Lecture Notes by Raul Guerra

Binary Search in Suffix Arrays

"A suffix array is basically a sorted list of all the suffixes of P" (p.320, Manber,Myers). For these lecture notes, P is the pattern and T is the text.

Running time from a typical binary search between the suffix array of P and T (to find the suffix of P that matches the biggest amount of characters with T) is `O(|P|*log|T|)`, the `|P|*` comes up because of the characters in the strings that need to be compared.

With the help of some precomputed information a binary search between T and the suffix array of P can be improved from `O(|P|*log|T|)` to `O(|P|+log|T|)`.This precomputed information is the LCP. We utilize the LCP to figure out whether to recurse to the left side or right side of the middle point M in a binary search.

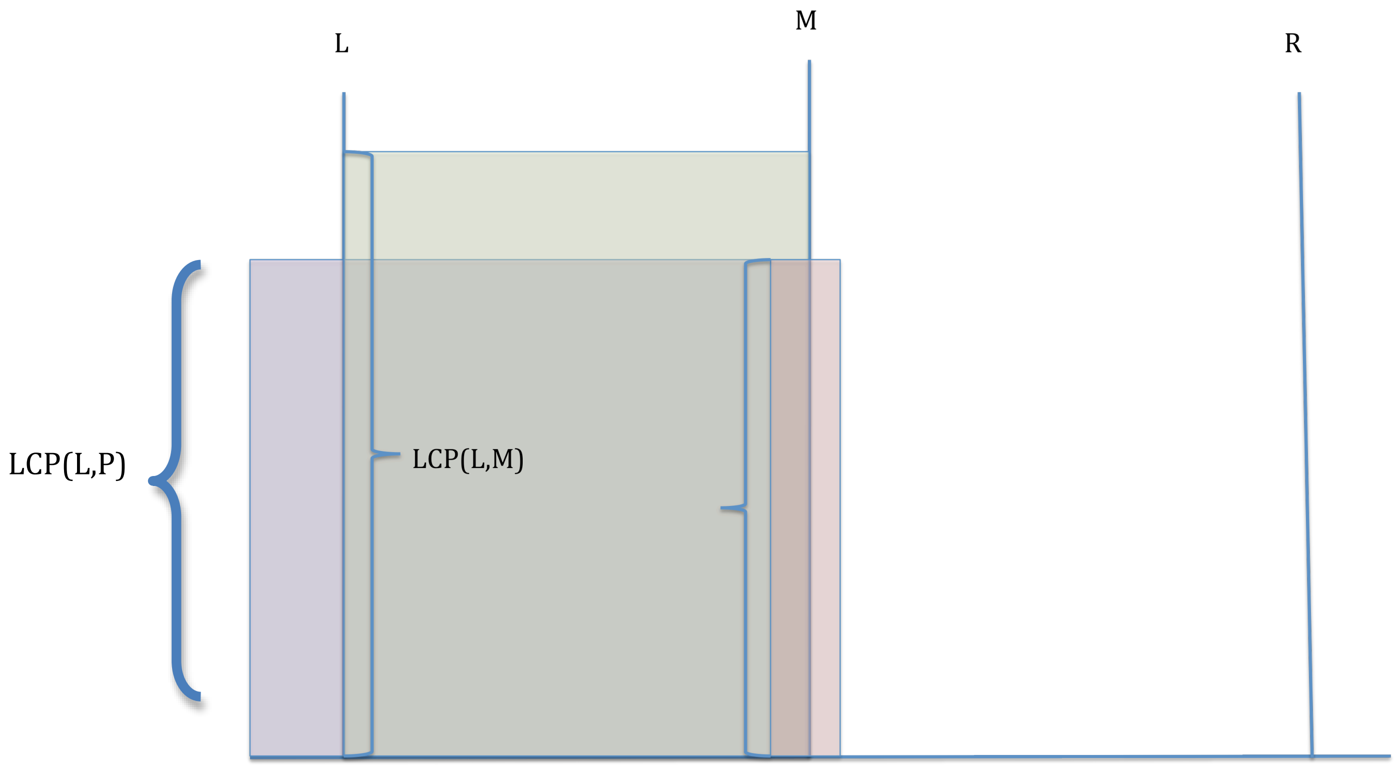
```
LCP(x,y) //LENGTH of the Longest Common Prefix of Strings x & y

Lets define h = max(LCP(L,P) , LCP(P,R))

L //Left boundary of Binary search, L is a suffix of P in the suffix
array

R //Right boundary of Binary search, R is a suffix of P in the suffix
array

M //Midpoint suffix between L and R in the Binary search, M is a suffix
of P in the suffix array
```

