String Matching

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

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

We say that pattern $P$ occurs with shift $s$ in text $T$ (or, equivalently, that pattern $P$ occurs beginning at position $s + 1$ in text $T$) if $0 \leq s \leq n - m$ and $T[s + 1..s + m] = P[1..m]$ (that is, if $T[s + j] = P[j]$, for $1 \leq j \leq m$). If $P$ occurs with shift $s$ in $T$, then we call $s$ a valid shift; otherwise, we call $s$ an invalid shift. The string-matching problem is the problem of finding all valid shifts with which a given pattern $P$ occurs in a given text $T$. Figure 32.1 illustrates these definitions.

![Figure 32.1](image)

**Figure 32.1** The string-matching problem. The goal is to find all occurrences of the pattern $P = abaa$ in the text $T = abcabaabcabac$. The pattern occurs only once in the text, at shift $s = 3$. The shift $s = 3$ is said to be a valid shift. Each character of the pattern is connected by a vertical line to the matching character in the text, and all matched characters are shown shaded.


<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Preprocessing time</th>
<th>Matching time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive</td>
<td>0</td>
<td>$O((n - m + 1)m)$</td>
</tr>
<tr>
<td>Rabin-Karp</td>
<td>$\Theta(m)$</td>
<td>$O((n - m + 1)m)$</td>
</tr>
<tr>
<td>Finite automaton</td>
<td>$O(m</td>
<td>\Sigma</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>$\Theta(m)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

Figure 32.2 The string-matching algorithms in this chapter and their preprocessing and matching times.

Except for the naive brute-force algorithm, which we review in Section 32.1, each string-matching algorithm in this chapter performs some preprocessing based on the pattern and then finds all valid shifts; we will call this latter phase “matching.” Figure 32.2 shows the preprocessing and matching times for each of the algorithms in this chapter. The total running time of each algorithm is the sum of the preprocessing and matching times. Section 32.2 presents an interesting string-matching algorithm, due to Rabin and Karp. Although the $\Theta((n - m + 1)m)$ worst-case running time of this algorithm is no better than that of the naive method, it works much better on average and in practice. It also generalizes nicely to other pattern-matching problems. Section 32.3 then describes a string-matching algorithm that begins by constructing a finite automaton specifically designed to search for occurrences of the given pattern $P$ in a text. This algorithm takes $O(m |\Sigma|)$ preprocessing time but only $\Theta(n)$ matching time. The similar but much cleverer Knuth-Morris-Pratt (or KMP) algorithm is presented in Section 32.4; the KMP algorithm has the same $\Theta(n)$ matching time, and it reduces the preprocessing time to only $\Theta(m)$.

Notation and terminology

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

We say that a string $w$ is a prefix of a string $x$, denoted $w \sqsubseteq x$, if $x = wy$ for some string $y \in \Sigma^*$. Note that if $w \sqsubseteq x$, then $|w| \leq |x|$. Similarly, we say that a string $w$ is a suffix of a string $x$, denoted $w \sqsupseteq x$, if $x = yw$ for some $y \in \Sigma^*$. It follows from $w \sqsubseteq x$ that $|w| \leq |x|$. The empty string $\epsilon$ is both a suffix and a prefix of every string. For example, we have $ab \sqsubseteq abccca$ and $cca \sqsupseteq abccca$. It is useful to note that for any strings $x$ and $y$ and any character $a$, we have $x \sqsupseteq y$ if
Figure 32.3 A graphical proof of Lemma 32.1. We suppose that $x \sqsubset z$ and $y \sqsubset z$. The three parts of the figure illustrate the three cases of the lemma. Vertical lines connect matching regions (shown shaded) of the strings. (a) If $|x| \leq |y|$, then $x \sqsupset y$. (b) If $|x| \geq |y|$, then $y \sqsupset x$. (c) If $|x| = |y|$, then $x = y$.

and only if $xa \sqsupset ya$. Also note that $\sqsubset$ and $\sqsupset$ are transitive relations. The following lemma will be useful later.

**Lemma 32.1 (Overlapping-suffix lemma)**

Suppose that $x$, $y$, and $z$ are strings such that $x \sqsubset z$ and $y \sqsubset z$. If $|x| \leq |y|$, then $x \sqsupset y$. If $|x| \geq |y|$, then $y \sqsupset x$. If $|x| = |y|$, then $x = y$.

**Proof** See Figure 32.3 for a graphical proof.

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

In our pseudocode, we allow two equal-length strings to be compared for equality as a primitive operation. If the strings are compared from left to right and the comparison stops when a mismatch is discovered, we assume that the time taken by such a test is a linear function of the number of matching characters discovered. To be precise, the test "$x = y$" is assumed to take time $\Theta(t+1)$, where $t$ is the length of the longest string $z$ such that $z \sqsupset x$ and $z \sqsubset y$. (We write $\Theta(t+1)$ rather than $\Theta(t)$ to handle the case in which $t = 0$; the first characters compared do not match, but it takes a positive amount of time to perform this comparison.)
### 32.1 The naive string-matching algorithm

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1..m] = T[s + 1..s + m]$ for each of the $n - m + 1$ possible values of $s$.

**NAIVE-STRING-MATCHER**(T, P)

1. $n \leftarrow \text{length}[T]$
2. $m \leftarrow \text{length}[P]$
3. for $s \leftarrow 0$ to $n - m$
   
   do if $P[1..m] = T[s + 1..s + m]$
   
   then print "Pattern occurs with shift" $s$

The naive string-matching procedure can be interpreted graphically as sliding a "template" containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in the text, as illustrated in Figure 32.4. The for loop beginning on line 3 considers each possible shift explicitly. The test on line 4 determines whether the current shift is valid or not; this test involves an implicit loop to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift $s$.

Procedure **NAIVE-STRING-MATCHER** takes time $O((n - m + 1)m)$, and this bound is tight in the worst case. For example, consider the text string $a^n$ (a string of $n$ a’s) and the pattern $a^m$. For each of the $n - m + 1$ possible values of the shift $s$, the implicit loop on line 4 to compare corresponding characters must execute $m$ times to validate the shift. The worst-case running time is thus $\Theta((n - m + 1)m)$, which is $\Theta(n^2)$ if $m = \lfloor n/2 \rfloor$. The running time of **NAIVE-STRING-MATCHER** is equal to its matching time, since there is no preprocessing.

As we shall see, **NAIVE-STRING-MATCHER** is not an optimal procedure for this problem. Indeed, in this chapter we shall show an algorithm with a worst-case preprocessing time of $\Theta(m)$ and a worst-case matching time of $\Theta(n)$. The naive string-matcher is inefficient because information gained about the text for one value of $s$ is entirely ignored in considering other values of $s$. Such information can be very valuable, however. For example, if $P = aaba$ and we find that $s = 0$ is valid, then none of the shifts 1, 2, or 3 are valid, since $T[4] = b$. In the following sections, we examine several ways to make effective use of this sort of information.

**Exercises**

**32.1-1**

Show the comparisons the naive string matcher makes for the pattern $P = 0001$ in the text $T = 000010001010001$. 
Figure 32.4 The operation of the naive string matcher for the pattern $P = aab$ and the text $T = acaaabc$. We can imagine the pattern $P$ as a "template" that we slide next to the text. (a)–(d) The four successive alignments tried by the naive string matcher. In each part, vertical lines connect corresponding regions found to match (shown shaded), and a jagged line connects the first mismatched character found, if any. One occurrence of the pattern is found, at shift $s = 2$, shown in part (c).

32.1-2
Suppose that all characters in the pattern $P$ are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time $O(n)$ on an $n$-character text $T$.

