

Prefix function

Theory

Practice

66% completed, 2 problems solved

Theory

🕒 22 minutes reading

Verify to skip

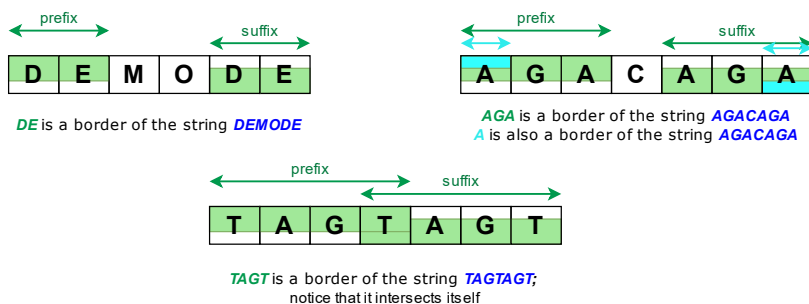
Start practicing

When we search inside a string, there are certain questions that come to mind. Does a string have a pattern? And if it has some recurrent parts, how many of them are there? We need to know how to answer them because they can direct our actions: for example, if a part of a string repeats itself, we can easily compress this string, and so on.

In this topic, we will introduce a unique structure called prefix function that will allow us to investigate the borders of substrings. The algorithm for its computation may look unusual, but fear not. Remember that every problem has a better solution if you start thinking about it from a new perspective.

§1. Border and its length

Recall that a **border** of a string s is a proper prefix of s that is also a suffix of that string. It is worth noting that a string usually has more than one border and it is completely normal for borders to intersect themselves. Let's escape from abstraction by taking a look at a couple of examples:



We are ready to define our main tool now. First of all, let's grasp the main idea:

1. Prefix functions do not live on their own. You need to choose a string or an array of symbols to calculate its prefix function. Let's choose the string *SYMBOLS* as an example.
2. The result of our calculations will look like an array of numbers.
3. To find the numbers, we will need to examine the substrings of our *SYMBOLS*. We will start with the substring *S*, then continue with *SY*, after that – with *SYM* and so on.
4. Every number in our array shows what is the length of the border in a substring that we examine. So, the first number is 0, then 0 again. Well, our string is not so repetitive, but what can we do.
5. It seems like only the last substring, namely *SYMBOLS*, has a border *S*. So we can finally write down 1 instead of 0, because the length of the border here is 1.

To sum up: the prefix function of *SYMBOLS* is $[0, 0, 0, 0, 0, 0, 1]$.

§2. Definition of the prefix function

Let's hope you've understood the informal explanation. Now, let's move on to the official one. Given a string s of length n . One can define the **prefix function** of s as an array p of length n , where $p[i]$ is the length of the longest border of the substring $s[0..i]$. It is obvious that $p[0] = 0$, since a string of length 1 can't have nonempty proper prefixes.

1 required topic

✓ Searching a substring

1 dependent topic

Knuth-Morris-Pratt algorithm

Table of contents:

↑ Prefix function

§1. Border and its length

§2. Definition of the prefix function

§3. Improving the naive approach

§4. Efficient algorithm

§5. Complexity analysis

§6. Why prefix function

§7. Conclusion

Discussion

In order to get a better understanding of this notion, let's calculate the prefix function of some strings. Take the string $s = ABACABB$ and another string $w = AAAABA$. First of all, we will examine the prefixes of those two strings and find the longest borders for each one of them:

$s[0,0]$	A						
$s[0,1]$	A	B					
$s[0,2]$	A	B	A				
$s[0,3]$	A	B	A	C			
$s[0,4]$	A	B	A	C	A		
$s[0,5]$	A	B	A	C	A	B	
$s[0,6]$	A	B	A	C	A	B	B

$w[0,0]$	A						
$w[0,1]$	A	A					
$w[0,2]$	A	A	A				
$w[0,3]$	A	A	A	A			
$w[0,4]$	A	A	A	A	B		
$w[0,5]$	A	A	A	A	B	A	

Following the definition, after some observations (see the picture on the left), we conclude that the prefix function of the first string is $[0, 0, 1, 0, 1, 2, 0]$. For example, $p[5] = 2$, because the longest border of the substring $s[0, 5] = ABACAB$ is AB , whose length is 2.

