Theory: Prefix function

31 minutes 6 / 6 problems solved

The prefix function is a structure that allows us to solve different string processing problems efficiently. For example, it can be used to implement a substring searching algorithm that works in linear time in the worst case. In this lesson, we will get familiar with this structure, build an efficient algorithm for its calculation, and see how it works in one simple example.

Current topic:

Prefix function

Topic depends on:

String basics

Topic is required for:

Knuth-Morris-Pratt algorithm

§1. A definition of the prefix function

For a string s, the prefix function is an array p of length $|s|$ where $p[i]$ is the length of the longest prefix of $s[0..i+1]$ that is also a suffix of $s[0..i+1]$. Prefixes equal to the string itself are not considered. Usually, we think that function is a map $f(x) \to y$. And in that case, an algorithm is also called a function, because in some way we get values for every index in the string. So we make a function $p(i) = y$, where i is the string index and y is the value of prefix function on such prefix. By convention, $p[0] = 0$. Given below is an example of the prefix function for a string $s = ABACABAD$:

	0	1	2	3	4	5	6	7
s	A	B	A	C	A	B	A	D
p	0	0	1	0	1	2	3	0

The array p was calculated as follows:

- $p[0] = 0$ by convention.
- $p[1] = 0$, since $s[0..2] = AB$ contains no prefixes equal to its suffixes.
- $p[2] = 1$, since the longest prefix of $s[0..3] = ABA$, which is also a suffix is A.
- $p[3] = 0$, since $s[0..4] = ABAC$ contains no prefixes equal to its suffixes.
- $p[4] = 1$, since the longest prefix of $s[0..5] = ABACA$, which is also a suffix is A.
- $p[5] = 2$, since the longest prefix of $s[0..6] = ABACAB$, which is also a suffix is AB.
- $p[6] = 3$, since the longest prefix of $s[0..7] = ABACABA$, which is also a suffix is ABA.
- $p[7] = 0$, since $s[0..8] = ABACABAD$ contains no prefixes equal to its suffixes.

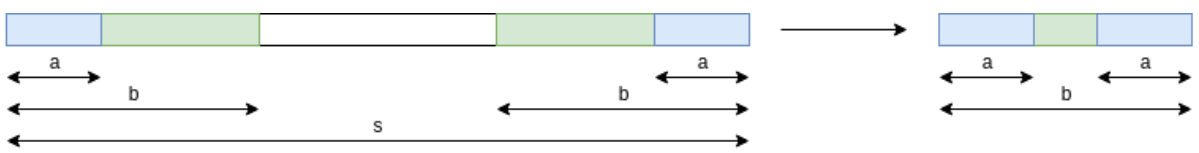
Further, we will denote a prefix of s, which is also a suffix as a **border** of s.

Our next goal is to implement an efficient algorithm for calculation of the prefix function. To do this, we first need to learn a few properties of borders.

§2. Properties of borders

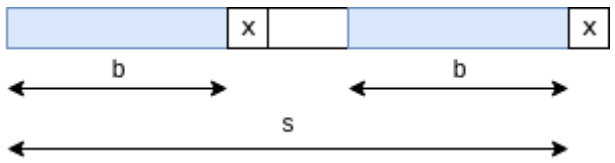
Keep in mind that the border of string s is such string t that is at the same time a prefix and a suffix of string s. Importantly, the borders may intersect. For example, if we have a string $AAAAA$, it would have 4 borders: A, AA, AAA and $AAAA$.

Property 1. Let a and b be borders of a string s, and $|a| < |b|$ Then a is a border of b. The following figure illustrates the statement:



Since a is a prefix of s, it is a prefix of b as well. Similarly, a is a suffix of s and therefore is a suffix of b. So, a is a border of b. From this property, we can conclude that if b is the longest border of s, then the next longest border of s can be found as the longest border of b.

Property 2. Suppose b is a border of a string s. Then b can be extended by the symbol x if bx is a border of sx. The figure below explains the statement:



This property will be quite useful in the future. For now, though, it may not be very clear why do we need it, but soon you will get it.

§3. An algorithm for calculation of the prefix function

Now, let's see how using the properties of borders, we can find the prefix function p for a string s. The array of prefix function is evaluated step by step, so first, we compute $p[1]$, then $p[2]$ and so on until $p[9]$. To calculate i-th value of prefix function, we need previous values only. The idea is to calculate all elements of p sequentially, starting from the first, then calculating the second and so on until the last element. At each step, we use previously calculated elements of p to find the longest border that can be extended by the current symbol or s. Let's consider this step in more detail for the following example:

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

Here, the elements $p[0], p[1], ..., p[8]$ are already calculated and the last step is to calculate $p[9]$. To do this, we check whether we can extend the longest border of $s[0..9]$ (see the left figure below). This border is $s[0..p[8]-1] = s[0..3] = ACCA$ (shown in yellow).

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

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

We can see that $s[p[8]] = s[4] = B$, while $s[9] = C$. Since $B \neq C$, the border cannot be extended. Then, we need to check whether it is possible to extend the next longest border of $s[0..9]$. According to the first property, such border can be found as the longest border of $ACCA$. This border is $s[0..p[3]-1] = s[0] = A$ (see the right figure above). Since $s[1] = C$ and $s[9] = C$, the border can be extended and $p[9] = p[3] + 1 = 1 + 1 = 2$.

Note that in some cases, there might be no border that can be extended. Then, the corresponding value of p is assigned to 0.