32.1-3
Suppose that pattern $P$ and text $T$ are randomly chosen strings of length $m$ and $n$, respectively, from the $d$-ary alphabet $\Sigma_d = \{0, 1, \ldots, d-1\}$, where $d \geq 2$. Show that the expected number of character-to-character comparisons made by the implicit loop in line 4 of the naive algorithm is

$$(n - m + 1) \frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1)$$

over all executions of this loop. (Assume that the naive algorithm stops comparing characters for a given shift once a mismatch is found or the entire pattern is matched.) Thus, for randomly chosen strings, the naive algorithm is quite efficient.

32.1-4
Suppose we allow the pattern $P$ to contain occurrences of a gap character $\diamond$ that can match an arbitrary string of characters (even one of zero length). For example, the pattern $ab\diamond ba\diamond c$ occurs in the text $cabcbacabacab$ as

\[
\begin{array}{ccccccc}
\hline
\text{c} & \text{ab} & \text{cc} & \text{ba} & \text{cba} & \text{c} & \text{ab} \\
\text{ab} & \diamond & \text{ba} & \diamond & \text{c} \\
\hline
\end{array}
\]

and as

\[
\begin{array}{ccccccc}
\hline
\text{c} & \text{ab} & \text{ccbac} & \text{ba} & \text{bc} & \text{c} & \text{ab} \\
\text{ab} & \diamond & \text{ba} & \diamond & \text{c} \\
\hline
\end{array}
\]

Note that the gap character may occur an arbitrary number of times in the pattern but is assumed not to occur at all in the text. Give a polynomial-time algorithm
to determine if such a pattern $P$ occurs in a given text $T$, and analyze the running time of your algorithm.

## 32.2 The Rabin-Karp algorithm

Rabin and Karp have proposed a string-matching algorithm that performs well in practice and that also generalizes to other algorithms for related problems, such as two-dimensional pattern matching. The Rabin-Karp algorithm uses $\Theta(m)$ preprocessing time, and its worst-case running time is $\Theta((n-m+1)m)$. Based on certain assumptions, however, its average-case running time is better.

This algorithm makes use of elementary number-theoretic notions such as the equivalence of two numbers modulo a third number. You may want to refer to Section 31.1 for the relevant definitions.

For expository purposes, let us assume that $\Sigma = \{0, 1, 2, \ldots, 9\}$, so that each character is a decimal digit. (In the general case, we can assume that each character is a digit in radix-$d$ notation, where $d = |\Sigma|$.) We can then view a string of $k$ consecutive characters as representing a length-$k$ decimal number. The character string 31415 thus corresponds to the decimal number 31415. Given the dual interpretation of the input characters as both graphical symbols and digits, we find it convenient in this section to denote them as we would digits, in our standard text font.

Given a pattern $P[1..m]$, we let $p$ denote its corresponding decimal value. In a similar manner, given a text $T[1..n]$, we let $t_s$ denote the decimal value of the length-$m$ substring $T[s+1..s+m]$, for $s = 0, 1, \ldots, n-m$. Certainly, $t_s = p$ if and only if $T[s+1..s+m] = P[1..m]$; thus, $s$ is a valid shift if and only if $t_s = p$. If we could compute $p$ in time $\Theta(m)$ and all the $t_s$ values in a total of $\Theta(n-m+1)$ time,\(^1\) then we could determine all valid shifts $s$ in time $\Theta(m) + \Theta(n-m+1) = \Theta(n)$ by comparing $p$ with each of the $t_s$'s. (For the moment, let's not worry about the possibility that $p$ and the $t_s$'s might be very large numbers.)

We can compute $p$ in time $\Theta(m)$ using Horner's rule (see Section 30.1):

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1]) \cdots)) .$$

The value $t_0$ can be similarly computed from $T[1..m]$ in time $\Theta(m)$.

