String Algorithms

14

This chapter deals with topics related to string processing.

Section 14.1 presents the trie structure which maintains a set of strings. After this, dynamic programming algorithms for determining longest common subsequences and edit distances are discussed.

Section 14.2 discusses the string hashing technique which is a general tool for creating efficient string algorithms. The idea is to compare hash values of strings instead of their characters, which allows us to compare strings in constant time.

Section 14.3 introduces the Z-algorithm which determines for each string position the longest substring which is also a prefix of the string. The Z-algorithm is an alternative for many string problems that can also be solved using hashing.

Section 14.4 discusses the suffix array structure, which can be used to solve some more advanced string problems.

14.1 Basic Topics

Throughout the chapter, we assume that all strings are zero indexed. For example, a string $\mathtt{s}$ of length n consists of characters $\mathtt{s}[0]$, $\mathtt{s}[1]$, ..., $\mathtt{s}[n-1]$.

A *substring* is a sequence of consecutive characters in a string. We use the notation $\mathtt{s}[a \ldots b]$ to refer to a substring of $\mathtt{s}$ that starts at position a and ends at position b. A *prefix* is a substring that contains the first character of a string, and a *suffix* is a substring that contains the last character of a string.

A *subsequence* is any sequence of characters in a string in their original order. All substrings are subsequences, but the converse is not true (Fig. 14.1).

A. Laaksonen, *Guide to Competitive Programming*, Undergraduate
Topics in Computer Science, https://doi.org/10.1007/978-3-319-72547-5_14

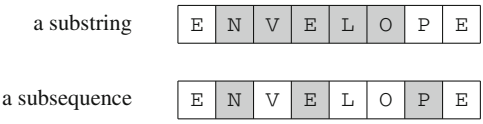

Fig. 14.1 NVELO is a substring, NEP is a subsequence

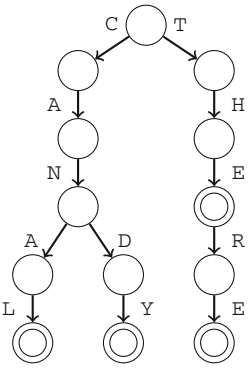

Fig. 14.2 A trie that contains the strings CANAL, CANDY, THE, and THERE

14.1.1 Trie Structure

A *trie* is a rooted tree that maintains a set of strings. Each string in the set is stored as a character chain that starts at the root node. If two strings have a common prefix, they also have a common chain in the tree. As an example, the trie in Fig. 14.2 corresponds to the set {CANAL, CANDY, THE, THERE}. A circle in a node means that a string in the set ends at the node.

After constructing a trie, we can easily check whether it contains a given string by following the chain that starts at the root node. We can also add a new string to the trie by first following the chain and then adding new nodes if necessary. Both the operations work in $O(n)$ time where n is the length of the string.

A trie can be stored in an array

```
int trie[N][A];
```

where N is the maximum number of nodes (the maximum total length of the strings in the set) and A is the size of the alphabet. The trie nodes are numbered 0, 1, 2, ... in such a way that the number of the root is 0, and `trie`$[s][c]$ specifies the next node in the chain when we move from node s using character c.

There are several ways how we can extend the trie structure. For example, suppose that we are given queries that require us to calculate the number of strings in the set that have a certain prefix. We can do this efficiently by storing for each trie node the number of strings whose chain goes through the node.

		O	P	E	R	A
	T	0	0	0	0	0
	O	1	1	1	1	1
	U	1	1	1	1	1
	R	1	1	1	2	2

Fig. 14.3 The values of the `lcs` function for determining the longest common subsequence of `TOUR` and `OPERA`

14.1.2 Dynamic Programming

Dynamic programming can be used to solve many string problems. Next we will discuss two examples of such problems.

Longest Common Subsequence The *longest common subsequence* of two strings is the longest string that appears as a subsequence in both strings. For example, the longest common subsequence of `TOUR` and `OPERA` is `OR`.