Use of LCP values in binary search

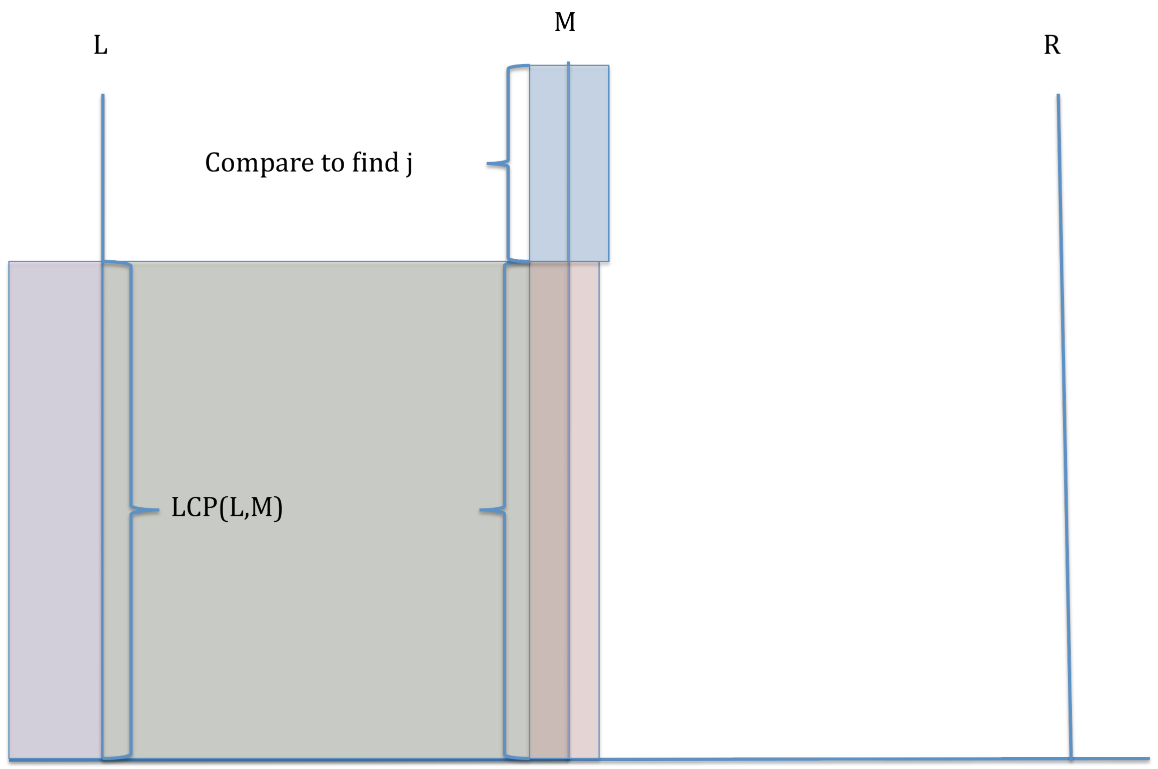
```
//Assumming  LCP(L,P)  >= LCP(P,R) we can focus only on the left side.
```

