# The Review Report of *The LCA Problem Revisited*

**Yukuan DING**
**18082849d**

## Background and Motivation

On a tree $T$ with given root, the solution of the Least Common Ancestor (LCA) Problem aims to return a shared ancestor of two nodes $u$ and $v$, which has the longest distance from the root node among all ancestors. Originally, without preprocessing, a brute-force algorithm is capable to process any single LCA question $LCA(T,u,v)$ through a Depth-first Search of tree $T$ in $O(n)$ time, where $n$ is the number of nodes in tree $T$.

When multiple LCA questions were raised on the same tree, however, conducting a Depth-first Search for every question appears to be repetitive. In order to reduce the time complexity of every single question, preprocessing was then introduced, enabling that a question $LCA_T(u,v)$ could be solved through a simple and fast query towards the data generated by preprocessing. Consequently, the optimization of the LCA problem should cover both the preprocessing time and query time.

While the LCA problem had been theoretically proved to have an algorithm with constant query time and linear preprocessing time by Harel and Tarjan (1984), the complicatedness of the algorithm became an insurmountable obstacle to efficient implementation. Even in a simplified version presented by Schieber and Vishkin (1988) of the above algorithm, the preprocessing not only requires the maintenance of extra tree structure with contextual data (i.e., parent and child nodes), but also creates intricated lists of new attributes with complicated calculation.

Considering the algorithm above, an easily implementable algorithm should be constructed on simple data structure without complex relations among the new attributes generated. Berkman *et al* noticed an existing theory where the LCA problem on a Cartesian tree could be reduced from the problem for the range maxima queries problem on an array with $O(1)$ query time (1989). In the range maxima queries problem on a length-$n$ array, the preprocessing was verified to perform in $O(lglgn)$ time with $\frac{12}{lglgn}$ processors using the PRAM CRCW model, which allows random access from multiple CPUs to a shared storage unit with concurrent operations permitted. The only restriction of this algorithm was the difficulty of implementation brought by parallel computing with concurrent operation. Although the detail of this algorithm is not available from Internet, it could be guessed that the idea of transformation between LCA problem and the range maxima queries problem inspired and motivated the development of the fast and easily implementable LCA algorithm applicable on all trees by Bender and Farach-Colton (2000).

## Problem Definitions

In this part, we briefly define the problems mentioned in the context.

**Problem 1. The Range Maxima Queries problem:**

**Data to Preprocess:** A length-$n$ array $A$ (index starts at 0) of numeric data

**Query Input:** Two integer $i, j$ where $0 \leq i \leq j \leq n-1$

**Query Output:** The index of the largest element in $A$ between $A[i]$ and $A[j]$

**Problem 2. The Least Common Ancestor (LCA) problem:**

**Data to Preprocess:** A rooted tree T having n nodes (could be any form of tree representation (e.g., adjacency list, adjacency matrix, tree map))

**Query Input:** Two nodes, $u$ and $v$, in $T$

**Query Output:** A node among the shared ancestors of $u$ and $v$ with the largest distance from the root

**Problem 3. The Range Minimum Queries (RMQ) problem:**

**Data to Preprocess:** A length-$n$ array $A$ (index starts at 0) of numeric data. Notice if the difference of all pairs of adjacent elements in $A$ is 1 or -1 ($A$ satisfies $\pm\mathbf{1}$-$\boldsymbol{property}$, this RMQ problem is specially called RMQ$\pm 1$ problem.

**Query Input:** Two integer $i, j$ where $0 \leq i \leq j \leq n-1$

**Query Output:** The index of the smallest element in $A$ between $A[i]$ and $A[j]$. Later we may use RMQ to represent also the range minimum value.

## Results

We use $\{P, Q\}$ to indicate the time complexity of a preprocessing-query based algorithm, where $P, Q$ respectively represent the complexity of preprocessing and query.

The RMQ problem on an array of length-$n$, in which the abstract value of the difference between every pair of the adjacent elements (RMQ$\pm 1$), should be 1, has time complexity of $\{O(n), O(1)\}$.

The LCA problem on a tree with m nodes could be reconstructed as a (RMQ$\pm 1$) problem on an array of length $(2m-1)$ in $O(m)$ time, presenting an $\{O(m), O(1)\}$ solution for the LCA problem.

The RMQ problem on an array of length-$n$ without the restriction on the difference of adjacent elements could be reduced to a LCA problem on a Cartesian tree with $n$ nodes. As LCA is proved to have an $\{O(n), O(1)\}$ solution, there is a $\{O(n), O(1)\}$ solution for RMQ.

## Techniques Overview

### Problem Reduction

Initially, the reduction from LCA problem to RMQ$\pm 1$ problem is essential to simplify the data structure of the algorithm. As the result of a RMQ query would be indexed between two query inputs, the LCA problem is expected to preprocess

the tree into an array where the result node or its representative is indexed between the two nodes or their representatives. Among the existing traversing methods for a tree, the Euler tour, a DFS that also records the path back to the former split point when it reaches a leaf node, appears to be a suitable choice. Applying RMQ on an array representing the Euler tour is however a vain attempt, stemming from the fact that the array stores only the representatives of nodes (e.g., node ID) that are irrelevant to the contextual information of the tree structure. Thus, in the same order of the Euler tour, another array storing the level, defined as the distance from the root, of each node is created simultaneously while conducting DFS for Euler tour.

It could be clearly observed that the least common ancestor of two nodes is the node with lowest level between the query nodes in the order of Euler tour. As the difference on level between adjacent nodes are 1 or -1, the LCA problem on a tree is then converted to an RMQ$\pm 1$ problem on the level array in Euler tour order.

## Optimization of RMQ Problem

The article progressively discusses the preprocessing methodology of an RMQ problem. In general, the core operation of preprocessing could be concluded as computing a partition of all possible queries in advance, with an optimization strategy aiming to reduce the proportion of that partition through allocating calculation burden to single query on the premise of preserving the asymptotical time complexity of a query. If the query is expected to only have array access operations without extra calculations, the preprocessing could hardly perform in asymptotically less than $O(n^2)$ time. The $O(n^2)$ preprocessing, which requires $n(n-1)$ constant-time comparisons, is an application of Dynamic Programming, where (Assume $i \neq j$):

$$RMQ_A(i,j) = \begin{cases} \min(A[i], A[j]) & , \quad abs(i-j) = 1 \\ \min\big(RMQ(i, j-1), RMQ(i+1, j)\big), & \quad else \end{cases}$$

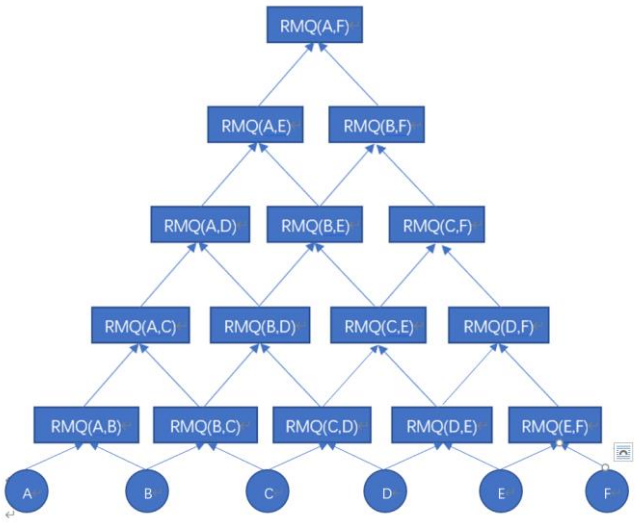


Figure 1. Naïve query table using dynamic programming

For example, for an array $\{A, B, C, D, E, F\}$ of length 6, an RMQ query table could be constructed as in $Figure\ 1$ with ${}_6C_2 = 15$ comparisons. Nevertheless, calculating the results for all the RMQ queries in advance would waste time if a large proportion of the possible queries have not been called. An idea is to increase the sparsity of the query table, transferring part of the comparisons to each query. Continuously deleting the table entries from top in the above figure until asymptotic time decrease was achieved would unreasonably overload the queries involved in those comparisons, as these queries would have to process $O(n)$ lost comparisons themselves (e.g., $RMQ(A, F)$).

An acceptable strategy would be deleting the entries distributed among the layers in the above figure, promising every query to fetch the result directly or after constant comparisons among existing entries. The strategy could be implemented by a query table only maintaining the results of queries on all intervals with length of a power of 2 implements. For the queries on power-of-2 intervals, the result certainly could be retrieved with array access. Because there definitely exist two intersected power-of-2 intervals covering the queried one, the queries on other intervals could be answered with single comparison between the directly retrieved results of these 2 intervals. The logarithm-based preprocessing could generate a query table of length $O(nlgn)$ in $O(nlgn)$ time, promising $O(1)$ query time. The preprocessing performs also with Dynamic Programming like the above figure, where (Assume $j > 0$ and $i + 2^j - 1$ does not exceed the maximum index of the array):

$$RMQ_A(i, i + 2^j - 1) = \begin{cases} \min(A[i], A[i+1]) & , \quad j = 1 \\ \min\left(RMQ(i, i + 2^{j-1} - 1), RMQ(i + 2^{j-1}, i + 2^j - 1)\right), & \quad j > 1 \end{cases}$$

## Taking the Advantage of RMQ$\pm 1$ Problem

While the optimization on RMQ problem terminate on $\{O(nlgn), O(1)\}$ time complexity, the uniqueness of the RMQ $\pm 1$ problem proceeds further optimization. It is observed that to construct a length-$k$ numerical array satisfying $\pm 1\text{-}property$, a length-$(k-1)$ list of operations chosen from $\{+1, -1\}$ could be conducted sequentially on an arbitrary number (e.g., applying $\{+1, -1, +1, +1\}$ to $2$ results in an array $\{2,3,2,3,4\}$). Based on this binary operation sequence, a length-$k$ numerical array satisfying $\pm 1\text{-}property$ could be classified into $2^{k-1}$ categories. There is now an unneglectable observation that when 2 length-$k$ numerical arrays satisfying $\pm 1\text{-}property$ belong to the same category, they would result the same from an RMQ query. Thus, when dealing with RMQ$\pm 1$ problem on multiply length-$k$ arrays in the same category, clearly, they could share the same preprocessing, saving time from repetitive operations.

According to the above observation, a single numerical length-$n$ array satisfying $\pm 1\text{-}property$ could be divided into blocks of equal length, in order to formulate multiple arrays that could be categorized. Instead of preprocessing a

logarithm query table for all these blocks, the preprocessing is conducted on each category of the blocks. Hence it would be possible for $m$ blocks to have equal classification result, avoiding m-1 redundant preprocessing. For $RMQ(i,j)$ where the elements indexed as $i$ and $j$ are in the same block, the result could be fetched through checking the query table of the block's category in $O(1)$ time.

The preprocessing would also store the RMQ of each block in an array $A'$, as well as construct an RMQ query table for this array. The block size is chosen to be $\frac{lgn}{2}$ in the article without further explanation. An array $B$ would be used for storing the indexes in $A$ of the elements in $A'$. When dealing with $RMQ(i,j)$ where $i$ and $j$ are in different blocks, block partitioning shows its contribution to reducing time complexity as a method corresponding to Divide and Conquer principle. Assuming $i < j$, $RMQ_A(i,j)$ would result in the index of the minimum among:

$$A[RMQ_A(i, end\ of\ i's\ block)],$$
$$A\left[B\left[RMQ_{A'}[i's\ next\ block, j's\ former\ block]\right]\right],$$
$$A[RMQ_A(beginning\ of\ j's\ block, j)],$$

where essentially $RMQ_{A'}$ is divided into RMQ problems on several blocks.

**The Wrapping-up of LCA Problem**

In the preprocessing the tree $T$ given in a LCA problem, two arrays $A\ and\ E$ respectively recording the nodes and their levels in the order of Euler tour is constructed. By partitioning the array into blocks, and calculating the RMQ table for each kind of block as well as an array of RMQs in each block, we finished the preprocessing for an RMQ problem. Let $f(i)$ be the index of the first appearance of the node $i$ in array $E$. Query $LCA_T(i,j)$ could be answered by $A\left[RMQ_E\left(f(i), f(j)\right)\right]$ with $\{O(n), O(1)\}$ time complexity.

**Improvement on General RMQ Problem**

If a Cartesian tree $\boldsymbol{C}$ is constructed from an array $\boldsymbol{A}$, then $\boldsymbol{A}[RMQ_A(i,j)] = LCA_C(i,j)$. In Cartesian tree $\boldsymbol{C}$, the left and right subtree of a node $v$ represent the subarrays on the left and right side of the element referred by $v$ respectively. The root represents the minimum element in $\boldsymbol{A}$ and each node also represents the minimum element in the subarray of $\boldsymbol{A}$ referred to by its subtree. Thus, the RMQ of element $i$ and $j$ in $\boldsymbol{A}$ would also be element corresponding to the LCA of the nodes representing $i$ and $j$. Through only maintaining the rightmost path in $C$ with constant edge adding or deleting operation for each node, the tree $C$ could be constructed in $O(n)$ time where n is the length of array $\boldsymbol{A}$. As the LCA problem had already been solve in $\{O(n), O(1)\}$ time, the RMQ now has a $\{O(n), O(1)\}$ solution.

## Example Problem and Solution

Consider a tree in the figure below, where the blue point is the root and the numbers are the nodes' IDs.

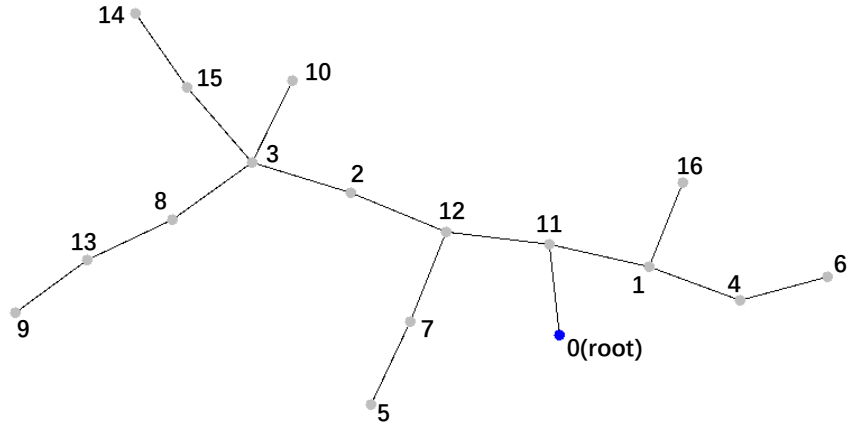


Figure 2. A rooted tree with 17 nodes

### Data Process

*An array of the Euler tour $A$:*

$\{0,11,1,4,6,4,1,16,1,11,12,2,3,10,3,15,14,15,3,8,13,9,13,8,3,2,12,7,5,7,12,11,0\}$

*An array of the levels of nodes in the order of Euler tour $A'$:*

$\{0,1,2,3,4,3,2,3,2,1,2,3,4,5,4,5,6,5,4,5,6,7,6,5,4,3,2,3,4,3,2,1,0\}$

*Partitioned blocks in RMQ preprocessing*

{0,1} {2,3} {4,3} {2,3} {2,1} {2,3} {4,5} {4,5} {6,5} {4,5} {6,7} {6,5} {4,3} {2,3} {4,3} {2,1} {0}

Here we have only two classes: {+1} and {-1}. Also, here is an unclassifiable block {0}. We could either perform {+1} operation on it to make it a normalized block, or create an independent query table for it. For the convenience of illustration, we convert it to a normalized block.

### Query Tables

Figure 5. Query table for the RMQs of all blocks

| Start Index \ Interval Length | 2 | 4 | 8 | 16 |
|---|---|---|---|---|
| 0 | **0** | **0** | **0** | **0** |
| 1 | **1** | **4** | **4** | **** |
| 2 | **3** | **4** | **4** | **** |
| 3 | **4** | **4** | **4** | **** |
| 4 | **4** | **4** | **4** | **** |
| 5 | **5** | **5** | **5** | **** |
| 6 | **6** | **6** | **13** | **** |
| 7 | **7** | **7** | **13** | **** |
| 8 | **9** | **9** | **13** | **** |
| 9 | **9** | **12** | **13** | **** |
| 10 | **11** | **13** | **** | **** |
| 11 | **12** | **13** | **** | **** |
| 12 | **13** | **15** | **** | **** |
| 13 | **14** | **15** | **** | **** |
| 14 | **15** | **** | **** | **** |
| 15 | **16** | **** | **** | **** |

