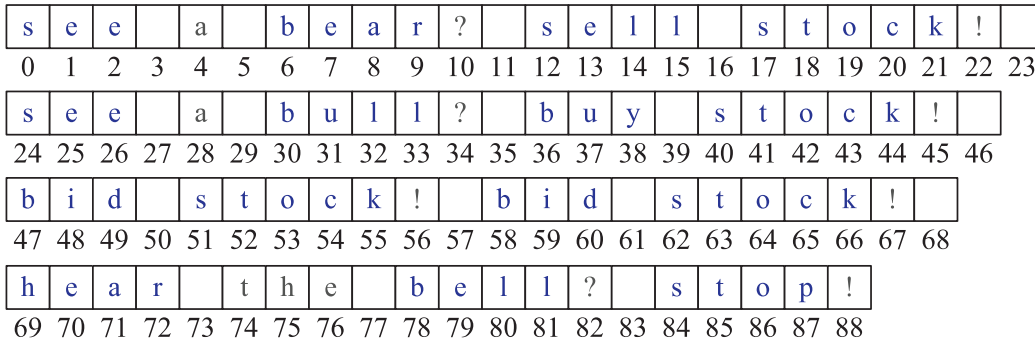
- *Every internal node of T has at most d children*
- *T has s external nodes*
- *The height of T is equal to the length of the longest string in S*
- *The number of nodes of T is $O(n)$*

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.

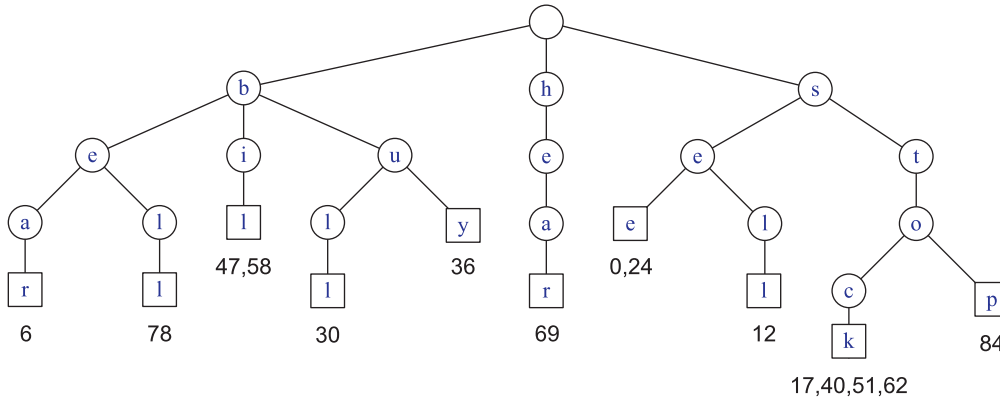
A trie T for a set S of strings can be used to implement a dictionary whose keys are the strings of S. Namely, we perform a search in T for a string X by tracing down from the root the path indicated by the characters in X. If this path can be traced and terminates at an external node, then we know X is in the dictionary. For example, in the trie in Figure 12.9, 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 12.9, 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). 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 search 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.

From the discussion above, 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 12.10.) 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.

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 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 descendents 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 12.10: Word matching and prefix matching with a standard trie: (a) text to be searched; (b) standard trie for the words in the text (articles and prepositions, which are also known as ***stop words***, excluded), with external nodes augmented with indications of the 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. We discuss the compressed trie next.

12.5.2 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 12.11.) 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 12.9 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 $v_1, \ldots, v_k$.

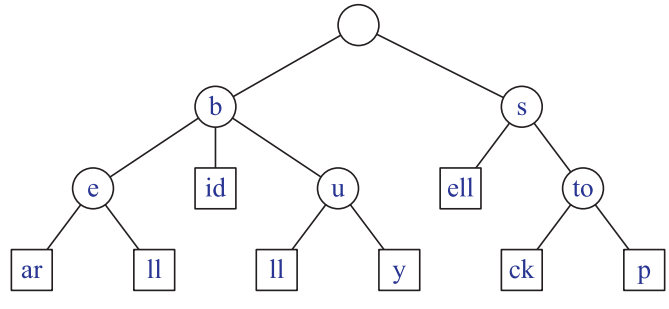


Figure 12.11: Compressed trie for the strings {bear, bell, bid, bull, buy, sell, stock, stop}. Compare this with the standard trie shown in Figure 12.9.