<u>Cases</u>

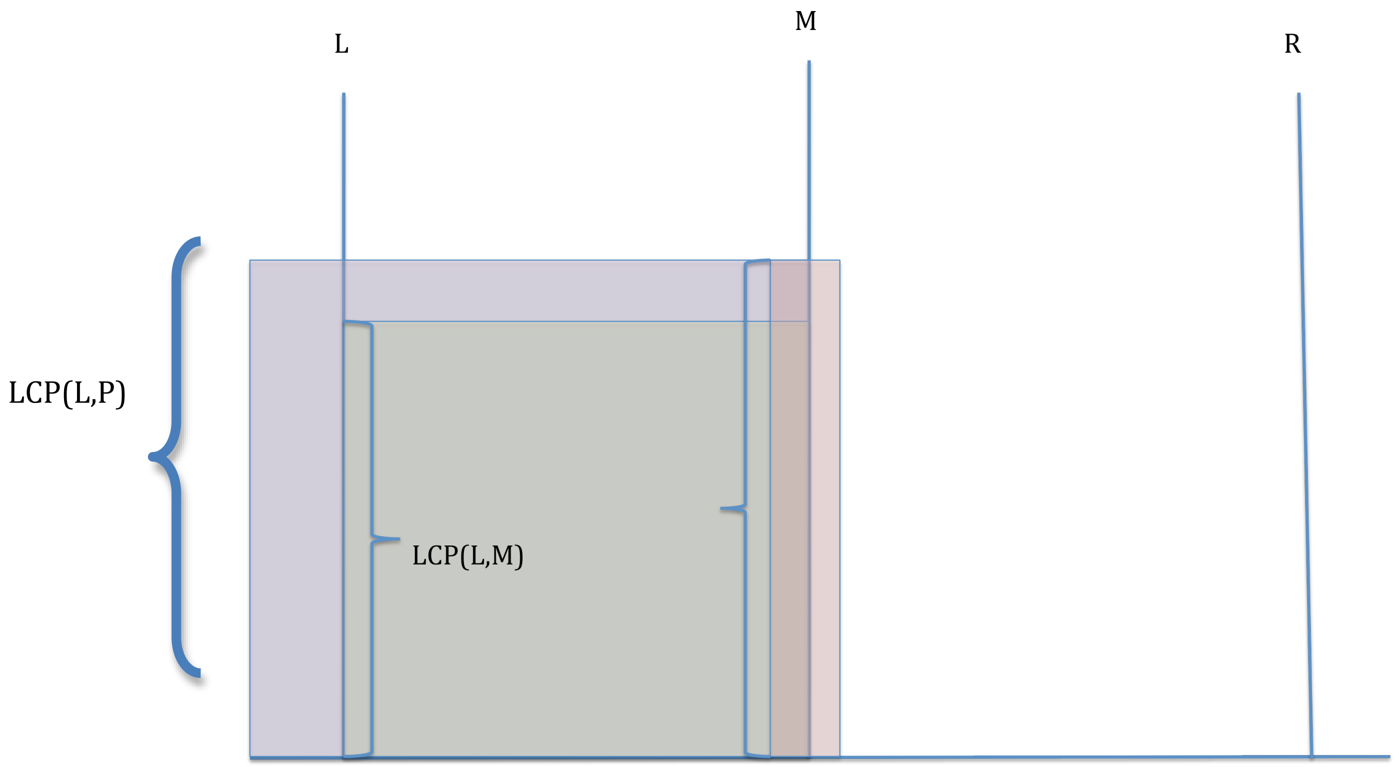
```
if LCP(L,M) > LCP(L,P)
 then    recurse on right and LCP(L,P) remains unchanged
```



```
if LCP(L,M) == LCP(L,P)
 then
     Compare with middle to see if it is to the right of the Left
     part, match characters to decide where side to recurse on

     We need to compare only the LPC(L,P)+1st symbol, LPC(L,P)+2nd
     symbol, and so on, until we find one, say LPC(L,P) + j, such that
     the LPC(L,P) + jth symbol determines whether to recurse on the
     Left side or Right side. In either case the new value for
     LCP(L,P) or LCP(P,R) is LCP(L,P)+j. //See pag. 322 of Manber and
     Myers paper.
```



```
if LCP(L,M) < LCP(L,P)
 then
      recurse on left and LCP(P,R) = LCP(L,M)
```



