

# SUFFIX AUTOMATON by- saisumit

January 26, 2016July 28, 2016

What is suffix – automaton ?

*Suffix – Automaton of a string “S” in simple terms is a directed a-cyclic graph where vertices or nodes are called “**states**” and the arcs or the edges between these nodes is called the “**transition**” between these states.*

*One of the state(node) denoted by “**To**” is called the “**Initial state**” of the suffix-automaton from where we can reach to all other states in the automaton.*

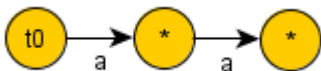
*One or more of this states are marked as “**Terminal states**” such that if we go from “**To**” to any of these terminal states and note down the labels of the edges what we get is a suffix of the original string “S”.*

Why am i reading this shit ? what use it is for me ?

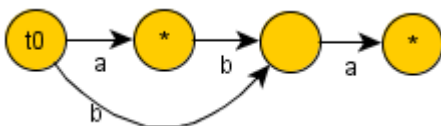
Hey ‘) don’t worry , its one of the best structures to solve almost all problems related to string and the best thing about this is that almost 20-30 lines of code can solve all the under listed problems that too with a Time Complexity of  $O(n)$  and space complexity of  $O(n)$  . Wow that’s awesome man 😊

What it looks like? I know many of them, have a look 😊

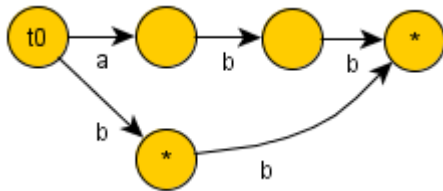
That's for string “aa” ->



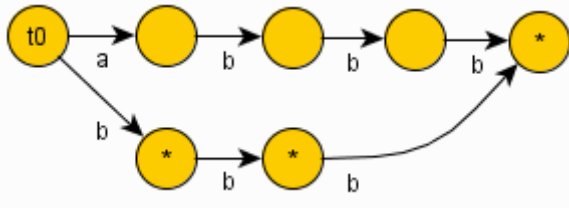
That's for string “aba” ->



That's for string "**abb**" ->



That's for string "**abbb**" ->



Still not satisfied go to this link (preferably with firefox) enter the string

### PROBLEMS IT SOLVES

1. Number of different substrings of a given string .
2. Total length of various substrings.
3. Lexographically  $k^{\text{th}}$  substring.
4. smallest Cyclic shift.
5. No. of occurrences of a pattern in the given Text.
6. Position of all Occurrences.
7. Longest common substring.
8. Longest Common substring of multiple substring.
9. Search for shortest substring that is not included in this string.

So Lets Begin 😊

### Properties of Suffix Automaton

- It contains all the information about all the substrings of the string. Take any path from the "**Initial State ( $t_0$ )**" and write down the label of the edges and terminate that path at any state ( not necessarily a terminal state ) , what we obtain is a substring of the given string and if we

write down all such paths we get all distinct substrings. Conversely any substring of the string “S” corresponds to a path in suffix automaton starting from “Initial State ( $t_0$ )” .

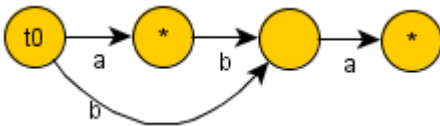
**The “endpos” class – “the building block of suffix-automaton”**: Consider a non-empty substring “T” of “S” , then  $endpos(T)$  is the set of all the positions where T ends in S .We call two strings  $s_1, s_2$  endpos-equivalent if their sets of endings are identical. That is  $(endpos(s_1) == endpos(s_2))$  .

*Help Please! I don't understand without examples:*

**Consider string “aba”**

1.  $endpos(“aba”) = \{3\}$
2.  $endpos(“ba”) = \{3\}$
3.  $endpos(“a”) = \{1,3\}$
4.  $endpos(“ab”) = \{2\}$
5.  $endpos(“b”) = \{2\}$
6.  $endpos(“”) = \{-1,3\}$

Therefore strings “aba” and “ba” are enpos equivalent and are part of same class , “b” and “ab” are equivalent they form a same class . Hey what about “a”? Well , he is a loner he forms a separate class LOL :). The last one is an empty substring , we take its endpos to be  $\{-1, length(s)\}$  , just assume it for now . Here is the pic of the example



This forms the basis of constructing a suffix automaton, To generalize , **“number of states or nodes in a suffix automaton is equal to number of endpos classes”** ..In this example it is equal to 4 .