Using dynamic programming, we can determine the longest common subsequence of two strings `x` and `y` in $O(nm)$ time, where n and m denote the lengths of the strings. To do this, we define a function $\texttt{lcs}(i, j)$ that gives the length of the longest common subsequence of the prefixes $\texttt{x}[0 \ldots i]$ and $\texttt{y}[0 \ldots j]$. Then, we can use the recurrence

$$\texttt{lcs}(i, j) = \begin{cases} \texttt{lcs}(i-1, j-1) + 1 & \texttt{x}[i] = \texttt{y}[j] \\ \max(\texttt{lcs}(i, j-1), \texttt{lcs}(i-1, j)) & \text{otherwise.} \end{cases}$$

The idea is that if characters $\texttt{x}[i]$ and $\texttt{y}[j]$ are equal, we match them and increase the length of the longest common subsequence by one. Otherwise, we remove the last character from either `x` or `y`, depending on which choice is optimal.

For example, Fig. 14.3 shows the values of the `lcs` function in our example scenario.

Edit Distances The *edit distance* (or *Levenshtein distance*) between two strings denotes the minimum number of editing operations that transform the first string into the second string. The allowed editing operations are as follows:

- insert a character (e.g., `ABC` → `ABCA`)
- remove a character (e.g., `ABC` → `AC`)
- modify a character (e.g., `ABC` → `ADC`)

For example, the edit distance between `LOVE` and `MOVIE` is 2, because we can first perform the operation `LOVE` → `MOVE` (modify) and then the operation `MOVE` → `MOVIE` (insert).

We can calculate the edit distance between two strings `x` and `y` in $O(nm)$ time, where n and m are the lengths of the strings. Let $\texttt{edit}(i, j)$ denote the edit distance between the prefixes $\texttt{x}[0 \ldots i]$ and $\texttt{y}[0 \ldots j]$. The values of the function can be calculated using the recurrence

Fig. 14.4 The values of the `edit` function for determining the edit distance between `LOVE` and `MOVIE`

		M	O	V	I	E
L	1	2	3	4	5
O	2	1	2	3	4
V	3	2	1	2	3
E	4	3	2	2	2

$$\begin{aligned}\texttt{edit}(a,b) = \min(&\texttt{edit}(a,b-1)+1,\\ &\texttt{edit}(a-1,b)+1,\\ &\texttt{edit}(a-1,b-1)+\texttt{cost}(a,b)),\end{aligned}$$

where $\texttt{cost}(a,b) = 0$ if $\texttt{x}[a] = \texttt{y}[b]$, and otherwise $\texttt{cost}(a,b) = 1$. The formula considers three ways to edit the string `x`: insert a character at the end of `x`, remove the last character from `x`, or match/modify the last character of `x`. In the last case, if $\texttt{x}[a] = \texttt{y}[b]$, we can match the last characters without editing.

For example, Fig. 14.4 shows the values of the `edit` function in our example scenario.

14.2 String Hashing

Using *string hashing* we can efficiently check whether two strings are equal by comparing their hash values. A *hash value* is an integer that is calculated from the characters of the string. If two strings are equal, their hash values are also equal, which makes it possible to compare strings based on their hash values.

14.2.1 Polynomial Hashing

A usual way to implement string hashing is *polynomial hashing*, which means that the hash value of a string `s` of length n is

$$(\texttt{s}[0]A^{n-1} + \texttt{s}[1]A^{n-2} + \cdots + \texttt{s}[n-1]A^0) \bmod B,$$

where $s[0], s[1], \ldots, s[n-1]$ are interpreted as character codes, and A and B are prechosen constants.

For example, let us calculate the hash value of the string `ABACB`. The character codes of `A`, `B`, and `C` are 65, 66, and 67. Then, we need to fix the constants; suppose that $A = 3$ and $B = 97$. Thus, the hash value is

$$(65 \cdot 3^4 + 66 \cdot 3^3 + 65 \cdot 3^2 + 66 \cdot 3^1 + 67 \cdot 3^0) \bmod 97 = 40.$$

When polynomial hashing is used, we can calculate the hash value of any substring of a string s in $O(1)$ time after an $O(n)$ time preprocessing. The idea is to construct an array h such that h[k] contains the hash value of the prefix s[$0 \ldots k$]. The array values can be recursively calculated as follows:

$$\begin{aligned} \texttt{h}[0] &= \texttt{s}[0] \\ \texttt{h}[k] &= (\texttt{h}[k-1]A + \texttt{s}[k]) \bmod B \end{aligned}$$