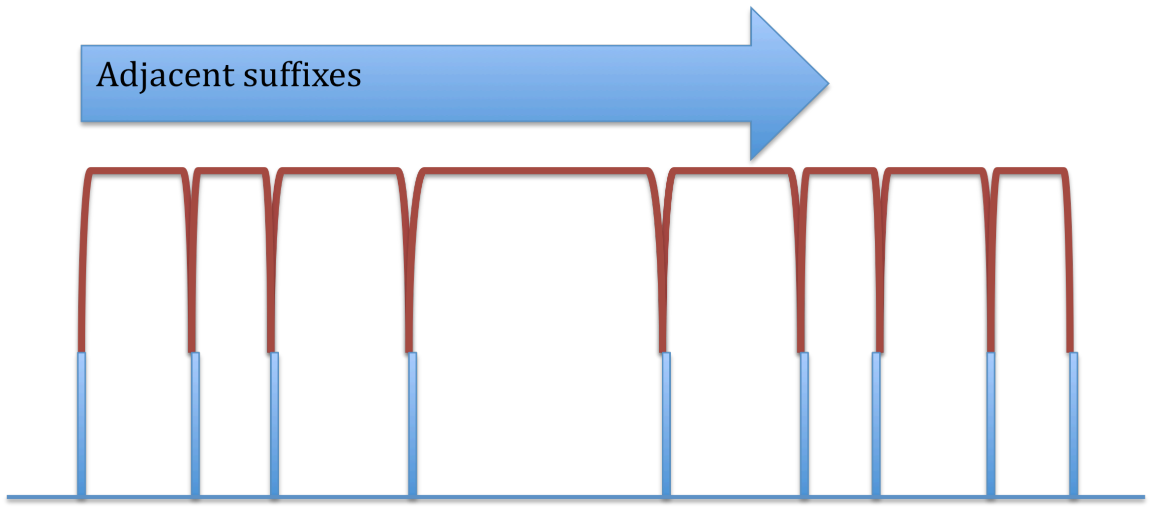
If we assume the LCP values are given, then work at each given step of the binary search is either constant or consists in comparing a small part of the characters in the middle string.

