Date: Thursday, February 18th

Lecture: Dr. Mihai Pop

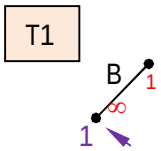
Scribe: Hyoungtae Cho

Administrivia

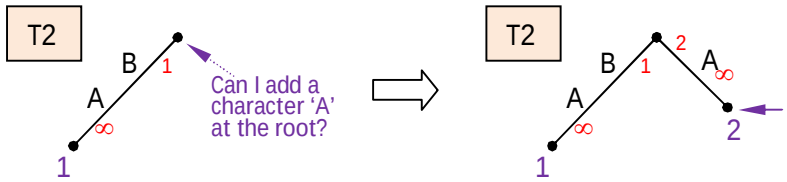
1. Gusfield text for chapter 5&6 about suffix trees are scanned and uploaded on the web
2. List of Project ideas is uploaded

Building Suffix trees

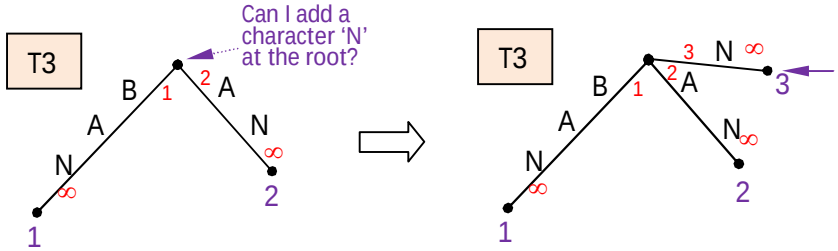
Example string: BANANA$



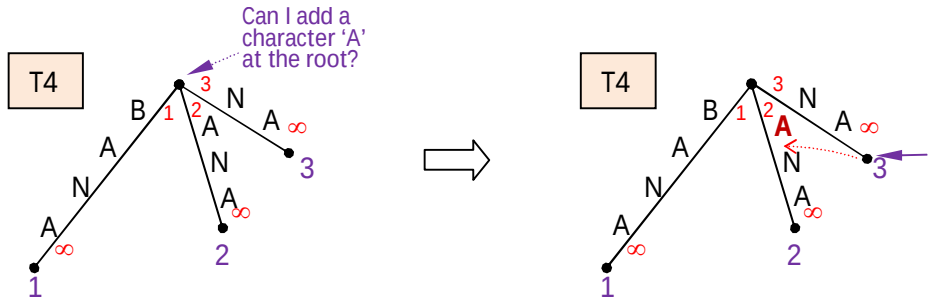
There's only one suffix in T1, which is suffix 1. The label 1 is a starting position where the suffix starts and ∞ specifies this edge would keep continuing to grow throughout the algorithm. The pointer points out the node which is last explicitly created in the tree.



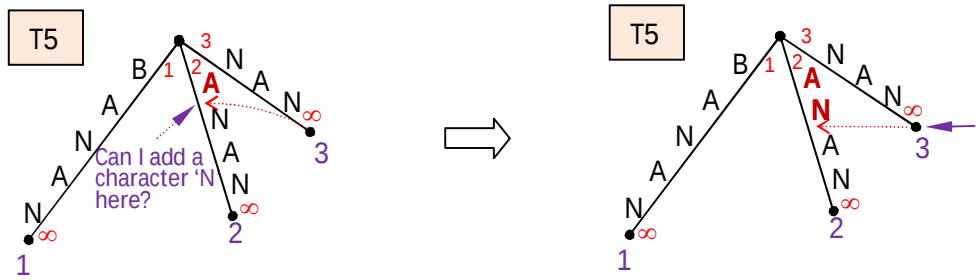
After adding a character 'A', then jump through the suffix link to the root and ask if a new suffix would be added. In this case, we added a new suffix since the tree doesn't have a suffix starting with a character 'A'.



In the same manner, we add a new suffix for a character 'N'.