Figure 3. Query table for Class 1 {+1}

| Start Index \ Interval Length | 2 |
|---|---|
| 0 | **0** |

Figure 4. Query table for Class 2 {-1}

| Start Index \ Interval Length | 2 |
|---|---|
| 0 | **1** |

**Sample Query: $LCA_T(10,7)$**

1. Index of the first appearance of node 10 and 7 in array $A$: 13 and 27.
2. Process $RMQ_{A'}(13,27)$.
3. Index 13 is the 2nd in $block_{index=6,Class=\{+1\}}$.
   Index 27 is the 2nd in $block_{index=13,Class=\{+1\}}$
4. According to the query table of Class 1 $\{+1\}$, $block_{index=6,Class=\{+1\}}$'s RMQ is $A'[12]$, $block_{index=13,Class=\{+1\}}$'s RMQ is $A'[26]$. According to the query table for the RMQs of all blocks, the RMQ in between $block_{index=7}$ and $block_{index=10}$ is $A'[14]$, the RMQ in between $block_{index=11}$ and $block_{index=12}$ is $A'[23]$. $\min(A'[12], A'[26], A'[14], A'[23]) = A'[26]$, thus $RMQ_{A'}(13,27)$ returns 26.
5. $A[26]$ is node 12, which is the LCA of node 10 and 7.

## Implementation and Experiments

We implemented the given LCA algorithm and a DFS based O(n) brute-force algorithm without preprocessing.

In an example test, a graph with 526 nodes is randomly generated. For each query, the brute-force algorithm could be faster than the RMQ based algorithm only when the queried nodes are close to the root (i.e., approximately having distance of 3 nodes) in the order of DFS, which terminates the DFS in an early state. As the levels of queried nodes raises, the speed of a single RMQ-based query is observed to be approximately 5 times faster than the naïve brute-force query.

In a formal test, we randomly generate 25 graphs, each of which has 1260 nodes. 200 random LCA queries are raised on each of these graphs. For the RMQ-based algorithm, the preprocessing time is included when calculating average query time. On average, the RMQ-based algorithm would have lower average query time than the brute-force algorithm after approximately 30 queries.

The RMQ-based algorithm has an average running time of about 0.008ms, which is around 80% faster than the brute-force algorithm's average, 0.04ms.

The general RMQ algorithm which could be reduced to LCA problem on a Cartesian tree was not implemented, for that it is not the key point in the article. However, as the $\{O(n), O(1)\}$ algorithm based on reduction is supposed to have a relatively large coefficient on the $O(n)$ preprocessing, it could be hypothesized that the sparse-table preprocessing with $O(nlgn)$ time performs faster with a low magnitude of $n$. Simultaneously, the query on a table generated by sparse-table preprocessing clearly requires less array access and element comparisons in most cases, appearing to be constantly faster. Thus, the magnitude of n and the queries would have been raised could possibly determine the choice of RMQ algorithms.

## Reviews and Opinions

The algorithm is complete, fast and implementable, though, there are some

possible amendments.

In the construction of the logarithm-based RMQ query table, if a simple 2-Dimensional array is used as container, there would be waste in storage resources. In each column with interval length $j$ (e.g., Figure 5. Query table for the RMQs of all blocks), there would be $j-2$ wasted space. As n becomes larger, the query table for each block would have more columns, triggering larger waste space. When there is a graph of $2^{15}$ nodes (e.g., a map), it is easy to calculate the space waste in all $2^{13}$ length-8 blocks, which is approximately 256KB. In personal opinion, 256KB of memory might not be a neglectable amount in compressed computers on aircrafts or missiles. If saving this memory is important, a hash table implementation with $j$ as key and an array of constant length as value could be used.

While an array is partitioned into blocks, the article does not mention that how to process a possible leftover part (i.e., the last block with shorter length than other blocks). In the implementation, an extra query table is created for this part in $O(lgnlg(lgn))$ time. However, it could be normalized through continuously conducting +1 operations until it has the same length of a normalized block (e.g., $\{3,1,2\}$ could be normalized as $\{3,1,2,3,4,5,6\ldots\}$), in order to reduce redundant processing.

In addition, it is possible that not all $O(\sqrt{n})$ normalized blocks have their instances among all the blocks, which may be correlated to the structure of the original tree. The algorithm could calculate only the normalized blocks with existing instances.

## Bibliography

Bender, M. A., & Farach-Colton, M. (2000, April). The LCA problem revisited. In Latin American Symposium on Theoretical Informatics (pp. 88-94). Springer, Berlin, Heidelberg.

Harel, D., & Tarjan, R. E. (1984). Fast algorithms for finding nearest common ancestors. siam Journal on Computing, 13(2), 338-355.

Schieber, B., & Vishkin, U. (1988). On finding lowest common ancestors: Simplification and parallelization. SIAM Journal on Computing, 17(6), 1253-1262.