Last update: August 2, 2024 Translated From: e-maxx.ru

# Suffix Array

## Definition

Let $s$ be a string of length $n$. The $i$-th suffix of $s$ is the substring $s[i \dots n-1]$.

A **suffix array** will contain integers that represent the **starting indexes** of the all the suffixes of a given string, after the aforementioned suffixes are sorted.

As an example look at the string $s = abaab$. All suffixes are as follows

0. $abaab$
1. $baab$
2. $aab$
3. $ab$
4. $b$

After sorting these strings:

2. $aab$
3. $ab$
0. $abaab$
4. $b$
1. $baab$

Therefore the suffix array for $s$ will be $(2,\ 3,\ 0,\ 4,\ 1)$.

As a data structure it is widely used in areas such as data compression, bioinformatics and, in general, in any area that deals with strings and string matching problems.

## Construction

### $O(n^2 \log n)$ approach

This is the most naive approach. Get all the suffixes and sort them using quicksort or mergesort and simultaneously retain their original indices. Sorting uses $O(n \log n)$ comparisons, and since comparing two strings will additionally take $O(n)$ time, we get the final complexity of $O(n^2 \log n)$.

### $O(n \log n)$ approach

Strictly speaking the following algorithm will not sort the suffixes, but rather the cyclic shifts of a string. However we can very easily derive an algorithm for sorting suffixes from it: it is enough to append an arbitrary character to the end of the string which is smaller than any character from the string. It is common to use the symbol \$. Then the order of the sorted cyclic shifts is equivalent to the order of the sorted suffixes, as demonstrated here with the string $dabbb$.

$$\begin{array}{ll} 1. & abbb\$d \quad abbb \\ 4. & b\$dabb \quad b \\ 3. & bb\$dab \quad bb \\ 2. & bbb\$da \quad bbb \\ 0. & dabbb\$ \quad dabbb \end{array}$$

Since we are going to sort cyclic shifts, we will consider **cyclic substrings**. We will use the notation $s[i \dots j]$ for the substring of $s$ even if $i > j$. In this case we actually mean the string $s[i \dots n-1] + s[0 \dots j]$. In addition we will take all indices modulo the length of $s$, and will omit the modulo operation for simplicity.

The algorithm we discuss will perform $\lceil \log n \rceil + 1$ iterations. In the $k$-th iteration ($k = 0 \dots \lceil \log n \rceil$) we sort the $n$ cyclic substrings of $s$ of length $2^k$. After the $\lceil \log n \rceil$-th iteration the substrings of length $2^{\lceil \log n \rceil} \geq n$ will be sorted, so this is equivalent to sorting the cyclic shifts altogether.

In each iteration of the algorithm, in addition to the permutation $p[0 \dots n-1]$, where $p[i]$ is the index of the $i$-th substring (starting at $i$ and with length $2^k$) in the sorted order, we will also maintain an array $c[0 \dots n-1]$, where $c[i]$ corresponds to the **equivalence class** to which the substring belongs. Because some of the substrings will be identical, and the algorithm needs to treat them equally. For convenience the classes will be labeled by numbers started from zero. In addition the numbers $c[i]$ will be assigned in such a way that they preserve information about the order: if one substring is smaller than the other, then it should also have a smaller class label. The number of equivalence classes will be stored in a variable $\text{classes}$.

Let's look at an example. Consider the string $s = aaba$. The cyclic substrings and the corresponding arrays $p[]$ and $c[]$ are given for each iteration:

$$\begin{array}{lcll} 0: & (a,\ a,\ b,\ a) & p = (0,\ 1,\ 3,\ 2) & c = (0,\ 0,\ 1,\ 0) \\ 1: & (aa,\ ab,\ ba,\ aa) & p = (0,\ 3,\ 1,\ 2) & c = (0,\ 1,\ 2,\ 0) \\ 2: & (aaba,\ abaa,\ baaa,\ aaab) & p = (3,\ 0,\ 1,\ 2) & c = (1,\ 2,\ 3,\ 0) \end{array}$$

It is worth noting that the values of $p[]$ can be different. For example in the $0$-th iteration the array could also be $p = (3,\ 1,\ 0,\ 2)$ or $p = (3,\ 0,\ 1,\ 2)$. All these options permutation the substrings into a sorted order. So they are all valid. At the same time the array $c[]$ is fixed, there can be no ambiguities.

Let us now focus on the implementation of the algorithm. We will write a function that takes a string $s$ and returns the permutations of the sorted cyclic shifts.

```
vector<int> sort_cyclic_shifts(string const& s) {
    int n = s.size();
    const int alphabet = 256;
```

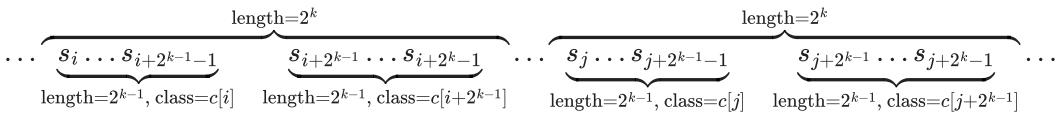
At the beginning (in the $0$**-th iteration**) we must sort the cyclic substrings of length $1$, that is we have to sort all characters of the string and divide them into equivalence classes (same symbols get assigned to the same class). This can be done trivially, for example, by using **counting sort**. For each character we count how many times it

appears in the string, and then use this information to create the array $p[]$. After that we go through the array $p[]$ and construct $c[]$ by comparing adjacent characters.

```
vector<int> p(n), c(n), cnt(max(alphabet, n), 0);
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;
}
```

Now we have to talk about the iteration step. Let's assume we have already performed the $k-1$-th step and computed the values of the arrays $p[]$ and $c[]$ for it. We want to compute the values for the $k$-th step in $O(n)$ time. Since we perform this step $O(\log n)$ times, the complete algorithm will have a time complexity of $O(n \log n)$.

To do this, note that the cyclic substrings of length $2^k$ consists of two substrings of length $2^{k-1}$ which we can compare with each other in $O(1)$ using the information from the previous phase - the values of the equivalence classes $c[]$. Thus, for two substrings of length $2^k$ starting at position $i$ and $j$, all necessary information to compare them is contained in the pairs $(c[i],\ c[i+2^{k-1}])$ and $(c[j],\ c[j+2^{k-1}])$.




This gives us a very simple solution: **sort** the substrings of length $2^k$ **by these pairs of numbers**. This will give us the required order $p[]$. However a normal sort runs in $O(n \log n)$ time, with which we are not satisfied. This will only give us an algorithm for constructing a suffix array in $O(n \log^2 n)$ times.

How do we quickly perform such a sorting of the pairs? Since the elements of the pairs do not exceed $n$, we can use counting sort again. However sorting pairs with counting sort is not the most efficient. To achieve a better hidden constant in the complexity, we will use another trick.

We use here the technique on which **radix sort** is based: to sort the pairs we first sort them by the second element, and then by the first element (with a stable sort, i.e. sorting without breaking the relative order of equal elements). However the second elements were already sorted in the previous iteration. Thus, in order to sort the pairs by the second elements, we just need to subtract $2^{k-1}$ from the indices in $p[]$ (e.g. if the smallest substring of length $2^{k-1}$ starts at position $i$, then the substring of length $2^k$ with the smallest second half starts at $i - 2^{k-1}$).

So only by simple subtractions we can sort the second elements of the pairs in $p[]$. Now we need to perform a stable sort by the first elements. As already mentioned, this can be accomplished with counting sort.

The only thing left is to compute the equivalence classes $c[]$, but as before this can be done by simply iterating over the sorted permutation $p[]$ and comparing neighboring pairs.

Here is the remaining implementation. We use temporary arrays $pn[]$ and $cn[]$ to store the permutation by the second elements and the new equivalent class indices.

```
    vector<int> pn(n), cn(n);
    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;
        }
        fill(cnt.begin(), cnt.begin() + classes, 0);
        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++) {
            pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
            pair<int, int> prev = {c[p[i-1]], c[(p[i-1] + (1 << h)) % n]};
            if (cur != prev)
                ++classes;
            cn[p[i]] = classes - 1;
        }
        c.swap(cn);
    }
    return p;
}
```

The algorithm requires $O(n \log n)$ time and $O(n)$ memory. For simplicity we used the complete ASCII range as alphabet.

If it is known that the string only contains a subset of characters, e.g. only lowercase letters, then the implementation can be optimized, but the optimization factor would likely be insignificant, as the size of the alphabet only matters on the first iteration. Every other iteration depends on the number of equivalence classes, which may quickly reach $O(n)$ even if initially it was a string over the alphabet of size $2$.

Also note, that this algorithm only sorts the cycle shifts. As mentioned at the beginning of this section we can generate the sorted order of the suffixes by appending a character that is smaller than all other characters of the string, and sorting this resulting string by cycle shifts, e.g. by sorting the cycle shifts of $s + \$$. This will obviously give the suffix array of $s$, however prepended with $|s|$.

```
vector<int> suffix_array_construction(string s) {
    s += "$";
    vector<int> sorted_shifts = sort_cyclic_shifts(s);
    sorted_shifts.erase(sorted_shifts.begin());
    return sorted_shifts;
}
```

## Applications

### Finding the smallest cyclic shift

The algorithm above sorts all cyclic shifts (without appending a character to the string), and therefore $p[0]$ gives the position of the smallest cyclic shift.

## Finding a substring in a string

The task is to find a string $s$ inside some text $t$ online - we know the text $t$ beforehand, but not the string $s$. We can create the suffix array for the text $t$ in $O(|t| \log |t|)$ time. Now we can look for the substring $s$ in the following way. The occurrence of $s$ must be a prefix of some suffix from $t$. Since we sorted all the suffixes we can perform a binary search for $s$ in $p$. Comparing the current suffix and the substring $s$ within the binary search can be done in $O(|s|)$ time, therefore the complexity for finding the substring is $O(|s| \log |t|)$. Also notice that if the substring occurs multiple times in $t$, then all occurrences will be next to each other in $p$. Therefore the number of occurrences can be found with a second binary search, and all occurrences can be printed easily.

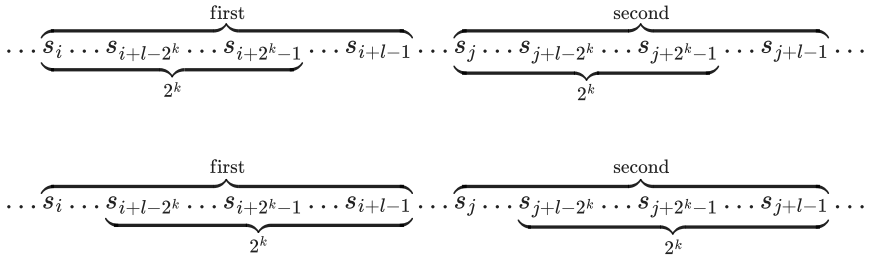
## Comparing two substrings of a string

We want to be able to compare two substrings of the same length of a given string $s$ in $O(1)$ time, i.e. checking if the first substring is smaller than the second one.

For this we construct the suffix array in $O(|s| \log |s|)$ time and store all the intermediate results of the equivalence classes $c[]$.

Using this information we can compare any two substring whose length is equal to a power of two in O(1): for this it is sufficient to compare the equivalence classes of both substrings. Now we want to generalize this method to substrings of arbitrary length.

Let's compare two substrings of length $l$ with the starting indices $i$ and $j$. We find the largest length of a block that is placed inside a substring of this length: the greatest $k$ such that $2^k \leq l$. Then comparing the two substrings can be replaced by comparing two overlapping blocks of length $2^k$: first you need to compare the two blocks starting at $i$ and $j$, and if these are equal then compare the two blocks ending in positions $i + l - 1$ and $j + l - 1$:




Here is the implementation of the comparison. Note that it is assumed that the function gets called with the already calculated $k$. $k$ can be computed with $\lfloor \log l \rfloor$, but it is more efficient to precompute all $k$ values for every $l$. See for instance the article about the Sparse Table, which uses a similar idea and computes all $\log$ values.

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

### Longest common prefix of two substrings with additional memory

For a given string $s$ we want to compute the longest common prefix (**LCP**) of two arbitrary suffixes with position $i$ and $j$.

The method described here uses $O(|s| \log |s|)$ additional memory. A completely different approach that will only use a linear amount of memory is described in the next section.

We construct the suffix array in $O(|s| \log |s|)$ time, and remember the intermediate results of the arrays $c[]$ from each iteration.

Let's compute the LCP for two suffixes starting at $i$ and $j$. We can compare any two substrings with a length equal to a power of two in $O(1)$. To do this, we compare the strings by power of twos (from highest to lowest power) and if the substrings of this length are the same, then we add the equal length to the answer and continue checking for the LCP to the right of the equal part, i.e. $i$ and $j$ get added by the current power of two.

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

Here `log_n` denotes a constant that is equal to the logarithm of $n$ in base $2$ rounded down.

### Longest common prefix of two substrings without additional memory

We have the same task as in the previous section. We have compute the longest common prefix (**LCP**) for two suffixes of a string $s$.

Unlike the previous method this one will only use $O(|s|)$ memory. The result of the preprocessing will be an array (which itself is an important source of information about the string, and therefore also used to solve other tasks). LCP queries can be answered by performing RMQ queries (range minimum queries) in this array, so for different implementations it is possible to achieve logarithmic and even constant query time.

The basis for this algorithm is the following idea: we will compute the longest common prefix for each **pair of adjacent suffixes in the sorted order**. In other words we construct an array $\text{lcp}[0 \dots n-2]$, where $\text{lcp}[i]$ is equal to the length of the longest common prefix of the suffixes starting at $p[i]$ and $p[i+1]$. This array will give us an answer for any two adjacent suffixes of the string. Then the answer for arbitrary two suffixes, not necessarily neighboring ones, can be obtained from this array. In fact, let the request be to compute the LCP of the suffixes $p[i]$ and $p[j]$. Then the answer to this query will be $\min(lcp[i],\ lcp[i+1],\ \dots,\ lcp[j-1])$.

Thus if we have such an array $\text{lcp}$, then the problem is reduced to the RMQ, which has many wide number of different solutions with different complexities.

So the main task is to **build** this array $\text{lcp}$. We will use **Kasai's algorithm**, which can compute this array in $O(n)$ time.

Let's look at two adjacent suffixes in the sorted order (order of the suffix array). Let their starting positions be $i$ and $j$ and their $\text{lcp}$ equal to $k > 0$. If we remove the first letter of both suffixes - i.e. we take the suffixes $i + 1$ and $j + 1$ - then it should be obvious that the $\text{lcp}$ of these two is $k - 1$. However we cannot use this value and write it in the $\text{lcp}$ array, because these two suffixes might not be next to each other in the sorted order. The suffix $i + 1$ will of course be smaller than the suffix $j + 1$, but there might be some suffixes between them. However, since we know that the LCP between two suffixes is the minimum value of all transitions, we also know that the LCP between any two pairs in that interval has to be at least $k - 1$, especially also between $i + 1$ and the next suffix. And possibly it can be bigger.

Now we already can implement the algorithm. We will iterate over the suffixes in order of their length. This way we can reuse the last value $k$, since going from suffix $i$ to the suffix $i + 1$ is exactly the same as removing the first letter. We will need an additional array $\text{rank}$, which will give us the position of a suffix in the sorted list of suffixes.

```
vector<int> lcp_construction(string const& s, vector<int> const& p) {
    int n = s.size();
    vector<int> rank(n, 0);
    for (int i = 0; i < n; i++)
        rank[p[i]] = i;

    int k = 0;
    vector<int> lcp(n-1, 0);
    for (int i = 0; i < n; i++) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        int j = p[rank[i] + 1];
        while (i + k < n && j + k < n && s[i+k] == s[j+k])
            k++;
        lcp[rank[i]] = k;
        if (k)
            k--;
    }
    return lcp;
}
```

It is easy to see, that we decrease $k$ at most $O(n)$ times (each iteration at most once, except for $\text{rank}[i] == n - 1$, where we directly reset it to $0$), and the LCP between two strings is at most $n - 1$, we will also increase $k$ only $O(n)$ times. Therefore the algorithm runs in $O(n)$ time.

## Number of different substrings

We preprocess the string $s$ by computing the suffix array and the LCP array. Using this information we can compute the number of different substrings in the string.

To do this, we will think about which **new** substrings begin at position $p[0]$, then at $p[1]$, etc. In fact we take the suffixes in sorted order and see what prefixes give new substrings. Thus we will not overlook any by accident.

Because the suffixes are sorted, it is clear that the current suffix $p[i]$ will give new substrings for all its prefixes, except for the prefixes that coincide with the suffix $p[i - 1]$. Thus, all its prefixes except the first $\text{lcp}[i - 1]$ one. Since the length of the current suffix is $n - p[i]$, $n - p[i] - \text{lcp}[i - 1]$ new prefixes start at $p[i]$. Summing over all the suffixes, we get the final answer: