MAXimal

date added: 22 Jun 2008 13:37
edited: 1 Oct 2012 12:46

Suffix array

Contents [hide]

Given a string $s[0 \ldots n-1]$ length n.

i-th **suffix** of string is called substring $s[i \ldots n-1]$, $i = 0 \ldots n-1$.

Then the **suffix array** of a string s is called a permutation of indexes of suffixes $p[0 \ldots n-1]$, $p[i] \in [0; n-$ which specifies the order of suffixes in lexicographic sort order. In other words, you need to sort all suffixes of the given string.

For example, for the row of the $s = abaab$ suffix array will be equal to:

$$(2, 3, 0, 4, 1)$$

Building for $O(n \log n)$

Strictly speaking, the following algorithm performs the sorting of the suffixes and **circular shifts** of the string. However, this algorithm is easy to obtain and the algorithm for sorting suffixes: it is sufficient to ascribe to the end of line character, which is certainly less than any character, which may consist of a string (for example, it may be a dollar or sharp; in the C language this purpose you can use the existing null character).

Immediately, we note that since we sorted cyclic shifts, then the substring we consider **cyclic**: substring $s[i \ldots j]$, when $i > j$understood as a substring $s[i \ldots n-1] + s[0 \ldots j]$. In addition, all the indices are taken modulo the length of the string (in order to simplify the formulas, I will omit explicitly taking the indices modulo).

The considered algorithm consists of some $\log n$ phases. On k-th phase ($k = 0 \ldots \lceil \log n \rceil$) sorted cyclic substrings of length 2^k. At last, the $\lceil \log n \rceil$first phase, will be sorted substrings of length $2^{\lceil \log n \rceil} > n$that is equivalent to the sorting cyclic shifts.

At each phase of the algorithm in addition to permutation $p[0 \ldots n-1]$ of the indexes of cyclic substrings will support for each of the cyclic substring starting at the position i length 2^k, **the number** $c[i]$ **of the equivalence class**that a substring belongs to. In fact, among the substrings can be identical, and the algorithm would need information about it. The hotel$c[i]$ the equivalence classes will give so that they are kept and information about the order: if the same suffix is less than the other, then the number of the class it needs to get smaller. Classes will, for convenience, numbered from zero. The number of equivalence classes will be stored in a variable $\mathrm{classes}$.

We give **an example**. Consider a string $s = aaba$. The values of the arrays $p[]$ and $c[]$ at each stage with zero at second are:

$$\begin{array}{lll} 0: & p = (0, 1, 3, 2) & c = (0, 0, 1, 0) \\ 1: & p = (0, 3, 1, 2) & c = (0, 1, 2, 0) \\ 2: & p = (3, 0, 1, 2) & c = (1, 2, 3, 0) \end{array}$$

It should be noted that the array $p[]$ of possible ambiguity. For example, at zero phase, the array would be equal to: $p = (3, 1, 0, 2)$. Which option will depend on the specific implementation of the algorithm, but all variants are equally valid. At the same time, in the mountains $c[]$ no ambiguities could not be.

We now turn to the construction **of the algorithm**. Input:

```
char *s; // input string
int n; // length of strings

// constants
const int maxlen = ...; // maximum string length
const int alphabet = 256; // size of the alphabet, <= maxlen
```

At **zero phase,** we need to sort cyclic substrings of length 1, i.e. individual characters of the string, and divide them into equivalence classes (just the same symbols will be assigned to the same equivalence class). This can be done, for example, trivial to sort by count. For each symbol, count how many times it was found. Then use this information to reconstruct the array $p[]$. After that, pass the array $p[]$ and compare characters, built the array $c[]$.

```
int p[maxlen], cnt[maxlen], c[maxlen];
memset (cnt, 0, alphabet * sizeof(int));
for (int i=0; i<n; ++i)
        ++cnt[s[i]];
for (int i=1; i<alphabet; ++i)
 cnt[i] += cnt[i-1];
for (int i=0; i<n; ++i)
 p[--cnt[s[i]]] = i;
c[p[0]] = 0;
int classes = 1;
for (int i=1; i<n; ++i) {
        if (s[p[i]] != s[p[i-1]])  ++classes;
 c[p[i]] = classes-1;
}
```

Further, let we have completed $k-1$the first phase (i.e. calculated values of the arrays $p[]$ and $c[]$ for her), now learn for $O(n)$ you to perform **the following, kth, phase**. Because phases only $O(\log n)$, it will give us the required algorithm with the time $O(n \log n)$.

