Last update: August 20, 2023 Translated From: e-maxx.ru

Prefix function. Knuth–Morris–Pratt algorithm

Prefix function definition

You are given a string s of length n. The **prefix function** for this string is defined as an array π of length n, where $\pi[i]$ is the length of the longest proper prefix of the substring $s[0 \dots i]$ which is also a suffix of this substring. A proper prefix of a string is a prefix that is not equal to the string itself. By definition, $\pi[0] = 0$.

Mathematically the definition of the prefix function can be written as follows:

$$\pi[i] = \max_{k = 0 \dots i} \{k : s[0 \dots k-1] = s[i-(k-1) \dots i]\}$$

For example, prefix function of string "abcabcd" is $[0, 0, 0, 1, 2, 3, 0]$, and prefix function of string "aabaaab" is $[0, 1, 0, 1, 2, 2, 3]$.

Trivial Algorithm

An algorithm which follows the definition of prefix function exactly is the following:

```
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 0; i < n; i++)
        for (int k = 0; k <= i; k++)
            if (s.substr(0, k) == s.substr(i-k+1, k))
                pi[i] = k;
    return pi;
}
```

It is easy to see that its complexity is $O(n^3)$, which has room for improvement.

Efficient Algorithm

This algorithm was proposed by Knuth and Pratt and independently from them by Morris in 1977. It was used as the main function of a substring search algorithm.

First optimization

The first important observation is, that the values of the prefix function can only increase by at most one.

Indeed, otherwise, if $\pi[i + 1] > \pi[i] + 1$, then we can take this suffix ending in position $i + 1$ with the length $\pi[i + 1]$ and remove the last character from it. We end up with a suffix ending in position i with the length $\pi[i + 1] - 1$, which is better than $\pi[i]$, i.e. we get a contradiction.

The following illustration shows this contradiction. The longest proper suffix at position i that also is a prefix is of length 2, and at position $i+1$ it is of length 4. Therefore the string $s_0 ~ s_1 ~ s_2 ~ s_3$ is equal to the string $s_{i-2} ~ s_{i-1} ~ s_i ~ s_{i+1}$, which means that also the strings $s_0 ~ s_1 ~ s_2$ and $s_{i-2} ~ s_{i-1} ~ s_i$ are equal, therefore $\pi[i]$ has to be 3.

$$\underbrace{\overbrace{s_0 ~ s_1}^{\pi[i] = 2} ~ s_2 ~ s_3}_{\pi[i+1] = 4} ~ \dots ~ \underbrace{s_{i-2} ~ \overbrace{s_{i-1} ~ s_{i}}^{\pi[i] = 2} ~ s_{i+1}}_{\pi[i+1] = 4}$$

Thus when moving to the next position, the value of the prefix function can either increase by one, stay the same, or decrease by some amount. This fact already allows us to reduce the complexity of the algorithm to $O(n^2)$, because in one step the prefix function can grow at most by one. In total the function can grow at most n steps, and therefore also only can decrease a total of n steps. This means we only have to perform $O(n)$ string comparisons, and reach the complexity $O(n^2)$.

Second optimization

Let's go further, we want to get rid of the string comparisons. To accomplish this, we have to use all the information computed in the previous steps.

So let us compute the value of the prefix function π for $i + 1$. If $s[i+1] = s[\pi[i]]$, then we can say with certainty that $\pi[i+1] = \pi[i] + 1$, since we already know that the suffix at position i of length $\pi[i]$ is equal to the prefix of length $\pi[i]$. This is illustrated again with an example.

$$\underbrace{\overbrace{s_0 ~ s_1 ~ s_2}^{\pi[i]} ~ \overbrace{s_3}^{s_3 = s_{i+1}}}_{\pi[i+1] = \pi[i] + 1} ~ \dots ~ \underbrace{\overbrace{s_{i-2} ~ s_{i-1} ~ s_{i}}^{\pi[i]} ~ \overbrace{s_{i+1}}^{s_3 = s_{i+1}}}_{\pi[i+1] = \pi[i] + 1}$$

If this is not the case, $s[i+1] \neq s[\pi[i]]$, then we need to try a shorter string. In order to speed things up, we would like to immediately move to the longest length $j < \pi[i]$, such that the prefix property in the position i holds, i.e. $s[0 \dots j-1] = s[i-j+1 \dots i]$:

$$\overbrace{\underbrace{s_0 ~ s_1}_j ~ s_2 ~ s_3}^{\pi[i]} ~ \dots ~ \overbrace{s_{i-3} ~ s_{i-2} ~ \underbrace{s_{i-1} ~ s_{i}}_j}^{\pi[i]} ~ s_{i+1}$$

Indeed, if we find such a length j, then we again only need to compare the characters $s[i+1]$ and $s[j]$. If they are equal, then we can assign $\pi[i+1] = j + 1$. Otherwise we will need to find the largest value smaller than j, for which the prefix property holds, and so on. It can happen that this goes until $j = 0$. If then $s[i+1] = s[0]$, we assign $\pi[i+1] = 1$, and $\pi[i+1] = 0$ otherwise.

So we already have a general scheme of the algorithm. The only question left is how do we effectively find the lengths for j. Let's recap: for the current length j at the position i for which the prefix property holds, i.e. $s[0 \dots j-1] = s[i-j+1 \dots i]$, we want to find the greatest $k < j$, for which the prefix property holds.

$$\overbrace{\underbrace{s_0 ~ s_1}_k ~ s_2 ~ s_3}^j ~ \dots ~ \overbrace{s_{i-3} ~ s_{i-2} ~ \underbrace{s_{i-1} ~ s_i}_k}^j ~ s_{i+1}$$

The illustration shows, that this has to be the value of $\pi[j-1]$, which we already calculated earlier.

Final algorithm

So we finally can build an algorithm that doesn't perform any string comparisons and only performs $O(n)$ actions.

Here is the final procedure:

- We compute the prefix values $\pi[i]$ in a loop by iterating from $i = 1$ to $i = n-1$ ($\pi[0]$ just gets assigned with 0).
- To calculate the current value $\pi[i]$ we set the variable j denoting the length of the best suffix for $i-1$. Initially $j = \pi[i-1]$.
- Test if the suffix of length $j+1$ is also a prefix by comparing $s[j]$ and $s[i]$. If they are equal then we assign $\pi[i] = j+1$, otherwise we reduce j to $\pi[j-1]$ and repeat this step.
- If we have reached the length $j = 0$ and still don't have a match, then we assign $\pi[i] = 0$ and go to the next index $i+1$.

Implementation

The implementation ends up being surprisingly short and expressive.

```
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            j++;
        pi[i] = j;
    }
    return pi;
}
```

This is an **online** algorithm, i.e. it processes the data as it arrives - for example, you can read the string characters one by one and process them immediately, finding the value of prefix function for each next character. The algorithm still requires storing the string itself and the previously calculated values of prefix function, but if we know beforehand the maximum value M the prefix function can take on the string, we can store only $M+1$ first characters of the string and the same number of values of the prefix function.

Applications

Search for a substring in a string. The Knuth-Morris-Pratt algorithm

The task is the classical application of the prefix function.

Given a text t and a string s, we want to find and display the positions of all occurrences of the string s in the text t.

For convenience we denote with n the length of the string s and with m the length of the text t.

We generate the string $s + \# + t$, where $\#$ is a separator that appears neither in s nor in t. Let us calculate the prefix function for this string. Now think about the meaning of the values of the prefix function, except for the first $n+1$ entries (which belong to the string s and the separator). By definition the value $\pi[i]$ shows the longest length of a substring ending in position i that coincides with the prefix. But in our case this is nothing more than the largest block that coincides with s and ends at position i. This length cannot be bigger than n due to the separator. But if equality $\pi[i] = n$ is achieved, then it means that the string s appears completely in at this position, i.e. it ends at position i. Just do not forget that the positions are indexed in the string $s + \# + t$.

Thus if at some position i we have $\pi[i] = n$, then at the position $i - (n+1) - n + 1 = i - 2n$ in the string t the string s appears.

As already mentioned in the description of the prefix function computation, if we know that the prefix values never exceed a certain value, then we do not need to store the entire string and the entire function, but only its beginning. In our case this means that we only need to store the string $s + \#$ and the values of the prefix function for it. We can read one character at a time of the string t and calculate the current value of the prefix function.

Thus the Knuth-Morris-Pratt algorithm solves the problem in $O(n+m)$ time and $O(n)$ memory.

Counting the number of occurrences of each prefix

Here we discuss two problems at once. Given a string s of length n. In the first variation of the problem we want to count the number of appearances of each prefix $s[0 \dots i]$ in the same string. In the second variation of the problem another string t is given and we want to count the number of appearances of each prefix $s[0 \dots i]$ in t.

First we solve the first problem. Consider the value of the prefix function $\pi[i]$ at a position i. By definition it means that the prefix of length $\pi[i]$ of the string s occurs and ends at position i, and there is no longer prefix that follows this definition. At the same time shorter prefixes can end at this position. It is not difficult to see, that we have the same question that we already answered when we computed the prefix function itself: Given a prefix of length j that is a suffix ending at position i, what is the next smaller prefix $< j$ that is also a suffix ending at position i. Thus at the position i ends the prefix of length $\pi[i]$, the prefix of length $\pi[\pi[i]-1]$, the prefix $\pi[\pi[\pi[i]-1]-1]$, and so on, until the index becomes zero. Thus we can compute the answer in the following way.

```
vector<int> ans(n + 1);
for (int i = 0; i < n; i++)
    ans[pi[i]]++;
for (int i = n-1; i > 0; i--)
    ans[pi[i-1]] += ans[i];
for (int i = 0; i <= n; i++)
    ans[i]++;
```

Here for each value of the prefix function we first count how many times it occurs in the array π, and then compute the final answers: if we know that the length prefix i appears exactly $\text{ans}[i]$ times, then this number must be added to the number of occurrences of its longest suffix that is also a prefix. At the end we need to add 1 to each result, since we also need to count the original prefixes also.

Now let us consider the second problem. We apply the trick from Knuth-Morris-Pratt: we create the string $s + \# + t$ and compute its prefix function. The only differences to the first task is, that we are only interested in the prefix values that relate to the string t, i.e. $\pi[i]$ for $i \ge n + 1$. With those values we can perform the exact same computations as in the first task.

The number of different substring in a string

Given a string s of length n. We want to compute the number of different substrings appearing in it.

We will solve this problem iteratively. Namely we will learn, knowing the current number of different substrings, how to recompute this count by adding a character to the end.

So let k be the current number of different substrings in s, and we add the character c to the end of s. Obviously some new substrings ending in c will appear. We want to count these new substrings that didn't appear before.

We take the string $t = s + c$ and reverse it. Now the task is transformed into computing how many prefixes there are that don't appear anywhere else. If we compute the maximal value of the prefix function $\pi_{\max}$ of the reversed string t, then the longest prefix that appears in s is $\pi_{\max}$ long. Clearly also all prefixes of smaller length appear in it.

Therefore the number of new substrings appearing when we add a new character c is $|s| + 1 - \pi_{\max}$.

So for each character appended we can compute the number of new substrings in $O(n)$ times, which gives a time complexity of $O(n^2)$ in total.

It is worth noting, that we can also compute the number of different substrings by appending the characters at the beginning, or by deleting characters from the beginning or the end.

Compressing a string

Given a string s of length n. We want to find the shortest "compressed" representation of the string, i.e. we want to find a string t of smallest length such that s can be represented as a concatenation of one or more copies of t.

It is clear, that we only need to find the length of t. Knowing the length, the answer to the problem will be the prefix of s with this length.

Let us compute the prefix function for s. Using the last value of it we define the value $k = n - \pi[n-1]$. We will show, that if k divides n, then k will be the answer, otherwise there is no effective compression and the answer is n.

Let n be divisible by k. Then the string can be partitioned into blocks of the length k. By definition of the prefix function, the prefix of length $n-k$ will be equal with its suffix. But this means that the last block is equal to the block before. And the block before has to be equal to the block before it. And so on. As a result, it turns out that all blocks are equal, therefore we can compress the string s to length k.

Of course we still need to show that this is actually the optimum. Indeed, if there was a smaller compression than k, than the prefix function at the end would be greater than $n-k$. Therefore k is really the answer.

Now let us assume that n is not divisible by k. We show that this implies that the length of the answer is n. We prove it by contradiction. Assuming there exists an answer, and the compression has length p (p divides n). Then the last value of the prefix function has to be greater than $n-p$, i.e. the suffix will partially cover the first block. Now consider the second block of the string. Since the prefix is equal with the suffix, and both the prefix and the suffix cover this block and their displacement relative to each other k does not divide the block length p (otherwise k divides n), then all the characters of the block have to be identical. But then the string consists of only one character repeated over and over, hence we can compress it to a string of size 1, which gives $k=1$, and k divides n. Contradiction.

$$\overbrace{s_0\ s_1\ s_2\ s_3}^{p}\ \overbrace{s_4\ s_5\ s_6\ s_7}^{p}$$

$$s_0\ s_1\ s_2\ \underbrace{\overbrace{s_3\ s_4\ s_5\ s_6}^{p}\ s_7}_{\pi[7]=5}$$

$$s_4 = s_3,\ s_5 = s_4,\ s_6 = s_5,\ s_7 = s_6\ \Rightarrow\ s_0 = s_1 = s_2 = s_3$$

Building an automaton according to the prefix function

Let's return to the concatenation to the two strings through a separator, i.e. for the strings s and t we compute the prefix function for the string $s + \# + t$. Obviously, since $\#$ is a separator, the value of the prefix function will never exceed $|s|$. It follows, that it is sufficient to only store the string $s + \#$ and the values of the prefix function for it, and we can compute the prefix function for all subsequent character on the fly:

$$\underbrace{s_0\ s_1\ \dots\ s_{n-1}\ \#}_{\text{need to store}}\ \underbrace{t_0\ t_1\ \dots\ t_{m-1}}_{\text{do not need to store}}$$

Indeed, in such a situation, knowing the next character $c \in t$ and the value of the prefix function of the previous position is enough information to compute the next value of the prefix function, without using any previous characters of the string t and the value of the prefix function in them.

In other words, we can construct an **automaton** (a finite state machine): the state in it is the current value of the prefix function, and the transition from one state to another will be performed via the next character.

Thus, even without having the string t, we can construct such a transition table $(\text{old}_\pi, c) \to \text{new}_\pi$ using the same algorithm as for calculating the transition table:

```
void compute_automaton(string s, vector<vector<int>>& aut) {
    s += '#';
    int n = s.size();
    vector<int> pi = prefix_function(s);
    aut.assign(n, vector<int>(26));
    for (int i = 0; i < n; i++) {
        for (int c = 0; c < 26; c++) {
            int j = i;
            while (j > 0 && 'a' + c != s[j])
                j = pi[j-1];
            if ('a' + c == s[j])
                j++;
            aut[i][c] = j;
        }
    }
}
```

However in this form the algorithm runs in $O(n^2 26)$ time for the lowercase letters of the alphabet. Note that we can apply dynamic programming and use the already calculated parts of the table. Whenever we go from the value j to the value $\pi[j-1]$, we actually mean that the transition (j, c) leads to the same state as the transition as $(\pi[j-1], c)$, and this answer is already accurately computed.

```
void compute_automaton(string s, vector<vector<int>>& aut) {
    s += '#';
    int n = s.size();
    vector<int> pi = prefix_function(s);
    aut.assign(n, vector<int>(26));
    for (int i = 0; i < n; i++) {
        for (int c = 0; c < 26; c++) {
            if (i > 0 && 'a' + c != s[i])
                aut[i][c] = aut[pi[i-1]][c];
            else
                aut[i][c] = i + ('a' + c == s[i]);
        }
    }
}
```

As a result we construct the automaton in $O(26n)$ time.

When is such a automaton useful? To begin with, remember that we use the prefix function for the string $s + \# + t$ and its values mostly for a single purpose: find all occurrences of the string s in the string t.

Therefore the most obvious benefit of this automaton is the **acceleration of calculating the prefix function** for the string $s + \# + t$. By building the automaton for $s + \#$, we no longer need to store the string s or the values of the prefix function in it. All transitions are already computed in the table.

But there is a second, less obvious, application. We can use the automaton when the string t is a **gigantic string constructed using some rules**. This can for instance be the Gray strings, or a string formed by a recursive combination of several short strings from the input.

For completeness we will solve such a problem: given a number $k \le 10^5$ and a string s of length $\le 10^5$. We have to compute the number of occurrences of s in the k-th Gray string. Recall that Gray's strings are define in the following way:

$$\begin{aligned}g_1 &= \text{"a"}\\ g_2 &= \text{"aba"}\\ g_3 &= \text{"abacaba"}\\ g_4 &= \text{"abacabadabacaba"}\end{aligned}$$

In such cases even constructing the string t will be impossible, because of its astronomical length. The k-th Gray string is $2^k - 1$ characters long. However we can calculate the value of the prefix function at the end of the string effectively, by only knowing the value of the prefix function at the start.

In addition to the automaton itself, we also compute values $G[i][j]$ - the value of the automaton after processing the string g_i starting with the state j. And additionally we compute values $K[i][j]$ - the number of occurrences of s in g_i, before during the processing of g_i starting with the state j. Actually $K[i][j]$ is the number of times that the prefix function took the value $|s|$ while performing the operations. The answer to the problem will then be $K[k][0]$.

How can we compute these values? First the basic values are $G[0][j] = j$ and $K[0][j] = 0$. And all subsequent values can be calculated from the previous values and using the automaton. To calculate the value for some i we remember that the string g_i consists of g_{i-1}, the i character of the alphabet, and g_{i-1}. Thus the automaton will go into the state:

$$\text{mid} = \text{aut}[G[i-1][j]][i]$$

$$G[i][j] = G[i-1][\text{mid}]$$

The values for $K[i][j]$ can also be easily counted.

$$K[i][j] = K[i-1][j] + (\text{mid} == |s|) + K[i-1][\text{mid}]$$

So we can solve the problem for Gray strings, and similarly also a huge number of other similar problems. For example the exact same method also solves the following problem: we are given a string s and some patterns t_i, each of which is specified as follows: it is a string of ordinary characters, and there might be some recursive insertions of the previous strings of the form t_k^{cnt}, which means that at this place we have to insert the string t_k cnt times. An example of such patterns:

$$\begin{aligned}t_1 &= \text{"abdeca"}\\ t_2 &= \text{"abc"} + t_1^{30} + \text{"abd"}\\ t_3 &= t_2^{50} + t_1^{100}\\ t_4 &= t_2^{10} + t_3^{100}\end{aligned}$$

The recursive substitutions blow the string up, so that their lengths can reach the order of 100^{100}.

We have to find the number of times the string s appears in each of the strings.

The problem can be solved in the same way by constructing the automaton of the prefix function, and then we calculate the transitions in for each pattern by using the previous results.

Practice Problems

- UVA # 455 "Periodic Strings"
- UVA # 11022 "String Factoring"
- UVA # 11452 "Dancing the Cheeky-Cheeky"
- UVA 12604 - Caesar Cipher
- UVA 12467 - Secret Word
- UVA 11019 - Matrix Matcher
- SPOJ - Pattern Find
- SPOJ - A Needle in the Haystack
- Codeforces - Anthem of Berland
- Codeforces - MUH and Cube Walls
- Codeforces - Prefixes and Suffixes