

# Lyndon factorization

## Table of Contents

- [Lyndon factorization](#)
- [Duval algorithm](#)
  - [Implementation](#)
  - [Complexity](#)
- [Finding the smallest cyclic shift](#)
- [Problems](#)

## Lyndon factorization

First let us define the notion of the Lyndon factorization.

A string is called **simple** (or a Lyndon word), if it is strictly **smaller than** any of its own nontrivial **suffixes**. Examples of simple strings are:  $a$ ,  $b$ ,  $ab$ ,  $aab$ ,  $abb$ ,  $ababb$ ,  $abcd$ . It can be shown that a string is simple, if and only if it is strictly **smaller than** all its nontrivial **cyclic shifts**.

Next, let there be a given string  $s$ . The **Lyndon factorization** of the string  $s$  is a factorization  $s = w_1 w_2 \dots w_k$ , where all strings  $w_i$  are simple, and they are in non-increasing order  $w_1 \geq w_2 \geq \dots \geq w_k$ .

It can be shown, that for any string such a factorization exists and that it is unique.

# Duval algorithm

The Duval algorithm constructs the Lyndon factorization in  $O(n)$  time using  $O(1)$  additional memory.

First let us introduce another notion: a string  $t$  is called **pre-simple**, if it has the form  $t = ww \dots w\overline{w}$ , where  $w$  is a simple string and  $\overline{w}$  is a prefix of  $w$  (possibly empty). A simple string is also pre-simple.

The Duval algorithm is greedy. At any point during its execution, the string  $s$  will actually be divided into three strings  $s = s_1 s_2 s_3$ , where the Lyndon factorization for  $s_1$  is already found and finalized, the string  $s_2$  is pre-simple (and we know the length of the simple string in it), and  $s_3$  is completely untouched. In each iteration the Duval algorithm takes the first character of the string  $s_3$  and tries to append it to the string  $s_2$ . If  $s_2$  is no longer pre-simple, then the Lyndon factorization for some part of  $s_2$  becomes known, and this part goes to  $s_1$ .

Let's describe the algorithm in more detail. The pointer  $i$  will always point to the beginning of the string  $s_2$ . The outer loop will be executed as long as  $i < n$ . Inside the loop we use two additional pointers,  $j$  which points to the beginning of  $s_3$ , and  $k$  which points to the current character that we are currently comparing to. We want to add the character  $s[j]$  to the string  $s_2$ , which requires a comparison with the character  $s[k]$ . There can be three different cases:

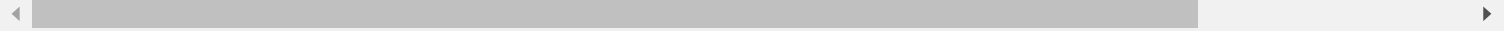
- $s[j] = s[k]$ : if this is the case, then adding the symbol  $s[j]$  to  $s_2$  doesn't violate its pre-simplicity. So we simply increment the pointers  $j$  and  $k$ .
- $s[j] > s[k]$ : here, the string  $s_2 + s[j]$  becomes simple. We can increment  $j$  and reset  $k$  back to the beginning of  $s_2$ , so that the next character can be compared with the beginning of the simple word.
- $s[j] < s[k]$ : the string  $s_2 + s[j]$  is no longer pre-simple. Therefore we will split the pre-simple string  $s_2$  into its simple strings and the remainder, possibly empty. The simple string will have the length  $j - k$ . In the next iteration we start again with the remaining  $s_2$ .

## Implementation

Here we present the implementation of the Duval algorithm, which will return the desired Lyndon factorization of a given string  $s$ .

```
vector<string> duval(string const& s) {
    int n = s.size();
    int i = 0;
    vector<string> factorization;
    while (i < n) {
        int j = i + 1, k = i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j])
                k = i;
            else
                k++;
            j++;
        }
    }
}
```

```
        while (i <= k) {
            factorization.push_back(s.substr(i, j - i));
            i += j - k;
        }
    }
    return factorization;
}
```



## Complexity

Let us estimate the running time of this algorithm.

The **outer while loop** does not exceed  $n$  iterations, since at the end of each iteration  $i$  increases. Also the second inner while loop runs in  $O(n)$ , since it only outputs the final factorization.

So we are only interested in the **first inner while loop**. How many iterations does it perform in the worst case? It's easy to see that the simple words that we identify in each iteration of the outer loop are longer than the remainder that we additionally compared. Therefore also the sum of the remainders will be smaller than  $n$ , which means that we only perform at most  $O(n)$  iterations of the first inner while loop. In fact the total number of character comparisons will not exceed  $4n - 3$ .

## Finding the smallest cyclic shift

Let there be a string  $s$ . We construct the Lyndon factorization for the string  $s + s$  (in  $O(n)$  time). We will

look for a simple string in the factorization, which starts at a position less than  $n$  (i.e. it starts in the first instance of  $s$ ), and ends in a position greater than or equal to  $n$  (i.e. in the second instance) of  $s$ ). It is stated, that the position of the start of this simple string will be the beginning of the desired smallest cyclic shift. This can be easily verified using the definition of the Lyndon decomposition.

The beginning of the simple block can be found easily - just remember the pointer  $i$  at the beginning of each iteration of the outer loop, which indicated the beginning of the current pre-simple string.

So we get the following implementation:

```
string min_cyclic_string(string s) {
    s += s;
    int n = s.size();
    int i = 0, ans = 0;
    while (i < n / 2) {
        ans = i;
        int j = i + 1, k = i;
        while (j < n && s[k] <= s[j]) {
            if (s[k] < s[j])
                k = i;
            else
                k++;
            j++;
        }
        while (i <= k)
            i += j - k;
    }
}
```

```
}  
    return s.substr(ans, n / 2);  
}
```

## Problems

- [UVA #719 - Glass Beads](#)

(c) 2014-2019 translation by <http://github.com/e-maxx-eng>

07:80/112