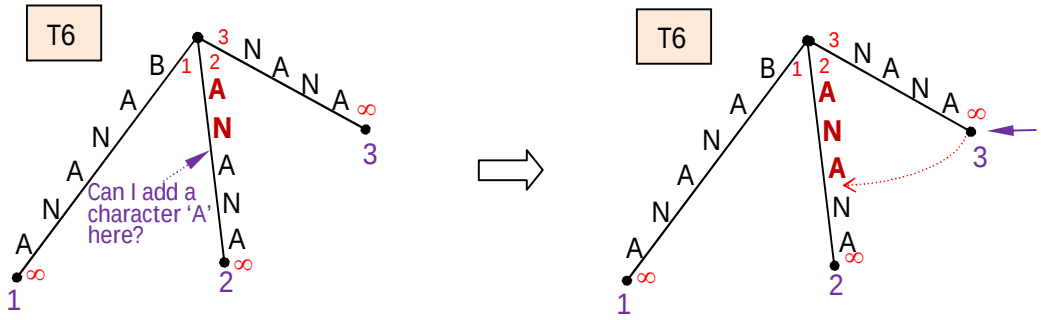


This is a bit tricky part. Following a suffix link to the root, find a suffix starting with a character 'A' to determine if a new suffix should be added. In this case, suffix 2 has already had 'A' as the first character in the edge so that we don't do anything. The pointer stays at the node that is edited last time which is the same position in T3. Then create and keep a virtual pointer(dotted in red) from the suffix tree 3 to one node away from the root.

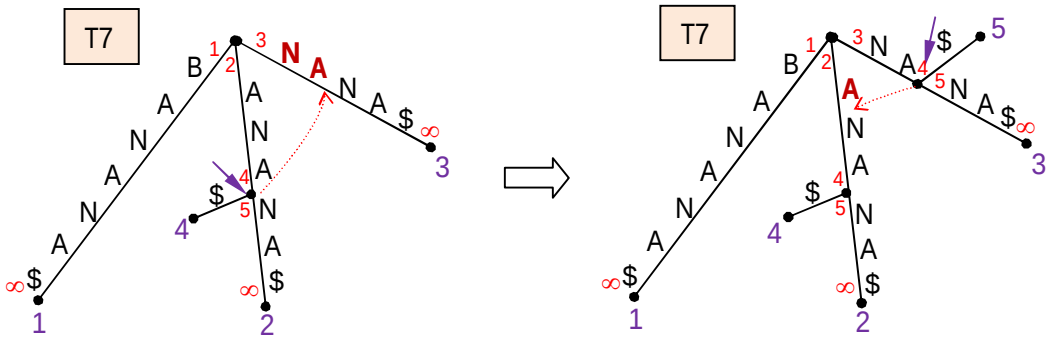


Following the suffix link, go to one node down from the root in the suffix 2, move one node down without doing anything for the tree since there is a character 'N', and update the virtual pointer(link).

`

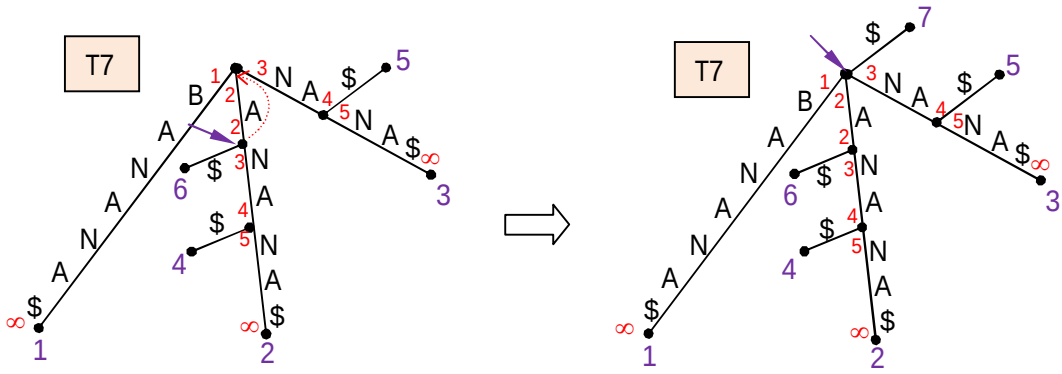


Now we have a full string. In the same manner in T5, we move one node down updating the virtual link without modifying the tree. We keep track of the location where we are using a virtual link as well as the pointer that indicates the node edited last time in order to save computation time.



