9.1 Strings and Pattern Matching Algorithms

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

9.1.1 String Operations

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:

P = "CGTAAACTGCTTTAATCAAACGC"

R = "U.S. Men Win Soccer World Cup!"

S = "http://www.wiley.com/college/goodrich/".

The first string, P, comes from DNA applications, the last string, S, is the Internet address (URL) for the Web site that accompanies this book, and the middle string, R, is a fictional news headline. In this section, we present some of the useful operations that are supported by the string ADT for processing strings such as these.

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

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

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

The Pattern Matching Problem

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

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

That is,

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

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

To allow for fairly general notions of a character string, we typically do restrict the characters in T and P to come explicitly from a well-known charac set, like the ASCII or Unicode character sets. Instead, we typically use the gene symbol Σ to denote the character set, or ***alphabet***, from which the characters T and P can come. This alphabet Σ can, of course, be a subset of the ASCII Unicode character sets, but it could also be more general and is even allowed be infinite. Nevertheless, since most document processing algorithms are used applications where the underlying character set is finite, we usually assume that size of the alphabet Σ, denoted with $|\Sigma|$, is a fixed constant.

Example 9.1: *Suppose we are given the text string*

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

and the pattern string

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

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

In this section, we present three different pattern matching algorithms.

9.1.2 Brute Force Pattern Matching

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.

