CSE 4502/5717 Big Data Analytics — with Dr. Sanguthevar Rajasekaran
Lecture #14, 3/26/2018 — Notes from Katherine Riedling

Recap: Suffix trees can occupy a lot of space when the alphabet size is large. Suffix arrays require less space and less time for construction.

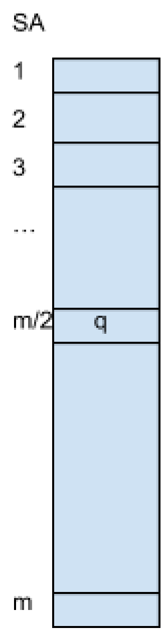
PATTERN-MATCHING USING A SUFFIX ARRAY:

INPUT: $T = t_1\, t_2 \ldots t_m$

$P = p_1\, p_2 \ldots p_n$

OUTPUT: occurrences of P in T.

IDEA: Construct a suffix array SA[1:m] for the next T.
Conduct a binary search with respect to the suffixes as shown here:

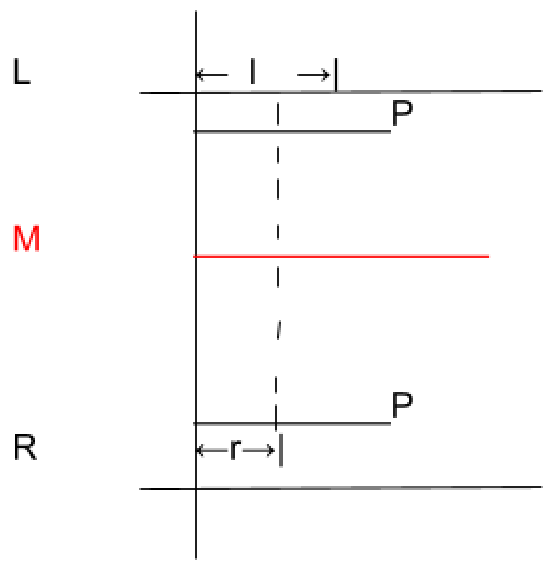


We start comparing P with the suffix starting from position SA[m/2]. If there is a match, we look for other matches for P in the neighborhood of SA[m/2]. After outputting all the matches, we stop. If there is no match of P with the suffix starting at SA[m/2], and P is larger than the suffix starting at SA[m/2] then the search continues in the interval SA[m/2+1, m]; otherwise the search continues in the interval SA[1: (m/2)-1]. In the following figure, we assume that SA[m/2]=q.

This binary search on SA[1:m] takes O(n log m) time.

Claim: We can do pattern-matching in O(n + log m) time.
Proof:

Let $M = \lfloor \frac{L+R}{2} \rfloor$. By suffix i we mean the suffix that starts at position i in T. We also refer to the suffix starting from position i in T as S_i. At any time in the binary search, we have a range [L, R] within which P is known to fall (if at all). P will be compared next with the suffix M.



We always keep the length l of the longest common prefix of L and P; we also keep the length r of the longest common prefix of P and R. Let mlr = Min{l, r} and let MLR = Max{l, r}. Note that when P is compared with the suffix M it suffices to start comparing from position mlr+1. In practice, this observation could improve the performance significantly. If we can always start comparing from position MLR+1, that will be even better!

For any two integers i and j, let LCP(i,j) be the length of the longest common prefix of S_i and S_j. Assume that this information can be obtained in constant time $\forall$ i,j. We'll later show how to construct a data structure for doing this.
$S_i \rightarrow$ suffix $t_i\ t_{i+1} \ldots t_m$.
$S_j \rightarrow$ suffix $t_j\ t_{j+1} \ldots t_m$.

Example: T = a a b a b b b a a a b a b a

1 2 3 4 5 6 7 8 9 10 11 12 13 14

LCP(2,6) = 0.
LCP(1,8) = 2.
LCP(1,9) = 5.

<u>Case 1:</u> l = r.
In this case, compare P with M starting from position MLR+1 because if l = r then the first l characters will be the same for all the suffixes L through R.

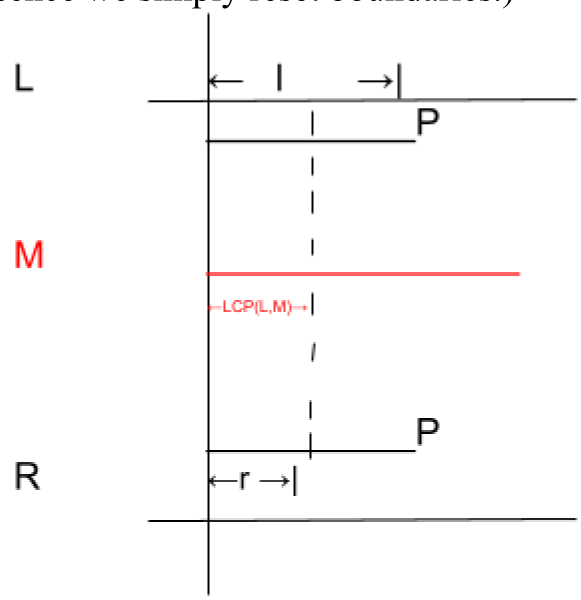
<u>Case 2:</u> l > r.

    <u>Case 2a:</u> LCP(L,M) > l:

        We set L=M;
        Move onto the next step in binary search.