Finally we add a special character '$' that is not expected to be in the string. After adding it on suffix 1, 2, and 3, move the pointer through a virtual link on the suffix 3 to 'ANA' on the suffix 2. Since there is no '$', we create a new node of the character for suffix 4 and split the edge into two with labels of (2, 4) and (5, ∞).

How to find a position for suffix 5? Dr. Pop explain it that we can move to 'NA' on suffix 3 through a suffix link by asking the root which edge we need to take and how many nodes we are going down. This computation can be done in linear time using skip/count trick.



At the end, we have suffix 6 and 7 through suffix links.

Suffix Tree built in Linear Time Algorithm

Hence, we can build a suffix tree in a linear time proving three parts:

1) Once a leaf created, we don't have to touch it again.
2) Once we find a suffix on the other suffix edges, we don't have to work additionally in the tree except expanding a new leaf.
3) In the final tree, the number of jumps through suffix links is at most O(n).

Suffix tree is always a good data structure?

- ➔ Good: Solve the longest common substring problem efficiently!
- ➔ Bad: Suffix tree is based on pointers that require much memory. Considering a huge string such as DNA sequence, a suffix tree may require too much space to be practical. Is there any compact data structure better than this?
- ➔ Taking an advantage of suffix tree that efficiently stores all the suffixes in the text, why don't we sort the suffix indexes to use a binary search? **Suffix Array**

Suffix Array

Example string: BANANA

1	B	A	N	A	N	A
2		A	N	A	N	A
3			N	A	N	A
4				A	N	A
5					N	A
6						A

sort ➔

6	A					
4	A	N	A			
2	A	N	A	N	A	
1	B	A	N	A	N	A
5	N	A				
3	N	A	N	A		

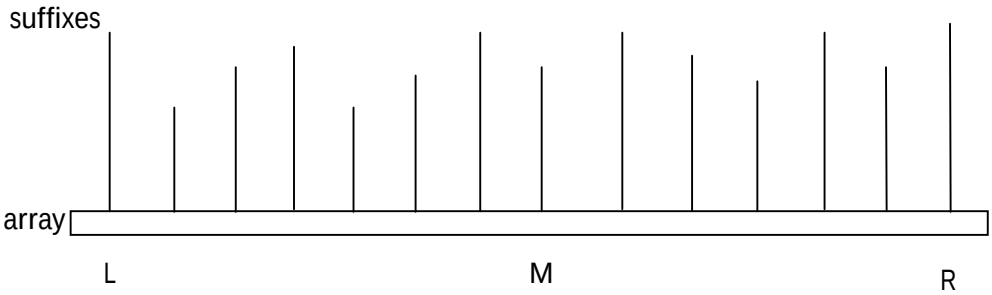
However, this simple solution on the left uses $O(N^2)$ memory so that we sort suffixes and store only their indexes like on the right using $O(N\log N)$ memory – N for the number of suffixes and logN for binary search. Sometimes logN part can be ignored, though.

As suffix tree is built in linear time, suffix array can be generated in linear time from suffix tree using a lexical depth-first search.