In addition, we construct an array p where $\texttt{p}[k] = A^k \bmod B$:

$$\begin{aligned} \texttt{p}[0] &= 1 \\ \texttt{p}[k] &= (\texttt{p}[k-1]A) \bmod B. \end{aligned}$$

Constructing the above arrays takes $O(n)$ time. After this, the hash value of any substring s[$a \ldots b$] can be calculated in $O(1)$ time using the formula

$$(\texttt{h}[b] - \texttt{h}[a-1]\texttt{p}[b-a+1]) \bmod B$$

assuming that $a > 0$. If $a = 0$, the hash value is simply h[b].

14.2.2 Applications

We can efficiently solve many string problems using hashing, because it allows us to compare arbitrary substrings of strings in $O(1)$ time. In fact, we can often simply take a brute force algorithm and make it efficient by using hashing.

Pattern Matching A fundamental string problem is the *pattern matching* problem: given a string s and a pattern p, find the positions where p occurs in s. For example, the pattern ABC occurs at positions 0 and 5 in the string ABCABABCA (Fig. 14.5).

We can solve the pattern matching problem in $O(n^2)$ time using a brute force algorithm that goes through all positions where p may occur in s and compares strings character by character. Then, we can make the brute force algorithm efficient using hashing, because each comparison of strings then only takes $O(1)$ time. This results in an $O(n)$ time algorithm.

Distinct Substrings Consider the problem of counting the number of *distinct* substrings of length k in a string. For example, the string ABABAB has two distinct substrings of length 3: ABA and BAB. Using hashing, we can calculate the hash value of each substring and reduce the problem to counting the number of distinct integers in a list, which can be done in $O(n \log n)$ time.

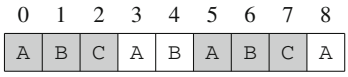

Fig. 14.5 The pattern ABC appears two times in the string ABCABABCA

Minimal Rotation A *rotation* of a string can be created by repeatedly moving the first character of the string to the end of the string. For example, the rotations of ATLAS are ATLAS, TLASA, LASAT, ASATL, and SATLA. Next we will consider the problem of finding the lexicographically *minimal* rotation of a string. For example, the minimal rotation of ATLAS is ASATL.

We can efficiently solve the problem by combining string hashing and *binary search*. The key idea is that we can find out the lexicographic order of two strings in logarithmic time. First, we calculate the length of the common prefix of the strings using binary search. Here hashing allows us to check in $O(1)$ time whether two prefixes of a certain length match. After this, we check the next character after the common prefix, which determines the order of the strings.

Then, to solve the problem, we construct a string that contains two copies of the original string (e.g., ATLASATLAS) and go through its substrings of length n maintaining the minimal substring. Since each comparison can be done in $O(\log n)$ time, the algorithm works in $O(n \log n)$ time.

14.2.3 Collisions and Parameters

An evident risk when comparing hash values is a *collision*, which means that two strings have different contents but equal hash values. In this case, an algorithm that relies on the hash values concludes that the strings are equal, but in reality they are not, and the algorithm may give incorrect results.

Collisions are always possible, because the number of different strings is larger than the number of different hash values. However, the probability of a collision is small if the constants A and B are carefully chosen. A usual way is to choose random constants near 10^9, for example, as follows:

$$A = 911382323$$
$$B = 972663749$$

Using such constants, the `long long` type can be used when calculating hash values, because the products AB and BB will fit in `long long`. But is it enough to have about 10^9 different hash values?

Let us consider three scenarios where hashing can be used:

Scenario 1: Strings x and y are compared with each other. The probability of a collision is $1/B$ assuming that all hash values are equally probable.

Scenario 2: A string x is compared with strings $y_1, y_2, \ldots, y_n$. The probability of one or more collisions is

$$1 - (1 - 1/B)^n.$$

Scenario 3: All pairs of strings $x_1, x_2, \ldots, x_n$ are compared with each other. The probability of one or more collisions is

$$1 - \frac{B \cdot (B-1) \cdot (B-2) \cdots (B-n+1)}{B^n}.$$

Table 14.1 Collision probabilities in hashing scenarios when $n = 10^6$

Constant B	Scenario 1	Scenario 2	Scenario 3
10^3	0.00	1.00	1.00
10^6	0.00	0.63	1.00
10^9	0.00	0.00	1.00
10^{12}	0.00	0.00	0.39
10^{15}	0.00	0.00	0.00
10^{18}	0.00	0.00	0.00

Table 14.1 shows the collision probabilities for different values of B when $n = 10^6$. The table shows that in Scenarios 1 and 2, the probability of a collision is negligible when $B \approx 10^9$. However, in Scenario 3 the situation is very different: a collision will almost always happen when $B \approx 10^9$.

The phenomenon in Scenario 3 is known as the *birthday paradox*: if there are n people in a room, the probability that *some* two people have the same birthday is large even if n is quite small. In hashing, correspondingly, when all hash values are compared with each other, the probability that some two hash values are equal is large.

We can make the probability of a collision smaller by calculating *multiple* hash values using different parameters. It is unlikely that a collision would occur in all hash values at the same time. For example, two hash values with parameter $B \approx 10^9$ correspond to one hash value with parameter $B \approx 10^{18}$, which makes the probability of a collision very small.

Some people use constants $B = 2^{32}$ and $B = 2^{64}$, which is convenient, because operations with 32- and 64-bit integers are calculated modulo 2^{32} and 2^{64}. However, this is *not* a good choice, because it is possible to construct inputs that always generate collisions when constants of the form 2^x are used [23].

14.3 Z-Algorithm

The *Z-array* $\texttt{z}$ of a string $\texttt{s}$ of length n contains for each $k = 0, 1, \ldots, n-1$ the length of the longest substring of $\texttt{s}$ that begins at position k and is a prefix of $\texttt{s}$. Thus, $\texttt{z}[k] = p$ tells us that $\texttt{s}[0 \ldots p-1]$ equals $\texttt{s}[k \ldots k+p-1]$, but $\texttt{s}[p]$ and $\texttt{s}[k+p]$ are different characters (or the length of the string is $k+p$).

For example, Fig. 14.6 shows the Z-array of ABCABCABAB. In the array, for example, $\texttt{z}[3] = 5$, because the substring ABCAB of length 5 is a prefix of $\texttt{s}$, but the substring ABCABA of length 6 is not a prefix of $\texttt{s}$.

Fig. 14.6 The Z-array of ABCABCABAB

0	1	2	3	4	5	6	7	8	9
A	B	C	A	B	C	A	B	A	B
–	0	0	5	0	0	2	0	2	0

Fig. 14.7 Scenario 1: Calculating the value of z[3]

0	1	2	3	4	5	6	7	8	9
A	B	C	A	B	C	A	B	A	B
–	0	0	?	?	?	?	?	?	?

			x				y		
0	1	2	3	4	5	6	7	8	9
A	B	C	A	B	C	A	B	A	B
–	0	0	5	?	?	?	?	?	?

14.3.1 Constructing the Z-Array

Next we describe an algorithm, called the *Z-algorithm* which efficiently constructs the Z-array in $O(n)$ time.[1] The algorithm calculates the Z-array values from left to right by both using information already stored in the array and by comparing substrings character by character.

To efficiently calculate the Z-array values, the algorithm maintains a range $[x, y]$ such that $\mathtt{s}[x \ldots y]$ is a prefix of $\mathtt{s}$, the value of $\mathtt{z}[x]$ has been determined, and y is as large as possible. Since we know that $\mathtt{s}[0 \ldots y - x]$ and $\mathtt{s}[x \ldots y]$ are equal, we can use this information when calculating subsequent array values. Suppose that we have calculated the values of $\mathtt{z}[0]$, $\mathtt{z}[1]$, $\ldots$, $\mathtt{z}[k-1]$ and we want to calculate the value of $\mathtt{z}[k]$. There are three possible scenarios:

Scenario 1: $y < k$. In this case, we do not have information about the position k, so we calculate the value of $\mathtt{z}[k]$ by comparing substrings character by character. For example, in Fig. 14.7, there is no $[x, y]$ range yet, so we compare the substrings starting at positions 0 and 3 character by character. Since $\mathtt{z}[3] = 5$, the new $[x, y]$ range becomes $[3, 7]$.

