字符串英文合集

1.AC自动机 Aho-Corasick algorithm

Let there be a set of strings with the total length m (sum of all lengths). The Aho-Corasick algorithm constructs a data structure similar to a trie with some additional links, and then constructs a finite state machine (automaton) in $O(mk)$ time, where k is the size of the used alphabet.

The algorithm was proposed by Alfred Aho and Margaret Corasick in 1975.

Construction of the trie

Formally a trie is a rooted tree, where each edge of the tree is labeled by some letter. All outgoing edge from one vertex mush have different labels.

Consider any path in the trie from the root to any vertex. If we write out the labels of all edges on the path, we get a string that corresponds to this path. For any vertex in the trie we will associate the string from the root to the vertex.

Each vertex will also have a flag leaf which will be true, if any string from the given set corresponds to this vertex.

Accordingly to build a trie for a set of strings means to build a trie such that each leaf vertex will correspond to one string from the set, and conversely that each string of the set corresponds to one leaf vertex.

We now describe how to construct a trie for a given set of strings in linear time with respect to their total length.

We introduce a structure for the vertices of the tree.

```
const int K = 26;

struct Vertex {
    int next[K];
    bool leaf = false;

    Vertex() {
        fill(begin(next), end(next), -1);
    }
};

vector<Vertex> trie(1);
```

Here we store the trie as an array of Vertex. Each Vertex contains the flag leaf, and the edges in the form of ans array $\text{next}[]$, where $\text{next}[i]$ is the index to the vertex that we reach by following the character i, or -1, if there is no such edge. Initially the trie consists of only one vertex - the root - with the index 0.

Now we implement a function that will add a string s to the trie. The implementation is extremely simple: we start at the root node, and as long as there are edges corresponding to the characters of s we follow them. If there is no edge for one character, we simply generate a new vertex and connect it via an edge. At the end of the process we mark the last vertex with flag leaf.

```
void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (trie[v].next[c] == -1) {
            trie[v].next[c] = trie.size();
            trie.emplace_back();
        }
        v = trie[v].next[c];
    }
    trie[v].leaf = true;
}
```

The implementation obviously runs in linear time. And since every vertex store k links, it will use $O(mk)$ memory.

It is possible to decrease the memory consumption to $O(m)$ by using a map instead of an array in each vertex. However this will increase the complexity to $O(n \log k)$.

Construction of an automaton

Suppose we have built a trie for the given set of strings. Now let's look at it from a different side. If we look at any vertex. The string that corresponds to it is a prefix of one or more strings in the set, thus each vertex of the trie can be interpreted as a position in one or more strings from the set.

In fact the trie vertices can be interpreted as states in a **finite deterministic automaton**. From any state we can transition - using some input letter - to other states, i.e. to another position in the set of strings. For example, if there is only one string in the trie abc, and we are standing at vertex 2 (which corresponds to the string ab), then using the letter c we can transition to the state 3.

Thus we can understand the edges of the trie as transitions in an automaton according to the corresponding letter. However for an automaton we cannot restrict the possible transitions for each state. If we try to perform a transition using a letter, and there is no corresponding edge in the trie, then we nevertheless must go into some state.

More strictly, let us be in a state p corresponding to the string t, and we want to transition to a different state with the character c. If there is an edge labeled with this letter c, then we can simply go over this edge, and get the vertex corresponding to $t + c$. If there is no such edge, then we must find the state corresponding to the longest proper suffix of the string t (the longest available in the trie), and try to perform a transition via c from there.

For example let the trie be constructed by the strings ab and bc, and we are currently at the vertex corresponding to ab, which is a leaf. For a transition with the letter c, we are forced to go to the state corresponding to the string b, and from there follow the edge with the letter c.

A **suffix link** for a vertex p is a edge that points to the longest proper suffix of the string corresponding to the vertex p. The only special case is the root of the trie, the suffix link will point to itself. Now we can reformulate the statement about the transitions in the automaton like this: while from the current vertex of the trie there is no transition using the current letter (or until we

reach the root), we follow the suffix link.

Thus we reduced the problem of constructing an automaton to the problem of finding suffix links for all vertices of the trie. However we will build these suffix links, oddly enough, using the transitions constructed in the automaton.

Note that if we want to find a suffix link for some vertex v, then we can go to the ancestor p of the current vertex (let c be the letter of the edge from p to v), then follow its suffix link, and perform from there the transition with the letter c.

Thus the problem of finding the transitions has been reduced to the problem of finding suffix links, and the problem of finding suffix links has been reduced to the problem of finding a suffix link and a transition, but for vertices closer to the root. So we have a recursive dependence that we can resolve in linear time.

Let's move to the implementation. Note that we now will store the ancestor p and the character pch of the edge from p to v for each vertex v. Also at each vertex we will store the suffix link link (or -1 if it hasn't been calculated yet), and in the array $\text{go}[k]$ the transitions in the machine for each symbol (again -1 if it hasn't been calculated yet).

```
const int K = 26;

struct Vertex {
    int next[K];
    bool leaf = false;
    int p = -1;
    char pch;
    int link = -1;
    int go[K];

    Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
        fill(begin(next), end(next), -1);
        fill(begin(go), end(go), -1);
    }
};

vector<Vertex> t(1);

void add_string(string const& s) {
    int v = 0;
    for (char ch : s) {
        int c = ch - 'a';
        if (t[v].next[c] == -1) {
            t[v].next[c] = t.size();
            t.emplace_back(v, ch);
        }
        v = t[v].next[c];
    }
    t[v].leaf = true;
}

int go(int v, char ch);

int get_link(int v) {
    if (t[v].link == -1) {
        if (v == 0 || t[v].p == 0)
            t[v].link = 0;
```

```
        else
            t[v].link = go(get_link(t[v].p), t[v].pch);
    }
    return t[v].link;
}

int go(int v, char ch) {
    int c = ch - 'a';
    if (t[v].go[c] == -1) {
        if (t[v].next[c] != -1)
            t[v].go[c] = t[v].next[c];
        else
            t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
    }
    return t[v].go[c];
}
```

It is easy to see, that due to the memorization of the found suffix links and transitions the total time for finding all suffix links and transitions will be linear.

Applications

Find all strings from a given set in a text

Given a set of strings and a text. We have to print all occurrences of all strings from the set in the given text in $O(\text{len} + \text{ans})$, where len is the length of the text and ans is the size of the answer.

We construct an automaton for this set of strings. We will now process the text letter by letter, transitioning during the different states. Initially we are at the root of the trie. If we are at any time at state v, and the next letter is c, then we transition to the next state with $\text{go}(v, c)$, thereby either increasing the length of the current match substring by 1, or decreasing it by following a suffix link.

How can we find out for a state v, if there are any matches with strings for the set? First, it is clear that if we stand on a leaf vertex, then the string corresponding to the vertex ends at this position in the text. However this is by no means the only possible case of achieving a match: if we can reach one or more leaf vertices by moving along the suffix links, then there will be also a match corresponding to each found leaf vertex. A simple example demonstrating this situation can be creating using the set of strings $dabce$, abc, bc and the text $dabc$.

Thus if we store in each leaf vertex the index of the string corresponding to it (or the list of indices if duplicate strings appear in the set), then we can find in $O(n)$ time the indices of all strings which match the current state, by simply following the suffix links from the current vertex to the root. However this is not the most efficient solution, since this gives us $O(n \text{ len})$ complexity in total. However this can be optimized by computing and storing the nearest leaf vertex that is reachable using suffix links (this is sometimes called the **exit link**). This value we can compute lazily in linear time. Thus for each vertex we can advance in $O(1)$ time to the next marked vertex in the suffix link path, i.e. to the next match. Thus for each match we spend $O(1)$ time, and therefore we reach the complexity $O(\text{len} + \text{ans})$.

If you only want to count the occurrences and not find the indices themselves, you can calculate the number of marked vertices in the suffix link path for each vertex v. This can be calculated in $O(n)$ time in total. Thus we can sum up all matches in $O(\text{len})$.

Finding the lexicographical smallest string of a given length that doesn't match any given strings

A set of strings and a length L is given. We have to find a string of length L, which does not contain any of the string, and derive the lexicographical smallest of such strings.

We can construct the automaton for the set of strings. Let's remember, that the vertices from which we can reach a leaf vertex are the states, at which we have a match with a string from the set. Since in this task we have to avoid matches, we are not allowed to enter such states. On the other hand we can enter all other vertices. Thus we delete all "bad" vertices from the machine, and in the remaining graph of the automaton we find the lexicographical smallest path of length L. This task can be solved in $O(L)$ for example by depth first search.

Finding the shortest string containing all given strings

Here we use the same ideas. For each vertex we store a mask that denotes the strings which match at this state. Then the problem can be reformulated as follows: initially being in the state $(v = \text{root},\ \text{mask} = 0)$, we want to reach the state $(v,\ \text{mask} = 2^n - 1)$, where n is the number of strings in the set. When we transition from one state to another using a letter, we update the mask accordingly. By running a breath first search we can find a path to the state $(v,\ \text{mask} = 2^n - 1)$ with the smallest length.