Brute-Force Pattern Matching

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 approach, shown in Algorithm 9.1, 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."
```

Algorithm 9.1: Brute-force pattern matching.

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.

The running time of brute-force pattern matching in the worst case is not good, however, because, for each candidate index in T, we can perform up to m character comparisons to discover that P does not match T at the current index. Referring to Algorithm 9.1, we see that the outer for-loop is executed at most $n-m+1$ times, and the inner loop is executed at most m times. Thus, the running time of the brute-force method is $O((n-m+1)m)$, which is $O(nm)$. Thus, in the worst case, when n and m are roughly equal, this algorithm has a quadratic running time.

In Figure 9.2 we illustrate the execution of the brute-force pattern matching algorithm on the strings T and P from Example 9.1.

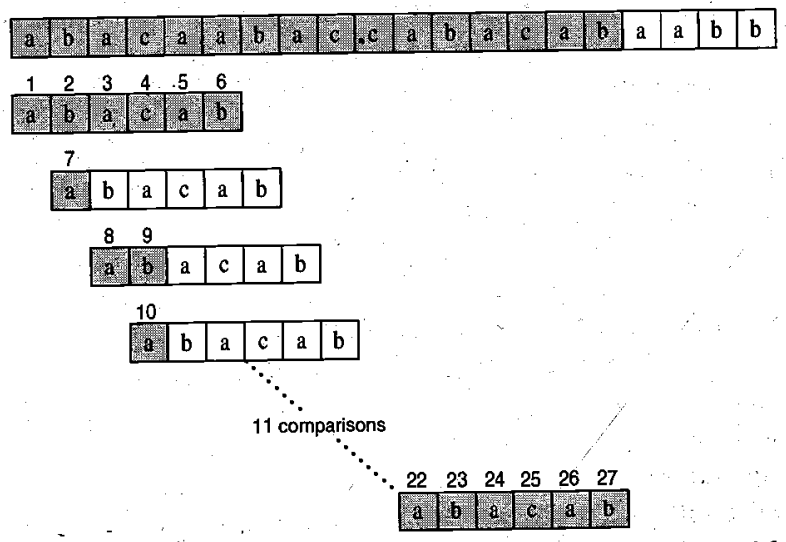


Figure 9.2: Example run of the brute-force pattern matching algorithm. The algorithm performs 27 character comparisons, indicated above with numerical labels.

9.1.3 The Boyer-Moore Algorithm

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

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

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

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

We will formalize these heuristics shortly, but at an intuitive level, they work as an integrated team. The looking-glass heuristic sets up the other heuristic to allow us to avoid comparisons between P and whole groups of characters in T. In this case at least, we can get to the destination faster by going 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.

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

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

If characters can be used as indices in arrays, then the last function can be easily implemented as a lookup table. We leave the method for computing this table efficiently as a simple exercise (R-9.6). The last function will give us all the information we need to perform the character-jump heuristic. In Algorithm 9.3, we show the BM pattern matching method. The jump step is illustrated in Figure 9.4.

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

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

Algorithm 9.3: The Boyer-Moore pattern matching algorithm.

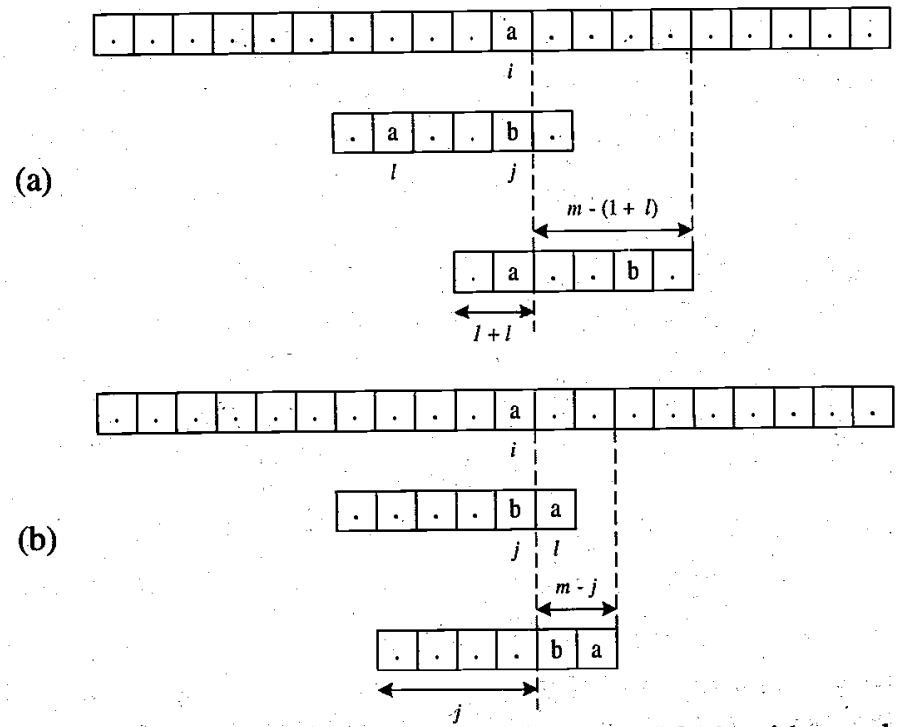
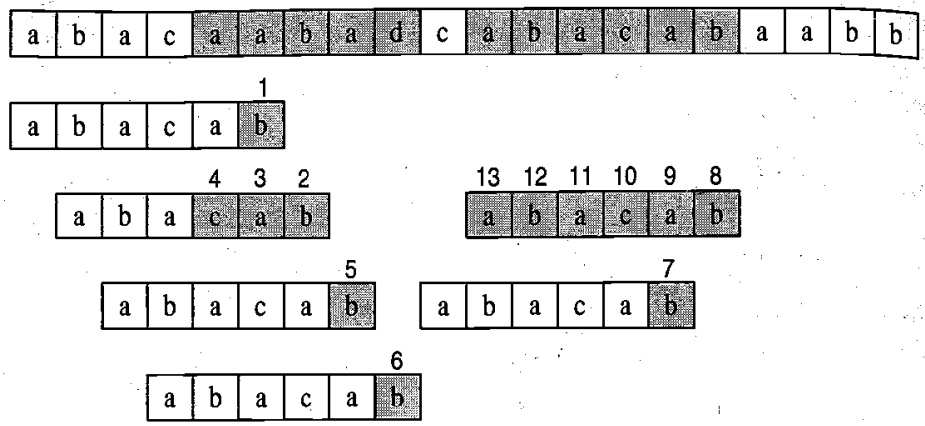


Figure 9.4: Illustration of the jump step in the BM algorithm, where l denotes last($T[i]$). We distinguish two cases: (a) $1+l \leq j$, where we shift the pattern by $j-l$ units; (b) $j < 1+l$, where we shift the pattern by one unit.

In Figure 9.5, we illustrate the execution of the Boyer-Moore pattern matchi algorithm on a similar input string as in Example 9.1.



The last(c) function:

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

Figure 9.5: An illustration of 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. For last(c) is the location of the *last* occurrence of c in P.

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

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

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

Indeed, the BM algorithm is often able to skip over large portions of the text. (See Figure 9.6.) There is experimental evidence that on English text, the average number of comparisons done per text character is approximately 0.24 for a five-character pattern string. The payoff is not as great for binary strings or for very short patterns, however, in which case the KMP algorithm, discussed in Section 9.1.4, or for very short patterns, the brute-force algorithm, may be better.

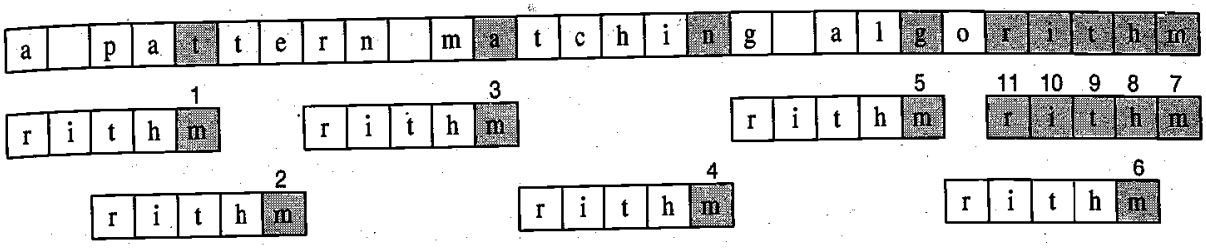


Figure 9.6: Execution of the Boyer-Moore algorithm on an English text and pattern, where a significant speedup is achieved. Note that not all text characters are examined.