Scenario 2: $y \geq k$ and $k + \mathtt{z}[k - x] \leq y$. In this case we know that $\mathtt{z}[k] = \mathtt{z}[k - x]$, because $\mathtt{s}[0 \ldots y - x]$ and $\mathtt{s}[x \ldots y]$ are equal and we stay inside the $[x, y]$ range. For example, in Fig. 14.8, we conclude that $\mathtt{z}[4] = \mathtt{z}[1] = 0$.

Scenario 3: $y \geq k$ and $k + \mathtt{z}[k - x] > y$. In this case we know that $\mathtt{z}[k] \geq y - k + 1$. However, since we do not have information after the position y, we have to compare substrings character by character starting at positions $y - k + 1$ and $y + 1$. For example, in Fig. 14.9, we know that $\mathtt{z}[6] \geq 2$. Then, since $\mathtt{s}[2] \neq \mathtt{s}[8]$, it turns out that, in fact, $\mathtt{z}[6] = 2$.

[1] Gusfield [13] presents the Z-algorithm as the simplest known method for linear-time pattern matching and attributes the original idea to Main and Lorentz [22].

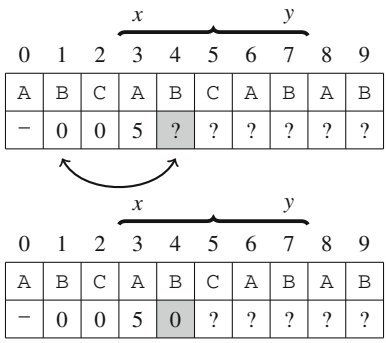

Fig. 14.8 Scenario 2: Calculating the value of z[4]

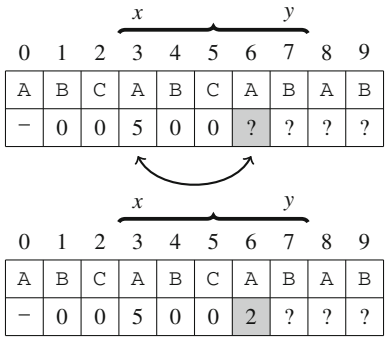

Fig. 14.9 Scenario 3: Calculating the value of z[6]

The resulting algorithm works in $O(n)$ time, because always when two characters match when comparing substrings character by character, the value of y increases. Thus, the total work needed for comparing substrings is only $O(n)$.

14.3.2 Applications

The Z-algorithm provides an alternative way to solve many string problems that can be also solved using hashing. However, unlike hashing, the Z-algorithm always works and there is no risk of collisions. In practice, it is often a matter of taste whether to use hashing or the Z-algorithm.

Pattern Matching Consider again the pattern matching problem, where our task is to find the occurrences of a pattern p in a string s. We already solved the problem using hashing, but now we will see how the Z-algorithm handles the problem.

A recurrent idea in string processing is to construct a string that consists of multiple individual parts separated by special characters. In this problem, we can construct a string $p\#s$, where p and s are separated by a special character # that does not occur in the strings. Then, the Z-array of $p\#s$ tells us the positions where p occurs in s, because such positions contain the length of p.

Fig. 14.10 Pattern matching using the Z-algorithm

0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	#	A	B	C	A	B	A	B	C	A
–	0	0	0	3	0	0	2	0	3	0	0	1

Fig. 14.11 Finding borders using the Z-algorithm

0	1	2	3	4	5	6	7	8	9	10
A	B	A	C	A	B	A	C	A	B	A
–	0	1	0	7	0	1	0	3	0	1

Fig. 14.12 The suffix array of the string ABAACBAB

0	1	2	3	4	5	6	7
2	6	0	3	7	1	5	4

Fig. 14.13 Another way to represent the suffix array

0	2	AACBAB
1	6	AB
2	0	ABAACBAB
3	3	ACBAB
4	7	B
5	1	BAACBAB
6	5	BAB
7	4	CBAB

Figure 14.10 shows the Z-array for $s =$ABCABABCA and $p =$ABC. Positions 4 and 9 contain the value 3, which means that p occurs in positions 0 and 5 in s.