NOTES

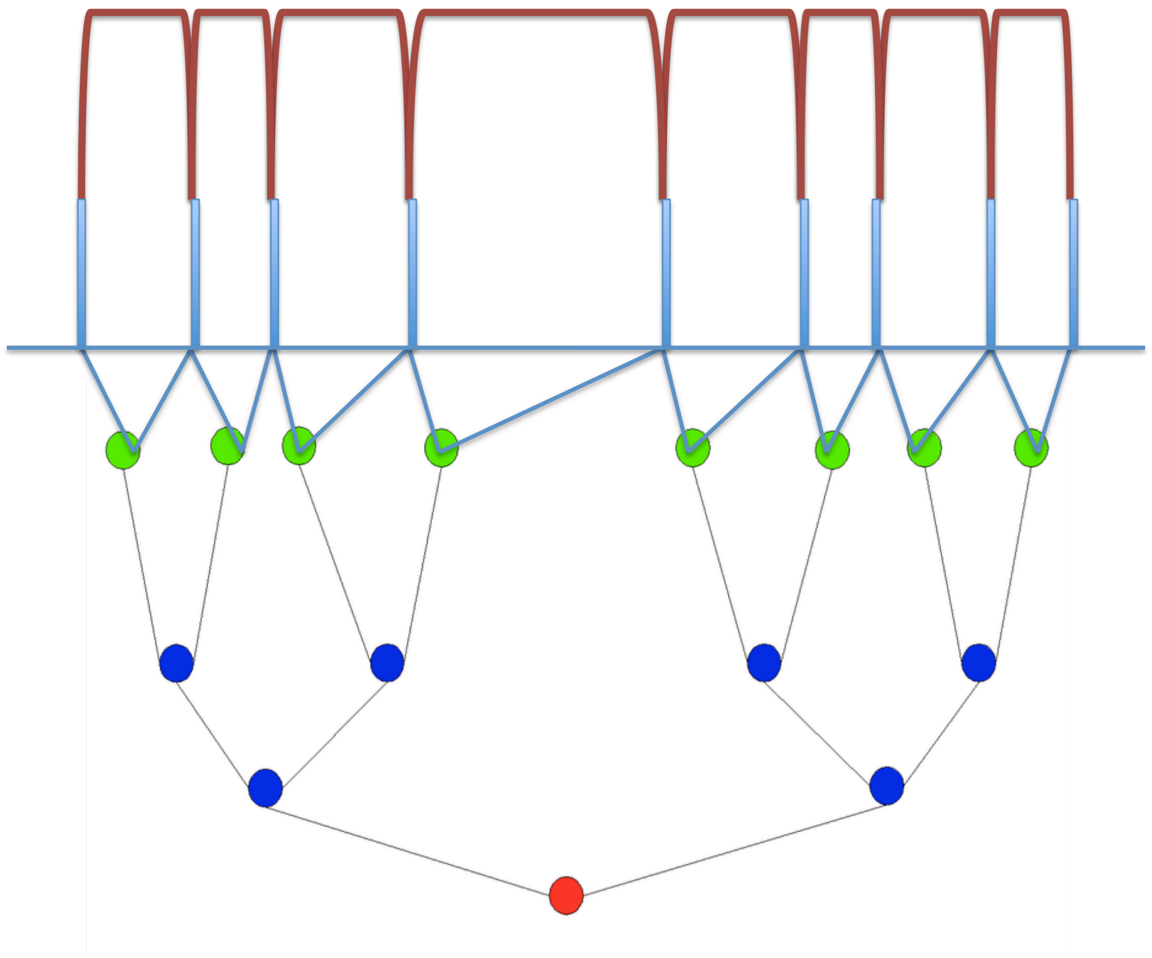
As the range in the Binary Search decreases LCP increases. Also we don't compare middle characters twice

Computation of LCP values from suffix trees

How many (L,R) pairs do we need to compute the LCP values we need?



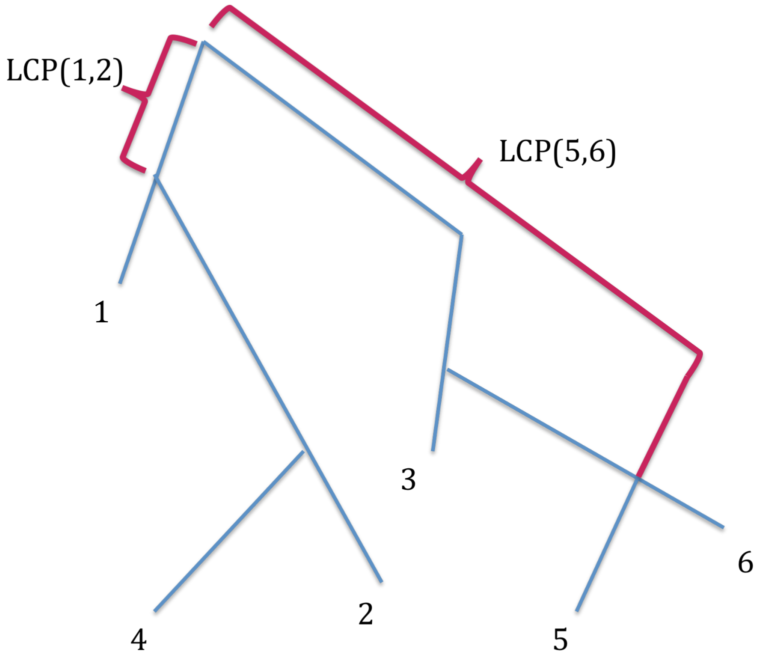
We first compute the LCP of adjacent suffixes, LCP 1,2|2,3|3,4| ... Then we figure out which are the suffixes that can arise in the inner loop of the binary search. We compute the LCP for every node that would be in a binary search tree of the suffixes. The LCP of a parent node in the tree will equal the minimum LCP of the children.