Did you notice something strange while listing the endpos of various substrings. Infact they are listed in suffix order “aba” -> “ba” -> “a” . The “ab” -> “b” . Well there is a reason behind this

$$\begin{cases} endpos(w) \subset endpos(u) & \text{if } u \text{ --- suffix } w, \\ endpos(u) \cap endpos(w) = \emptyset & \text{otherwise.} \end{cases}$$

The proof of this is obvious. Another important observation regarding a given class is that if sort all the substrings in a given class in increasing order of length, **“each substring will be shorter than the next one by one unit”**, this follows immediately from the fact that the substrings included in one equivalent class are infact suffixes of each other , and take all sorts of different length from  $[ minlen(v) , len(v) ]$  where **“len(v) is the length of the longest string included in a class and minlen(v) is the length of smallest string included in a particular class”**. Therefore if we consider above example of “aba” and let “v” be the class consisting of {“aba”, “ba”} , then  $len(v) = 3$  and  $minlen(v) = 2$  ..

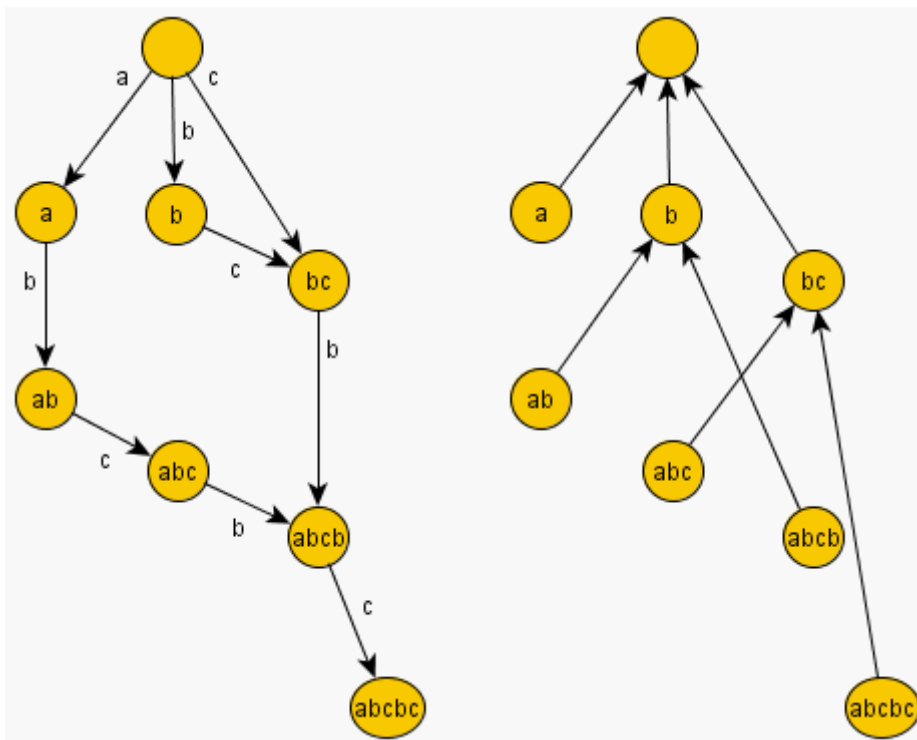
## Suffix – Links

Consider Some state of automaton  $v \neq (t_0)$ . As we know this state corresponds to some class of strings with identical values of endpos and let " $w$ " be the longest of these strings, therefore all other strings are suffixes of " $w$ ", let first few suffixes of this string be contained in the same equivalence class (Considering suffixes in decreasing order of length) and all other suffixes in some other classes, *then suffix link of " $v$ " leads to the state that contains the longest suffix of " $w$ " that is in different endpos class, this leads to two properties.*

1.  $\text{minlen}(v) = \text{len}(\text{link}(v)) + 1.$
2.  $\text{endpos}(v) \subset \text{endpos}(\text{link}(v)),$

In Our example let  $w = "aba"$  and  $v = \{ "aba", "ba" \}$  therefore suffix link of  $v$  will correspond to state  $\{ "a" \}$ , lets call it " $state\ r$ ", applying same thing to  $r$  suffix link of  $r$  will point to empty substring. Another important observation is that every node has a suffix link except  $(t_0)$  because each node will have a suffix link to  $(t_0)$  if there is no other node satisfying the condition.