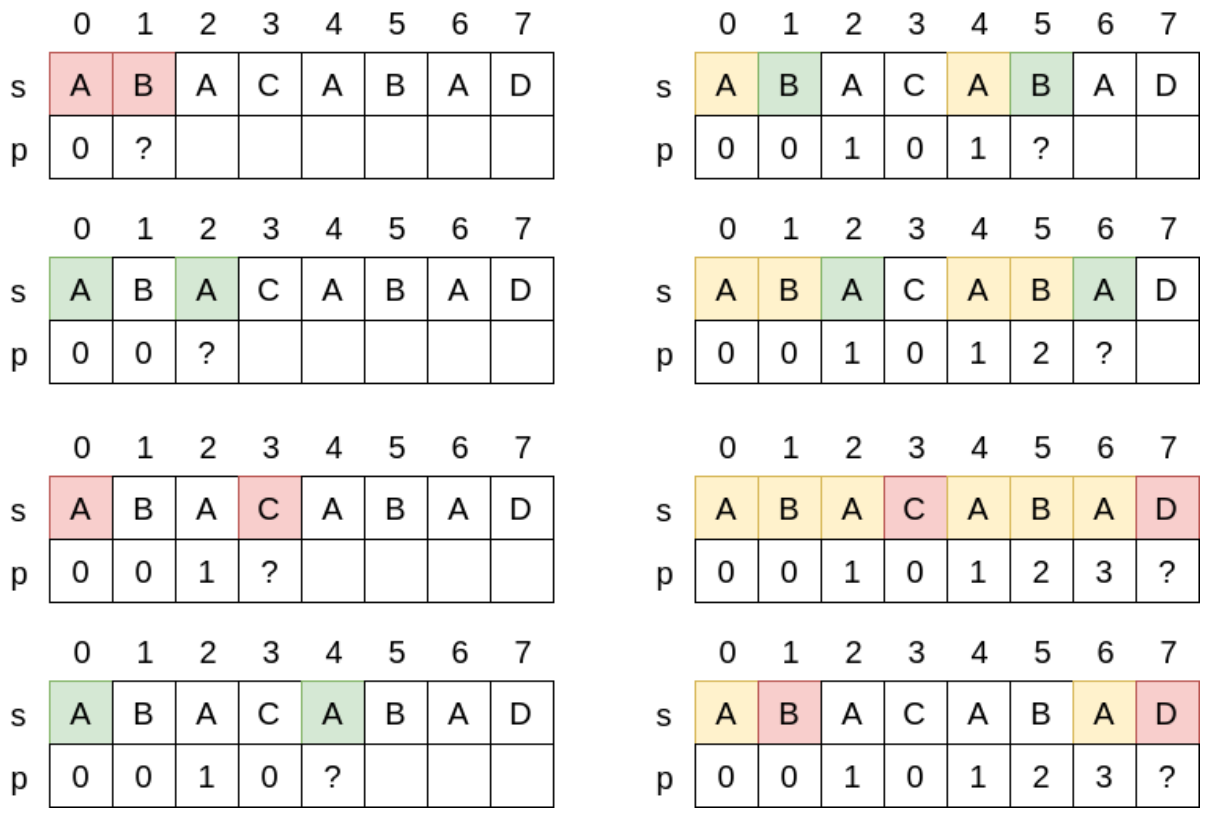
To sum up, the steps of the algorithm are:

1. Set $p[0]$ to 0.
2. For $i = 1..|s|$, set $p[i]$ to $l + 1$, where l is the length of the longest border of $s[0..i]$ that can be extended by $s[i]$. If there is no border that can be extended, set $p[i]$ to 0.

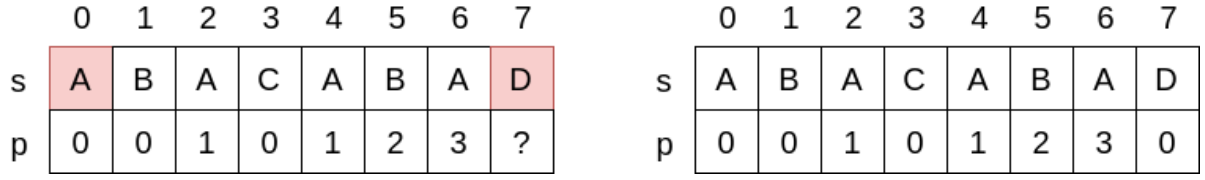
After all elements of p are calculated, the array will correspond to the prefix function for s.

§4. An example

Let's see how the described algorithm works for $s = ABACABAD$. The figures below illustrate the steps of the algorithm:



First, we need to calculate $p[1]$. At this step, the longest border of $s[0..1] = A$ is empty and since $s[0] \neq s[1]$, it cannot be extended. So, $p[1]$ is set to 0. At the next step, the longest border of $s[0..2]$ is empty again, but it can be extended, since $s[0] = s[2]$. In this case, $s[2]$ is set to 1.



Continuing the same procedure, we will finally get the same prefix function $p = [0, 0, 1, 0, 1, 2, 3, 0]$ as the example above.

§5. Complexity analysis

While calculating the prefix function p for a string s, we perform one iteration for each element of p, thus $|s|$ iterations in total. At each of these iterations, a new value of the prefix function can either increase by one or decrease. Therefore, the total number of increases, as well as the total number of decreases, is limited by $|s|$. From this, we can conclude that the total running time of the algorithm is $O(|s|)$.

Report a typo

35 users liked this theory. 24 didn't like it. What about you?

Start practicing

Comments (6) Hints (0) Useful links (2) Show discussion

Table of contents: