ITMO Academy. Suffix Array

Ivan Dyachenko

August 31, 2020

Contents

1 Suffix array. Base algorithm

1.1 What is a suffix array?

Consider a string s:

s = "ababba"

Let's write all its suffixes in lexicographical order. We get this an array of strings:

a
ababba
abba
ba
babba
bba

This sequence of suffixes is called the **suffix array**. How will we store it? If you store it as strings, then it will occupy $\mathcal{O}(n^2)$ memory. To keep it smaller, let's notice that the suffix can be identified by the index of the first character. In our example, suffixes will have these numbers:

a	**b**	**a**	**b**	**b**	**a**	
0	1	2	3	4	5	6

6	
5	`a`
0	`ababba`
2	`abba`
4	`ba`
1	`babba`
3	`bba`
p	

We will store the suffix array as an array p of suffix numbers in sorted order. Then it will occupy $\mathcal{O}(n)$ memory. Now our first task is this: given the string s, build an array p.

To make the algorithm simpler, we will make some preparations. First, add the symbol `$` to the end of the string. This will be a special character that is smaller than all characters in the string. As a result, the symbol `$` will be added to each suffix. Let's notice that the order of strings has not changed, because `$` is smaller than all characters.

a	**b**	**a**	**b**	**b**	**a**	**$**
0	1	2	3	4	5	6

6	`$`
5	`a$`
0	`ababba$`
2	`abba$`
4	`ba$`
1	`babba$`
3	`bba$`
p	

Now let's make the length of all the strings the same. To do this, let's write after `$` all the other characters in the string in a cycle. Again, note that order of the strings has not changed, because the characters after `$` do not affect the string ordering:

a	**b**	**a**	**b**	**b**	**a**	**$**
0	1	2	3	4	5	6

p	
6	`$ababba`
5	`a$ababb`
0	`ababba$`
2	`abba$ab`
4	`ba$abab`
1	`babba$a`
3	`bba$aba`

Now the last preparation. Let's make the length of the strings the power of two, for this we add more characters in a cycle until the length of the string becomes a power of two. In our example, the string length is 7, so we need to add one character:

a	**b**	**a**	**b**	**b**	**a**	**$**
0	1	2	3	4	5	6

p	
6	`$ababba$`
5	`a$ababba`
0	`ababba$a`
2	`abba$aba`
4	`ba$ababb`
1	`babba$ab`
3	`bba$abab`

1.2 Algorithm idea

We will build this table by columns, each time increasing the number of columns twice. That is, first we construct the first column, then the first two, then four, and so on. In other words, we divide the algorithm into $\log n$ iterations, and on the iteration k we construct an array consisting of the strings $s[i..i + 2^k - 1]$, sorted in lexicographical order (we assume that the string is cyclic, that is, the character with index i corresponds to the character with index $i \mod n$ of the original string).

k=0	k=1	k=2	k=3
6 $	**6** $a	**6** $aba	**6** $ababba$
0 a	**5** a$	**5** a$ab	**5** a$ababba
2 a	**0** ab	**0** abab	**0** ababba$a
5 a	**2** ab	**2** abba	**2** abba$aba
1 b	**1** ba	**4** ba$a	**4** ba$ababb
3 b	**4** ba	**1** babb	**1** babba$ab
4 b	**3** bb	**3** bba$	**3** bba$abab

Let's start from the base, if $k = 0$, then we need to sort the strings of one character $s[i..i]$. This can be done by any sort in $\mathcal{O}(n \log n)$.

Now make the transition from k to $k + 1$. We need to sort the strings of length 2^{k+1}, using the order of the strings of length 2^k. To do this, let's learn how to quickly compare strings of length 2^{k+1}. Suppose we have two strings A and B, both have lengths 2^{k+1}. Let's divide both strings into two halves, then each half has a length 2^k.

Let's compare their left halves. If they are not equal, for example, if $A.left$ is less than $B.left$. This means that they have some common prefix, and after it comes a character that is not equal, and this character in the string A is less than in the string B. In this case, the entire string A is less than the string B, and the right halves should not be compared.

If the left halves are equal, then let's compare the right halves. Suppose, for example, $A.right$ is less than $B.right$. This again means that there is a common prefix, and then a character that is less in the string A than in the string B. Again we get that the string A is less than the string B.

Thus, we get that $A < B \iff A.left < B.left \vee (A.left = B.left \wedge A.right < B.right)$. If we learn to quickly compare strings of length 2^k, then we will make a comparator that quickly compares strings of length 2^{k+1}.

Now we need to learn how to compare strings of length 2^k quickly. We will do it as follows. Let's go through the strings of length 2^k in sorted order, and assign them integer numbers so that a smaller string corresponds to a smaller number. We call these numbers **equivalence classes**. Now, instead of comparing strings, we will compare their equivalence classes. Thus, we can compare strings of length 2^{k+1} in $O(1)$.

For example, for the string that we analyzed above, for $k = 1$ we get the following order of strings of length 2. Let's go through these strings assign them equivalence classes (array c).

6	`$a`	**0**
5	`a$`	**1**
0	`ab`	**2**
2	`ab`	**2**
1	`ba`	**3**
4	`ba`	**3**
3	`bb`	**4**
p		**c**

Now, if, for example, we want to compare the strings *abab* and *abba*. Divide them into two halves and write the equivalence classes for each of the halves, get the pair $(2, 2)$ for the string *abab*, and the pair $(2, 3)$ for the string *abba*. Now compare these pairs, we get that $(2, 2) < (2, 3)$, which means that $abab < abba$.

1.3 Algorithm

Thus, we get the following algorithm. First, we form strings of one character and sort them by any sort algorithm in $O(n \log n)$. Next, $\log n$ times we make the transition from k to $k + 1$. Each transition is done like this: take sorted strings of length 2^k, assign equivalence classes to them, then assign to each string of length 2^{k+1} a pair of numbers: equivalence classes of its halves, sort these pairs and get a sorted order for a string of length 2^{k+1}.

The time complexity this algorithm will be $O(n \log^2 n)$, because on at each of the $\log n$ iterations we do the sorting in $O(n \log n)$ time.

1.4 Improved algorithm

1.4.1 Using Radix sort

At the last step, we got the algorithm working in the time $O(n \log^2 n)$. To make the algorithm faster, we need to get rid of the sorting in $O(n \log n)$ on each transition. Let's notice that we need to sort the pairs of numbers (a_i, b_i), with each of the numbers a_i and b_i being integers in the range from 0 to $n - 1$. We can use the **radix sort** to sort these pairs.

1.4.2 Radix sort

Radix sort works as follows. We sort the pairs by the second element, and then sort them again using stable sorting algorithm by the first element. Since each time the key is an integer from 0 to $n - 1$, in both cases we can use **counting sort**.

1.4.3 Counting sort

Counting sort works as follows. We will calculate how many times each element occurs, after that we create the buckets of the required size for each element, and arrange the elements into buckets.

1.4.4 Conclusion

Thus, each sort will work in linear time, and the total time of the algorithm will be $O(n \log n)$. Thus, each sort will work in linear time, and the total time of the algorithm will be $O(n \log n)$.

1.4.5 Additional optimizations

To make the algorithm even a little faster and easier, the following fact can be noted. At iteration k we get sorted strings of length 2^k. If we add another 2^k characters to the beginning of each string, we get strings of length 2^{k+1}, sorted by right halves. After that, it will be enough to do only the second phase of radix sort, sorting the pairs by the first element.