For Suffix Array, O(N) space and O(NlogN) search

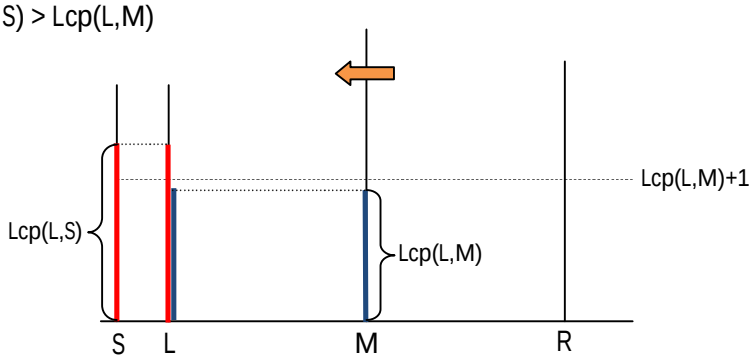
- ➔ Can we fix this searching time?

Sol) Using '**Longest Common Prefix**'(Lcp) of two strings would be a good help!



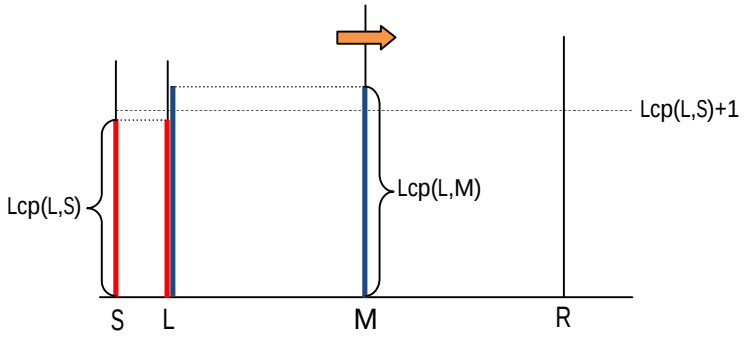
Doing a binary search using comparing Lcp(L,S), Lcp value of Left(L) and Middle(M) with Lcp(L,R), Lcp value of Left(L) and Right(R). There are three cases for the comparison and we only work for the case 3 that both Lcp values are identical.

1. Lcp(L,S) > Lcp(L,M)



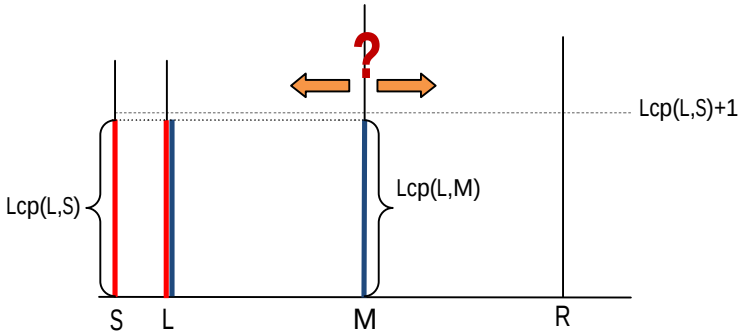
→Go Left! The common prefix of suffix L and S is longer than the common prefix of suffix L and M so that the Lcp(L,M)+1 characters of S and L are identical and lexically greater than character Lcp(L,M)+1 of suffix M. In this case, R is changed to M while L and Lcp(L,S) remains unchanged, and S must occur to the left of position M.

2. lcp(L,S) < lcp(L,M)



→Go Right! The common prefix of suffix L and S is smaller than the common prefix of suffix L and M so that the Lcp(L,S)+1 characters of L and M are identical and lexically less than character Lcp(L,S)+1 of suffix S. In this case, L is changed to M while R and Lcp(R,S) remains unchanged, and S must occur to the right of position M.

3. lcp(L,S) = lcp(L,M)



→ Compare! Since the prefix of suffix L and suffix M are identical, we need to lexically compare S to M to determine which of L or R change. In other words, we will check if S must occur to left position or right position of M based on lexical comparisons between S and M.

Hence, using the Lcp values the search can be done in O(N+logN) time instead of O(NlogN).
(N: character comparison for Lcp, logN: binary search)