### *Suffix Automaton and Suffix Link structure of string "abcbc"*



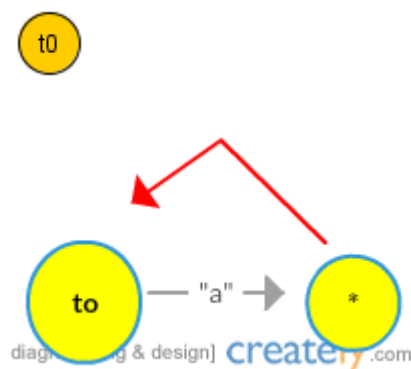
## Construction of Suffix Automaton in Linear Time

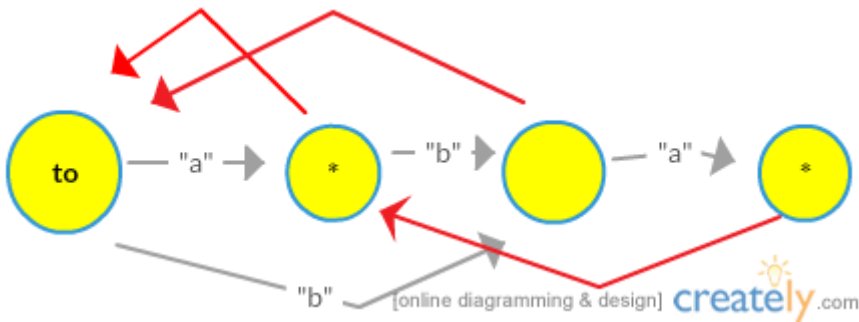
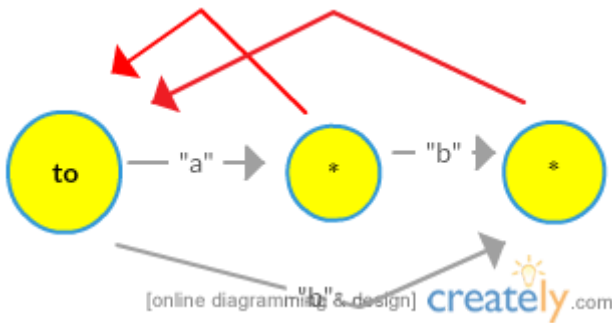
1. The algorithm for construction of suffix-automaton is online i.e we construct by adding a single character to our previous string, modifying the previous structure, lets do it for string " $aba$ ".
2. To achieve linear memory each state will consist of two values **len** (length of longest suffix), **link** (suffix link of the state).

3. Initially our structure consist of a single state ( $t_0$ ), which we shall assume to be zeroth state ( all other states will get numbers 1,2,3...). Assign this state  $len = 0$  and  $link = -1$  ( meaning non-realistic link) .
4. Accordingly the whole task now is to add a single character " $c$ " to the end of the current line.
5. Let ***last*** is the state( node ) that corresponds to the entire current line before adding the symbol ( initially  $last = 0$  ) and after adding each new character , we will change the value to ( 1 , 2 , 3 .... and so on ).
6. After adding a new character we will make a new node ( state ) " $cur$ " , this is obvious as *adding a new character increase the number of endpos classes by atleast 1 as no previous substring can end at the position of " $c$ " and also number of states == number of classes, this we have already discussed . we will assign  $len(cur) = len(last) + 1$  .For finding the suffix-link of this state and modifying the tree we will run a loop as described below.*
7. Till this time we have a created a new state , initialized it but we haven't added it to our structure . For doing so we will run a loop , initially we are at the "***last***" node of tree , if there is no edge/transition with label " $c$ " we will add an edge between "***last***" and " $cur$ " with the label " $c$ " and move to the node pointed by the suffix link of "***last***" change  $last$  to  $suffix\_link(last)$  , we will keep on doing this until we reach ( $t_0$ ) or we encounter the condition mentioned in next point.
8. If at some node the transition with label " $c$ " already exists , we will stop at that node , let that state be represented by " $p$ " , let the state which is connected to " $p$ " via transition with label " $c$ " be represented by " $q$ " , Now we have two cases depending upon if  $len(p) + 1 = len(q)$  or not –
  - if  $len(p) + 1 = len(q)$  , we can simply assign  $link("cur") = "q"$  and break , this happened because this case satisfied the condition of suffix link we discussed in "*suffix link*" section and hence we found the required suffix link.
  - Otherwise we will have to create a clone of state  $q$  with everything remaining same as state  $q$  except for the value of  $len$  such that  $len(clone) = len(p) + 1$  and assign  $link("cur") = "clone"$  and break.
9. If point no. 8 never occurred we reach dummy state -1 ( link of ( $t_0$ ) ) , in this case we simply assigned  $link("cur") = 0$  and exit.