Finding the lexicographical smallest string of length L containing k strings

As in the previous problem, we calculate for each vertex the number of matches that correspond to it (that is the number of marked vertices reachable using suffix links). We reformulate the problem: the current state is determined by a triple of numbers $(v,\ \text{len},\ \text{cnt})$, and we want to reach from the state $(\text{root},\ 0,\ 0)$ the state $(v,\ L,\ k)$, where v can be any vertex. Thus we can find such a path using depth first search (and if the search looks at the edges in their natural order, then the found path will automatically be the lexicographical smallest).

2.后缀数组 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
4. $abaab$
5. b
6. $baab$

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 $dabbb$.

1. $abbb\$d$ $abbb$
2. $b\$dabb$ b
3. $bb\$dab$ bb
4. $bbb\$da$ bbb
5. $dabbb\$$ $dabbb$

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 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:

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

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.

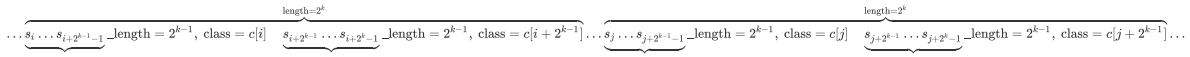
```
vector<int> sort_cyclic_shifts(string const& s) {
    int n = s.size();
    const int alphabet = 256;
```

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. However if we take the size of the alphabet k into account, then it uses $O((n+k) \log n)$ time and $O(n+k)$ memory.

For simplicity we used the complete ASCII range as alphabet. If we know that the string only contains a subset of characters, e.g. only lowercase letters, then this implementation can obviously be optimized. However not by much, since the alphabet size only appears with a factor of $O(\log n)$ in the complexity.

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 $j+l-1$:

$$\ldots \overbrace{s_i \ldots s_{i+l-2^k} \ldots s_{i+2^k-1}}^{\text{first}} \underbrace{\_2^k \ldots s_{i+l-1}} \ldots \overbrace{s_j \ldots s_{j+l-2^k} \ldots s_{j+2^k-1}}^{\text{second}} \underbrace{\_2^k \ldots s_{j+l-1}} \ldots$$

$$\ldots \overbrace{s_i \ldots s_{i+l-2^k} \ldots s_{i+2^k-1} \ldots s_{i+l-1}}^{\text{first}} \underbrace{\_2^k} \ldots \overbrace{s_j \ldots s_{j+l-2^k} \ldots s_{j+2^k-1} \ldots s_{j+l-1}}^{\text{second}} \underbrace{\_2^k} \ldots$$

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] == c[k][j]) {
            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 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 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 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 lcp of these two is $k-1$. However we cannot use this value and write it in the 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. Will will need an additional array 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 suffixes start at $p[i]$. Summing over all the suffixes, we get the final answer: $\sum_{i=0}^{n-1}(n-p[i]) - \sum_{i=0}^{n-2}\text{lcp}[i] = \frac{n^2+n}{2} - \sum_{i=0}^{n-2}\text{lcp}[i]$

3.后缀自动机 Suffix Automaton

A **suffix automaton** is a powerful data structure that allows solving many string-related problems.

For example, you can search for all occurrences of one string in another, or count the amount of different substrings of a given string. Both tasks can be solved in linear time with the help of a suffix automaton.

Intuitively a suffix automaton can be understood as compressed form of **all substrings** of a given string. An impressive fact is, that the suffix automaton contains all this information in a highly compressed form. For a string of length n it only requires $O(n)$ memory. Moreover, it can also be build in $O(n)$ time (if we consider the size k of the alphabet as a constant), otherwise both the memory and the time complexity will be $O(n \log k)$.

The linearity of the size of the suffix automaton was first discovered in 1983 by Blumer et al., and in 1985 the first linear algorithms for the construction was presented by Crochemore and Blumer.

Definition of a suffix automaton

A suffix automaton for a given string s is a minimal **DFA** (deterministic finite automaton / deterministic finite state machine) that accepts all the suffixes of the string s.

In other words:

- A suffix automaton is an oriented acyclic graph. The vertices are called **states**, and the edges are called **transitions** between states.

- One of the states t_0 is the **initial state**, and it must be the source of the graph (all other states are reachable from t_0).
- Each **transition** is labeled with some character. All transitions originating from a state must have **different** labels.
- One or multiple states are marked as **terminal states**. If we start from the initial state t_0 and move along transitions to a terminal state, then the labels of the passed transitions must spell one of the suffixes of the string s. Each of the suffixes of s must be spellable using a path from t_0 to a terminal state.
- The suffix automaton contains the minimum number of vertices among all automata satisfying the conditions described above.

Substring property

The simplest and most important property of a suffix automaton is, that it contains information about all substrings of the string s. Any path starting at the initial state t_0, if we write down the labels of the transitions, forms a **substring** of s. And conversely every substring of s corresponds to a certain path starting at t_0.

In order to simplify the explanations, we will say that the substrings **corresponds** to that path (starting at t_0 and the labels spell the substring). And conversely we say that any path **corresponds** to the string spelled by its labels.

One or multiple paths can lead to a state. Thus, we will say that a state **corresponds** to the set of strings, which correspond to these paths.

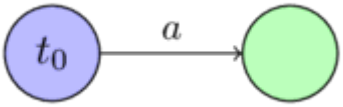
Examples of constructed suffix automata

Here we will show some examples of suffix automata for several simple strings.

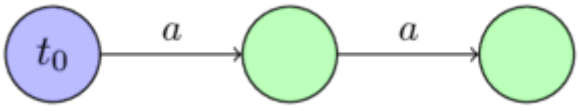
We will denote the initial state with blue and the terminal states with green.

For the string $s = $ "":

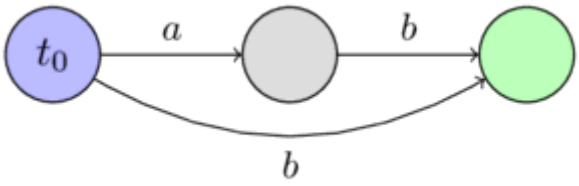
For the string $s = $ "a":



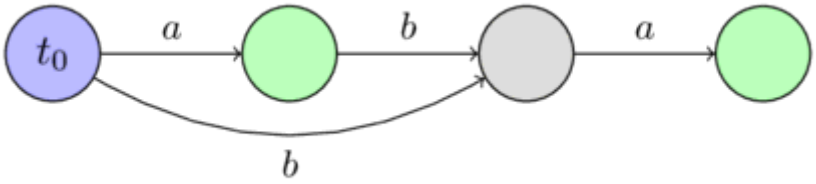
For the string $s = $ "aa":



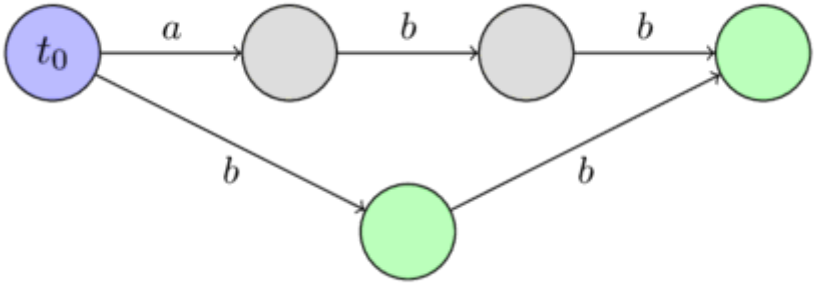
For the string $s = $ "ab":



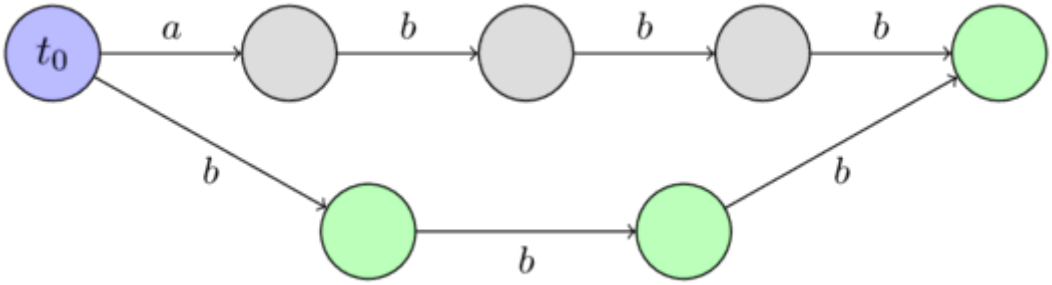
For the string $s =$ " aba ":



For the string $s =$ " abb ":



For the string $s =$ " $abbb$ ":



Construction in linear time

Before we describe the algorithm to construct a suffix automaton in linear time, we need to introduce several new concepts and simple proofs, which will be very important in understanding the construction.

End positions $endpos$

Consider any non-empty substring t of the string s. We will denote with $endpos(t)$ the set of all positions in the string s, in which the occurrences of t end. For instance, we have $endpos($" bc "$) = \{2, 4\}$ for the string " $abcbc$ ".

