Program Name- Longest Common Extension / LCE | Set 2 (Reduction to RMQ)

Project Category- Strings

Pre-requisites-

Longest Common Extension using Naive Method discussed in previous project

http://www.geeksforgeeks.org/suffix-array-set-2-a-nlognlogn-algorithm/

http://www.geeksforgeeks.org/%C2%AD%C2%ADkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/

Explanation-

In Set 1 we explained about the naive method to find the length of the LCE of a string on many queries.

In this set we will show how a LCE problem can be reduced to a RMQ problem, hence decreasing the asymptotic time complexity of the naive method.

Reduction of LCE to RMQ-

We assume a string **S**, and we have to find **LCE (L, R)**. We also have the suffix array – **Suff[]** and the **lcp** array.

The longest common extension between two suffixes S_L and S_R of **S** can be obtained from the **lcp** array in the following way. Let **low** be the rank of S_L among the suffixes of **S** (that is, **Suff [low] = L**). Let **high** be the rank of S_R among the suffixes of **S**. Without loss of generality, we assume that **low** < **high**. Then the longest common extension of S_L and S_R is **lcp (low, high)** = **min** $_{(low<=k<high)}$ **lcp [k]**.

Proof.

Let $S_L = s_L...s_{L+c}...s_n$ and $S_R = s_R...s_{R+c}...s_n$, and let **c** be the longest common extension of S_L and S_R (i.e. $s_L...s_{L+c-1} = s_n...s_{R+c-1}$). We assume that the string **S** has a sentinel character so that no suffix of **S** is a prefix of any other suffix of **S** but itself.

If **low** = **high − 1** then **i** = **low** and **lcp [low]** = **c** is the longest common extension of S_L and S_R and we are done.

If **low** < **high − 1** then select **i** such **lcp [i]** is the minimum value in the interval [**low**, **high**] of the **lcp** array. We then have two possible cases:

➔ If **c** < **lcp[i]** we have a contradiction because $s_L \ldots s_{L+lcp[i]-1} = s_R \ldots s_{R+lcp[i]-1}$ by the definition of the LCP table, and the fact that the entries of **lcp** correspond to sorted suffixes of **S**.

➔if **c** > **lcp[i]**, let **high** = **Suff[i]**, so that S_{high} is the suffix associated with position **i**. S_i is such that $s_{high} \ldots s_{high+lcp[i]-1} = s_L \ldots s_{L+lcp[i]-1}$ and $s_{high} \ldots s_{high+lcp[i]-1} = s_R \ldots s_{R+lcp[i]-1}$, but since $s_L \ldots s_{L+c-1} = s_R \ldots s_{R+c-1}$ we have that the **lcp** array should be wrongly sorted which is a contradiction.

Therefore we have **c** = **lcp[i]**

Thus we have reduced our longest common extension query to a range minimum-query over a range in **lcp**.

Algorithm-

To find low and high, we must have to compute the suffix array first and then from the suffix array we compute the inverse suffix array.

We also need lcp array, hence we use Kasai's Algorithm to find lcp array from the suffix array. [See- http://www.geeksforgeeks.org/%C2%AD%C2%ADkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/]

Once the above things are done, we simply find the minimum value in lcp array from index - low to high (as proved above) for each query.

The minimum value is the length of the LCE for that query.

Time Complexity-

To construct the lcp and the suffix array it takes O(N.logN) time.
To answer each query it takes O(|invSuff[R] – invSuff[L]|).

Hence the overall time complexity is O(N.logN + Q. (|invSuff[R] – invSuff[L]|))

Although we can construct the lcp array and the suffix array in O(N) time using other algorithms.

where,
Q = Number of LCE Queries.
N = Length of the input string.
invSuff[] = Inverse suffix array of the input string.

Although this may seems like an inefficient algorithm but this algorithm generally outperforms all other algorithms to answer the LCE queries.

We will give a detail description of the performance of this method in the next set.

Auxiliary Space -

We use O(N) auxiliary space to store lcp, suffix and inverse suffix arrays.

References-

http://www.sciencedirect.com/science/article/pii/S1570866710000377