TRYING OUR PROCEDURE ON "*aba*"

**NOTE** – Red ones are suffix links and grey ones are transitions/edges



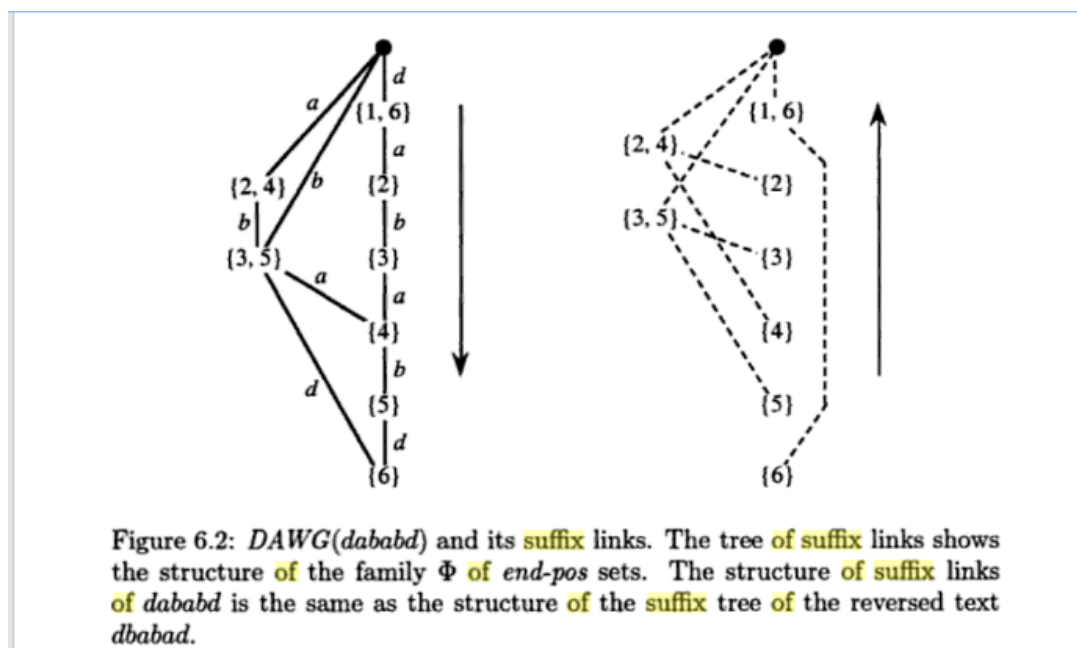


### NUMBER OF NODES IN SUFFIX AUTOMATON

Well to prove the linearity of the algorithm I need to prove that number of nodes made are linear in terms of number of characters in the string. We have earlier discussed that nodes correspond to different "Endposn Classes". and we all proved earlier that any two classes are either "**disjoint or one is a subset of other**". That is

$$\begin{cases} \text{endpos}(w) \subset \text{endpos}(u) & \text{if } u \text{ --- suffix } w, \\ \text{endpos}(u) \cap \text{endpos}(w) = \emptyset & \text{otherwise.} \end{cases}$$

This forms the basis of the proof. The above property induces a tree structure of suffix automaton. I will explain with this example



All leaves will be pairwise disjoint and thus there can be at most  $n$  leaves in a suffix automaton. Let's partition the nodes into two disjoint sets whether the  $\text{val}()$  (longest string in a state) is a prefix of the main string or not. the number of nodes in the first case would be exactly  $n + 1$  corresponding to each prefix of the string and  $\text{end\_posn}$  class of each will consist of exactly one index corresponding to the end-index of prefix.

Eg. Let the string be "abc" then prefixes will be "", "a", "ab", "abc"

We now count the number of nodes in the other subset of partition. Let  $v$  be the node and  $\text{val}(v)$  (longest string in that state) is not the prefix of the main string. Then  $\text{val}(v)$  is a non empty word that occurs at at least two different places, then only it will be disjoint to the previous partition of prefixes.

Thus it will have at least two children "p" and "q" corresponding to  $\text{suffix}[p] = v$  and  $\text{suffix}[q] = v$ , but number of leaves are restricted to " $n$ " therefore at best we can have " $n + n/2 + n/4 + n/8 \dots = 2*n$ " nodes which is an upper bound of number of nodes. To be exact it is  $2*n - 2$  and the worst case scenario occurs for strings like these "abbbb..." .

