Program Name- Longest Common Extension / LCE | Set 3 (Segment Tree Method)

Project Category- Strings

Pre-requisites-

Longest Common Extension using Naive Method and Longest Common Extension by conversion to RMQ as discussed in previous projects

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/

http://www.geeksforgeeks.org/segment-tree-set-1-range-minimum-query/

Explanation-

In this set we will discuss about the Segment Tree approach to solve the LCE problem.

In Set 2 we saw that an LCE problem can be converted into a RMQ problem.

To process the RMQ efficiently we will build a segment tree on the **lcp** array and then efficiently answer the LCE queries.

Algorithm-

We use the concept from Set 2.

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.

Without proving we will use the direct result (deduced after mathematical proofs)-

LCE (L, R) = RMQ $_{lcp}$ (invSuff[R], invSuff[L]-1)

The subscript – lcp means that we have to perform RMQ on the lcp array and hence we will build a segment tree on the lcp array.

Time Complexity-

To construct the lcp and the suffix array it takes O(N.logN) time.
To answer each query it takes O(log N)

Hence the overall time complexity is O(N.logN + Q.logN))

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.

Auxiliary Space -

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

Comparison of Performances-

We have seen three algorithm to compute the length of the LCE.

Set 1 → Naive Method [O(N.Q)]
Set 2 → RMQ-Direct Minimum Method [O(N.logN + Q. (|invSuff[R] – invSuff[L]|))]
Set 3 → Segment Tree Method [O(N.logN + Q.logN))]

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

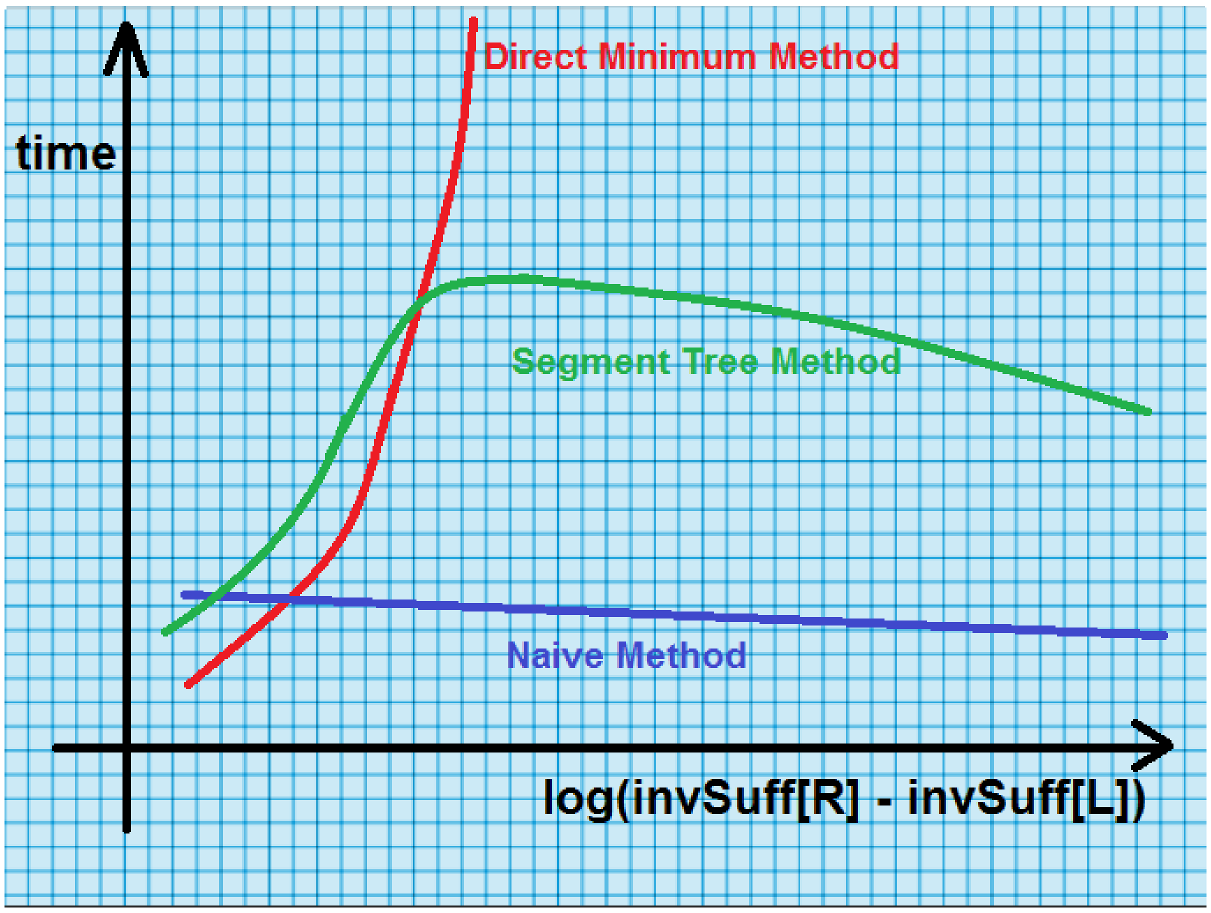
Which of them is more efficient ?

From the asymptotic time complexity it seems that the Segment Tree method is most efficient and the other two are very inefficient.

But when it comes to practical world this is not the case.

If we plot a graph between time vs log((|invSuff[R] – invSuff[L]|) for typical files having random strings for various runs, then the result is as shown below.

Note- The graph below is not meant to be scaled.



From the above graph it is clear that the Naive Method (discussed in Set 1) performs the best (better than Segment Tree Method).

This is surprising as the asymptotic time complexity of Segment Tree Method is much lesser than that of the Naive Method.

In fact, the naive method is generally 5-6 times faster than the Segment Tree Method on typical files having random strings. Also not to forget that the Naive Method is an in-place algorithm, thus making it the most desirable algorithm to compute LCE .

The bottom-line is that the naive method is the most optimal choice for answering the LCE queries when it comes to average-case performance.

Such kind of thinks rarely happens in Computer Science when a faster-looking algorithm gets beaten by a less efficient one in practical tests.

What we learn is that the although asymptotic analysis is one of the most effective way to compare two algorithms on paper, but in practical uses sometimes things may happen the other way round.

References-

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