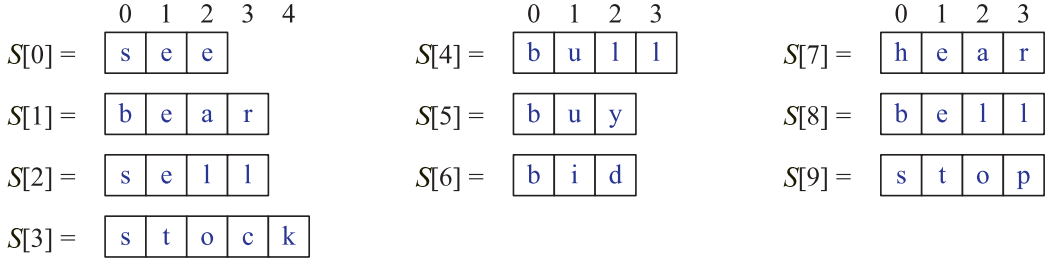
Thus, nodes in a compressed trie are labeled with strings, which are substrings of strings in the collection, rather than with individual characters. The advantage of a compressed trie over a standard trie is that the number of nodes of the compressed trie is proportional to the number of strings and not to their total length, as shown in the following proposition (compare with Proposition 12.8).

Proposition 12.9: *A compressed trie storing a collection S of s strings from an alphabet of size d has the following properties:*

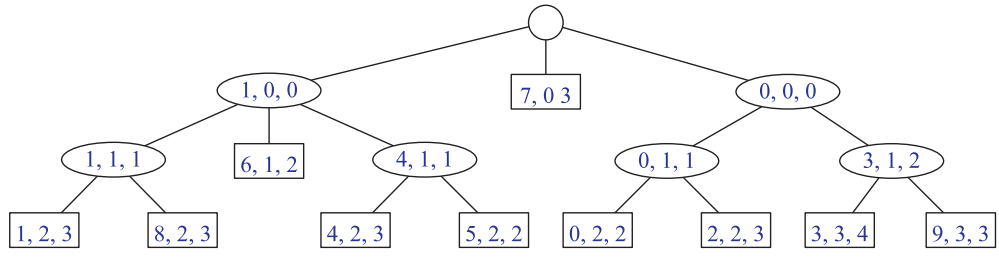
- *Every internal node of T has at least two children and most d children*
- *T has s external nodes*
- *The number of nodes of T is $O(s)$*

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.

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 jth to the kth included. (See the example in Figure 12.12. Also compare with the standard trie of Figure 12.10.)



(a)



(b)

Figure 12.12: (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.

12.5.3 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 12.13(a) 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, the label of each vertex is a pair (i, j) indicating the string $X[i..j]$. (See Figure 12.13(b).) 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 Σ 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 proposition.

Proposition 12.10: *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 12.5.1. 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.