## IMPLEMENTATION AND CODE

*Application of Suffix – Automaton*

```

1  struct state {
2      int len, link;
3      map<char,int>next;
4  };
5
6  const int MAXLEN = 100000;
7  state st[MAXLEN*2];
8  int sz, last;
9
10 void sa_init() {
11     sz = last = 0;
12     st[0].len = 0;
13     st[0].link = -1;
14     ++sz;}
15
16 void sa_extend (char c) {
17     int cur = sz++;
18     st[cur].len = st[last].len + 1;
19     int p;
20     for (p=last; p!=-1 && !st[p].next.count(c); p=st[p].link)
21         st[p].next[c] = cur;
22     if (p == -1)
23         st[cur].link = 0;
24     else {
25         int q = st[p].next[c];
26         if (st[p].len + 1 == st[q].len)
27             st[cur].link = q;
28         else {
29             int clone = sz++;
30             st[clone].len = st[p].len + 1;
31             st[clone].next = st[q].next;
32             st[clone].link = st[q].link;
33             for (; p!=-1 && st[p].next[c]==q; p=st[p].link)
34                 st[p].next[c] = clone;
35             st[q].link = st[cur].link = clone;
36         }
37     }
38     last = cur;
39 }

```

1 . *Finding number of distinct substrings* : Hey I know its been quite long reading this , but did you remember the 2nd paragraph of this blog which mentioned that every path in a suffix automaton corresponds to a substring , so yes what we need to do is just find the number of different paths which is indeed equal to the number of distinct substrings. Here is the code for that. Everything remains same except instead of map we used a 26 word array , its actually just a *DFS* of the suffix automaton.

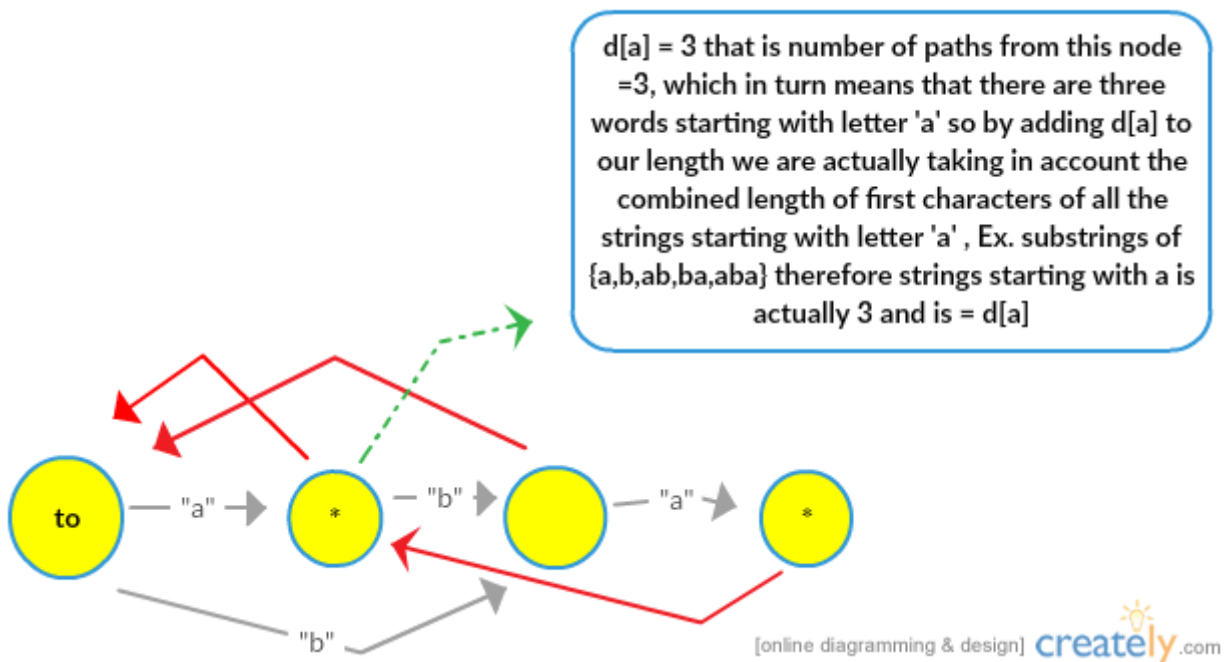


```

1  vector<int>d(MAXLEN,0);
2
3  int distsub(int ver)
4  {  int  tp = 1  ;
5
6      if(d[ver])
7          return d[ver];
8
9      for(int i=0;i<26;i++)
10         if( st[ver].next[i] )
11             tp+= distsub(st[ver].next[i]);
12
13     d[ver]=tp;
14     return d[ver];
15 }

```

2 .*Total Length of all distinct substrings* : For this we need to look at things more closely. Consider



HERE IS THE CODE FOR FINDING THE LENGTH OF ALL SUBSTRINGS

```

1  vector<int>ans(MAXLEN,0);
2
3  int lesub( int ver)
4  { int  tp = 0  ;//To Find Words just add length of words instead of 1
5
6      if(ans[ver])
7          return ans[ver];
8
9      for(int i=0;i<26;i++)
10         if( st[ver].next[i] )
11             tp = lesub(st[ver].next[i]) + d[st[ver].next[i]];
12
13     ans[ver]=tp;
14     return ans[ver];
15
16 }

```

### 3) The Lexographically kth substring

The solution to this problem is based on the same idea as previous two tasks. To find the kth Lexographically substring we searched for the kth path moving from the root and rest is taken care by our map or array which already have sorted keys/alphabets so that we always pick the transition with smallest possible character thereby maintaining the lexicographical order.

#### HERE IS THE C++ CODE FOR FINDING THE KTH LEXICOGRAPHICAL SUBSTRING

```

1  void klex(int ver)
2  {
3      vector<int>ans;
4      for(int i=0;i<26;i++)
5          if( st[ver].next[i] )
6              { path++;
7                if (path==k)
8                    { ans.push_back((char)('a' + i));
9                      return;}
10
11                 klex(st[ver].next[i],ans);
12                 if (path==k)
13                     { ans.push_back((char)('a' + i));
14                       return;}
15
16                 }
17
18 }

```

### 4) Smallest Cyclic Shift to obtain lexicographical smallest of All possible

Consider string “aba” then the minimum cyclic shift is ‘1’ because by shifting the pattern by one we get string “aab” which is lexicographically smallest of all possible ( “aba” , “aab” , “baa” ) . For solving this problem we build the suffix automaton of “**String + String**” Now the problem is reduced to above problem where  $k = 1$  , solving until we get substring of length = length of given string,

#### C++ IMPLEMENTATION

Don't worry about the fpos part that is done to find the start position of minimum lexicographical string so that we can find number of shifts.

```

1  int  tp = 0 ;
2  string s;
3  int t = s.length();
4  s = s+s;
5  void  minshift(int ver)
6  {
7
8
9      for(int i=0;i<26;i++)
10         if( st[ver].next[i] )
11             { tp++;
12
13                 if(tp==t)
14                     {cout<<st[ver].fpos - t + 2<<endl;
15                     break ;}
16
17                     minshift(st[ver].next[i]);
18                     break;
19
20             }
21 }
22

```

### 5) POSITION OF FIRST OCCURRENCE

Given a text T and we receive request of the form : given a pattern P we need to know the position of first occurrence of P in T .

To solve this we will build the suffix automaton of the Text T , we also need to add in the preprocessing finding fpos ( first position of occurrence) for all states of automaton, such that  $fpos[cur] = len[cur] - 1$  and  $fpos[clone] = fpos[q]$  , then answer to our query is simply equal to  $fpos[t] - p.length() + 1$  where "t" is the state corresponding to pattern , p is the pattern. Here is the code

```

1  st[cur].fpos = st[cur].len - 1;
2  st[clone].fpos = st[q].fpos;
3  at appropriate positions in sa_extend function */
4
5  int firstpostn()
6  {  int s_t = 0 ;
7      int vtx = 0;
8      int u = s_t;
9      while(s_t<p.length())
10         {vtx = st[vtx].next[p[s_t] - 'a'];
11         s_t++;}
12
13     cout<<st[vtx].fpos - p.length() + 1 <<endl;
14 }
15

```

## 6) COUNT NUMBER OF OCCURRENCES

To solve this we need to build suffix automaton of  $S$ , along with this we also need to make a preprocessing for each state  $v$  to find  $count[v]$  which is equal to the size of  $endpos(v)$  since it would not be possible to store all suffixes in  $endpos(v)$ , we will instead store its size. For each state, other than initial state or states obtained by cloning we initially assign  $count[v] = 1$ . Then we will go over all the states in descending order of their length  $len$  and calculate  $count[v]$  using  $count[link[v]] += count[v]$ . After this answer to query is  $count[t]$  where " $t$  is the state of the pattern". Here is the code which find occurrences of pattern string " $pa$ " in main string " $s$ ".

```

1  set<pair<int,int> base ;
2  int s_t = 0
3  /* ADD these 4 lines at appropriate places in sa_extend function
4
5      cnt[cur] = 1 ;
6      base.insert(make_pair(st[cur].len, cur));
7
8      cnt[clone]=0;
9      base.insert(make_pair(st[clone].len,clone));
10
11  */
12
13  int repstr()
14  {
15      set::reverse_iterator it;
16      for( it=base.rbegin(); it!=base.rend(); it++ )
17          cnt[ st[ it->second ].link ] += cnt[ it->second ];
18
19      int vtx = 0;
20
21      while(s_t<=pa.length()-1)
22          {vtx = st[vtx].next[pa[s_t] - 'a'];
23            s_t++;}
24
25      cout<<cnt[vtx]<<endl;
26  }
```

## 7) LONGEST COMMON SUBSTRING OF TWO STRINGS

We now go on line  $T$  and look for each prefix of the longest suffix prefix  $S$  occurring. In other words, for each position in the line  $T$  we want to find of the longest common substring  $S$  and  $T$  ending at that position.