(Source http://commons.wikimedia.org/wiki/File:Binary_tree.png)

To calculate the LCP of two adjacent suffixes we utilize a Suffix tree.

SUFFIX TREE

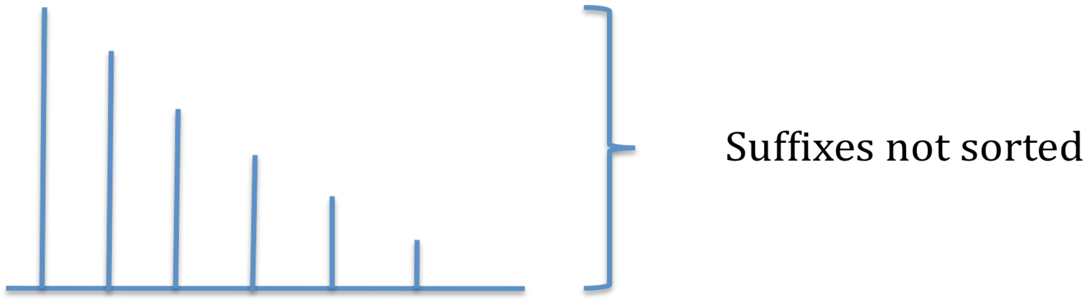


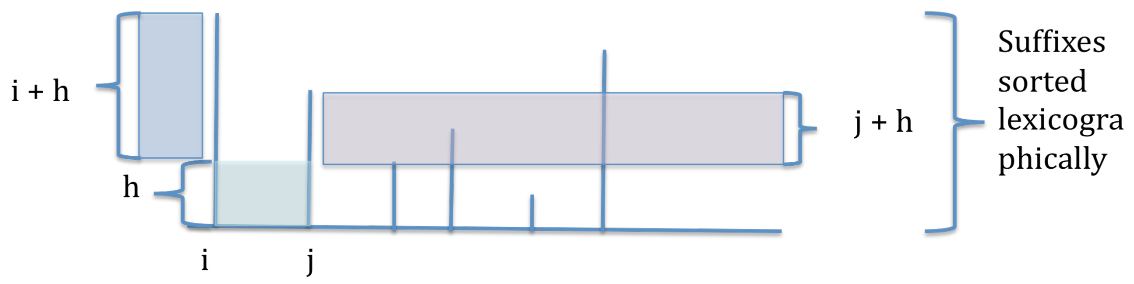
LEXICOGRAPHICAL ORDER

The LCP value can be calculated by keeping track of the highest point between two lexicographically adjacent leaves. In the image above, the leaves are ordered "lexicographically". I use quotations because all suffixes that have equal p-prefixes must appear in consecutive positions in the lexicographically ordered suffix tree. However that is not the case in the tree. However in this example Prof. Pop wanted to show the concept of what the highest point between two leaves would be.

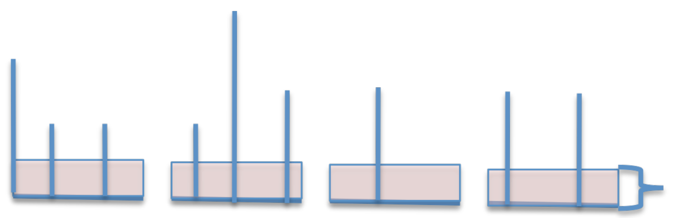
Trick to sorting the suffixes in linear time

The trick is Radix sort





If the first `h` characters are equal between `i` and `j`, then their lexicographical ordering depends on the ordering of the suffixes `(i + h)` and `(j + h)`. We can utilize this idea to sort the suffixes.


