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 $\texttt{h}[k]$ contains the hash value of the prefix $\texttt{s}[0\ldots k]$. 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 $\texttt{s}[a\ldots b]$ 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 $\texttt{h}[b]$.

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.

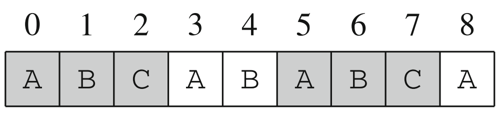

Fig. 14.5 The pattern ABC appears two times in the string ABCABABCA

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:

$$\begin{aligned} A &= 911382323 \\ B &= 972663749 \end{aligned}$$

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* `z` of a string `s` of length n contains for each $k = 0, 1, \ldots, n-1$ the length of the longest substring of `s` that begins at position k and is a prefix of `s`. Thus, `z`$[k] = p$ tells us that `s`$[0 \ldots p-1]$ equals `s`$[k \ldots k+p-1]$, but `s`$[p]$ and `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, `z`$[3] = 5$, because the substring `ABCAB` of length 5 is a prefix of `s`, but the substring `ABCABA` of length 6 is not a prefix of `s`.