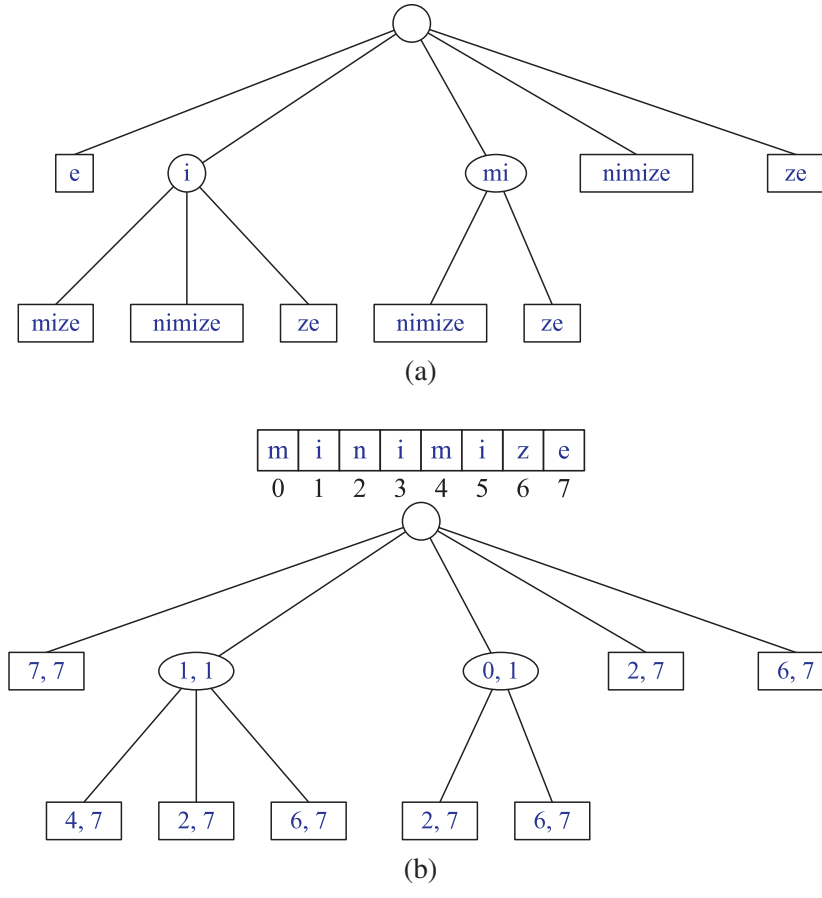


Figure 12.13: (a) Suffix trie T for the string $X =$ ``minimize''. (b) Compact representation of T, where pair (i, j) denotes $X[i..j]$.

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 search down the trie T assumes that nodes in T store some additional information, with respect to 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.

12.5.4 Search Engines

The World Wide Web contains a huge collection of text documents (Web pages). Information about these pages are 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 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:

1. An array storing the occurrence lists of the terms (in no particular order)
2. 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 (Section 12.5.1). Namely, we find the keyword in the trie and we return the associated occurrence list.

When multiple keywords are given and the desired output are 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 (see, for example, the generic merge computation discussed in Section 11.4).

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.

12.6 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

Reinforcement

R-12.1 What is the best way to multiply a chain of matrices with dimensions that are 10×5, 5×2, 2×20, 20×12, 12×4, and 4×60? Show your work.

R-12.2 Design an efficient algorithm for the matrix chain multiplication problem that outputs a fully parenthesized expression for how to multiply the matrices in the chain using the minimum number of operations.

R-12.3 List the prefixes of the string $P =$ `"aaabbaaa"` that are also suffixes of P.

R-12.4 Draw a figure illustrating the comparisons done by brute-force pattern matching for the text `"aaabaadaabaaa"` and pattern `"aabaaa"`.

R-12.5 Repeat the previous problem for the BM pattern matching algorithm, not counting the comparisons made to compute the $\mathsf{last}(c)$ function.

R-12.6 Repeat the previous problem for the KMP pattern matching algorithm, not counting the comparisons made to compute the failure function.

R-12.7 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):

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

R-12.8 Assuming that the characters in alphabet Σ can be enumerated and can be used to index arrays, give an $O(m + |\Sigma|)$-time method for constructing the last function from an m-length pattern string P.

R-12.9 Compute a table representing the KMP failure function for the pattern string `"cgtacgttcgtac"`.

R-12.10 Draw a standard trie for the following set of strings:

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

R-12.11 Draw a compressed trie for the set of strings given in Exercise R-12.10.

R-12.12 Draw the compact representation of the suffix trie for the string

"minimize minime".

R-12.13 What is the longest prefix of the string "cgtacgttcgtacg" that is also a suffix of this string?

R-12.14 Draw the frequency array and Huffman tree for the following string:

"dogs do not spot hot pots or cats".

R-12.15 Show the longest common subsequence array L for the two strings

$$X = \text{"skullandbones"}$$
$$Y = \text{"lullabybabies"}.$$

What is a longest common subsequence between these strings?