All suffixes in each of these bucket share the first character. h = 1
Buckets

Create buckets and put in them all the suffixes that share the first h characters. Then fuse buckets `log N` times, doubling the value of `h` each time.

Example

One letter

h=1

A		
ANANA		NANA
ANA	BANANA	NA
A	**B**	**N**

Two letters

h=2

	ANA		NANA
A	ANANA	BANANA	NA

A$ **AN** **BA** **NA**

Four letters

h=4

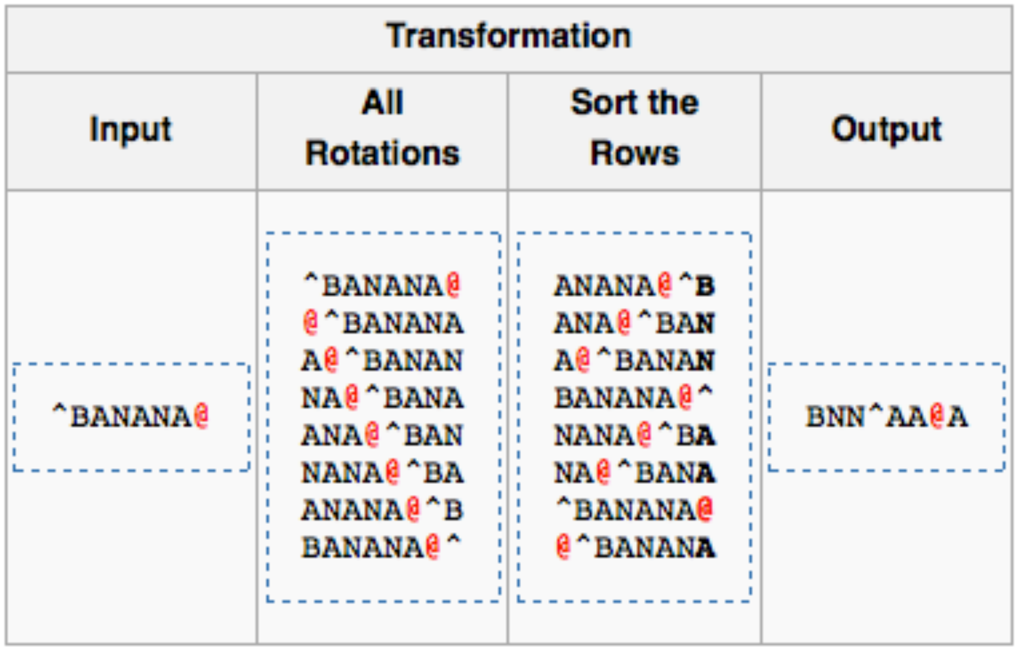
A	ANA$	ANANA
A$$$	**ANA$**	**ANAN**
BANANA	NA$$	NANA
BANA	**NA$$**	**NANA**

This sort is `O(NLogN)` as opposed to `O(N^2LogN)`

```
Memory Complexity = (NLogN) //The LogN comes from the bits used to
store the numbers
```

Burrows-wheeler transform

Construction



Transformation			
Input	All Rotations	Sort the Rows	Output
^BANANA@	^BANANA@ @^BANANA A@^BANAN NA@^BANA ANA@^BAN NANA@^BA ANANA@^B BANANA@^	ANANA@^B ANA@^BAN A@^BANAN BANANA@^ NANA@^BA NA@^BANA ^BANANA@ @^BANANA	BNN^AA@A

(Source Wikipedia's Burrows–Wheeler transform entry)