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

Initially,  $p = t_0$  and  $l = 0$  i.e empty match .

Now let us consider the character  $T[i]$  and we want to recalculate the answer for it.

- If the state of  $v$  the automaton has a transition with label  $T[i]$ , we simply commit this change and increase  $l$  by one.
- And if the state  $v$  is not of the desired transition, then we should try to shorten the current matching portion for which it is necessary to go to suffix link:

```
v = link(v).
```

```
l = len(v).
```

Unless we find a new state with required transition or we reach a fictitious state  $-1$ , we have to go on the suffix link.

The answer to the problem will be a maximum of the values  $l$  for all time rounds.

Here is the code For that

```

1  string lcs (string s, string t) {
2      sa_init();
3      for (int i=0; i<(int)s.length(); ++i)
4          sa_extend (s[i]);
5
6      int v = 0, l = 0,
7          best = 0, bestpos = 0;
8      for (int i=0; i<(int)t.length(); ++i) {
9          while (v && ! st[v].next.count(t[i])) {
10             v = st[v].link;
11             l = st[v].length;
12          }
13          if (st[v].next.count(t[i])) {
14             v = st[v].next[t[i]];
15             ++l;
16          }
17          if (l > best)
18             best = l, bestpos = i;
19      }
20      return t.substr (bestpos-best+1, best);
21  }
```

After this long post we have come to an end for a new beginning , hope you enjoyed reading this post 😊 , I will love to hear from you , do reply and give me an opportunity to improve my self .  
Happy coding 😊

For further reference read translate version of –

SUFFIX AUTOMATA ([http://e-maxx.ru/algo/suffix\\_automata](http://e-maxx.ru/algo/suffix_automata)).

OR

JEWELS OF STRINGOLOGY ([https://books.google.co.in/books?id=9NdohJXtlyYC&pg=PA70&lpg=PA70&dq=number+of+nodes+in+suffix+automaton&source=bl&ots=ln9h5BJ6Ia&sig=8-](https://books.google.co.in/books?id=9NdohJXtlyYC&pg=PA70&lpg=PA70&dq=number+of+nodes+in+suffix+automaton&source=bl&ots=ln9h5BJ6Ia&sig=8-s8u2KDSJEnIzy4b524PK1pu3I&hl=en&sa=X&ved=0ahUKEwjRmNGdrJXOAhVFRo8KHcSBD68Q6AEIUzAJ#v=onepage&q=number%20of%20nodes%20in%20suffix%20automaton&f=true)


[s8u2KDSJEnIzy4b524PK1pu3I&hl=en&sa=X&ved=0ahUKEwjRmNGdrJXOAhVFRo8KHcSBD68Q6AEIUzAJ#v=onepage&q=number%20of%20nodes%20in%20suffix%20automaton&f=true](https://books.google.co.in/books?id=9NdohJXtlyYC&pg=PA70&lpg=PA70&dq=number+of+nodes+in+suffix+automaton&source=bl&ots=ln9h5BJ6Ia&sig=8-s8u2KDSJEnIzy4b524PK1pu3I&hl=en&sa=X&ved=0ahUKEwjRmNGdrJXOAhVFRo8KHcSBD68Q6AEIUzAJ#v=onepage&q=number%20of%20nodes%20in%20suffix%20automaton&f=true)).

## Advertisements

AUTOMATIC

**We're hiring  
backend developers.  
Join us!**

APPLY



REPORT THIS AD

## 16 thoughts on “SUFFIX AUTOMATON by- saisumit”

1. [RADN](#) [February 6, 2016 at 2:30 pm](#) [REPLY](#)

Can you explain the automata for abcbc ? Thanks for the tutorial

[SAISUMIT](#) [February 7, 2016 at 8:51 am](#) [REPLY](#)

If u have any problem with construction of suffix automata, try using this

<http://rain.ifmo.ru/cat/view.php/vis/strings/suffix-automaton-build-2009> , its a step wise guide or else tell me where exactly you are having a problem building it.. 😊

2. [ANON](#) [February 16, 2016 at 8:23 pm](#) [REPLY](#)

Why is it  $O(N)$ ? How do you prove that the clone part of the algo doesn't make it  $O(N^2)$  ?