\(^1\)We write $\Theta(n-m+1)$ instead of $\Theta(n-m)$ because there are $n-m+1$ different values that $s$ takes on. The "+1" is significant in an asymptotic sense because when $m = n$, computing the lone $t_s$ value takes $\Theta(1)$ time, not $\Theta(0)$ time.
To compute the remaining values \( t_1, t_2, \ldots, t_{n-m} \) in time \( \Theta(n-m) \), it suffices to observe that \( t_{s+1} \) can be computed from \( t_s \) in constant time, since

\[
t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1].
\]

(32.1)

For example, if \( m = 5 \) and \( t_s = 31415 \), then we wish to remove the high-order digit \( T[s+1] = 3 \) and bring in the new low-order digit (suppose it is \( T[s+5+1] = 2 \)) to obtain

\[
t_{s+1} = 10(31415 - 10000 \cdot 3) + 2
\]

\[
= 14152.
\]

Subtracting \( 10^{m-1}T[s+1] \) removes the high-order digit from \( t_s \), multiplying the result by 10 shifts the number left one position, and adding \( T[s+m+1] \) brings in the appropriate low-order digit. If the constant \( 10^{m-1} \) is precomputed (which can be done in time \( O(\lg m) \) using the techniques of Section 31.6, although for this application a straightforward \( O(m) \)-time method is quite adequate), then each execution of equation (32.1) takes a constant number of arithmetic operations. Thus, we can compute \( p \) in time \( \Theta(m) \) and compute \( t_0, t_1, \ldots , t_{n-m} \) in time \( \Theta(n-m+1) \), and we can find all occurrences of the pattern \( P[1..m] \) in the text \( T[1..n] \) with \( \Theta(m) \) preprocessing time and \( \Theta(n-m+1) \) matching time.

The only difficulty with this procedure is that \( p \) and \( t_s \) may be too large to work with conveniently. If \( P \) contains \( m \) characters, then assuming that each arithmetic operation on \( p \) (which is \( m \) digits long) takes “constant time” is unreasonable. Fortunately, there is a simple cure for this problem, as shown in Figure 32.5: compute \( p \) and the \( t_s \)'s modulo a suitable modulus \( q \). Since the computation of \( p, t_0, \) and the recurrence (32.1) can all be performed modulo \( q \), we see that we can compute \( p \) modulo \( q \) in \( \Theta(m) \) time and all the \( t_s \)'s modulo \( q \) in \( \Theta(n-m+1) \) time. The modulus \( q \) is typically chosen as a prime such that \( 10q \) just fits within one computer word, which allows all the necessary computations to be performed with single-precision arithmetic. In general, with a \( d \)-ary alphabet \( \{0, 1, \ldots, d-1\} \), we choose \( q \) so that \( dq \) fits within a computer word and adjust the recurrence equation (32.1) to work modulo \( q \), so that it becomes

\[
t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \mod q,
\]

(32.2)

where \( h \equiv d^{m-1} \mod q \) is the value of the digit “1” in the high-order position of an \( m \)-digit text window.

The solution of working modulo \( q \) is not perfect, however, since \( t_s \equiv p \mod q \) does not imply that \( t_s = p \). On the other hand, if \( t_s \not\equiv p \mod q \), then we definitely have that \( t_s \neq p \), so that shift \( s \) is invalid. We can thus use the test \( t_s \equiv p \mod q \) as a fast heuristic test to rule out invalid shifts \( s \). Any shift \( s \) for which \( t_s \equiv p \mod q \) must be tested further to see if \( s \) is really valid or we just have a **spurious hit**. This testing can be done by explicitly checking the condition
Figure 32.5  The Rabin-Karp algorithm. Each character is a decimal digit, and we compute values modulo 13.  (a) A text string. A window of length 5 is shown shaded. The numerical value of the shaded number is computed modulo 13, yielding the value 7.  (b) The same text string with values computed modulo 13 for each possible position of a length-5 window. Assuming the pattern \( P = 31415 \), we look for windows whose value modulo 13 is 7, since \( 31415 \equiv 7 \) (mod 13). Two such windows are found, shown shaded in the figure. The first, beginning at text position 7, is indeed an occurrence of the pattern, while the second, beginning at text position 13, is a spurious hit.  (c) Computing the value for a window in constant time, given the value for the previous window. The first window has value 31415. Dropping the high-order digit 3, shifting left (multiplying by 10), and then adding in the low-order digit 2 gives us the new value 14152. All computations are performed modulo 13, however, so the value for the first window is 7, and the value computed for the new window is 8.
$P[1..m] = T[s + 1..s + m]$. If $q$ is large enough, then we can hope that spurious hits occur infrequently enough that the cost of the extra checking is low.