Finding Borders A *border* is a string that is both a prefix and a suffix of a string, but not the entire string. For example, the borders of ABACABACABA are A, ABA, and ABACABA. All borders of a string can be efficiently found using the Z-algorithm, because a suffix at position k is a border exactly when $k + z[k] = n$ where n is the length of the string. For example, in Fig. 14.11, $4 + \text{z}[4] = 11$, which means that ABACABA is a border of the string.

14.4 Suffix Arrays

The *suffix array* of a string describes the lexicographic order of its suffixes. Each value in the suffix array is a starting position of a suffix. For example, Fig. 14.12 shows the suffix array of the string ABAACBAB.

It is often convenient to represent the suffix array vertically and also show the corresponding suffixes (Fig. 14.13). However, note that the suffix array itself only contains the starting positions of the suffixes and not their characters.

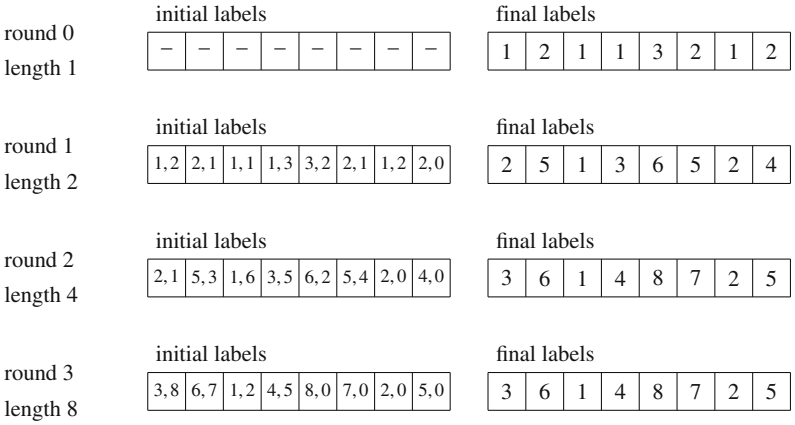

Fig. 14.14 Constructing the labels for the string ABAACBAB

14.4.1 Prefix Doubling Method

A simple and efficient way to create the suffix array of a string is to use a prefix doubling construction, which works in $O(n \log^2 n)$ or $O(n \log n)$ time, depending on the implementation.[2] The algorithm consists of rounds numbered $0, 1, \ldots, \lceil \log_2 n \rceil$, and round i goes through substrings whose length is 2^i. During a round, each substring x of length 2^i is given an integer label $l(x)$ such that $l(a) = l(b)$ exactly when $a = b$ and $l(a) < l(b)$ exactly when $a < b$.

On round 0, each substring consists of only one character, and we can, for example, use labels A $= 1$, B $= 2$, and so on. Then, on round i, where $i > 0$, we use the labels for substrings of length 2^{i-1} to construct labels for substrings of length 2^i. To give a label $l(x)$ for a substring x of length 2^i, we divide x into two halves a and b of length 2^{i-1} whose labels are $l(a)$ and $l(b)$. (If the second half begins outside the string, we assume that its label is 0.) First, we give x an *initial* label that is a pair $(l(a), l(b))$. Then, after all substrings of length 2^i have been given initial labels, we sort the initial labels and give *final* labels that are consecutive integers 1, 2, 3, etc. The purpose of giving the labels is that after the last round, each substring has a *unique* label, and the labels show the lexicographic order of the substrings. Then, we can easily construct the suffix array based on the labels.

Figure 14.14 shows the construction of the labels for ABAACBAB. For example, after round 1, we know that l(AB) $= 2$ and l(AA) $= 1$. Then, on round 2, the initial label for ABAA is (2, 1). Since there are two smaller initial labels ((1, 6) and (2, 0)), the final label is l(ABAA) $= 3$. Note that in this example, each label is unique already

[2] The idea of prefix doubling is due to Karp, Miller, and Rosenberg [17]. There are also more advanced $O(n)$ time algorithms for constructing suffix arrays; Kärkkäinen and Sanders [16] provide a quite simple such algorithm.