To do this, note that a cyclic substring of length 2^k consists of two substrings of length 2^{k-1}that we can compare for $O(1)$using information from the previous phase — the numbers of $c[]$ the equivalence classes. Thus, for a substring of length 2^kstarting at position i, all necessary information is contained in a pair of numbers $(c[i], c[i+2^{k-1}])$ (again, we use the array $c[]$ from the previous phase).

$$\ldots \overbrace{\underbrace{s_i \ldots s_{i+2^{k-1}-1}}_{\text{length}=2^{k-1},\text{class}=c[i]} \quad \underbrace{s_{i+2^{k-1}} \ldots s_{i+2^k-1}}_{\text{length}=2^{k-1},\ \text{class}=c[i+2^{k-1}]}}^{\text{length}=2^k} \ldots \overbrace{\underbrace{s_j \ldots s_{j+2^{k-1}-1}}_{\text{length}=2^{k-1},\text{class}=c[j]} \quad \underbrace{s_{j+2^{k-1}} \ldots s_{j+2^k-1}}_{\text{length}=2^{k-1},\text{class}=c[j+2^{k-1}]}}^{\text{length}=2^k} \ldots$$

This gives us a very simple solution: **sort the** substrings of length 2^k just **at** those **pairs of numbers**, it will give us the required order, i.e. the array $p[]$. However, the ordinary sort, running over time $O(n \log n)$, we are not satisfied is will give the algorithm for constructing suffix array with time $O(n \log^2 n)$ (but this algorithm is somewhat easier to write than described below).

How to do this sort of pairs? Since the elements do not exceed par n, then you can sort by count. However, to achieve the best hidden in the asymptotic constants instead of sorting couples come to sort simple numbers.

We use here a technique, which is based on the so-called **digital sorting**: to sort the pairs, sort them first by the second elements, and first elements (but necessarily stable sorting, ie, does not violate the relative order of elements with equal). However, separate second parts are already ordered — this order is specified in the array $p[]$ from the previous phase. Then to order a pair for the second set of elements, you just have each element of the array $p[]$ to take 2^{k-1} — this will give us the sort order pairs on the second elements (after all, $p[]$ gives the ordering of the string2^{k-1} and when you go to line twice the length of these substrings become their second halves, so the position of the second half minus a length of the first half).

Thus, using only subtraction from the array elements $p[]$, we perform the sort on the second elements of pairs. Now it is necessary to produce a stable sort by the first elements of the pairs, it can be done $O(n)$, using counting sort.

It remains only to count the numbers $c[]$ of equivalence classes of, but they're easy to get, just passing on the resulting new permutation $p[]$ and comparing the adjacent elements (again, comparing pairs of two numbers).

We give **the implementation** the implementation of all phases of the algorithm, except for the zero. Introduced additional temporary arrays pn and cn (pn — a permutation in sorting order of the second elements of pairs cn — new numbers are the equivalence classes).

```
int pn[maxlen], cn[maxlen];
for (int h=0; (1<<h)<n; ++h) {
        for (int i=0; i<n; ++i) {
 pn[i] = p[i] - (1<<h);
                if (pn[i] < 0) pn[i] += n;
        }
        memset (cnt, 0, classes * sizeof(int));
        for (int i=0; i<n; ++i)
                ++cnt[c[pn[i]]];
        for (int i=1; i<classes; ++i)
 cnt[i] += cnt[i-1];
        for (int i=n-1; i>=0; --i)
 p[--cnt[c[pn[i]]]] = pn[i];
 cn[p[0]] = 0;
 classes = 1;
        for (int i=1; i<n; ++i) {
                int mid1 = (p[i] + (1<<h)) % n, mid2 = (p[i-1] + (1<<h)) % n;
                if (c[p[i]] != c[p[i-1]] || c[mid1] != c[mid2])
                        ++classes;
 cn[p[i]] = classes-1;
        }
        memcpy (c, cn, n * sizeof(int));
}
```

This algorithm requires $O(n \log n)$ time and $O(n)$ memory. However, if we take into account the size k of the alphabet, then the operating time is $O((n+k) \log n)$and the size of memory $O(n+k)$.

Applications

Finding the smallest cyclic shift of the row

The above algorithm sorts the cyclic shifts (if the line is not attributed to the dollar), and therefore $p[0]$ will give the desired position of the smallest cyclic shift. Working time — $O(n \log n)$.