The following procedure makes these ideas precise. The inputs to the procedure are the text $T$, the pattern $P$, the radix $d$ to use (which is typically taken to be $|\Sigma|$), and the prime $q$ to use.

RABIN-KARP-MATCHER($T, P, d, q$)

1. $n \leftarrow \text{length}[T]$  
2. $m \leftarrow \text{length}[P]$  
3. $h \leftarrow d^{m-1} \mod q$  
4. $p \leftarrow 0$  
5. $t_0 \leftarrow 0$

6. for $i \leftarrow 1$ to $m$ \hspace{1cm} \triangleright Preprocessing. 
7. do $p \leftarrow (dp + P[i]) \mod q$  
8. \hspace{1cm} $t_0 \leftarrow (dt_0 + T[i]) \mod q$

9. for $s \leftarrow 0$ to $n - m$ \hspace{1cm} \triangleright Matching. 
10. do if $p = t_s$
11. \hspace{1cm} then if $P[1..m] = T[s + 1..s + m]$  
12. \hspace{2cm} then print "Pattern occurs with shift" $s$
13. \hspace{1cm} if $s < n - m$
14. \hspace{2cm} then $t_{s+1} \leftarrow (d(t_s - T[s + 1]h) + T[s + m + 1]) \mod q$

The procedure RABIN-KARP-MATCHER works as follows. All characters are interpreted as radix-$d$ digits. The subscripts on $t$ are provided only for clarity; the program works correctly if all the subscripts are dropped. Line 3 initializes $h$ to the value of the high-order digit position of an $m$-digit window. Lines 4–8 compute $p$ as the value of $P[1..m] \mod q$ and $t_0$ as the value of $T[1..m] \mod q$. The for loop of lines 9–14 iterates through all possible shifts $s$, maintaining the following invariant:

Whenever line 10 is executed, $t_s = T[s + 1..s + m] \mod q$.

If $p = t_s$ in line 10 (a "hit"), then we check to see if $P[1..m] = T[s + 1..s + m]$ in line 11 to rule out the possibility of a spurious hit. Any valid shifts found are printed out on line 12. If $s < n - m$ (checked in line 13), then the for loop is to be executed at least one more time, and so line 14 is first executed to ensure that the loop invariant holds when line 10 is again reached. Line 14 computes the value of $t_{s+1} \mod q$ from the value of $t_s \mod q$ in constant time using equation (32.2) directly.

RABIN-KARP-MATCHER takes $\Theta(m)$ preprocessing time, and its matching time is $\Theta(n - m + 1)m$ in the worst case, since (like the naive string-matching algorithm) the Rabin-Karp algorithm explicitly verifies every valid shift. If $P = a^m$
and \( T = a^n \), then the verifications take time \( \Theta((n - m + 1)m) \), since each of the \( n - m + 1 \) possible shifts is valid.

In many applications, we expect few valid shifts (perhaps some constant \( c \) of them); in these applications, the expected matching time of the algorithm is only \( O((n - m + 1) + cm) = O(n+m) \), plus the time required to process spurious hits. We can base a heuristic analysis on the assumption that reducing values modulo \( q \) acts like a random mapping from \( \Sigma^* \) to \( \mathbb{Z}_q \). (See the discussion on the use of division for hashing in Section 11.3.1. It is difficult to formalize and prove such an assumption, although one viable approach is to assume that \( q \) is chosen randomly from integers of the appropriate size. We shall not pursue this formalization here.) We can then expect that the number of spurious hits is \( O(n/q) \), since the chance that an arbitrary \( t_i \) will be equivalent to \( p \), modulo \( q \), can be estimated as \( 1/q \). Since there are \( O(n) \) positions at which the test of line 10 fails and we spend \( O(m) \) time for each hit, the expected matching time taken by the Rabin-Karp algorithm is