Creativity

C-12.1 A native Australian named Anatjari wishes to cross a desert carrying only a single water bottle. He has a map that marks all the watering holes along the way. Assuming he can walk k miles on one bottle of water, design an efficient algorithm for determining where Anatjari should refill his bottle in order to make as few stops as possible. Argue why your algorithm is correct.

C-12.2 Describe an efficient greedy algorithm for making change for a specified value using a minimum number of coins, assuming there are four denominations of coins, called quarters, dimes, nickels, and pennies, with values 25, 10, 5, and 1, respectively. Argue why your algorithm is correct.

C-12.3 Give an example set of denominations of coins so that a greedy change-making algorithm will not use the minimum number of coins.

C-12.4 In the ***art gallery guarding*** problem we are given a line L that represents a long hallway in an art gallery. We are also given a set $X = \{x_0, x_1, \ldots, x_{n-1}\}$ of real numbers that specify the positions of paintings in this hallway. Suppose that a single guard can protect all the paintings within distance at most 1 of his or her position (on both sides). Design an algorithm for finding a placement of guards that uses the minimum number of guards to guard all the paintings with positions in X.

C-12.5 Let P be a convex polygon, a ***triangulation*** of P is an addition of diagonals connecting the vertices of P so that each interior face is a triangle. The ***weight*** of a triangulation is the sum of the lengths of the diagonals.

Assuming that we can compute lengths and add and compare them in constant time, give an efficient algorithm for computing a minimum-weight triangulation of P.

C-12.6 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-12.7 Give a justification of why the KMPFailureFunction function (Code Fragment 12.7) runs in $O(m)$ time on a pattern of length m.

C-12.8 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-12.9 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-12.10 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 (normal) 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-12.11 The KMP pattern matching algorithm can be modified to run faster on binary strings by redefining the failure function as

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

where $\widehat{p_k}$ denotes the complement of the kth 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 function for computing this failure function. Show that this function 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 12.3.3).

C-12.12 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-12.13 Given a string X of length n and a string Y of length m, describe an $O(n+m)$-time algorithm for finding the longest prefix of X that is a suffix of Y.

C-12.14 Give an efficient algorithm for deleting a string from a standard trie and analyze its running time.

C-12.15 Give an efficient algorithm for deleting a string from a compressed trie and analyze its running time.

C-12.16 Describe an algorithm for constructing the compact representation of a suffix trie, given its noncompact representation, and analyze its running time.

C-12.17 Let T be a 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.

C-12.18 Describe an efficient algorithm to find the longest palindrome that is a suffix of a string T of length n. Recall that a ***palindrome*** is a string that is equal to its reversal. What is the running time of your method?

C-12.19 Given a sequence $S = (x_0, x_1, x_2, \ldots, x_{n-1})$ of numbers, describe an $O(n^2)$-time algorithm for finding a longest subsequence $T = (x_{i_0}, x_{i_1}, x_{i_2}, \ldots, x_{i_{k-1}})$ of numbers, such that $i_j < i_{j+1}$ and $x_{i_j} > x_{i_{j+1}}$. That is, T is a longest decreasing subsequence of S.

C-12.20 Define the ***edit distance*** between two strings X and Y of length n and m, respectively, to be the number of edits that it takes to change X into Y. An edit consists of a character insertion, a character deletion, or a character replacement. For example, the strings `"algorithm"` and `"rhythm"` have edit distance 6. Design an $O(nm)$-time algorithm for computing the edit distance between X and Y.

C-12.21 Design a greedy algorithm for making change after someone buys some candy costing x cents and the customer gives the clerk \$1. Your algorithm should try to minimize the number of coins returned.

a. Show that your greedy algorithm returns the minimum number of coins if the coins have denominations \$0.25, \$0.10, \$0.05, and \$0.01.

b. Give a set of denominations for which your algorithm may not return the minimum number of coins. Include an example where your algorithm fails.

C-12.22 Give an efficient algorithm for determining if a pattern P is a subsequence (not substring) of a text T. What is the running time of your algorithm?