0	2	AACBAB	0	2	AACBAB	0	2	AACBAB
1	6	AB	1	6	AB	1	6	AB
2	0	ABAACBAB	2	0	ABAACBAB	2	0	ABAACBAB
3	3	ACBAB	3	3	ACBAB	3	3	ACBAB
4	7	B	4	7	**B**	4	7	B
5	1	BAACBAB	5	1	**B**AACBAB	5	1	**BA**ACBAB
6	5	BAB	6	5	**B**AB	6	5	**BA**B
7	4	CBAB	7	4	CBAB	7	4	CBAB

Fig. 14.15 Finding the occurrences of BA in ABAACBAB using a suffix array

after round 2, because the first four characters of the substrings completely determine their lexicographical order.

The resulting algorithm works in $O(n \log^2 n)$ time, because there are $O(\log n)$ rounds and we sort a list of n pairs on each round. In fact, an $O(n \log n)$ implementation is also possible, because we can use a linear-time sorting algorithm to sort the pairs. Still, a straightforward $O(n \log^2 n)$ time implementation just using the C++ `sort` function is usually efficient enough.

14.4.2 Finding Patterns

After constructing the suffix array, we can efficiently find the occurrences of any given pattern in the string. This can be done in $O(k \log n)$ time, where n is the length of the string and k is the length of the pattern. The idea is to process the pattern character by character and maintain a range in the suffix array that corresponds to the prefix of the pattern processed so far. Using binary search, we can efficiently update the range after each new character.

For example, consider finding the occurrences of the pattern BA in the string ABAACBAB (Fig. 14.15). First, our search range is [0, 7], which spans the entire suffix array. Then, after processing the character B, the range becomes [4, 6]. Finally, after processing the character A, the range becomes [5, 6]. Thus, we conclude that BA has two occurrences in ABAACBAB in positions 1 and 5.

Compared to string hashing and the Z-algorithm discussed earlier, the advantage of the suffix array is that we can efficiently process *several* queries that are related to different patterns, and it is not necessary to know the patterns beforehand when constructing the suffix array.

14.4.3 LCP Arrays

The *LCP array* of a string gives for its each suffix a *LCP value*: the length of the *longest common prefix* of the suffix and the next suffix in the suffix array. Figure 14.16

0	1	AACBAB
1	2	AB
2	1	ABAACBAB
3	0	ACBAB
4	1	B
5	2	BAACBAB
6	0	BAB
7	–	CBAB

Fig. 14.16 The LCP array of the string ABAACBAB

shows the LCP array for the string ABAACBAB. For example, the LCP value of the suffix BAACBAB is 2, because the longest common prefix of BAACBAB and BAB is BA. Note that the last suffix in the suffix array does not have a LCP value.

Next we present an efficient algorithm, due to Kasai et al. [18], for constructing the LCP array of a string, provided that we have already constructed its suffix array. The algorithm is based on the following observation: Consider a suffix whose LCP value is x. If we remove the first character from the suffix and get another suffix, we immediately know that its LCP value has to be at least $x - 1$. For example, in Fig. 14.16, the LCP value of the suffix BAACBAB is 2, so we know that the LCP value of the suffix AACBAB has to be at least 1. In fact, it happens to be exactly 1.

We can use the above observation to efficiently construct the LCP array by calculating the LCP values in decreasing order of suffix length. At each suffix, we calculate its LCP value by comparing the suffix and the next suffix in the suffix array character by character. Now we can use the fact that we know the LCP value of the suffix that has one more character. Thus, the current LCP value has to be at least $x - 1$, where x is the previous LCP value, and we do not need to compare the first $x - 1$ characters of the suffixes. The resulting algorithm works in $O(n)$ time, because only $O(n)$ comparisons are done during the algorithm.

Using the LCP array, we can efficiently solve some advanced string problems. For example, to calculate the number of distinct substrings in a string, we can simply subtract the sum of all values in the LCP array from the total number of substrings, i.e., the answer to the problem is

$$\frac{n(n+1)}{2} - c,$$

where n is the length of the string and c is the sum of all values in the LCP array. For example, the string ABAACBAB has

$$\frac{8 \cdot 9}{2} - 7 = 29$$

distinct substrings.