String $w = AAAABA$, and it is a bit trickier! Keep in mind that a border is a proper prefix-suffix and it might intersect itself. Here, we conclude that $p = [0, 1, 2, 3, 0, 1]$ as in the picture on the right.

It might seem contradictory that the prefix function is an array, and not a function. Well, the definition of an array is a simple and correct way to get the idea of it. Formally, the prefix function is a function from the set $\{0, 1, \dots, n-1\}$ (the i of the prefixes $s[0, i]$) to $\{0, 1, \dots, n-1\}$ (the possible lengths of the borders). However, this won't play any role later on; it is simply to recognize that the word function in prefix function makes actual sense and is not just put randomly there.

§3. Improving the naive approach

So, how did we calculate the prefix functions above? Directly by definition – a method that is known as the naive approach. Namely, for each one of n prefixes $s[0, i]$ of the initial string, we find the longest border by comparing the prefixes and suffixes of every length to detect the borders, and then we choose the longest amongst them. Overall, there are n substrings, and for each one we find the longest border. It takes $O(n^2)$, since we compare two strings in linear time for every possible length. Hence the general time complexity is $O(n^3)$, which is too long.

In practice, strings can have enormous lengths, so we need to come up with a much more efficient algorithm. Borders and their properties come to the rescue. Let's make some observations that will allow us to considerably improve our algorithm's time complexity.

1. Skip unnecessary comparisons using the fact that $p[i+1] \leq p[i] + 1$. In other words, if we know the value of $p[i]$, we can conclude that the value of $p[i+1]$ can't increase by more than one. This means that we have to search the longest border among the prefix-suffixes of length no more than $p[i] + 1$, avoiding many unnecessary comparisons.

► Why does this assumption hold true?

2. Get rid of string comparisons by scanning only the last character of the string and using the already calculated values. Suppose we already know previous values, we intend to compute $p[i+1]$.

► We do as follows:

§4. Efficient algorithm

The observations above allow us to easily build a correct and efficient algorithm for calculating the prefix function of a given string. The steps can be described as below:

1. Set $p[0] = 0$.
2. For every prefix $s[0, i]$ of the string s calculate the value of $p[i]$ as follows:
define $j = p[i - 1]$.
3. Compare the symbols $s[j]$ and $s[i]$. If they match, set $p[i] = p[i - 1] + 1$.
4. Otherwise, take $j = p[j - 1]$ and repeat step 3. If we reach $j = 0$ and there is no match, we conclude that $p[i] = 0$.

Let us see how the algorithm works in an example. We are going to compute the prefix function for the string $s = ACCABACCAC$. For obvious reasons, we are not going to explain every iteration of the algorithm. Assume that we've already calculated the values of the prefix function for $i < 6$ and got the result $[0, 0, 0, 1, 0, 1]$ (try it!). Let's calculate $p[6]$ based on the algorithm. Define $j = p[5] = 1$. Check if $s[1] = s[6]$. In our case, $s[1] = C$ and $s[6] = C$ – it is a match, hence we assert that $p[6] = p[5] + 1 = 2$.

9 i

	0	1	2	3	4	5	6	7	8	9
s[i]	A	C	C	A	B	A	C	C	A	C
p[i]	0	0	0	1	0	1	2	-	-	-

9.1 i

	0	1	2	3	4	5	6	7	8	9
s[i]	A	C	C	A	B	A	C	C	A	C
p[i]	0	0	0	1	0	1	2	3	4	-

9.2 i

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

Similarly, we get that $p[7] = 3$ and $p[8] = 4$. For the sake of better understanding, we will explain in detail the computation of $p[9]$.