Search of a substring in a string

Suppose that you want the text t to find the line s in online mode (ie pre the string s should be considered unknown). Construct the suffix array for text t for $O(|t| \log |t|)$. Now a sub-string s will look as follows: note that the search entry should be the prefix of any suffix t. Since the suffixes are ordered from us (this gives us the suffix array), then the substring s can be searched by binary search on the suffix of the string. Comparison of current and suffix substrings s within a binary search can be carried out trivially, for $O(|p|)$. Then the asymptotics of the search substring in the text becomes $O(|p| \log |t|)$.

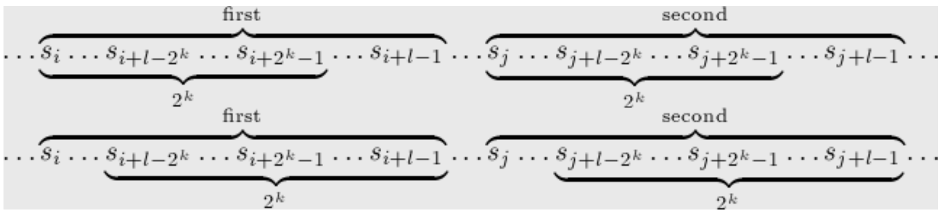
Comparison of two substrings of the string

Required string s, producing some of its preprocessing, learning for $O(1)$ responding to the comparison of any two substrings (i.e. checking that the first substring equal/less/greater than the second).

Construct the suffix array for $O(|s| \log |s|)$, keep deliverables: we need the arrays $c[]$ from each phase. Therefore, memory is also required $O(|s| \log |s|)$.

Using this information, we can $O(1)$ compare any two substrings of length equal to power of two: it is enough to compare the numbers of equivalence classes of the corresponding phase. Now generalize this method into substrings of arbitrary length.

Suppose now received another request for a comparison between two substrings of length l beginning at indices i and j. Find the largest block length that will fit inside the substring of that length, ie the largest k such that $2^k \le l$. Then the comparison of the two substrings can be replaced with a comparison of two pairs of overlapping blocks of length 2^k: first, we need to compare two blocks starting at positions i and j, and in case of equal — compare two blocks ending at positions $i+l-1$, and $j+l-1$:


first
second
$\ldots s_i \ldots s_{i+l-2^k} \ldots s_{i+2^k-1} \ldots s_{i+l-1} \ldots s_j \ldots s_{j+l-2^k} \ldots s_{j+2^k-1} \ldots s_{j+l-1} \ldots$
2^k
2^k
first
second
$\ldots s_i \ldots s_{i+l-2^k} \ldots s_{i+2^k-1} \ldots s_{i+l-1} \ldots s_j \ldots s_{j+l-2^k} \ldots s_{j+2^k-1} \ldots s_{j+l-1} \ldots$
2^k
2^k

Thus, the implementation turns out like this (here it is considered that the calling procedure itself calculates k, as to do this in constant time is not so easy (apparently the fastest — predpochla), but in any case it has nothing to do with the application of suffix array):

```
int compare (int i, int j, int l, int k) {
 pair<int,int> a = make_pair (c[k][i], c[k][i+l-(1<<k)]);
 pair<int,int> b = make_pair (c[k][j], c[k][j+l-(1<<k)]);
        return a == b ? 0 : a < b ? -1 : 1;
}
```

The greatest common prefix of two substrings: method with additional memory

Required string s, producing some of its preprocessing, learn for $O(\log |s|)$ to answer the queries of the largest common prefix (longest common prefix, lcp) for two arbitrary suffixes with the positions i and j.

The method described here requires $O(|s| \log |s|)$ additional memory; the other method that uses a linear amount of memory, but non-constant response time to the request that are described in the next section.

Construct the suffix array for $O(|s| \log |s|)$, keep deliverables: we need the arrays $c[]$ from each phase. Therefore, memory is also required $O(|s| \log |s|)$.

Suppose now received another request: a pair of indices i and j. Let's use the fact that we can $O(1)$ compare any two substrings of length that is a power of two. To do this, let's take the power of two (larger to smaller), and for the current extent check: if the substrings of this length match, then the answer is to add the powers of two and the greatest common prefix will continue to look to the right of the equal parts, i.e. to i and j it is necessary to add the current power of two.