\[
O(n) + O(m(v + n/q))
\]

where \( v \) is the number of valid shifts. This running time is \( O(n) \) if \( v = O(1) \) and we choose \( q \geq m \). That is, if the expected number of valid shifts is small \( O(1) \) and the prime \( q \) is chosen to be larger than the length of the pattern, then we can expect the Rabin-Karp procedure to use only \( O(n + m) \) matching time. Since \( m \leq n \), this expected matching time is \( O(n) \).

Exercises

32.2-1
Working modulo \( q = 11 \), how many spurious hits does the Rabin-Karp matcher encounter in the text \( T = 3141592653589793 \) when looking for the pattern \( P = 26 \)?

32.2-2
How would you extend the Rabin-Karp method to the problem of searching a text string for an occurrence of any one of a given set of \( k \) patterns? Start by assuming that all \( k \) patterns have the same length. Then generalize your solution to allow the patterns to have different lengths.

32.2-3
Show how to extend the Rabin Karp method to handle the problem of looking for a given \( m \times m \) pattern in an \( n \times n \) array of characters. (The pattern may be shifted vertically and horizontally, but it may not be rotated.)

32.2-4
Alice has a copy of a long \( n \)-bit file \( A = \langle a_{n-1}, a_{n-2}, \ldots, a_0 \rangle \), and Bob similarly has an \( n \)-bit file \( B = \langle b_{n-1}, b_{n-2}, \ldots, b_0 \rangle \). Alice and Bob wish to know if their
files are identical. To avoid transmitting all of $A$ or $B$, they use the following fast probabilistic check. Together, they select a prime $q > 1000n$ and randomly select an integer $x$ from $\{0, 1, \ldots, q - 1\}$. Then, Alice evaluates

$$A(x) = \left( \sum_{i=0}^{n-1} a_i x^i \right) \mod q$$

and Bob similarly evaluates $B(x)$. Prove that if $A \neq B$, there is at most one chance in 1000 that $A(x) = B(x)$, whereas if the two files are the same, $A(x)$ is necessarily the same as $B(x)$. (Hint: See Exercise 31.4-4.)

### 32.3 String matching with finite automata

Many string-matching algorithms build a finite automaton that scans the text string $T$ for all occurrences of the pattern $P$. This section presents a method for building such an automaton. These string-matching automata are very efficient: they examine each text character exactly once, taking constant time per text character. The matching time used—after preprocessing the pattern to build the automaton—is therefore $\Theta(n)$. The time to build the automaton, however, can be large if $\Sigma$ is large. Section 32.4 describes a clever way around this problem.

We begin this section with the definition of a finite automaton. We then examine a special string-matching automaton and show how it can be used to find occurrences of a pattern in a text. This discussion includes details on how to simulate the behavior of a string-matching automaton on a given text. Finally, we shall show how to construct the string-matching automaton for a given input pattern.

#### Finite automata

A **finite automaton** $M$ is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- $Q$ is a finite set of **states**,  
- $q_0 \in Q$ is the **start state**,  
- $A \subseteq Q$ is a distinguished set of **accepting states**,  
- $\Sigma$ is a finite **input alphabet**,  
- $\delta$ is a function from $Q \times \Sigma$ into $Q$, called the **transition function** of $M$.

The finite automaton begins in state $q_0$ and reads the characters of its input string one at a time. If the automaton is in state $q$ and reads input character $a$, it moves ("makes a transition") from state $q$ to state $\delta(q, a)$. Whenever its current state $q$ is a member of $A$, the machine $M$ is said to have **accepted** the string read so far.