- Define $j = p[8] = 4$.
- Check if $s[4] = s[9]$. We have $s[4] = B$ and $s[9] = C$.
- They are not the same, hence take $j = p[j - 1]$, i.e. $j = p[4 - 1] = 1$.
- Check again if $s[1] = s[9]$. We obtain $C = C$, a match. Hence we set $p[9] = p[3] + 1 = 2$.

All the values have been found, therefore, the prefix function of the string $s = ACCABACCAC$ is $[0, 0, 0, 1, 0, 1, 2, 3, 4, 2]$.

§5. Complexity analysis

After many efforts, we have come up with an effective algorithm. However, one question still remains: what is the time complexity of our new algorithm? The answer should cheer you up: the efficient algorithm works on $O(n)$, which is far better than $O(n^3)$ of the naive approach. Anyway, why $O(n)$ and not $O(n^2)$, if we have n iterations and during each iteration we reduce j until we get a match?

The answer is tricky: the total time of the algorithm depends on the total number of times we decrease the value of j . However, during every iteration, based on property 1, we know that its value can't increase by more than one. Since we start from $j = 0$, the maximum value of j can be $n - 1$. This means that during all the iterations, we can't decrease j more than n times, therefore, the time complexity of our algorithm is $O(n)$.

§6. Why prefix function

At first, the prefix function was introduced in order to speed up the process of substring search. Indeed, it provides us with sufficient information on the structure of the string, so that we can avoid repetitions and unnecessary comparisons. This gives us one of the most effective substring search algorithms as we will soon see.

However, except for the classical application, there are dozens of problems that we can solve by using the prefix function smartly. Let's shortly mention some of them:

1. Counting the number of occurrences of every prefix. Given a string s , we find the number of occurrences of each prefix $s[0, i]$ in it. Moreover, we can count the number of occurrences of these prefixes in another string w . This task is essential in bioinformatics while examining long sequences of DNA, for example.
2. Counting the number of different substrings in a string. The name is self-explanatory. Such a task is fundamental for the following problem.
3. Compressing a text. Given a string s , we want to find a shorter string w , such

that s is a concatenation of copies of w . Text compression is crucial: think of a long text file. If we store the information as it is, it would take an immense amount of space; however, after compressing it, we end up with a file of several kilobytes.

Unfortunately, the algorithms solving these problems are out of the scope of our platform. Nevertheless, you will find some Easter eggs in the problem section.

§7. Conclusion

Prefix function is a tool with many applications in string algorithms. An easy way to define it is as an array p consisting of the lengths of the longest prefix-suffix of every prefix. It provides us with essential information on the repeating parts of the string, and not only about that. You can calculate the prefix function of a string of a length n in $O(n)$ time using these steps:

1. Set $p[0] = 0$. Do the following for every i from 0 to $n - 1$:
2. Define $j = p[i - 1]$.
3. If $s[j] = s[i]$, set $p[i] = p[i - 1] + 1$.
4. Otherwise, take $j = p[j - 1]$ and repeat step 3. If we reach $j = 0$ and there is no match, we conclude that $p[i] = 0$.

In the following topic, we will see one of the most effective substring search algorithms, which uses the prefix function as a helping tool to avoid repetitions and unnecessary comparisons. For now, try to take in this theory and practice in the exercise section. Good job!

 Report a typo

13 users liked this piece of theory. 18 didn't like it. What about you?



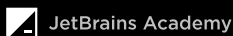
Start practicing

Verify to skip

[Comments \(7\)](#)

[Useful links \(0\)](#)

[Show discussion](#)



Tracks


Pricing

For organizations

About

Contribute

Careers


 Become beta tester

Be the first to see what's new



[Terms](#) [Support](#)



Made with  by Hyperskill and JetBrains