C-12.23 Let x and y be strings of length n and m respectively. Define $B(i, j)$ to be the length of the longest common substring of the suffix of length i in x and the suffix of length j in y. Design an $O(nm)$-time algorithm for computing all the values of $B(i, j)$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

C-12.24 Raji has just won a contest that allows her to take n pieces of candy out of a candy store for free. Raji is old enough to realize that some candy is expensive, while other candy is relatively cheap, costing much less. The jars of candy are numbered 0, 1, …, $m-1$, so that jar j has n_j pieces in it, with a price of c_j per piece. Design an $O(n+m)$-time algorithm that allows Raji to maximize the value of the pieces of candy she takes for her winnings. Show that your algorithm produces the maximum value for Raji.

C-12.25 Let three integer arrays, A, B, and C, be given, each of size n. Given an arbitrary integer x, design an $O(n^2 \log n)$-time algorithm to determine if there exist numbers, a in A, b in B, and c in C, such that $x = a + b + c$.

C-12.26 Give an $O(n^2)$-time algorithm for the previous problem.

Projects

P-12.1 Implement the LCS algorithm and use it to compute the best sequence alignment between some DNA strings that you can get online from GenBank.

P-12.2 Perform an experimental analysis, using documents found on the Internet, of the efficiency (number of character comparisons performed) of the brute-force and KMP pattern matching algorithms for varying-length patterns.

P-12.3 Perform an experimental analysis, using documents found on the Internet, of the efficiency (number of character comparisons performed) of the brute-force and BM pattern matching algorithms for varying-length patterns.

P-12.4 Perform an experimental comparison of the relative speeds of the brute-force, KMP, and BM pattern matching algorithms. Document the time taken for coding up each of these algorithms as well as their relative running times on documents found on the Internet that are then searched using varying-length patterns.

P-12.5 Implement a compression and decompression scheme that is based on Huffman coding.

P-12.6 Create a class that implements a standard trie for a set of ASCII strings. The class should have a constructor that takes as an argument a list of strings, and the class should have a method that tests whether a given string is stored in the trie.

P-12.7 Create a class that implements a compressed trie for a set of ASCII strings. The class should have a constructor that takes as an argument a list of strings, and the class should have a function that tests whether a given string is stored in the trie.

P-12.8 Create a class that implements a prefix trie for an ASCII string. The class should have a constructor that takes as an argument a string and a function for pattern matching on the string.

P-12.9 Implement the simplified search engine described in Section 12.5.4 for the pages of a small Web site. Use all the words in the pages of the site as index terms, excluding stop words such as articles, prepositions, and pronouns.

P-12.10 Implement a search engine for the pages of a small Web site by adding a page-ranking feature to the simplified search engine described in Section 12.5.4. Your page-ranking feature should return the most relevant pages first. Use all the words in the pages of the site as index terms, excluding stop words, such as articles, prepositions, and pronouns.

P-12.11 Write a program that takes two character strings (which could be, for example, representations of DNA strands) and computes their edit distance, showing the corresponding pieces. (See Exercise C-12.20.)

Chapter Notes

The KMP algorithm is described by Knuth, Morris, and Pratt in their journal article [61], and Boyer and Moore describe their algorithm in a journal article published the same year [14]. In their article, however, Knuth *et al.* [61] also prove that the BM algorithm runs in linear time. More recently, Cole [22] shows that the BM algorithm makes at most $3n$ character comparisons in the worst case, and this bound is tight. All of the algorithms discussed above are also discussed in the book chapter by Aho [3], although in a more theoretical framework, including the methods for regular-expression pattern matching. The reader interested in further study of string pattern matching algorithms is referred to the book by Stephen [90] and the book chapters by Aho [3] and Crochemore and Lecroq [26].

The trie was invented by Morrison [79] and is discussed extensively in the classic *Sorting and Searching* book by Knuth [60]. The name "Patricia" is short for "Practical Algorithm to Retrieve Information Coded in Alphanumeric" [79]. McCreight [69] 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 Web, is provided in the book by Baeza-Yates and Ribeiro-Neto [7].