Implementation:

```
int lcp (int i, int j) {
        int ans = 0;
        for (int k=log_n; k>=0; --k)
                if (c[k][i] == c[k][j]) {
 ans += 1<<k;
 i += 1<<k;
 j += 1<<k;
                }
        return ans;
}
```

Here in $\log\_n$ denotes the constant equal to the logarithm n base 2, rounded down.

The greatest common prefix of two substrings: method no additional memory. The greatest common prefix of two adjacent suffixes

Required string s, producing some of its preprocessing, learn to answer the queries of the largest common prefix (longest common prefix, lcp) for two arbitrary suffixes with the positions i and j.

Unlike the previous method described here is to perform a preprocessing of the string over $O(n \log n)$ time with $O(n)$ memory. The result of this preprocessing will be an array (which in itself is an important source of information about the line, and therefore be used for other tasks). Replies to the request will be made as the result of the query RMQ (minimum interval, the range minimum query) in this array, therefore, in various implementations, can be obtained as the logarithmic and constant time work.

The basis for this algorithm is the following idea: find some way the most common prefixes for each **adjacent in the sort order of pairs of suffixes**. In other words, build an array $\mathrm{lcp}[0 \ldots n-2]$where $\mathrm{lcp}[i]$ equal to the greatest common prefix of the suffixes $p[i]$ and $p[i+1]$. This array will give us the answer for any two adjacent suffix strings. Then the answer for any two suffixes, not necessarily adjacent, can be obtained by this array. In fact, even the request with some numbers suffixes i and j. We find these indices in the suffix array, i.e., let k_1 and k_2 their position in the array $p[]$ (sort them, i.e. let$k_1 < k_2$). The answer to this query will be low in the mountains lcp, taken on the segment $[k_1; k_2 - 1]$. In fact, the transition from suffix i to suffix j can replace a whole chain of transitions, starting with the suffix i and ending in the suffix j, but includes all the intermediate suffixes in the sort order between them.

Thus, if we have such array lcp, then the response to any request the largest common prefix is reduced to a request **of at least a segment** of the array lcp. This classical problem of minimum in the interval (range minimum query RMQ) has multiple solutions with different asymptotics described here.

So, our main task — **the construction** of this array lcp. To build it we will in the course of the algorithm for constructing suffix array: each current iteration will build the array lcp for cyclic substrings of current length.

After zero iteration, the array lcpmust obviously be zero.

Suppose now that we have completed $k-1$the first iteration received from her array lcp', and should the current k-th iteration to recalculate the array, getting its new value lcp. As we remember, in the algorithm for constructing suffix array cyclic substrings of length 2^k was divided in half into two substrings of length 2^{k-1}; we will use the same technique for constructing the array lcp.

So, let the current iteration of the algorithm for computing the suffix array did the job, found a new value of the permutation $p[]$ substrings. We now go through the array and look pair of adjacent substrings: $p[i]$ and $p[i+1]$, $i = 0 \ldots n-2$. Breaking each substring in half, we get two different situations: 1) the first half of substrings at the positions $p[i]$ and $p[i+1]$ differ, and 2) the first halves are the same (recall that such a comparison can easily be made by simply comparing the class numbers $c[]$ from the previous iteration). Let's consider each of these cases separately.

1) the First half of substrings differ. Note that when in the previous step, the first half must have been nearby. In fact, equivalence classes could not disappear (and can only appear), so all the different substrings of length 2^{k-1} will give (in first halves) in the current iteration of different substrings of length 2^k, and in the same order. Thus, to determine $\mathrm{lcp}[i]$ in this case you just have to take the corresponding value from the array lcp'.

2) the First halved match. Then the second half could both coincide and differ; in this case, if they differ, they do not necessarily have to be adjacent to the previous iteration. So in this case, there is no easy way to determine $\mathrm{lcp}[i]$. To determine it we must do the same as we are to compute a longest common prefix of any two suffixes: it is necessary to query at least (RMQ) on the relevant sector of the array lcp'.

Rate the **complexity** of this algorithm. As we saw in the analysis of these two cases, only the second case gives an increase in the number of equivalence classes. In other words, we can say that each new equivalence class appears with one query RMQ. Because of all these equivalence classes can be up to n, and look for the minimum we need for the asymptotics $O(\log n)$. And for this we need to use some data structure for low stretch; this data structure will have to be rebuilt on each iteration (which is just $O(\log n)$). A good choice of data structure is **a segment Tree**: it can be build $O(n)$, and then perform queries over $O(\log n)$that just gives us the total complexity $O(n \log n)$.