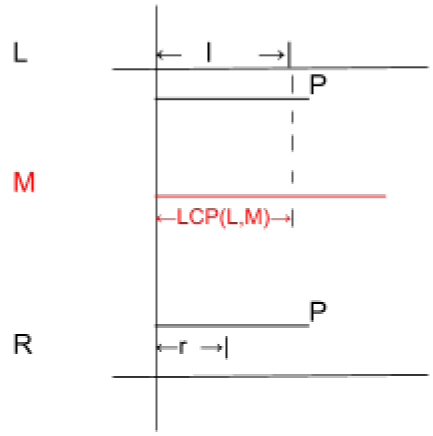
    <u>Case 2b:</u> LCP(L,M) < l:

        In this case, P is between L and M;
        Set R=M, r=LCP(L,M). Proceed to the next step in binary search.
        (In essence we simply reset boundaries.)



    <u>Case 2c:</u> LCP(L,M) = l:

        In this case, compare P and M starting from position MLR+1.



Case 3: l < r.

SIMILAR TO CASE 2.

Analysis: In the algorithm, in any step, we

1) Terminate the search;
2) We do not do any character comparisons; OR
3) We start comparisons in M from position MLR+1.

We call a character comparison in P redundant if this character has already been compared with a character of T.

In any step of the algorithm, if we compare P with M starting from position MLR+1, this comparison might have been done in a previous step. But the characters starting from position MLR+2 would not have been compared before.

This means that there is at most one redundant comparison per step.

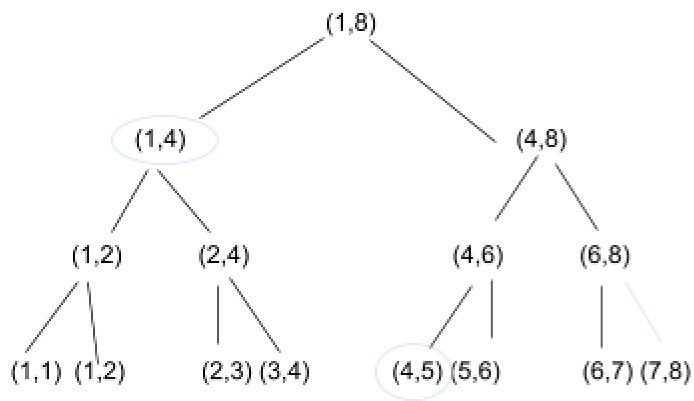
Therefore, the RUN TIME is O(n+log m).

CONSTRUCTION of the LCP Values:

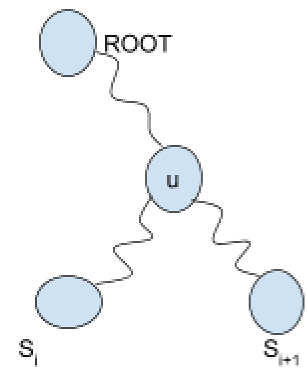
Consider a complete binary tree whose ROOT is (1,m). Any node (i,j) in the tree will have two children: $(i, \lfloor \frac{i+j}{2} \rfloor)$ and $(\lfloor \frac{i+j}{2} \rfloor, j)$.

To do binary search in SA[1:m], we only need the LCP values corresponding to every node in this tree.

An example tree for m=8 is shown below:



Computing LCP(i,i+1):



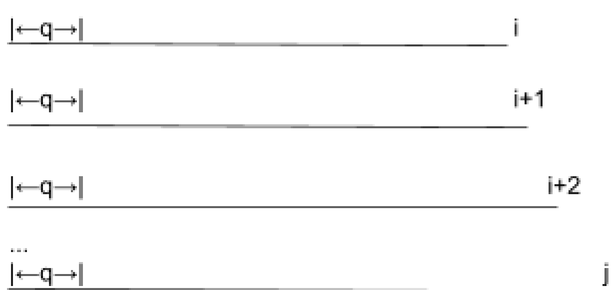
Do a lexical depth-first search in the suffix tree for T. Let u be the node closest to the root that is visited between S_i and S_{i+1}.

The string depth of u is LCP(i,i+1) for any i. This takes O(M) time.

Claim: For any j>(i+1), LCP(i,j) = $\mathrm{Min}^{j-1}_{k=i}$ (LCP(k,k+1)).

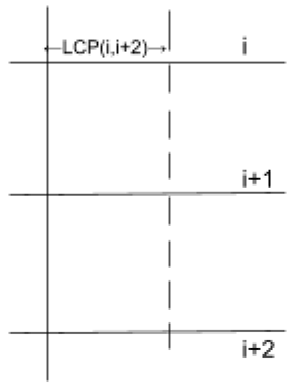
PROOF:

LCP(i,j) $\leq \mathrm{Min}^{j-1}_{k=i}$ (LCP(k,k+1))



Notice that the first "q" characters must all be the same, where q=$\mathrm{Min}^{j-1}_{k=i}$ (LCP(k,k+1)).

LCP(i,j) $\geq \mathrm{Min}^{j-1}_{k=i}$ (LCP(k,k+1)):



Proven by induction.

<u>CLAIM:</u> We can construct a suffix array in O(M) time without constructing a suffix tree.

In 2003, the following teams created algorithms that support this claim:

Kärkkäinen and Sanders
Ko and Aluru
Kim, Kim, Park, and Park

<u>The SKEW ALGORITHM of Kärkkäinen and Sanders</u>

Let $T = t_0\ t_1\ t_2 \dots t_{m-1}$

Assume that m=3q for some integer q.

The basic idea is to recursively sort (⅔)m of the suffixes, to sort the remaining (⅓)m of the suffixes using the above sorted list, and merge the two sorted suffix lists.

We will finish this algorithm in the next class.