We will call two substrings t_1 and t_2 $endpos$-equivalent, if their ending sets coincide: $endpos(t_1) = endpos(t_2)$. Thus all non-empty substrings of the string s can be decomposed into several **equivalence classes** according to their sets $endpos$.

It turns out, that in a suffix machine $endpos$-equivalent substrings **correspond to the same state**. In other words the number of states in a suffix automaton is equal to the number of equivalence classes among all substrings, plus the initial state. Each state of a suffix automaton corresponds to one or more substrings having the same value $endpos$.

We will later describe the construction algorithm using this assumption. We will then see, that all the required properties of a suffix automaton, except for the minimality, are fulfilled. And the minimality follows from Nerode's theorem (which will not be proven in this article).

We can make some important observations concerning the values $endpos$:

Lemma 1: Two non-empty substrings u and w (with $length(u) \le length(w)$) are $endpos$-equivalent, if and only if the string u occurs in s only in the form of a suffix of w.

The proof is obvious. If u and w have the same $endpos$ values, then u is a suffix of w and appears only in the form of a suffix of w in s. And if u is a suffix of w and appears only in the form as a suffix in s, then the values $endpos$ are equal by definition.

Lemma 2: Consider two non-empty substrings u and w (with $length(u) \le length(w)$). Then their sets $endpos$ either don't intersect at all, or $endpos(w)$ is a subset of $endpos(u)$. And it depends on if u is a suffix of w or not.

$$\begin{cases} endpos(w) \subseteq endpos(u) & \text{if } u \text{ is a suffix of } w \\ endpos(w) \cap endpos(u) = \emptyset & \text{otherwise} \end{cases}$$

Proof: If the sets $endpos(u)$ and $endpos(w)$ have at least one common element, then the strings u and w both end in that position, i.e. u is a suffix of w. But then at every occurrence of w also appears the substring u, which means that $endpos(w)$ is a subset of $endpos(u)$.

Lemma 3: Consider an $endpos$-equivalence class. Sort all the substrings in this class by non-increasing length. Then in the resulting sequence each substring will be one shorter than the previous one, and at the same time will be a suffix of the previous one. In other words, in a same equivalence class, the shorter substrings are actually suffixes of the longer substrings, and they take all possible lengths in a certain interval $[x; y]$.

Proof: Fix come $endpos$-equivalent class. If it only contains one string, then the lemma is obviously true. Now let's say that the number of strings in the class is greater than one.

According to Lemma 1, two different $endpos$-equivalent strings are always in such a way, that the shorter one is a proper suffix of the longer one. Consequently, there cannot be two strings of the same length in the equivalence class.

Let's denote by w the longest, and through u the shortest string in the equivalence class. According to Lemma 1, the string u is a proper suffix of the string w. Consider now any suffix of w with a length in the interval $[length(u); length(w)]$. It is easy to see, that this suffix is also contained in the same equivalence class. Because this suffix can only appear in the form of a suffix of w in the string s (since also the shorter suffix u occurs in s only in the form of a suffix of w). Consequently, according to Lemma 1, this suffix is $endpos$-equivalent to the string w.

Suffix links $link$

Consider some state $v \neq t_0$ in the automaton. As we know, the state v corresponds to the class of strings with the same $endpos$ values. And if we denote by w the longest of these strings, then all the other strings are suffixes of w.

We also know the first few suffixes of a string w (if we consider suffixes in descending order of their length) are all contained in this equivalent class, and all other suffixes (at least one other - the empty suffix) are in some other classes. We denote by t the biggest such suffix, and make a suffix link to it.

In other words, a **suffix link** $link(v)$ leads to the state that corresponds to the **longest suffix** of w that is another $endpos$-equivalent class.

Here we assume that the initial state t_0 corresponds to its own equivalence class (containing only the empty string), and for convenience we set $endpos(t) = -1, 0, \dots, length(s) - 1$.

Lemma 4: Suffix links form a **tree** with the root t_0.

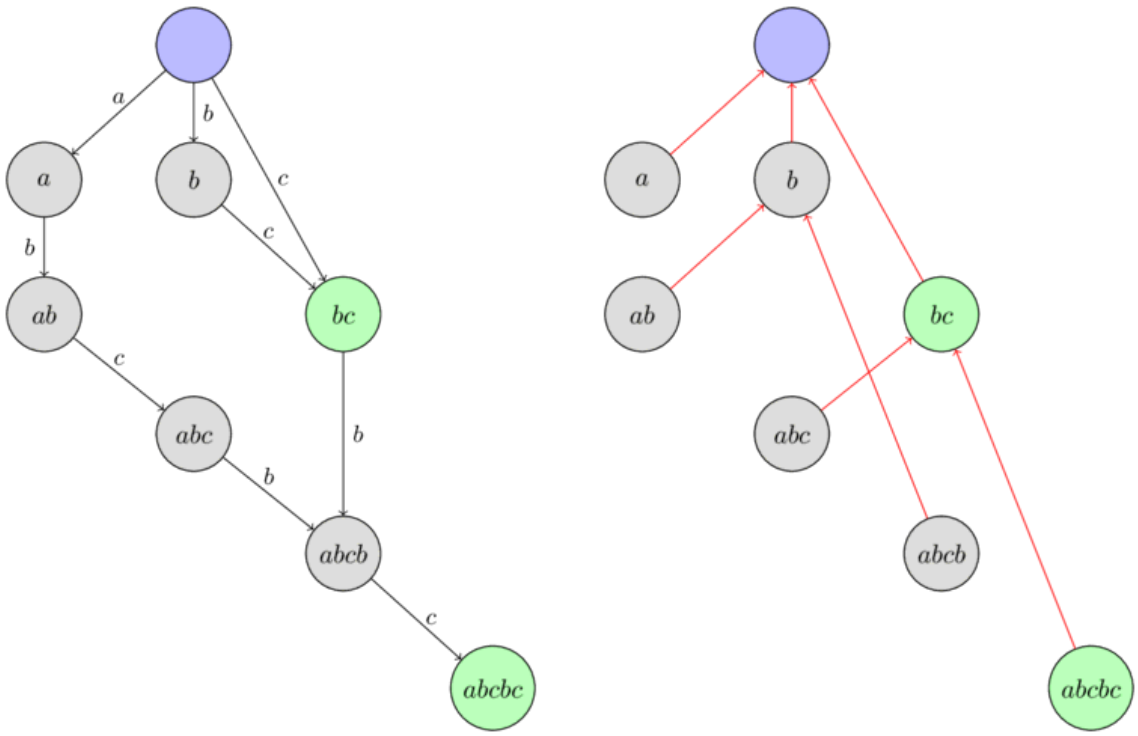
Proof: Consider an arbitrary state $v \neq t_0$. A suffix link $link(v)$ leads to a state corresponding to strings with strictly smaller length (this follows from the definition of the suffix links and from Lemma 3). Therefore, by moving along the suffix links, we will sooner or later come to the initial state t_0, which corresponds to the empty string.

Lemma 5: If we construct a tree using the sets $endpos$ (by the rule that the set of a parent node contains the sets of all children as subsets), then the structure will coincide with the tree of suffix links.

Proof: The fact that we can construct a tree using the sets $endpos$ follows directly from Lemma 2 (that any two sets either do not intersect of one is contained in the other).

Let us now consider an arbitrary state $v \neq t_0$, and its suffix link $link(v)$. From the definition of the suffix link and from Lemma 2 it follows that $endpos(v) \subseteq endpos(link(v))$, which together with the previous lemma proves the assertion: the tree of suffix links is essentially a tree of sets $endpos$.

Here is an **example** of a tree of suffix links in the suffix automaton build for the string " $abcbc$ ". The nodes are labeled with the longest substring from the corresponding equivalence class.



Recap

Before proceeding to the algorithm itself, we recap the accumulated knowledge, and introduce a few auxiliary notations.

- The substrings of the string s can be decomposed into equivalence classes according to their end positions $endpos$.
- The suffix automaton consists of the initial state t_0, as well as of one state for each $endpos$-equivalence class.
- For each state v one or multiple substrings match. We denote by $longest(v)$ the longest such string, and through $len(v)$ its length. We denote by $shortest(v)$ the shortest such substring, and its length with $minlen(v)$. Then all the strings corresponding to this state are different suffixes of the string $longest(v)$ and have all possible lengths in the interval $[minlength(v); len(v)]$.
- For each state $v \neq t_0$ a suffix link is defined as a link, that leads to a state that corresponds to the suffix of the string $longest(v)$ of length $minlen(v) - 1$. The suffix links form a tree with the root in t_0, and at the same time this tree forms an inclusion relationship between the sets $endpos$.
- We can express $minlen(v)$ for $v \neq t_0$ using the suffix link $link(v)$ as: $minlen(v) = len(link(v)) + 1$
- If we start from an arbitrary state v_0 and follow the suffix links, then sooner or later we will reach the initial state t_0. In this case we obtain a sequence of disjoint intervals $[minlen(v_i); len(v_i)]$, which in union forms the continuous interval $[0; len(v_0)]$.

Algorithm

Now we can proceed to the algorithm itself. The algorithm will be **online**, i.e. we will add the characters of the string one by one, and modify the automaton accordingly in each step.

To achieve linear memory consumption, we will only store the values len, $link$ and a list of transitions in each state. We will not label terminal states (but we will later show how to arrange these labels after constructing the suffix automaton).

Initially the automaton consists of a single state t_0, which will be the index 0 (the remaining states will receive the indices $1, 2, \ldots$). We assign it $len = 0$ and $link = -1$ for convenience (-1 will be a fictional, non-existing state).

Now the whole task boils down to implementing the process of **adding one character** c to the end of the current string. Let us describe this process:

- Let $last$ be the state corresponding to the entire string before adding the character c. (Initially we set $last = 0$, and we will change $last$ in the last step of the algorithm accordingly.)
- Create a new state cur, and assign it with $len(cur) = len(last) + 1$. The value $link(cur)$ is not know at the time.
- Now we to the following procedure: We start at the state $last$. While there isn't a transition through the letter c, we will add a transition to the state cur, and follow the suffix link. If at some point there already exists a transition through the letter c, then we will stop and denote this state with p.
- If it haven't found such a state p, then we reached the fictitious state -1, then we can just assign $link(cur) = 0$ and leave.
- Suppose now that we have found a state p, from which there exists a transition through the letter c. We will denote the state, to which the transition leads, with q.
- Now we have two cases. Either $len(p) + 1 = len(q)$, or not.
- If $len(p) + 1 = len(q)$, then we can simply assign $link(cur) = q$ and leave.

- Otherwise it is a bit more complicated. It is necessary to **clone** the state q: we create a new state $clone$, copy all the data from q (suffix link and transition) except the value len. We will assign $len(clone) = len(p) + 1$.

  After cloning we direct the suffix link from cur to $clone$, and also from q to clone.

  Finally we need to walk from the state p back using suffix links as long as there is a transition through c to the state q, and redirect all those to the state $clone$.
- In any of the three cases, after completing the procedure, we update the value $last$ with the state cur.

If we also want to know which states are **terminal** and which are not, the we can find all terminal states after constructing the complete suffix automaton for the entire string s. To do this, we take the state corresponding to the entire string (stored in the variable $last$), and follow its suffix links until we reach the initial state. We will mark all visited states as terminal. It is easy to understand that by doing so we will mark exactly the states corresponding to all the suffixes of the string s, which are exactly the terminal states.

In the next section we will look in detail at each step and show its **correctness**.

Here we only note that, since we only create one or two new states for each character of s, the suffix automaton contains a **linear number of states**.

The linearity of the number of transitions, and in general the linearity of the runtime of the algorithm is less clear, and they will be proven after we proved the correctness.

Correctness

- We will call a transition (p, q) **continuous** if $len(p) + 1 = len(q)$. Otherwise, i.e. when $len(p) + 1 < len(q)$, the transition will be called **non-continuous**.

  As we can see from the description of the algorithm, continuous and non-continuous transitions will lead to different cases of the algorithm. Continuous transitions are fixed, and will never change again. In contrast non-continuous transition may change, when new letters are added to the string (the end of the transition edge may change).
- To avoid ambiguity we will denote the string, for which the suffix automaton was built before adding the current character c, with s.
- The algorithm begins with creating a new state cur, which will correspond to the entire string $s + c$. It is clear why we have to create a new state. Together with the new character a new equivalence class is created.
- After creating a new state we traverse by suffix links starting from the state corresponding to the entire string s. For each state we try to add a transition with the character c to the new state cur. Thus we append to each suffix of s the character c. However we can only add these new transitions, if they don't conflict with an already existing one. Therefore as soon as we find an already existing transition with c we have to stop.
- In the simplest case we reached the fictitious state -1. This means we added the transition with c to all suffixes of s. This also means, that the character c hasn't been part of the string s before. Therefore the suffix link of cur has to lead to the state 0.
- In the second case we came across an existing transition (p, q). This means that we tried to add a string $x + c$ (where x is a suffix of s) to the machine that **already exists** in the machine (the string $x + c$ already appears as a substring of s). Since we assume that the automaton for the string s is build correctly, we should not add a new transition here.

However there is a difficulty. To which state should the suffix link from the state cur lead? We have to make a suffix link to a state, in which the longest string is exactly $x + c$, i.e. the len of this state should be $len(p) + 1$. However it is possible, that such a state doesn't yet exists, i.e. $len(q) > len(p) + 1$. In this case we have to create such a state, by **splitting** the state q.

- If the transition (p, q) turns out to be continuous, then $len(q) = len(p) + 1$. In this case everything is simple. We direct the suffix link from cur to the state q.
- Otherwise the transition is non-continuous, i.e. $len(q) > len(p) + 1$. This means that the state q corresponds to not only the suffix of $s + c$ with length $len(p) + 1$, but also to longer substrings of s. We can do nothing other than **splitting** the state q into two sub-states, so that the first one has length $len(p) + 1$.

  How can we split a state? We **clone** the state q, which gives us the state $clone$, and we set $len(clone) = len(p) + 1$. We copy all the transitions from q to $clone$, because we don't want to change the paths that traverse through q. Also we set the suffix link from $clone$ to the target of the suffix link of q, and set the suffix link of q to $clone$.

  And after splitting the state, we set the suffix link from cur to $clone$.

  In the last step we change some of the transitions to q, we redirect them to $clone$. Which transitions do we have to change? It is enough to redirect only the transitions corresponding to all the suffixes of the string $w + c$ (where w is the longest string of p), i.e. we need to continue to move along the suffix links, starting from the vertex p until we reach the fictitious state -1 or a transition that leads to a different state than q.

Linear number of operations

First we immediately make the assumption that the size of the alphabet is **constant**. If this is not the case, then it will not be possible to talk about the linear time complexity. The list of transitions from one vertex will be stored in a balanced tree, which allows you to quickly perform key search operations and adding keys. Therefore if we denote with k the size of the alphabet, then the asymptotic behavior of the algorithm will be $O(n \log k)$ with $O(n)$ memory. However if the alphabet is small enough, then you can sacrifice memory by avoiding balanced trees, and store the transitions at each vertex as an array of length k (for quick searching by key) and a dynamic list (to quickly traverse all available keys). Thus we reach the $O(n)$ time complexity for the algorithm, but at a cost of $O(nk)$ memory complexity.

So we will consider the size of the alphabet to be constant, i.e. each operation of searching for a transition on a character, adding a transition, searching for the next transition - all these operations can be done in $O(1)$.

If we consider all parts of the algorithm, then it contains three places in the algorithm in which the linear complexity is not obvious:

- The first place is the traversal through the suffix links from the state $last$, adding transitions with the character c.
- The second place is the copying of transitions when the state q is cloned into a new state $clone$.
- Third place is changing the transition leading to q, redirecting them to $clone$.

We use the fact that the size of the suffix automaton (both in number of states and in the number of transitions) is **linear**. (The proof of the linearity of the number of states is the algorithm itself, and the proof of linearity of the number of states is given below, after the implementation of the algorithm).

Thus the total complexity of the **first and second places** is obvious, after all each operation adds only one amortized new transition to the automaton.

It remains to estimate the total complexity of the **third place**, in which we redirect transitions, that pointed originally to q, to $clone$. We denote $v = longest(p)$. This is a suffix of the string s, and with each iteration its length decreases - and therefore the position v as the suffix of the string s increases monotonically with each iteration. In this case, if before the first iteration of the loop, the corresponding string v was at the depth k ($k \geq 2$) from $last$ (by counting the depth as the number of suffix links), then after the last iteration the string $v + c$ will be a 2-th suffix link on the path from cur (which will become the new value $last$).

Thus, each iteration of this loop leads to the fact that the position of the string $longest(link(link(last))$ as suffix of the current string will monotonically increase. Therefore this cycle cannot be executed more than n iterations, which was required to prove.

Implementation

First we describe a data structure that will store all information about a specific transition (len, $link$ and the list of transitions). If necessary you can add a terminal flag here, as well as other information. We will store the list of transitions in the form of a map, which allows us to achieve total $O(n)$ memory and $O(n \log k)$ time for processing the entire string.

```
struct state {
    int len, link;
    map<char, int> next;
};
```

The suffix automaton itself will be stored in an array of these structures $state$. We store the current size sz and also the variable $last$, the state corresponding to the entire string at the moment.

```
const int MAXLEN = 100000;
state st[MAXLEN * 2];
int sz, last;
```

We give a function that initializes a suffix automaton (creating a suffix automaton with a single state).

```
void sa_init() {
    st[0].len = 0;
    st[0].link = -1;
    sz++;
    last = 0;
}
```

And finally we give the implementation of the main function - which adds the next character to the end of the current line, rebuilding the machine accordingly.

```
void sa_extend(char c) {
    int cur = sz++;
    st[cur].len = st[last].len + 1;
    int p = last;
    while (p != -1 && !st[p].next.count(c)) {
        st[p].next[c] = cur;
```

```
            p = st[p].link;
        }
        if (p == -1) {
            st[cur].link = 0;
        } else {
            int q = st[p].next[c];
            if (st[p].len + 1 == st[q].len) {
                st[cur].link = q;
            } else {
                int clone = sz++;
                st[clone].len = st[p].len + 1;
                st[clone].next = st[q].next;
                st[clone].link = st[q].link;
                while (p != -1 && st[p].next[c] == q) {
                    st[p].next[c] = clone;
                    p = st[p].link;
                }
                st[q].link = st[cur].link = clone;
            }
        }
        last = cur;
    }
```

As mentioned above, if you sacrifice memory ($O(nk)$, where k is the size of the alphabet), then you can achieve the build time of the machine in $O(n)$, even for any alphabet size k. But for this you will have to store an array of size k in each state (for quickly jumping to the transition of the letter), and additional a list of all transitions (to quickly iterate over the transitions them).

Additional properties

Number of states

The number of states in a suffix automaton of the string s of length n **doesn't exceed** $2n - 1$ (for $n \geq 2$).

The proof is the construction algorithm itself, since initially the automaton consists of one state, and in the first and second iteration only a single state will be created, and in the remaining $n - 2$ steps at most 2 states will be created each.

However we can also **show** this estimation **without knowing the algorithm**. Let us recall that the number of states is equal to the number of different sets $endpos$. In addition theses sets $endpos$ form a tree (a parent vertex contains all children sets in his set). Consider this tree and transform it a little bit: as long as it has an internal vertex with only one child (which means that the set of the child misses at least one position from the parent set), we create a new child with the set of the missing positions. In the end we have a tree in which each inner vertex has a degree greater than one, and the number of leaves does not exceed n. Therefore there are no more than $2n - 1$ vertices in such a tree.

This bound of the number of states can actually be achieved for each n. A possible string is: " $abbb \ldots bbb$ " In each iteration, starting at the third one, the algorithm will split a state, resulting in exactly $2n - 1$ states.

Number of transitions

The number of transitions in a suffix automaton of a string s of length n **doesn't exceed** $3n - 4$ (for $n \geq 3$).

Let us prove this:

Let us first estimate the number of continuous transitions. Consider a spanning tree of the longest paths in the automaton starting in the state t_0. This skeleton will consist of only the continuous edges, and therefore their number is less than the number of states, i.e. it does not exceed $2n - 2$.

Now let us estimate the number of non-continuous transitions. Let the current non-continuous transition be (p, q) with the character c. We take the correspondent string $u + c + w$, where the string u corresponds to the longest path from the initial state to p, and w to the longest path from q to any terminal state. On one hand, each such string $u + c + w$ for each incomplete strings will be different (since the strings u and w are formed only by complete transitions). On the other hand each such string $u + c + w$, by the definition of the terminal states, will be a suffix of the entire string s. Since there are only n non-empty suffixes of s, and non of the strings $u + c + w$ can contain s (because the entire string only contains complete transitions), the total number of incomplete transitions does not exceed $n - 1$.

Combining these two estimates gives us the bound $3n - 3$. However, since the maximum number of states can only be achieved with the test case " $abbb \ldots bbb$ " and this case has clearly less than $3n - 3$ transitions, we get the tighter bound of $3n - 4$ for the number of transitions in a suffix automaton.

This bound can also be achieved with the string: " $abbb \ldots bbbc$ "

Applications

Here we look at some tasks that can be solved using the suffix automaton. For the simplicity we assume that the alphabet size k is constant, which allows us to consider the complexity of appending a character and the traversal as constant.

Check for occurrence

Given a text T, and multiple patters P. We have to check whether or not the strings P appear as a substring of T.

We build a suffix automaton of the text T in $O(length(T))$ time. To check if a pattern P appears in T, we follow the transitions, starting from t_0, according to the characters of P. If at some point there doesn't exists a transition, then the pattern P doesn't appear as a substring of T. If we can process the entire string P this way, then the string appears in T.

It is clear that this will take $O(length(P))$ time for each string P. Moreover the algorithm actually finds the length of the longest prefix of P that appears in the text.

Number of different substrings

Given a string S. You want to compute the number of different substrings.

Let us build a suffix automaton for the string S.

Each substring of S corresponds to some path in the automaton. Therefore the number of different substrings is equal to the number of different paths in the automaton starting at t_0.

Given that the suffix automaton is a directed acyclic graph, the number of different ways can be computed using dynamic programming.

Namely, let $d[v]$ be the number of ways, starting at the state v (including the path of length zero). Then we have the recursion: $d[v] = 1 + \sum_{w:(v,w,c)\in DAWG} d[w]$ I.e. $d[v]$ can be expressed as the sum of answers for all ends of the transitions of v.

The number of different substrings is the value $d[t_0] - 1$ (since we don't count the empty substring).

Total time complexity: $O(length(S))$

Total length of all different substrings

Given a string S. We want to compute the total length of all its various substrings.

The solution is similar to the previous one, only now it is necessary to consider two quantities for the dynamic programming part: the number of different substrings $d[v]$ and their total length $ans[v]$.

We already described how to compute $d[v]$ in the previous task. The value $ans[v]$ can be computed using the recursion: $ans[v] = \sum_{w:(v,w,c) \in DAWG} d[w] + ans[w]$ We take the answer of each adjacent vertex w, and add to it $d[w]$ (since every substrings is one character longer when starting from the state v).

Again this task can be computed in $O(length(S))$ time.

Lexicographically k-th substring

Given a string S. We have to answer multiple queries. For each given number K_i we have to find the K_i-th string in the lexicographically ordered list of all substrings.

The solution of this problem is based on the idea of the previous two problems. The lexicographically k-th substring corresponds to the lexicographically k-th path in the suffix automaton. Therefore after counting the number of paths from each state, we can easily search for the k-th path starting from the root of the automaton.

This takes $O(length(S))$ time for preprocessing and then $O(length(ans) \cdot k)$ for each query (where ans is the answer for the query and k is the size of the alphabet).

Smallest cyclic shift

Given a string S. We want to find the lexicographically smallest cyclic shift.

We construct a suffix automaton for the string $S + S$. Then the automaton will contain in itself as paths all the cyclic shifts of the string S.

Consequently the problem is reduced to finding the lexicographically smallest path of length $length(S)$, which can be done in a trivial way: we start in the initial state and greedily pass through the transitions with the minimal character.

Total time complexity is $O(length(S))$.

Number of occurrences

For a given text T. We have to answer multiple queries. For each given pattern P we have to find out how many times the string P appears in the string T as substring.

We construct the suffix automaton for the text T.

Next we do the following preprocessing: for each state v in the automaton we calculate the number $cnt[v]$ that is equal to the size of the set $endpos(v)$. In fact all strings corresponding to the same state v appear in the text T an equal amount of times, which is equal to the number of positions in the set $endpos$.

However we cannot construct the sets $endpos$ explicitly, therefore we only consider their sizes cnt.

To compute them we proceed as follows. For each state, if it was not created by cloning (and if it is not the initial state t_0), we initialize it with $cnt = 1$. Then we will go through all states in decreasing order of their length len, and add the current value $cnt[v]$ to the suffix links: $cnt[link(v)] \mathrel{+}= cnt[v]$ This gives the correct value for each state.

Why is this correct? The total stats obtained not obtained by cloning are exactly $length(T)$, and the first i of them appeared when we added the first i characters. Consequently for each of these states we count the corresponding position at which it was processed. Therefore initially we have $cnt = 1$ for each such state, and $cnt = 0$ for all other.

Then we apply the following operation for each v: $cnt[link(v)] \mathrel{+}= cnt[v]$. The meaning behind this is, that if a string v appears $cnt[v]$ times, then also all its suffixes appear at the exact same end positions, therefore also $cnt[v]$ times.

Why don't we overcount in this procedure (i.e. don't count some position twice)? Because we add the positions of a state to only one other state, so it can not happen that one state directs its positions to another state twice in two different ways.

Thus we can compute the quantities cnt for all states in the automaton in $O(length(T))$ time.

After that answering a query by just looking up the value $cnt[t]$, where t is the state corresponding to the pattern, if such a state exists. Otherwise answer with 0. Answering a query takes $O(length(P))$ time.

First occurrence position

Given a text T and multiple queries. For each query string P we want to find the position of the first occurrence of P in the string T (the position of the beginning of P).

We again construct a suffix automaton. Additionally we precompute the position $firstpos$ for all states in the automaton, i.e. for each state v we want to find the position $firstpos[v]$ of the end of the first occurrence. In other words, we want to find in advance the minimal element of each set $endpos$ (since obviously cannot maintain all sets $endpos$ explicitly).

To maintain these positions $firstpos$ we extend the function `sa_extend()`. When we create a new state cur, we set: $firstpos(cur) = len(cur) - 1$ And when we clone a vertex q as $clone$, we set: $firstpos(clone) = firstpos(q)$ (since the only other option for a value would be $firstpos(cur)$ which is definitely too big)

Thus the answer for a query is simply $firstpos(t) - length(P) + 1$, where t is the state corresponding to the string P. Answering a query again takes only $O(length(P))$ time.

All occurrence positions

This time we have to display all positions of the occurrences in the string T.

Again we construct a suffix automaton for the text T. Similar as in the previous task we compute the position $firstpos$ for all states.

Clearly $firstpos(t)$ is part of the answer, if t is the state corresponding to a query string P. So we took into account the state of the automaton containing P. What other states do we need to take into account? All states that correspond to strings for which P is a suffix. In other words we need to find all the states that can reach the state t via suffix links.

Therefore to solve the problem we need to save for each state a list of suffix references leading to it. The answer to the query then will then contain all $firstpos$ for each state that we can find on a DFS / BFS starting from the state t using only the suffix references.

This workaround will work in time $O(answer(P))$, because we will not visit a state twice (because only one suffix link leaves each state, so there cannot be two different paths leading to the same state).

We only must take into account that two different states can have the same $firstpos$ value. This happens if one state was obtained by cloning another. However, this doesn't ruin the complexity, since each state can only have at most one clone.

Moreover, we can also get rid of the duplicate positions, if we don't output the positions from the cloned states. In fact a state, that a cloned state can reach, is also reachable from the original state. Thus if we remember the flag `is_cloned` for each state, we can simply ignore the cloned states and only output $firstpos$ for all other states.

Here are some implementation sketches:

```
struct state {
    ...
    bool is_clone;
    int first_pos;
    vector<int> inv_link;
};

// after constructing the automaton
for (int v = 1; v < sz; v++) {
    st[st[v].link].inv_link.push_back(v);
}

// output all positions of occurrences
void output_all_occurrences(int v, int P_length) {
    if (!st[v].is_clone)
        cout << st[v].first_pos - P_length + 1 << endl;
    for (int u : st[v].inv_link)
        output_all_occurrences(u, P_length);
}
```

Shortest non-appearing string

Given a string S and a certain alphabet. We have to find a string of smallest length, that doesn't appear in S.

We will apply dynamic programming on the suffix automaton built for the string S.

Let $d[v]$ be the answer for the node v, i.e. we already processed part of the substring, are currently in the state v, and want to find the smallest number of characters that have to be added to find a non-existent transition. Computing $d[v]$ is very simple. If there is not transition using at least one character of the alphabet, then $d[v] = 1$. Otherwise one character is not enough, and so we need to take the minimum of all answers of all transitions:
$d[v] = 1 + \min_{w:(v,w,c)\in SA} d[w]$.

The answer to the problem will be $d[t_0]$, and the actual string can be restored using the computed array $d[]$.

Longest common substring of two strings

Given two strings S and T. We have to find the longest common substring, i.e. such a string X that appears as substring in S and also in T.

We construct a suffix automaton for the string S.

We will now take the string T, and for each prefix look for the longest suffix of this prefix in S. In other words, for each position in the string T, we want to find the longest common substring of S and T ending in that position.

For this we will use two variables, the **current state** v, and the **current length** l. These two variables will describe the current matching part: its length and the state that corresponds to it.

Initially $v = t_0$ and $l = 0$, i.e. the match is empty.

Now let us describe how we can add a character $T[i]$ and recalculate the answer for it.

- If there is a transition from v with the character $T[i]$, then we simply follow the transition and increase l by one.
- If there is no such transition, we have to shorten the current matching part, which means that we need to follow the suffix link: $v = link(v)$ At the same time, the current length has to be shortened. Obviously we need to assign $l = len(v)$, since after passing through the suffix link we end up in state whose corresponding longest string is a substring.
- If there is still no transition using the required character, we repeat and again go through the suffix link and decrease l, until we find a transition or we reach the fictional state -1 (which means that the symbol $T[i]$ doesn't appear at all in S, so we assign $v = l = 0$).

The answer to the task will be the maximum of all the values l.

The complexity of this part is $O(length(T))$, since in one move we can either increase l by one, or make several passes through the suffix links, each one ends up reducing the value l.

Implementation:

```
string lcs (string S, string T) {
    sa_init();
    for (int i = 0; i < S.size(); i++)
        sa_extend(S[i]);

    int v = 0, l = 0, best = 0, bestpos = 0;
    for (int i = 0; i < T.size(); i++) {
        while (v && !st[v].next.count(T[i])) {
            v = st[v].link ;
            l = st[v].length ;
        }
        if (st[v].next.count(T[i])) {
            v = st [v].next[T[i]];
            l++;
        }
        if (l > best) {
            best = l;
            bestpos = i;
        }
    }
    return t.substr(bestpos - best + 1, best);
}
```

Largest common substring of multiple strings

There are k strings S_i given. We have to find the longest common substring, i.e. such a string X that appears as substring in each string S_i.

We join all strings into one large string T, separating the strings by a special characters D_i (one for each string): $T = S_1 + D_1 + S_2 + D_2 + \cdots + S_k + D_k$.

Then we construct the suffix automaton for the string T.

Now we need to find a string in the machine, which is contained in all the strings S_i, and this can be done by using the special added characters. Note that if a substring is included in some string S_j, then in the suffix automaton exists a path starting from this substring containing the character D_j and not containing the other characters $D_1, \ldots, D_{j-1}, D_{j+1}, \ldots, D_k$.

Thus we need to calculate the attainability, which tells us for each state of the machine and each symbol D_i if there exists such a path. This can easily be computed by DFS or BFS and dynamic programming. After that, the answer to the problem will be the string $longest(v)$ for the state v, from which the paths were exists for all special characters.

4.字符串哈希 String Hashing

Hashing algorithms are helpful in solving a lot of problems.

The problem we want to solve is the problem, we want to compare strings efficiently. The brute force way of doing so is just to compare the letters of both strings, which has a time complexity of $O(\min(n_1, n_2))$ if n_1 and n_2 are the sizes of the two strings. We want to do better. The idea behind strings is the following: we convert each string into an integer, and compare those instead of the strings. Comparing two strings is then an $O(1)$ operation.

For the conversion we need a so-called **hash function**. The goal of it is to convert a string into a integer, the so-called **hash** of the string. The following condition has to hold: if two strings s and t are equal ($s = t$), then also their hashes have to be equal ($\text{hash}(s) = \text{hash}(t)$). Otherwise we will not be able to compare strings.

Notice, the opposite direction doesn't have to hold. If the hashes are equal ($\text{hash}(s) = \text{hash}(t)$), then the strings do not necessarily have to be equal. E.g. a valid hash function would be simply $\text{hash}(s) = 0$ for each s. Now, this is just a stupid example, because this function will be completely useless, but it is a valid hash function. The reason why the opposite direction doesn't have to hold, if because there are exponential many strings. If we only want this hash function to distinguish between all strings consisting of lowercase characters of length smaller than 15, then already the hash wouldn't fit into a 64 bit integer (e.g. unsigned long long) any more, because there are so many of them. And of course we don't want to compare arbitrary long integers, because this will also have the complexity $O(n)$.

So usually we want the hash function to map strings onto numbers of a fixed range $[0, m)$, then comparing strings is just comparison of two integers with fixed length. And of course we want $\text{hash}(s) \neq \text{hash}(t)$ to be very likely, if $s \neq t$.

That's the important part that you have to keep in mind. Using hashing will not be 100% deterministically correct, because two complete different strings might have the same hash (the hashes collide). However, in a wide majority of tasks this can be safely ignored as the probability of the hashes of two different strings colliding is still very small. And we will discuss some techniques in this article how to keep the probability of collisions very low.

Calculation of the hash of a string

The good and widely used way to define the hash of a string s of length n is

$$\begin{aligned}\text{hash}(s) &= s[0] + s[1] \cdot p + s[2] \cdot p^2 + \ldots + s[n-1] \cdot p^{n-1} \mod m \\ &= \sum_{i=0}^{n-1} s[i] \cdot p^i \mod m,\end{aligned}$$

where p and m are some chosen, positive numbers. It is called a **polynomial rolling hash function**.

It is reasonable to make p a prime number roughly equal to the number of characters in the input alphabet. For example, if the input is composed of only lowercase letters of English alphabet, $p = 31$ is a good choice. If the input may contain both uppercase and lowercase letters, then $p = 53$ is a possible choice. The code in this article will use $p = 31$.

Obviously m should be a large number, since the probability of two random strings colliding is about $\approx \frac{1}{m}$. Sometimes $m = 2^{64}$ is chosen, since then the integer overflows of 64 bit integers work exactly like the modulo operation. However there exists a method, which generates colliding strings (which work independent from the choice of p). So in practice $m = 2^{64}$ is not recommended. A good choice for m is some large prime number. The code in this article will just use $m = 10^9 + 9$. This is a large number, but still small enough so that we can perform multiplication of two values using 64 bit integers.

