[CP-Algorithms](https://cp-algorithms.com/)[Page Authors](https://github.com/e-maxx-eng/e-maxx-eng/commits/master/src/string/aho_corasick.md)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | |  |  | | --- | --- | |  |  | | |  | |

**Aho-Corasick algorithm**

**Table of Contents**

[Construction of the trie](https://cp-algorithms.com/string/aho_corasick.html#toc-tgt-0)

[Construction of an automaton](https://cp-algorithms.com/string/aho_corasick.html#toc-tgt-1)

[Applications](https://cp-algorithms.com/string/aho_corasick.html#toc-tgt-2)

[Find all strings from a given set in a text](https://cp-algorithms.com/string/aho_corasick.html#toc-tgt-3)

[Finding the lexicographical smallest string of a given length that doesn't match any given strings](https://cp-algorithms.com/string/aho_corasick.html#toc-tgt-4)

[Finding the shortest string containing all given strings](https://cp-algorithms.com/string/aho_corasick.html#toc-tgt-5)

[Finding the lexicographical smallest string of length LL containing kk strings](https://cp-algorithms.com/string/aho_corasick.html#toc-tgt-6)

[Problems](https://cp-algorithms.com/string/aho_corasick.html#toc-tgt-7)

Let there be a set of strings with the total length mm (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)O(mk) time, where kk 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 leafleaf 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 leafleaf vertex will correspond to one string from the set, and conversely that each string of the set corresponds to one leafleaf 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 VertexVertex. Each VertexVertex contains the flag leafleaf, and the edges in the form of ans array next[]next[], where next[i]next[i] is the index to the vertex that we reach by following the character ii, or −1−1, if there is no such edge. Initially the trie consists of only one vertex - the root - with the index 00.

Now we implement a function that will add a string ss 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 ss 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 leafleaf.

**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 kk links, it will use O(mk)O(mk) memory.

It is possible to decrease the memory consumption to O(m)O(m) by using a map instead of an array in each vertex. However this will increase the complexity to O(nlogk)O(nlog⁡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 abcabc, and we are standing at vertex 22 (which corresponds to the string abab), then using the letter cc we can transition to the state 33.

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

For example let the trie be constructed by the strings abab and bcbc, and we are currently at the vertex corresponding to abab, which is a leafleaf. For a transition with the letter cc, we are forced to go to the state corresponding to the string bb, and from there follow the edge with the letter cc.

A **suffix link** for a vertex pp is a edge that points to the longest proper suffix of the string corresponding to the vertex pp. 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 vv, then we can go to the ancestor pp of the current vertex (let cc be the letter of the edge from pp to vv), then follow its suffix link, and perform from there the transition with the letter cc.

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 pp and the character pchpch of the edge from pp to vv for each vertex vv. Also at each vertex we will store the suffix link linklink (or −1−1 if it hasn't been calculated yet), and in the array go[k]go[k] the transitions in the machine for each symbol (again −1−1 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(len+ans)O(len+ans), where lenlen is the length of the text and ansans 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 vv, and the next letter is cc, then we transition to the next state with go(v,c)go(v,c), thereby either increasing the length of the current match substring by 11, or decreasing it by following a suffix link.

How can we find out for a state vv, if there are any matches with strings for the set? First, it is clear that if we stand on a leafleaf vertex, then then 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 leafleaf vertices by moving along the suffix links, then there will be also a match corresponding to each found leafleaf vertex. A simple example demonstrating this situation can be creating using the set of strings {dabce,abc,bc}{dabce,abc,bc} and the text dabcdabc.

Thus if we store in each leafleaf 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)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 len)O(n len) complexity in total. However this can be optimized by computing and storing the nearest leafleaf 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)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)O(1) time, and therefore we reach the complexity O(len+ans)O(len+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 vv. This can be calculated in O(n)O(n) time in total. Thus we can sum up all matches in O(len)O(len).

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

A set of strings and a length LL is given. We have to find a string of length LL, 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 leafleaf 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 LL. This task can be solved in O(L)O(L) for example by [depth first search](https://cp-algorithms.com/graph/depth-first-search.html).

**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=root, mask=0)(v=root, mask=0), we want to reach the state (v, mask=2n−1)(v, mask=2n−1), where nn 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](https://cp-algorithms.com/graph/breadth-first-search.html) we can find a path to the state (v, mask=2n−1)(v, mask=2n−1) with the smallest length.

**Finding the lexicographical smallest string of length**LL**containing**kk**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, len, cnt)(v, len, cnt), and we want to reach from the state (root, 0, 0)(root, 0, 0) the state (v, L, k)(v, L, k), where vv 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).

**Problems**

* [UVA #11590 - Prefix Lookup](https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2637)
* [UVA #11171 - SMS](https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2112)
* [UVA #10679 - I Love Strings!!](https://uva.onlinejudge.org/index.php?option=onlinejudge&page=show_problem&problem=1620)
* [Codeforces - Frequency of String](http://codeforces.com/problemset/problem/963/D)

(c) 2014-2018 translation by <http://github.com/e-maxx-eng>