Implementation:

```
int lcp[maxlen], lcpn[maxlen], lpos[maxlen], rpos[maxlen];
memset (lcp, 0, sizeof lcp);
for (int h=0; (1<<h)<n; ++h) {
        for (int i=0; i<n; ++i)
 rpos[c[p[i]]] = i;
        for (int i=n-1; i>=0; --i)
 lpos[c[p[i]]] = i;

 ... all the steps to build a suffix. of the array, except the last lines (memcpy)...

 rmq_build (lcp, n-1);
        for (int i=0; i<n-1; ++i) {
                int a = p[i], b = p[i+1];
                if (c[a] != c[b])
 lcpn[i] = lcp[rpos[c[a]]];
                else {
                        int aa = (a + (1<<h)) % n, bb = (b + (1<<h)) % n;
 lcpn[i] = (1<<h) + rmq (lpos[c[aa]], rpos[c[bb]]-1);
 lcpn[i] = min (n, lcpn[i]);
                }
        }
        memcpy (lcp, lcpn, (n-1) * sizeof(int));

        memcpy (c, cn, n * sizeof(int));
}
```

Here, in addition to the array $\mathrm{lcp}[]$ introduces a temporary array $\mathrm{lcpn}[]$ to its new value. Also supported by the array $\mathrm{pos}[]$, which for each substring stores its position in the permutation $p[]$. A function $\mathrm{rmq\_build}$ is some function that builds a data structure for the low array to the first argument, it is passed as a second argument. The function rmq returns the minimum interval from the first argument by the second, inclusive.

The algorithm for constructing suffix array only had to make a copy of the array $c[]$, because during the computation lcp we need the old values of this array.

It is worth noting that our implementation finds the length common prefix for **the cyclic substrings**, while in practice often need the length of the common prefix for the suffixes in their usual meaning. In this case, you just have to restrict the values lcp at the end of the algorithm:

```
for (int i=0; i<n-1; ++i)
 lcp[i] = min (lcp[i], min (n-p[i], n-p[i+1]));
```

For **any** two suffixes of the length of their longest common prefix can now be found as a minimum on the relevant sector of the array lcp:

```
for (int i=0; i<n; ++i)
 pos[p[i]] = i;
rmq_build (lcp, n-1);
```

```
... received a request (i,j) on finding a LCP ...
int result = rmq (min(i,j)max(i,j)-1);
```

The number of different substrings

Perform the **preprocessing**described in the previous section: $O(n \log n)$ time and $O(n)$ memory we have for each pair of adjacent in the sort order of the suffixes, find the length of their longest common prefix. Find now with this information, the number of different substrings in the string.

To this end we will consider what a new substring starting at position $p[0]$, then the position $p[1]$, etc. in fact, we take another in the sort order suffix and see what it prefixes give a new substring. Thereby we obviously don't lose sight of any of the substrings.

Using the fact that the suffixes we have already sorted, it is easy to understand that the current suffix $p[i]$ will give as new substrings all their prefixes, in addition to matching the prefix of suffix $p[i-1]$. I.e., all its prefixes, except $\mathrm{lcp}[i-1]$ the first, will give a new substring. Since the length of the current suffix is equal $n - p[i]$, finally, we find that the current suffix $p[i]$ gives $n - p[i] - \mathrm{lcp}[i-1]$ new substrings. Summing this for all suffixes (for the very first, $p[0]$to take away nothing — just added $n - p[0]$), we obtain **the answer** to the problem:

$$\sum_{i=0}^{n} (n - p[i]) - \sum_{i=0}^{n-1} \mathrm{lcp}[i]$$

Tasks in online judges

Tasks that can be solved using suffix array:

- UVA #10679 **"I Love Strings!!!"** [difficulty: medium]

0 Комментариев e-maxx 1 Войти

Рекомендовать Поделиться Лучшее в начале

Начать обсуждение...

ВОЙТИ С ПОМОЩЬЮ

ИЛИ ЧЕРЕЗ DISQUS ?

Имя

Прокомментируйте первым.

ТАКЖЕ НА E-MAXX

MAXimal :: algo :: Дерево отрезков

2 комментариев • 4 года назад•

Иван Жаров — ты школьник

Поиск общих касательных к двум окружностям

1 комментарий • 4 года назад•

Алексей Никитин — К чему относится случай, когда окружности пересекаются? Вырожденный, но допустимый для алгоритма?Ну и, если у алгоритма

Правильные скобочные последовательности

MAXimal :: algo :: Поиск в глубину и его