[SAISUMIT](#) [February 17, 2016 at 4:39 pm](#) [REPLY](#)

Well , as time complexity in construction of suffix automaton is directly proportional to the number of states in suffix automaton and since size of suffix automaton is linear ( at most  $2*n - 1$  nodes ) where  $n$  is the length of the string , therefore the algorithm is linear and worst case occurs for the strings of type such as “abbbbb...” where at each step after initial step 2 nodes are added

3. [LAAKERI](#) [February 16, 2016 at 9:06 pm](#) [REPLY](#)

Your solution for kth substring is  $O(n^2)$ . To make it  $O(n)$  you need do the occurrences count dp beforehand.

[SAISUMIT](#) [February 17, 2016 at 4:45 pm](#) [REPLY](#)

Well its linear in  $O(\max(n, k))$  , its just like dfs ...

4. [SOMEONE\\_YOU\\_KNOW](#) [June 20, 2016 at 10:48 pm](#) [REPLY](#)

how can you prove that the building time of suffix automaton is directly proportional to nodes (doesn't makes any sense, how it could be) ? at an intuitive level it certainly looks  $O(n^2)$  . A prove will surely help

SAISUMIT July 28, 2016 at 8:06 am REPLY

I think you are having trouble with that inner loop , if u look at that loop carefully , it is dependent upon the character set we are working upon ( 256 ) . If there is link to the node ( refer point 8 ) in the construction we terminate the loop and if there is not a link to the character. we add that character to its links , so we have to traverse atmost the size of the character set we are using to reach the desired node and complexity reduces to  $O(256*n)$  , Hope this clears your doubt . I have also added why there are linear number of nodes in suffix automaton you can also have a look at that

5. KOUSHIK September 4, 2016 at 8:51 pm REPLY

Is the condition ( `st[p].next[c] == q` ), in line 33, in the code for the construction of suffix automaton necessary? Will it not be true always?

6. RANDOM\_CODER June 28, 2017 at 1:30 pm REPLY

Brilliant Translation and Explanation....Thanks a Lot.....  
btw in the 2nd solution(Total Length) on line 11: it's ' `tp+=....` '

7. DINOBLAST November 5, 2017 at 6:15 pm REPLY

Hi, I was trying to get a grasp about how this works, so I added a 'main' and a print part to your first code, after 'IMPLEMENTATION AND CODE':

```
int main(){
char string[]="abcbc";
int ix = -1;
while(string[++ix]){
sa_extend(string[ix]);
}
for(int j=0;j<ix;j++){
printf("len[%d] link[%d] edges["j,st[j].len);
for (int k=0;k<26;k++){
char c = ((char)((int)'a')+k);
if (st[j].next.count(c))
printf("(%c,%d), ",c,st[j].next[c]);
}
printf("]\n");
}
}
```

and i got this result

```
len[0] link[1] edges[(a,1), (b,2), ]
len[1] link[2] edges[(a,0), (b,2), ]
len[2] link[2] edges[(c,3), ]
len[3] link[3] edges[(b,4), ]
len[4] link[4] edges[(c,5), ]
```

What is weird for me is the second state, len[1] where there is this `map[a]==0`

That means state 1 has an edge to state 0?

That is an error? That is a loop right?

Did I make a mistake?

8. ANAND [January 21, 2018 at 7:28 am](#) [REPLY](#)

Can you please post the code for the longest common substring of several strings?

9. WRZZ [August 19, 2018 at 2:39 pm](#) [REPLY](#)

1. About "Construction of Suffix Automaton in Linear Time" , the 8th point and  $\text{len}(p) + 1 \neq \text{len}(q)$ .

it should assign  $\text{link}("q") = \text{"clone"}$  too.

2. And go through the state p of the suffix links, and for each next state check: if there was a transition to the state q, then redirect it to the state clone (and if not, then stop).

3. And don't we should update last.

like your code:

```
``cpp
for (; p!=-1 && st[p].next[c]==q; p=st[p].link)
st[p].next[c] = clone;
st[q].link = st[cur].link = clone;
``
```

And thanks. I got a lot from this essay.

10. Pingback: [test - Kanari's Blog](#)11. Pingback: [后缀自动机 - Kanari's Blog](#)12. [নাস্টম মল্লিক জয়](#) [April 28, 2020 at 10:43 pm](#) [REPLY](#)

You started very well at beginning of your explanation ,but not good at all.You tried it's good ....

SAISUMIT

*Living for Living*

[BLOG AT WORDPRESS.COM.](#) [DO NOT SELL MY PERSONAL INFORMATION](#)