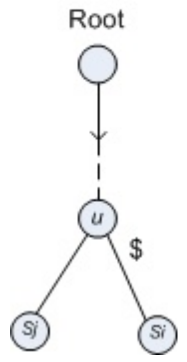
# <u>CSE 5095: Research Topics in Big Data Analytics</u>
# <u>Lecture 14 (March 11, 2014)</u>
**<u>Prof. Sanguthevar Rajasekaran</u>**
**<u>Prepared By: Mohammad Galib Hasan</u>**

**<u>Problem:</u>** All pairs suffix-prefix problem
**Input:** $S_1, S_2, S_3, \ldots, S_n$
**Output:** For every ordered *(i,j)*, the length of the longest suffix of $S_i$ that is a prefix of $S_j$

**<u>Claim:</u>** We can solve this problem in O($M+n^2$) time; where $M = \sum_{i=1}^{n} |S_i|$.
**<u>Proof:</u>**
Construct a generalized suffix tree *Q* for $S_1, S_2, S_3, \ldots, S_n$ in O(*M*) time.
Consider any string $S_j$. There exists a path starting from the root in *Q* corresponding to $S_j$. Let this path be called $P_j$. Call any edge as terminal if its label is \$ and one of its end points is a leaf. If *u* is any internal node along the path $P_j$, a terminal edge is incident on *u,* and the corresponding leaf corresponds to a suffix of $S_i$, then, clearly, the path label of node *u* is a suffix of $S_i$ that overlaps with a prefix of $S_j$. If *u* is the deepest such node for $S_i$, then, the path label of *u* will correspond to the longest suffix of $S_i$ that overlaps with a prefix of $S_j$.



We can use this observation to solve the all pairs suffix-prefix problem. Specifically, with a single traversal through $P_j$, we will be able to identify the longest suffix of $S_i$ that is a prefix of $S_j$, for every $i \in [1, n]$.

For any internal node $u \in Q$, we define a set *L(u)*. $i \in L(u)$ if there exists a terminal edge incident on *u* such that the corresponding leaf is a suffix of $S_i$. Note that we can compute *L*(*u*) for every node *u* in *Q* with a single traversal through *Q* in O(*M*) time.

We will employ *n* stacks, one for each string.
We do a DFS (Depth First Search) starting from the root. When we visit the node *u* in forward search, PUSH the node *u* onto top of stack *i*, for every $i \in L(u)$.

When we reach a leaf corresponding to the entire $S_j$ (i.e., a leaf node labeled (*j,* 1)), then the top of every stack corresponds to the answer we look for.
Specifically, if node *u* is at the top of stack *i*, then the path label of *u* is the longest suffix of $S_i$ that overlaps with a prefix of $S_j$. If any stack *k* is empty then there is no overlap between $S_k$ and $S_j$.

When a node *u* is visited in the reverse traversal, then we POP the top of stack *i*,
for every $i \in L(u)$.

**Analysis:**

The number of PUSH operations is ≤ *M* since we perform a PUSH at most once for every terminal edge.
The number of POP operations is ≤ *M* for a similar reason.
Time to output the answers is $O(n^2)$.
So, the total runtime = $O(M+n^2)$

## Suffix array:

**FACT:**

If we spend $O(m|\Sigma|)$ memory for a suffix tree on a string of size *m*, then we can search for a pattern of size *n* in O(*n*) time.
If we spend only O(*m*) space, then the pattern search will take the *Min* of $n \log M$ and $n \log |\Sigma|$ time.

An alternative is to use a suffix array.

**Definition:**

Let, $T = a_1 a_2 \cdots a_m \in \Sigma^m$ be an input string.
We can define a lexical ordering among the suffixes.

The suffix array for *T* is an array of integers SA[1:*m*] where SA[*i*] is the index of the *i*th smallest suffix, for $1 \le i \le m$.

**Example:**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *T*= | g | a | c | c | g | a | t | a | g | a | c | $ |

| | |
|---|---|
| ac | 10 |
| accgatagac | 2 |
| agac | 8 |
| atagac | 6 |
| c | 11 |
| ccgatagac | 3 |
| cgatagac | 4 |
| gac | 9 |
| gaccgatagac | 1 |

| | |
|---|---|
| gatagac | 5 |
| tagac | 7 |

The right column represents the suffix array.

**FACT:**

We can construct a suffix array for any string *T* of size *m* in O(*m*) time.

**Proof:**

Construct a suffix tree *Q* in O(*m*) time.
Perform a lexical DFS in *Q*. Specifically, at any node, choose the next edge to be the lexically smallest from out of the unexplored edges. In this case we will visit the leaves (i.e., the suffixes) in the right (i.e., lexicographical) order.

**Searching for a pattern *P* in a text *T*:**

Construct a suffix array for *T*. Let $|T|=m$.
Note that, all the occurrences of *P* in SA[1:*m*] (if any) will be contiguous.

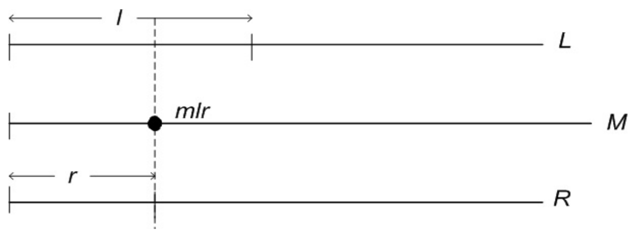
We can do a binary search in SA[1:*m*].
In each iteration of the binary search, we compare a prefix of *P* with a prefix of a suffix of *T*.
In the worst case, there can be *n* character comparisons in any iteration of the binary search.
[*n* = length of the pattern *P*].

Since there are O(log *m*) iterations in the binary search, the total number of character comparisons will be O(*n* log *m*).

**An improvement:**

Note that in any iteration of the binary search, we work with an interval of the suffix array.
Since the suffixes have been sorted we'll refer to them with integers 1 through *m*. In particular, by suffix *k,* we mean the $k^{th}$ smallest suffix of *T.*
Let, *L* be the left (or top) boundary and *R* be the right (or bottom) boundary.
To begin with, $L=1$, $R=m$. Let $M=(L+R)/2$.

Let, *l* be the length of the longest prefix of suffix *L* that matches a prefix of *P.*
Let, *r* be the longest prefix of suffix *R* that matches a prefix of *P*.



Let, *mlr= Min {l, r}*.
To compare *P* with suffix *M*, it suffices to start comparing from position *mlr*+1.

**Lemma:**

We can do string matching in O(*n* + log *m*) time.

**Proof:**

Call an examination of a character of *P* as redundant if the character has already been examined before.

When we compare *P* with suffix *M*, if we can start the comparison from position *max{l,r}*+1, then we can avoid redundant examinations.

In order to do that, we use LCP function.
LCP(*i,j*) = the length of the longest common prefix of suffix *i* and suffix *j*; ∀ *i,j*

**[to be continued…]**