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.

9.1.4 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 9.1, we should notice a major inefficiency. Specifically, we may perform many comparisons while testing a potential placement of the pattern against the text, yet if we 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 will discuss how to compute the failure function efficiently. The importance of this failure function is that it "encodes" repeated substrings inside the pattern itself.

Example 9.2: *Consider the pattern string* $P =$ "abacab" *from Example 9.* *KMP failure function* $f(j)$ *for the string* P *is as shown in the following tabl*

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

The KMP pattern matching algorithm, shown in Algorithm 9.7, increme processes the text string T comparing it to the pattern string P. Each time the match, we increment the current indices. On the other hand, if there is a mism and we have previously made progress in P, then we consult the failure func to determine the new index in P where we need to continue checking P agains Otherwise (there was a mismatch and we are at the beginning of P), we sim increment the index for T (and keep the index variable for P at its beginning). repeat this process until we find a match of P in T or the index for T reaches n, length of T (indicating that we did not find the pattern P in T).

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

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 indicatio that P is not a substring of T

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

Algorithm 9.7: The KMP pattern matching algorithm.

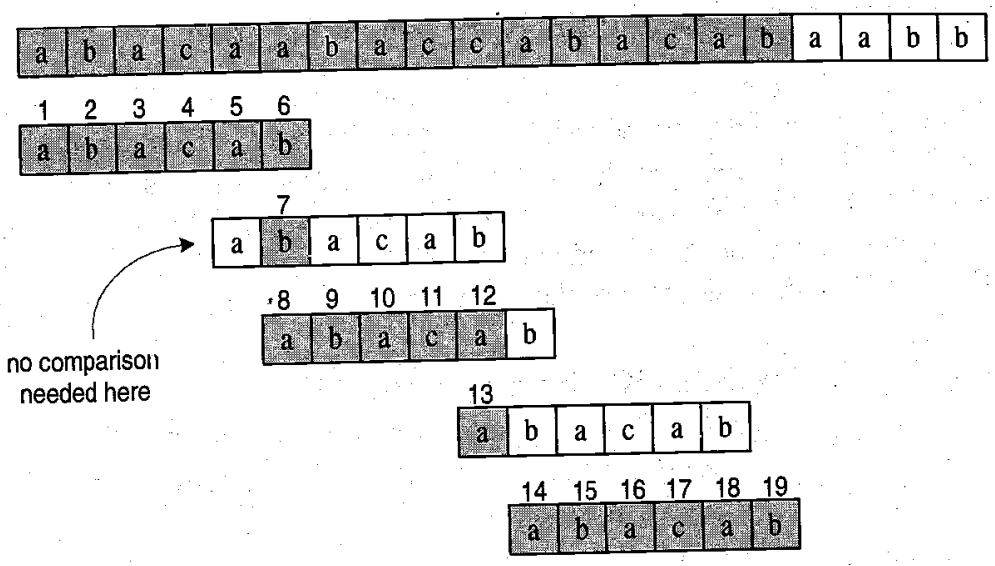


Figure 9.8: An illustration of the KMP pattern matching algorithm. The failure function f for this pattern is given in Example 9.2. The algorithm performs 19 character comparisons, which are indicated with numerical labels.

In Figure 9.8, we illustrate the execution of the KMP pattern matching algorithm on the same input strings as in Example 9.1. Note the use of the failure function to avoid redoing one of the comparisons between a character of the pattern and a character of the text. Also note that the algorithm performs fewer overall comparisons than the brute-force algorithm run on the same strings (Figure 9.2).

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 the analysis, let us define $k = i - j$. Intuitively, k is the total amount by which the pattern P has been shifted with respect to the text T. Note that throughout the execution of the algorithm, we have $k \leq n$. One of the following three cases occurs at each iteration of the loop.

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

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

Constructing the KMP Failure Function

To construct the failure function used in the KMP pattern matching algorith use the method shown in Algorithm 9.9. This algorithm is another examp "bootstrapping" process quite similar to that used in the KMPMatch alg We compare the pattern to itself as in the KMP algorithm. Each time we ha characters that match, we set $f(i) = j+1$. Note that since we have $i > j$ thro the execution of the algorithm, $f(j-1)$ is always defined when we need to u

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 lo prefix of P that is a suffix of $P[1..j]$

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

Algorithm 9.9: Computation of the failure function used in the KMP pattern matching algorithm. Note how the algorithm uses the previous values of the failure function to efficiently compute new values.

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

Theorem 9.3: *The Knuth-Morris-Pratt algorithm performs pattern matching on a text string of length n and a pattern string of length m in $O(n+m)$ time.*

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