Here is an example of calculating the hash of a string s, which contains only lowercase letters. We convert each character of s to an integer. Here we use the conversion $a \to 1$, $b \to 2$, ..., $z \to 26$. Converting $a \to 0$ is not a good idea, because then the hashes of the strings a, aa, aaa, ... all evaluate to 0.

```
long long compute_hash(string const& s) {
    const int p = 31;
    const int m = 1e9 + 9;
    long long hash_value = 0;
    long long p_pow = 1;
    for (char c : s) {
        hash_value = (hash_value + (c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }
    return hash_value;
}
```

Precomputing the powers of p might give a performance boost.

Example tasks

Search for duplicate strings in an array of strings

Problem: Given a list of n strings s_i, each no longer than m characters, find all the duplicate strings and divide them into groups.

From the obvious algorithm involving sorting the strings, we would get a time complexity of $O(nm \log n)$ where the sorting requires $O(n \log n)$ comparisons and each comparison take $O(m)$ time. However by using hashes, we reduce the comparison time to $O(1)$, giving us an algorithm that runs in $O(nm + n \log n)$ time.

We calculate the hash for each string, sort the hashes together with the indices, and then group the indices by identical hashes.

```
vector<vector<int>> group_identical_strings(vector<string> const& s) {
```

```
    int n = s.size();
    vector<pair<long long, int>> hashes(n);
    for (int i = 0; i < n; i++)
        hashes[i] = {compute_hash(s[i]), i};

    sort(hashes.begin(), hashes.end());

    vector<vector<int>> groups;
    for (int i = 0; i < n; i++) {
        if (i == 0 || hashes[i].first != hashes[i-1].first)
            groups.emplace_back();
        groups.back().push_back(hashes[i].second);
    }
    return groups;
}
```

Fast hash calculation of substrings of given string

Problem: Given a string s and indices i and j, find the hash of the substring $s[i \dots j]$.

By definition, we have: $\text{hash}(s[i \dots j]) = \sum_{k=i}^{j} s[k] \cdot p^{k-i} \mod m$ Multiplying by p^i gives:

$$\text{hash}(s[i \dots j]) \cdot p^i = \sum_{k=i}^{j} s[k] \cdot p^k \mod m$$

$$= \text{hash}(s[0 \dots j]) - \text{hash}(s[0 \dots i-1]) \mod m$$

So by knowing the hash value of each prefix of the string s, we can compute the hash of any substring directly using this formula. The only problem that we face in calculating it is that we must be able to divide $\text{hash}(s[0 \dots j]) - \text{hash}(s[0 \dots i-1])$ by p^i. Therefore we need to find the modular multiplicative inverse of p^i and then perform multiplication with this inverse. We can precompute the inverse of every p^i, which allows computing the hash of any substring of s in $O(1)$ time.

However, there does exist an easier way. In most cases, rather than calculating the hashes of substring exactly, it is enough to compute the hash multiplied by some power of p. Suppose we have two hashes of two substrings, one multiplied by p^i and the other by p^j. If $i < j$ then we multiply the first hash by p^{j-i}, otherwise we multiply the second hash by p^{i-j}. By doing this, we get both the hashes multiplied by the same power of p (which is the maximum of i and j) and now these hashes can be compared easily with no need for any division.

Applications of Hashing

Here are some typical applications of Hashing:

- Rabin-Karp algorithm for pattern matching in a string in $O(n)$ time
- Calculating the number of different substrings of a string in $O(n^2 \log n)$ (see below)
- Calculating the number of palindromic substrings in a string.

Determine the number of different substrings in a string

Problem: Given a string s of length n, consisting only of lowercase English letters, find the number of different substrings in this string.

To solve this problem, we iterate over all substring lengths $l = 1 \dots n$. For every substring length l we construct an array of hashes of all substrings of length l multiplied by the same power of p. The number of different elements in the array is equal to the number of distinct substrings of length l in the string. This number is added to the final answer.

For convenience we will use $h[i]$ as the hash of the prefix with i characters, and define $h[0] = 0$.

```
int count_unique_substrings(string const& s) {
    int n = s.size();

    const int p = 31;
    const int m = 1e9 + 9;
    vector<long long> p_pow(n);
    p_pow[0] = 1;
    for (int i = 1; i < n; i++)
        p_pow[i] = (p_pow[i-1] * p) % m;

    vector<long long> h(n + 1, 0);
    for (int i = 0; i < n; i++)
        h[i+1] = (h[i] + (s[i] - 'a' + 1) * p_pow[i]) % m;

    int cnt = 0;
    for (int l = 1; l <= n; l++) {
        set<long long> hs;
        for (int i = 0; i <= n - l; i++) {
            long long cur_h = (h[i + l] + m - h[i]) % m;
            cur_h = (cur_h * p_pow[n-i-1]) % m;
            hs.insert(cur_h);
        }
        cnt += hs.size();
    }
    return cnt;
}
```

Rabin-Karp Algorithm for string matching

This algorithm is based on the concept of hashing, so if you are not familiar with string hashing, refer to the string hashing article.

This algorithm was authored by Rabin and Karp in 1987.

Problem: Given two strings - a pattern s and a text t, determine if the pattern appears in the text and if it does, enumerate all its occurrences in $O(|s| + |t|)$ time.

Algorithm: Calculate the hash for the pattern s. Calculate hash values for all the prefixes of the text t. Now, we can compare a substring of length $|s|$ with s in constant time using the calculated hashes. So, compare each substring of length $|s|$ with the pattern. This will take a total of $O(|t|)$ time. Hence the final complexity of the algorithm is $O(|t| + |s|)$: $O(|s|)$ is required for calculating the hash of the pattern and $O(|t|)$ for comparing each substring of length $|s|$ with the pattern.

Implementation

```
vector<int> rabin_karp(string const& s, string const& t) {
    const int p = 31;
    const int m = 1e9 + 9;
    int S = s.size(), T = t.size();
```

```
    vector<long long> p_pow(max(S, T));
    p_pow[0] = 1;
    for (int i = 1; i < (int)p_pow.size(); i++)
        p_pow[i] = (p_pow[i-1] * p) % m;

    vector<long long> h(T + 1, 0);
    for (int i = 0; i < T; i++)
        h[i+1] = (h[i] + (t[i] - 'a' + 1) * p_pow[i]) % m;
    long long h_s = 0;
    for (int i = 0; i < S; i++)
        h_s = (h_s + (s[i] - 'a' + 1) * p_pow[i]) % m;

    vector<int> occurences;
    for (int i = 0; i + S - 1 < T; i++) {
        long long cur_h = (h[i+S] + m - h[i]) % m;
        if (cur_h == h_s * p_pow[i] % m)
            occurences.push_back(i);
    }
    return occurences;
}
```

Improve no-collision probability

Quite often the above mentioned polynomial hash is good enough, and no collisions will happen during tests. Remember, the probability that collision happens is only $\approx \frac{1}{m}$. For $m = 10^9 + 9$ the probability is $\approx 10^{-9}$ which is quite low. But notice, that we only did one comparison. What if we compared a string s with 10^6 different strings. The probability that the at least one collision happens is now $\approx 10^{-3}$. And if we want to compare 10^6 different strings with each other (e.g. by counting how many unique strings exists), then the probability of at least one collision happening is already ≈ 1. It is pretty much guaranteed that this task will end with a collision and returns the wrong result.

There is a really easy trick to get better probabilities. We can just compute two different hashes for each string (by using two different p, and/or different m, and compare these pairs instead. If m is about 10^9 for each of the two hash functions, than this is more or less equivalent as having one hash function with $m \approx 10^{18}$. When comparing 10^6 strings with each other, the probability that at least one collision happens is now reduced to $\approx 10^{-6}$.

7. 敏-洛伦兹Main and Lorentz. 找所有的重复串

Given a string s of length n.

A **repetition** is two occurrences of a string in a row. In other words a repetition can be described by a pair of indices $i < j$ such that the substring $s[i \dots j]$ consists of two identical strings written after each other.

The challenge is to **find all repetitions** in a given string s. Or a simplified task: find **any** repetition or find the **longest** repetition.

The algorithm described here was published in 1982 by Main and Lorentz.

Example

Consider the repetitions in the following example string: $acababaee$ The string contains the following three repetitions:

- $s[2 \dots 5] = abab$
- $s[3 \dots 6] = baba$
- $s[7 \dots 7] = ee$

Another example: $abaaba$ Here there are only two repetitions

- $s[0 \dots 5] = abaaba$
- $s[2 \dots 3] = aa$

Number of repetitions

In general there can be up to $O(n^2)$ repetitions in a string of length n. An obvious example is a string consisting of n times the same letter, in this case any substring of even length is a repetition. In general any periodic string with a short period will contain a lot of repetitions.

On the other hand this fact does not prevent computing the number of repetitions in $O(n \log n)$ time, because the algorithm can give the repetitions in compressed form, in groups of several pieces at once.

There is even the concept, that describes groups of periodic substrings with tuples of size four. It has been proven that we the number of such groups is at most linear with respect to the string length.

Also, here are some more interesting results related to the number of repetitions:

- The number of primitive repetitions (those whose halves are not repetitions) is at most $O(n \log n)$.
- If we encode repetitions with tuples of numbers (called Crochemore triples) $(i,\ p,\ r)$ (where i is the position of the beginning, p the length of the repeating substring, and r the number of repetitions), then all repetitions can be described with $O(n \log n)$ such triples.
- Fibonacci strings, defined as $t_0 = a,$ $t_1 = b,$ $t_i = t_{i-1} + t_{i-2},$ are "strongly" periodic. The number of repetitions in the Fibonacci string f_i, even in the compressed with Crochemore triples, is $O(f_n \log f_n)$. The number of primitive repetitions is also $O(f_n \log f_n)$.

Main-Lorentz algorithm

The idea behind the Main-Lorentz algorithm is **divide-and-conquer**.

It splits the initial string into halves, and computes the number of repetitions that lie completely in each halve by two recursive calls. Then comes the difficult part. The algorithm finds all repetitions starting in the first half and ending in the second half (which we will call **crossing repetitions**). This is the essential part of the Main-Lorentz algorithm, and we will discuss it in detail here.

The complexity of divide-and-conquer algorithms is well researched. The master theorem says, that we will end up with an $O(n \log n)$ algorithm, if we can compute the crossing repetitions in $O(n)$ time.

Search for crossing repetitions

So we want to find all such repetitions that start in the first half of the string, let's call it u, and end in the second half, let's call it v: $s = u + v$ Their lengths are approximately equal to the length of s divided by two.

Consider an arbitrary repetition and look at the middle character (more precisely the first character of the second half of the repetition). I.e. if the repetition is a substring $s[i \dots j]$, then the middle character is $(i + j + 1)/2$.

We call a repetition **left** or **right** depending on which string this character is located - in the string u or in the string v. In other words a string is called left, if the majority of it lies in u, otherwise we call it right.

We will now discuss how to find **all left repetitions**. Finding all right repetitions can be done in the same way.

Let us denote the length of the left repetition by $2l$ (i.e. each half of the repetition has length l). Consider the first character of the repetition falling into the string v (it is at position $|u|$ in the string s). It coincides with the character l positions before it, let's denote this position $cntr$.

We will fixate this position $cntr$, and **look for all repetitions at this position** $cntr$.

For example: $c \underset{cntr}{a} c \mid a\ d\ a$ The vertical lines divides the two halves. Here we fixated the position $cntr = 1$, and at this position we find the repetition $caca$.

It is clear, that if we fixate the position $cntr$, we simultaneously fixate the length of the possible repetitions: $l = |u| - cntr$. Once we know how to find these repetitions, we will iterate over all possible values for $cntr$ from 0 to $|u| - 1$, and find all left crossover repetitions of length $l = |u|,\ |u| - 1,\ \dots, 1$.

Criterion for left crossing repetitions

Now, how can we find all such repetitions for a fixated $cntr$? Keep in mind that there still can be multiple such repetitions.

Let's again look at a visualization, this time for the repetition $abcabc$: $\overbrace{a}^{l_1} \overbrace{\underset{cntr}{b}\ c}^{l_2} \overbrace{a}^{l_1} \mid \overbrace{b\ c}^{l_2}$ Here we denoted the lengths of the two pieces of the repetition with l_1 and l_2: l_1 is the length of the repetition up to the position $cntr - 1$, and l_2 is the length of the repetition from $cntr$ to the end of the half of the repetition. We have $2l = l_1 + l_2 + l_1 + l_2$ as the total length of the repetition.

Let us generate **necessary and sufficient** conditions for such a repetition at position $cntr$ of length $2l = 2(l_1 + l_2) = 2(|u| - cntr)$:

- Let k_1 be the largest number such that the first k_1 characters before the position $cntr$ coincide with the last k_1 characters in the string u:
  $u[cntr - k_1 \dots cntr - 1] = u[|u| - k_1 \dots |u| - 1]$
- Let k_2 be the largest number such that the k_2 characters starting at position $cntr$ coincide with the first k_2 characters in the string v: $u[cntr \dots cntr + k_2 - 1] = v[0 \dots k_2 - 1]$
- Then we have a repetition exactly for any pair $(l_1,\ l_2)$ with $\begin{matrix} l_1 \le k_1, \\ l_2 \le k_2. \end{matrix}$

To summarize:

- We fixate a specific position $cntr$.
- All repetition which we will find now have length $2l = 2(|u| - cntr)$. There might be multiple such repetitions, they depend on the lengths l_1 and $l_2 = l - l_1$.
- We find k_1 and k_2 as described above.

- Then all suitable repetitions are the ones for which the lengths of the pieces l_1 and l_2 satisfy

$$l_1 + l_2 = l = |u| - cntr$$

the conditions:

$$l_1 \leq k_1,$$

$$l_2 \leq k_2.$$

Therefore the only remaining part is how we can compute the values k_1 and k_2 quickly for every position $cntr$. Luckily we can compute them in $O(1)$ using the Z-function:

- To can find the value k_1 for each position by calculating the Z-function for the string $\overline{u}$ (i.e. the reversed string u). Then the value k_1 for a particular $cntr$ will be equal to the corresponding value of the array of the Z-function.
- To precompute all values k_2, we calculate the Z-function for the string $v + \# + u$ (i.e. the string u concatenated with the separator character $\#$ and the string v). Again we just need to look up the corresponding value in the Z-function to get the k_2 value.

So this is enough to find all left crossing repetitions.

Right crossing repetitions

For computing the right crossing repetitions we act similarly: we define the center $cntr$ as the character corresponding to the last character in the string u.

Then the length k_1 will be defined as the largest number of characters before the position $cntr$ (inclusive) that coincide with the last characters of the string u. And the length k_2 will be defined as the largest number of characters starting at $cntr + 1$ that coincide with the characters of the string v.

Thus we can find the values k_1 and k_2 by computing the Z-function for the strings $\overline{u} + \# + \overline{v}$ and v.

After that we can find the repetitions by looking at all positions $cntr$, and use the same criterion as we had for left crossing repetitions.

Implementation

The implementation of the Main-Lorentz algorithm finds all repetitions in form of peculiar tuples of size four: $(cntr,\ l,\ k_1,\ k_2)$ in $O(n \log n)$ time. If you only want to find the number of repetitions in a string, or only want to find the longest repetition in a string, this information is enough and the runtime will still be $O(n \log n)$.

Notice that if you want to expand these tuples to get the starting and end position of each repetition, then the runtime will be the runtime will be $O(n^2)$ (remember that there can be $O(n^2)$ repetitions). In this implementation we will do so, and store all found repetition in a vector of pairs of start and end indices.

```
vector<int> z_function(string const& s) {
    int n = s.size();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; i++) {
        if (i <= r)
            z[i] = min(r-i+1, z[i-l]);
        while (i + z[i] < n && s[z[i]] == s[i+z[i]])
            z[i]++;
        if (i + z[i] - 1 > r) {
            l = i;
```

```
            r = i + z[i] - 1;
        }
    }
    return z;
}

int get_z(vector<int> const& z, int i) {
    if (0 <= i && i < (int)z.size())
        return z[i];
    else
        return 0;
}

vector<pair<int, int>> repetitions;

void convert_to_repetitions(int shift, bool left, int cntr, int l, int k1, int
k2) {
    for (int l1 = max(1, l - k2); l1 <= min(l, k1); l1++) {
        if (left && l1 == l) break;
        int l2 = l - l1;
        int pos = shift + (left ? cntr - l1 : cntr - l - l1 + 1);
        repetitions.emplace_back(pos, pos + 2*l - 1);
    }
}

void find_repetitions(string s, int shift = 0) {
    int n = s.size();
    if (n == 1)
        return;

    int nu = n / 2;
    int nv = n - nu;
    string u = s.substr(0, nu);
    string v = s.substr(nu);
    string ru(u.rbegin(), u.rend());
    string rv(v.rbegin(), v.rend());

    find_repetitions(u, shift);
    find_repetitions(v, shift + nu);

    vector<int> z1 = z_function(ru);
    vector<int> z2 = z_function(v + '#' + u);
    vector<int> z3 = z_function(ru + '#' + rv);
    vector<int> z4 = z_function(v);

    for (int cntr = 0; cntr < n; cntr++) {
        int l, k1, k2;
        if (cntr < nu) {
            l = nu - cntr;
            k1 = get_z(z1, nu - cntr);
            k2 = get_z(z2, nv + 1 + cntr);
        } else {
            l = cntr - nu + 1;
            k1 = get_z(z3, nu + 1 + nv - 1 - (cntr - nu));
            k2 = get_z(z4, (cntr - nu) + 1);
        }
        if (k1 + k2 >= l)
            convert_to_repetitions(shift, cntr < nu, cntr, l, k1, k2);
```

```
    }
}
```

6. 林登分解 Lyndon factorization 字符环里字典序最小的起点

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. It 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 - k));
            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 is 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);
}
```