FARACH-COLTON AND BENDER ALGORITHM

CS302- ANALYSIS AND DESIGN OF ALGORITHMS

INDIAN INSTITUTE OF TECHNOLOGY ROPAR

SNEHA SAHU	2022CSB1129
HARTIK ARORA	2022CSB1314
PRATHISTHA PANDEY	2022CSB1105
SHRAVAN JINDAL	2022CSB1124
SIDDHARTH	2022CSB1125

INDEX

PROBLEM STATEMENT

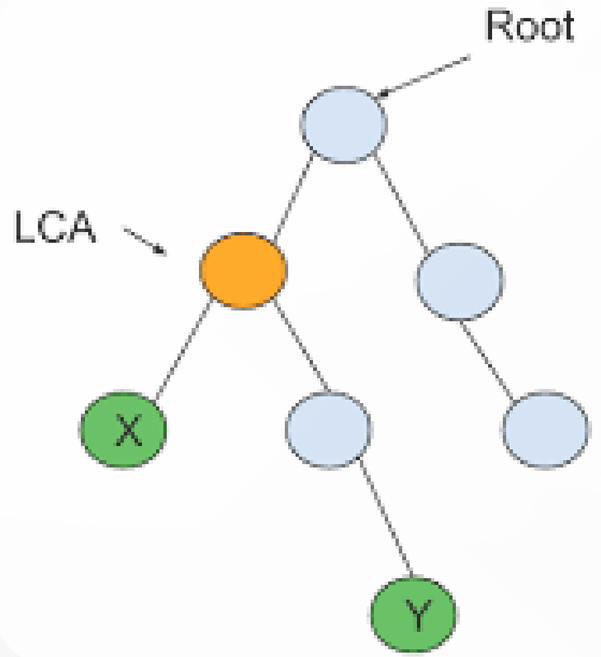
Given a rooted tree T with N nodes and two nodes u and v, find their lowest common ancestor (LCA).

Input:

- A tree T with N nodes
- Two nodes u and v (1 ≤ u, v ≤ N)
- Each node has a unique identifier
- Node 1 is considered the root

Output:

- The lowest common ancestor of u and v



APPROACH 1: DFS WITH PATH RECORDING

1. Preprocessing

Perform a DFS traversal to determine:

- Parent relationship of each node.
- Depth of each node relative to the root

2. Lifting Nodes to the Same Depth

If the two nodes are at different depths, move the deeper node upward in the tree until both nodes are at the same depth.

3. Simultaneously Move Up Until Convergence

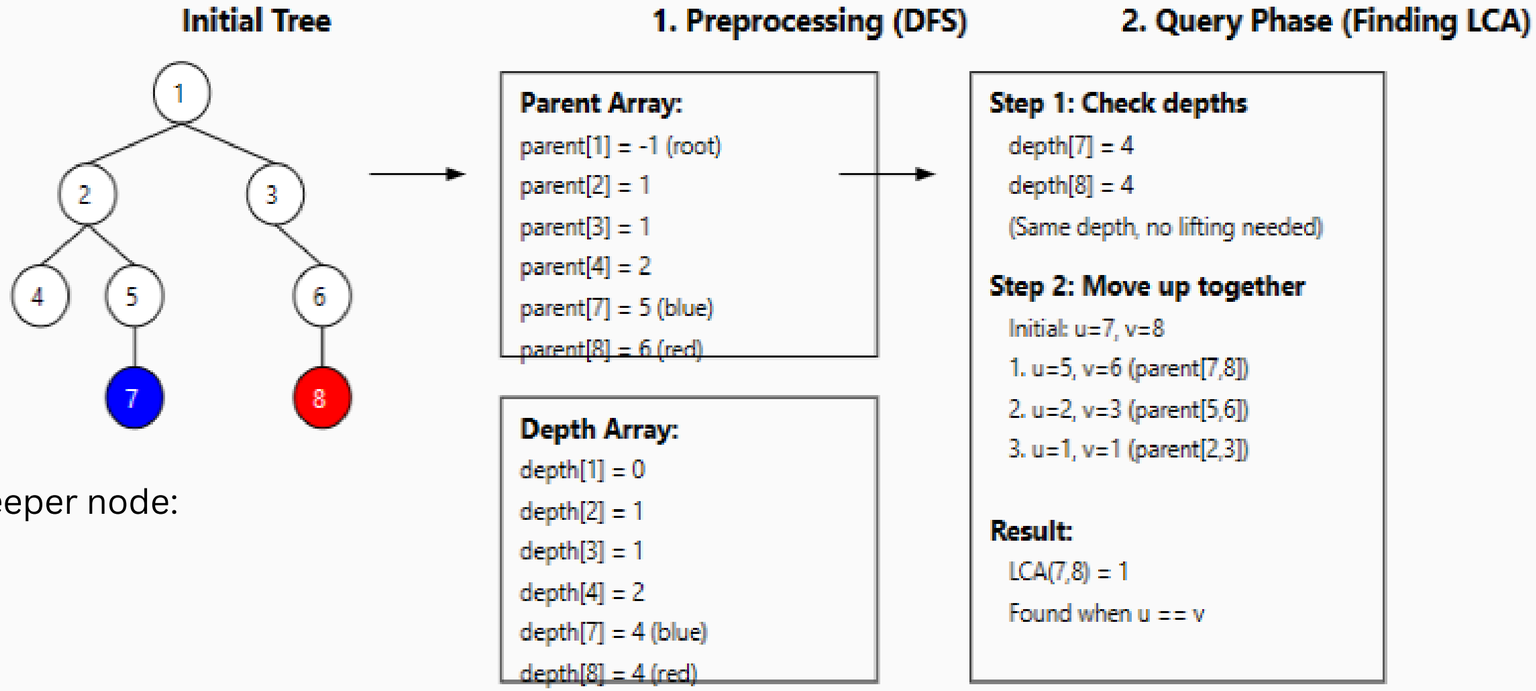
Move both nodes upward in the tree simultaneously (following parent pointers) until the two nodes meet. The node where they meet is the LCA.

APPROACH 1: DFS WITH PATH RECORDING

1. Preprocessing

1. Initialize:
   - parent[node] to store the parent of each node.
   - depth[node] to store the depth of each node.
   -
2. Perform a DFS from the root:
   - For each child of the current node, record:
   - parent[child] = current node
   - depth[child] = depth[current node] + 1
   - Recursively process all children.

Time Complexity - O(n)



2. Query (LCA)

- If nodes u and v are at different depths, "lift" the deeper node:
- While depth[u] > depth[v], set u = parent[u].
- Similarly, if depth[v] > depth[u], set v = parent[v].
- Once u and v are at the same depth:
- Move both nodes upward until u == v:
  - While u != v, set u = parent[u] and v = parent[v].
- The node where u == v is the LCA.

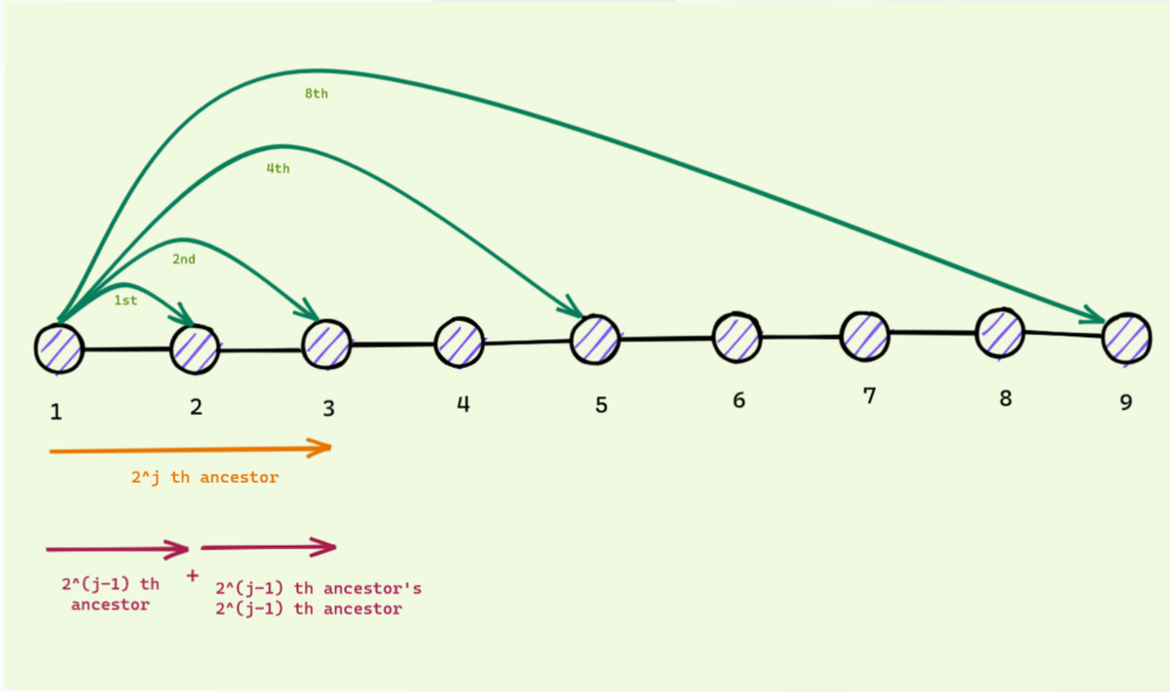
Time Complexity - O(n)

APPROACH 2: USING BINARY LIFTING

- Binary Lifting is a Dynamic Programming approach for trees where we precompute some ancestors of every node.
- This technique is based on the fact that every integer can be represented in binary form.
- Sparse table dp[i][v] stores the 2^ith parent of vertex v, where 0 <= i <= logN.
- When we want to get the kth parent of v, we can first break k into the binary representation

  k = 2^p1 + 2^p2 + 2^p3 + ... + 2^pj

  where p1, ..., pj are the indices where k has bit value 1
- Eg. 6 = 2^1 + 2^2, the answer can be P[1][P[2][v]] or P[2][P[1][v] Time complexity of getting the Kth parent from O(K) to O(logK)



APPROACH 2: USING BINARY LIFTING

1. Preprocessing

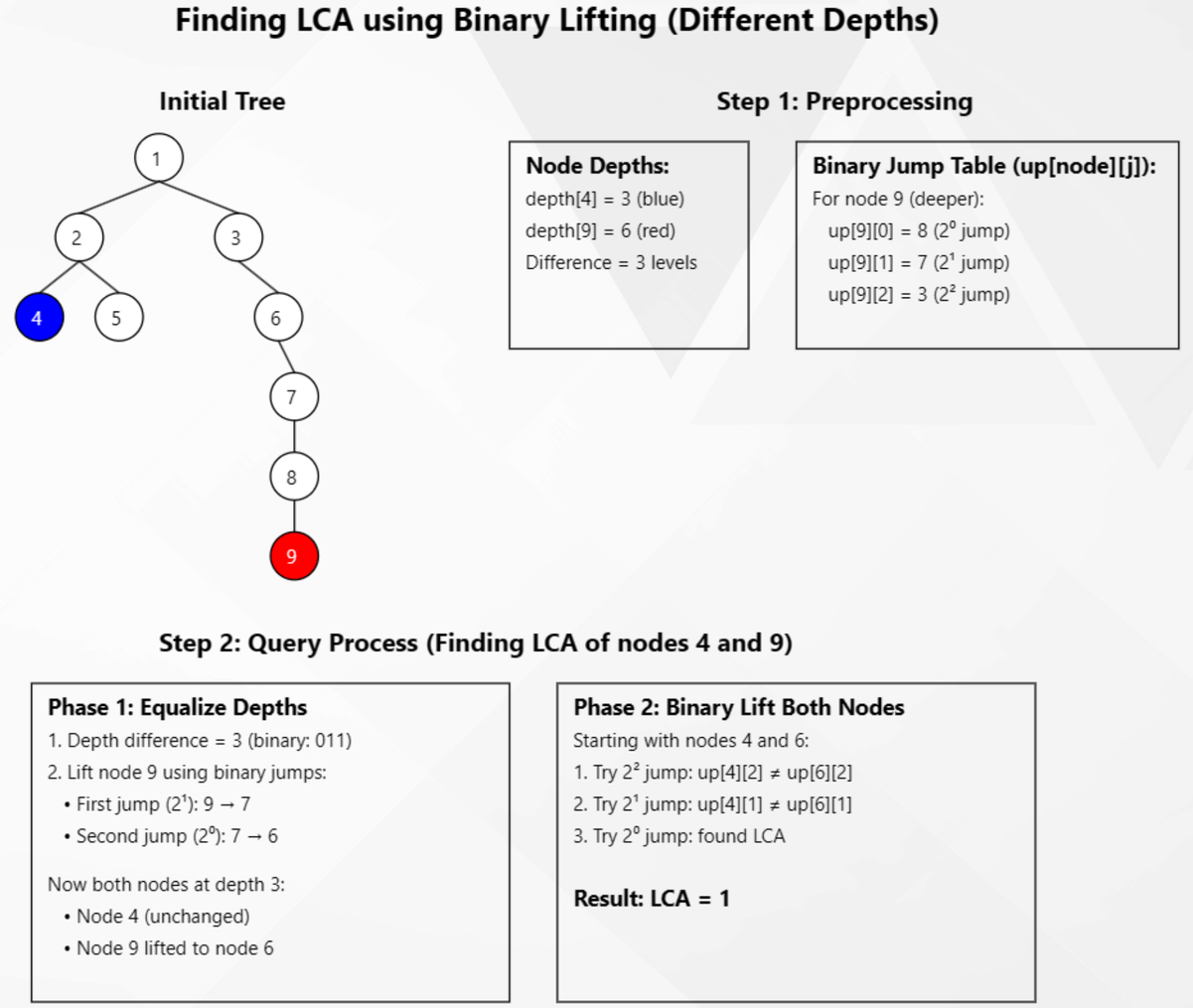
- Store Depths:
- Calculate the depth of each node from the root using a DFS.
- Create a Jump Table:
- Use a 2D array up[node][j], where up[node][j] represents the 2^j-th ancestor of the node.
- Base case: up[node][0] = parent[node].
- Transition: up[node][j] = up[up[node][j-1]][j-1], meaning the 2^j-th ancestor is computed using two 2^(j-1)-th ancestors.

Time Complexity - O(nlogn)

2. Query (LCA)

- Equalize Depths:
- If the two nodes u and v are at different depths, "lift" the deeper node to the same depth as the shallower one by jumping upwards in powers of 2.
- Find the LCA:
- Once u and v are at the same depth, lift both nodes simultaneously until they meet. The meeting node is the LCA.

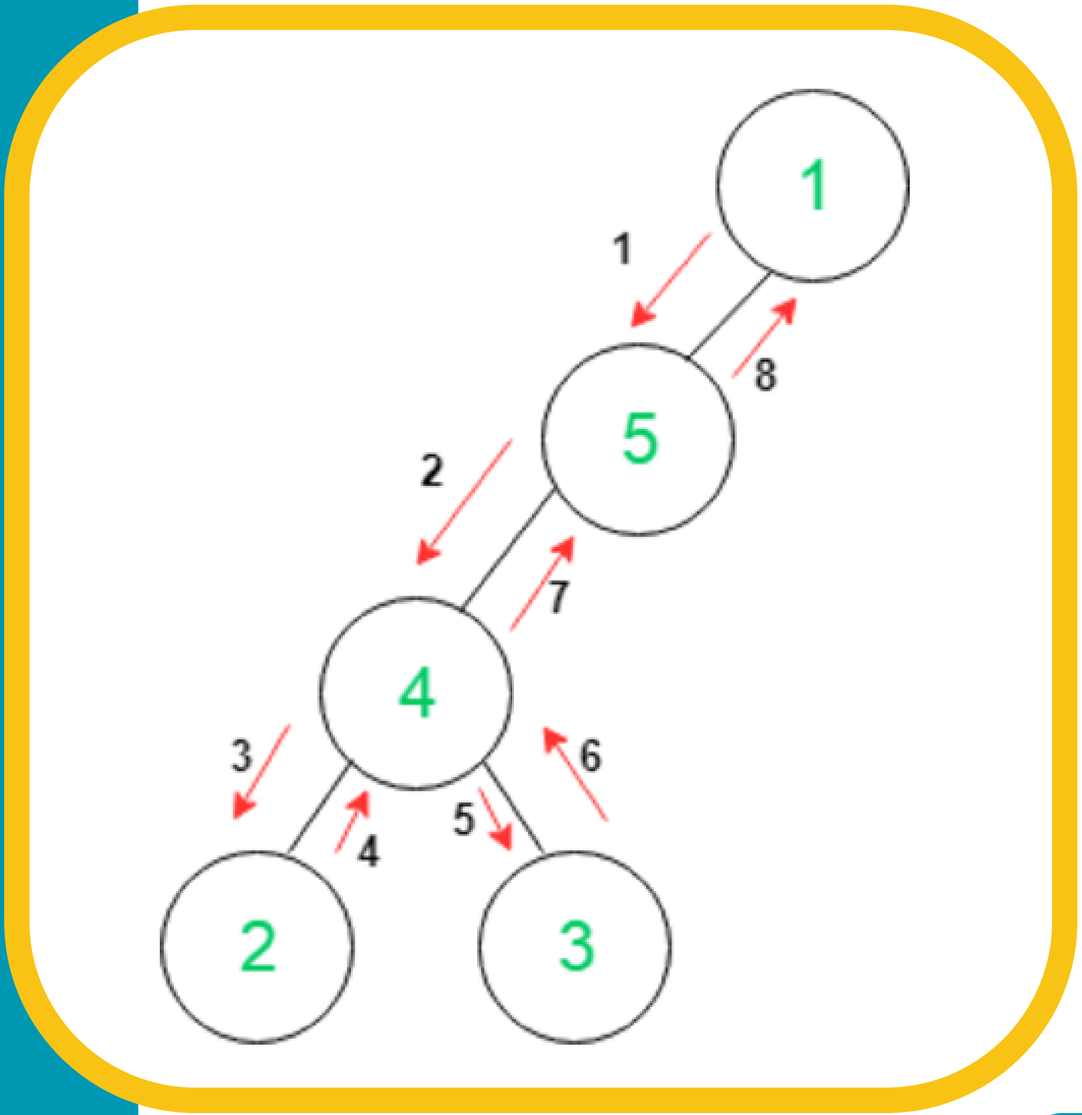
Time Complexity - O(logn)



Euler Tour Of Tree

An Euler tour is defined as a way of traversing a tree such that each vertex is added to the tour when we visit it (either moving down from the parent vertex or returning from a child vertex).

We start from the root and return to the root after visiting all vertices. It requires exactly 2N−1 vertices to store the Euler tour.

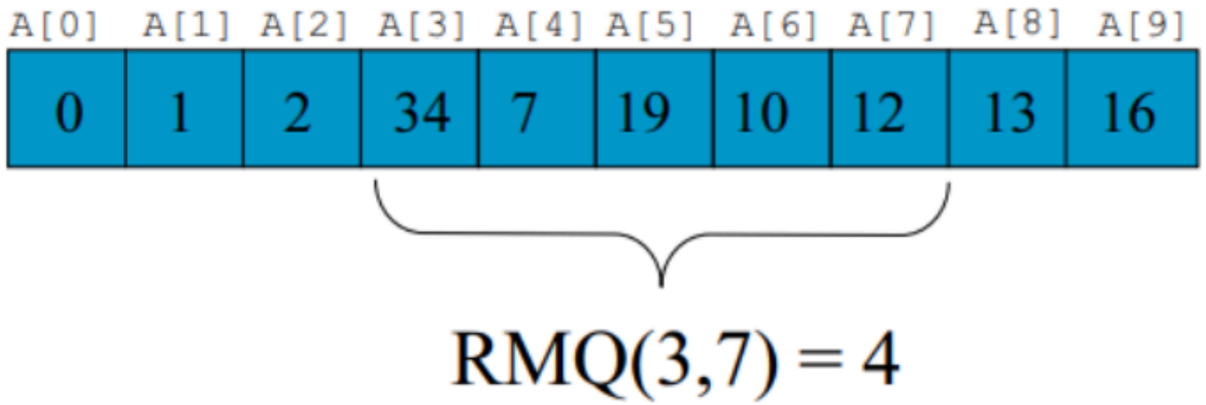


What is RMQ ?

Range Minimum Query (RMQ)

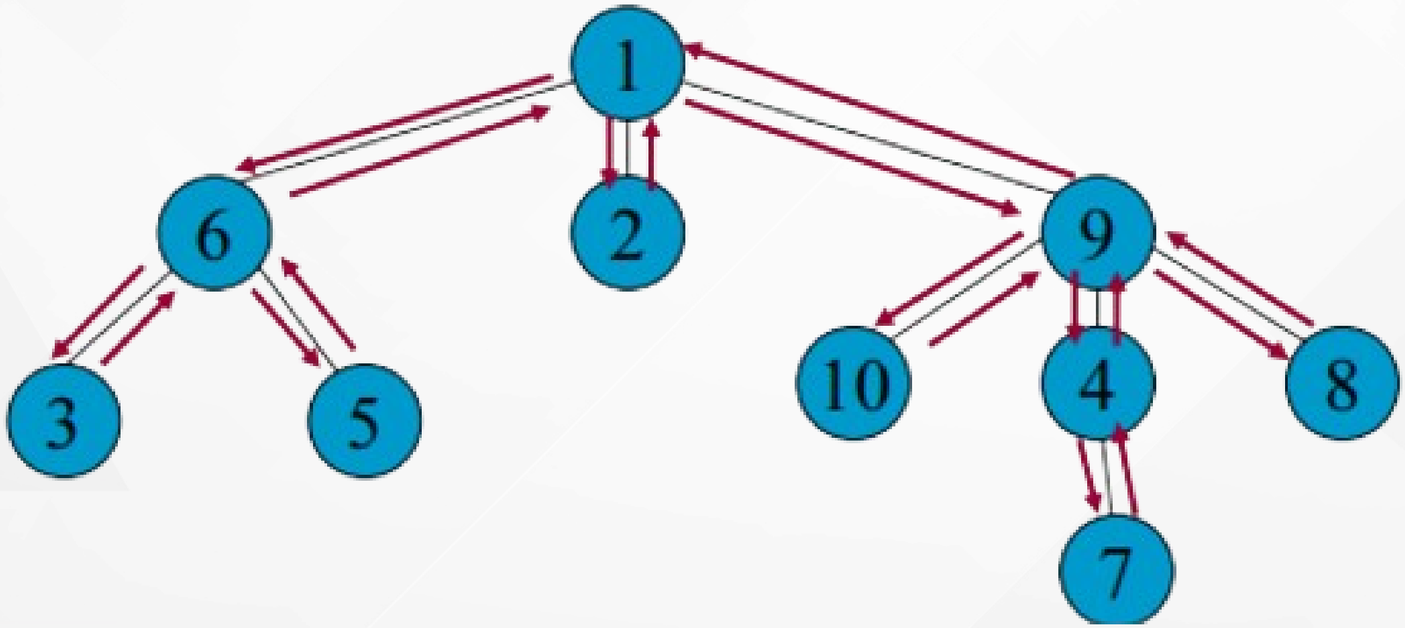
Given an array A of length n

RMQ(i,j) – returns the index of the smallest element in the subarray A[i..j].



OBSERVATION:

The Lowest Common Ancestor (LCA) of two nodes u and v is the shallowest (lowest depth) node in the tree that lies on the path between the first visit to u and v during a Depth-First Search (DFS) traversal of the tree T.



Reduction From LCA->RMQ

Lemma : **If a solution for *Range Minimum Query (RMQ)* exists with a time complexity of $\langle f(n), g(n) \rangle$, then a time complexity solution for *Lowest Common Ancestor (LCA)* can be achieved with $\langle f(2n-1) + O(n), g(2n-1) + O(1) \rangle$.**

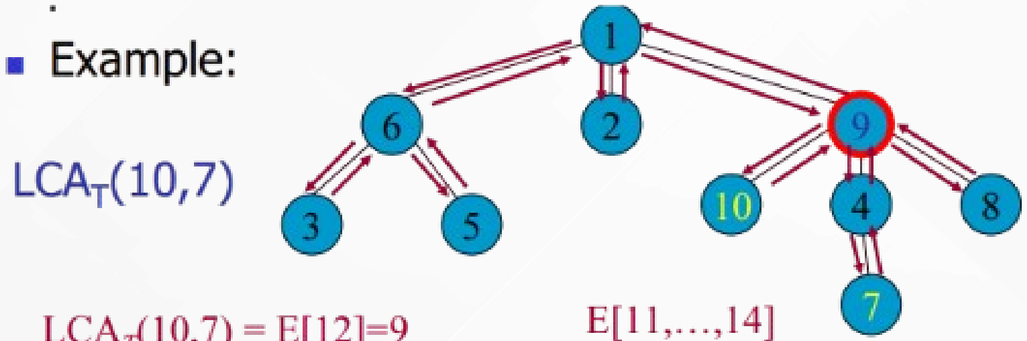
PreComputation :

1. Array $E[1, \ldots, 2n-1]$ stores the nodes traversed in an Euler Tour of the tree T. $E[i]$ is the label of the i^{th} node traversed in the Euler tour of T.
2. Let the level of a node be its distance from the root. The Level Array $L[1, \ldots, 2n-1]$ is computed, where $L[i]$ is the level of node $E[i]$ in the Euler Tour.
3. The representative of a node in an Euler Tour is the index of the first occurrence of the node in the tour, i.e., the representative of i is $\min\{j : E[j] = i\}$. The Representative Array $R[1, \ldots, n]$, where $R[i]$ is the representative of node i.

To Calculate LCA(X, Y)

1. Look at the segment of the Euler Tour between the first visits to u and v: $E[R[u], \ldots, R[v]]$ (or $E[R[v], \ldots, R[u]]$, depending on the order).
2. The **shallowest node** in this range is the one with the **minimum level**, found using a **Range Minimum Query (RMQ)** on the Level Array L for $[R[u], R[v]]$.
3. The RMQ result gives the index of the node with the minimum level, located at $E[\mathrm{RMQ}(R[u], R[v])]$, which is the **LCA** of u and v.

- Example:



$LCA_T(10,7)$

$LCA_T(10,7) = E[12]=9$

E[11,…,14]

E: 1 6 3 6 5 6 1 2 1 9 10 9 4 7 4 9 8 9 1

L: 0 1 2 1 2 1 0 1 0 1 2 1 2 3 2 1 2 1 0

R: 1 8 3 13 5 2 14 17 10 11

$RMQ_L(10,7) = 12$

R[7] R[10]

Pre Processing Complexity:

- Arrays L, R, and E are constructed in $O(n)$ time during a single DFS traversal.
- Preprocessing the Level Array L for Range Minimum Query (RMQ) takes $f(2n-1)$ time.

Query Complexity:

- An RMQ query on L has a complexity of $g(2n-1)$.
- Array references are $O(1)$ operations, with only three array lookups performed in total.

Overall Complexity: $\langle f(2n-1)+O(n), g(2n-1)+O(1)\rangle$

Note: The key is to optimize Range Minimum Query (RMQ) for better preprocessing and query performance.

Naive Algorithms to calculate RMQ

< O(N²), O(1)> SOLUTION

We precompute a 2D array lookup[i][j], where each entry stores the minimum value in the range [i,j].

Filling the Table:

- For every possible range [i,j] (0≤i≤j<N), compute the minimum value in that range and store it in lookup[i][j].
- This involves iterating over:
  - All starting indices i (total of N).
  - For each i, iterating over all possible end indices j (i≤j<N).
- Calculating the minimum for a range [i,j] requires traversing and comparing elements from array[i] to array[j], which could take up to j−i+1 steps.

Input Array:

2	5	1	4
[0]	[1]	[2]	[3]

Lookup Table (lookup[i][j]):

	j=0	j=1	j=2	j=3
i=0	2	2	1	1
i=1	5	5	1	1
i=2	1	1	1	1
i=3	4	4	4	4

Example Query: RMQ(0,2)

Find minimum in range [0..2]

Answer = lookup[0][2] = 1

Complexity:

Preprocessing: O(n²)

Query: O(1)

This approach achieves O(N²) for precomputing all possible ranges but allows constant time retrieval for queries.

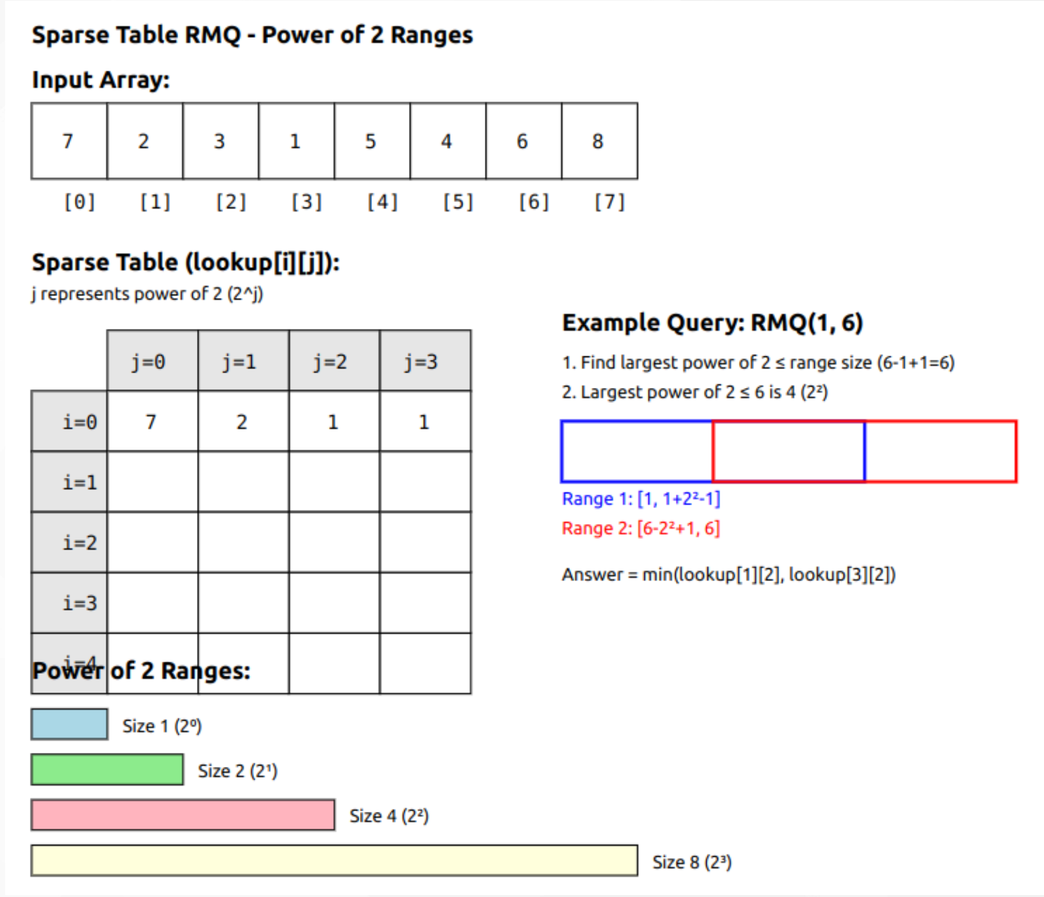
Solutions to implement - RMQ

< O(N log N), O(1)> SOLUTION

The idea is to precompute the minimum of all subarrays of size 2^j, where j varies from 0 to logn. We create a table lookup[i][j] such that lookup[i][j] stores the minimum of the subarray starting at index i and having a length of 2^j.

For any arbitrary range [L, R], we need to utilize subarrays whose sizes are powers of 2. The approach is to use the closest power of 2 that covers the range. Specifically, we perform at most one comparison to find the minimum between two subarrays of sizes that are powers of 2.

One subarray starts at index L and ends at L+2^k−1, where 2^k is the largest power of 2 less than or equal to the length of the range. The other subarray starts at R - 2^k + 1 and ends at R. By comparing these two subarrays, we obtain the minimum for the entire range [L,R].



Running time analysis of Sparse Tree < O(N log N), O(1)> SOLUTION

Range Splitting: The range [j, j + 2^i - 1] of length 2^i can be split into two smaller ranges [j, j + 2^{i-1} - 1] and [j + 2^{i-1}, j + 2^i - 1], each of length 2^{i-1}. This is the basis for precomputing the minimum of all subarrays efficiently.

Dynamic Programming for Table Generation: A table st[i][j] is created, where each entry stores the minimum of a subarray starting at index j with size 2^i. The pseudo code provided calculates these minimums using dynamic programming:

```
for (int i = 1; i <= K; i++)
    for (int j = 0; j + (1 << i) <= N; j++)
        st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i - 1))]);
```

The time complexity of precomputing the sparse table is O(NlogN), where N is the size of the array.

The minimum in the range [L,R] can be caulated in the following way :

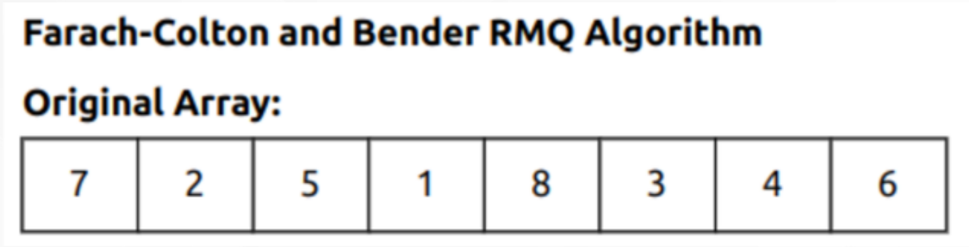
```
int i = lg[R - L + 1];
int minimum = min(st[i][L], st[i][R - (1 << i) + 1]);
```

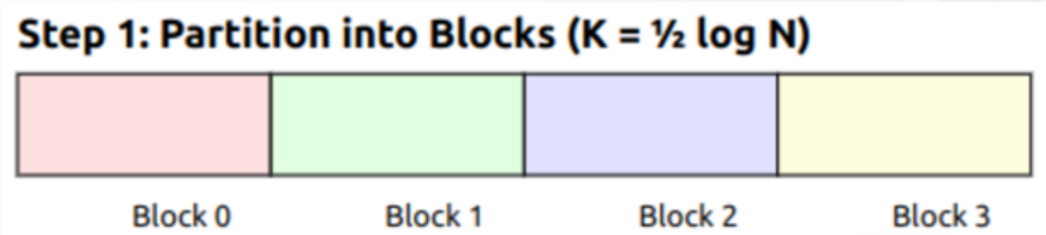
Farach-Colton and Bender Algorithm <O(N), O(1)> SOLUTION

Steps:

1 **Partition the Array:**

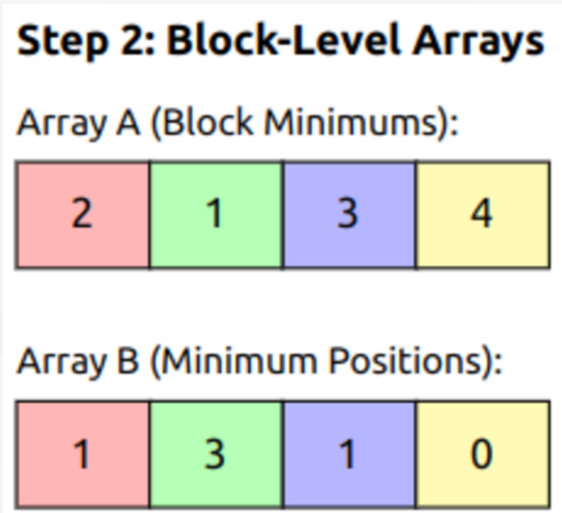
- Divide the input array into blocks of size $K = \frac{1}{2} \log N$.





2 **Define the Block-Level Arrays:**

- **Array A:**
  - Contains the minimum value of each block.
  - $A[i]$ is the minimum element in the i-th block of the original array.
- **Array B:**
  - Stores the position of the minimum value $A[i]$ within the i-th block.



3 **Sparse Table Preprocessing:**

- Precompute RMQs for **Array A** using a sparse table.
- This enables efficient querying for the blocks strictly between the blocks containing l and r in $O(1)$.

Step 3: Sparse Table for Array A

Sparse Table
Enables O(1) queries
between blocks

Query Example

1. Find block boundaries
2. Use sparse table for full blocks
3. Check partial blocks using preprocessed intra-block data

Step 4: Intra-Block Preprocessing

O(√N) types
Precompute all
possible RMQs

4 **Intra-Block RMQs:**

- Precompute all RMQs for every possible subrange within a block.
- Since all blocks can be normalized into $O(\sqrt{N})$ types, this preprocessing is efficient.

5 Query Decomposition:

- For a query range $RMQ(l, r)$:
  - If l and r are in the same block:
    - Directly use the precomputed intra-block RMQ.
  - If l and r span multiple blocks:
    - Compute the global minimum by combining three components:
      (a) The **suffix minimum** of l's block from index l to the end of the block.
      (b) The **inter-block minimum** from the sparse table, querying Array A for the blocks strictly between l's and r's blocks.
      (c) The **prefix minimum** of r's block from the start of the block to index r.
    - Combine the results of these three queries:

$$\text{RMQ}(l, r) = \min(\text{suffix\_min}, \text{inter\_block\_min}, \text{prefix\_min})$$

Time Complexity Analysis

PRECOMPUTATION

We divided the array A into blocks of size k= 0.5 log N
For each block we calculate the minimum element and store them in an array A
A has size N/K. We can construct a sparse table from the array A. The time taken will be

$$\frac{N}{K}\log\left(\frac{N}{K}\right) = \frac{2N}{\log(N)}\log\left(\frac{2N}{\log(N)}\right) =$$

$$= \frac{2N}{\log(N)}\left(1+\log\left(\frac{N}{\log(N)}\right)\right) \leq \frac{2N}{\log(N)} + 2N = O(N)$$

The precomputation for finding range minimum inside the block as explained in the next few slides is O (N)

Therefore total precomputation is = O (N) + O(N) = O (N)

QUERY

If the received range minimum query is [l, r] and l and r are in different blocks then the answer is the minimum of the following three values: the minimum of the suffix of block of l starting at l, the minimum of the prefix of block of r ending at r, and the minimum of the blocks between those. The minimum of the blocks in between can be answered in O(1) using the Sparse Table

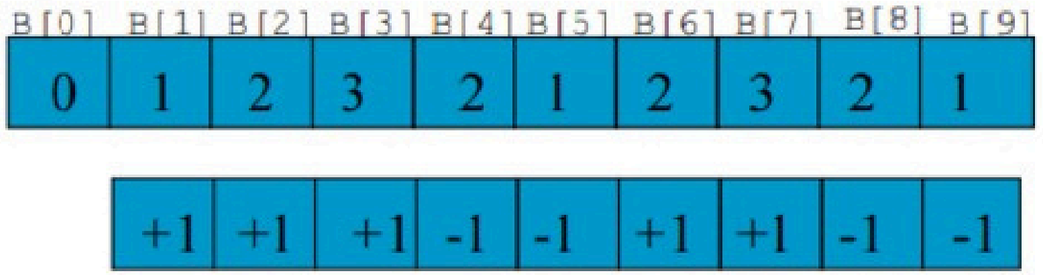
From the constructed block 3-dimensional array of masks, we can find the minimum for the remaining part inside the blocks in O (1) time.

Therefore total complexity is = O (1) + O(1) = O (1)

OBSERVATION – ARRAY FORMED IN EULER TOUR IS +-1 RMQ

The values in the array - which are just height values in the tree - will always differ by one. If we subtract the element from its adjacent element in the block, every block can be identified by a sequence of length K−1 consisting of the number +1 and -1.

A[i] = A[i] - A[i-1]



OBSERVATION - IF TWO ARRAYS X[1,2,.....K] AND Y[1,2,.....K] DIFFER BY SOME FIXED VALUE AT EACH POSITION, THAT IS, THERE IS A C SUCH THAT X[I] = Y[I] + C FOR EVERY I , THEN ALL RMQ ANSWERS WILL BE THE SAME FOR X AND Y. IN THIS CASE, WE CAN USE THE SAME PREPROCESSING FOR BOTH ARRAYS.

We can normalize a block by subtracting its initial offset from every element. We now use the +-1 property to show that there are very few kinds of normalized blocks.

Proof of correctness

Block Normalization Lemma:

Statement: For arrays satisfying the ± 1 property, blocks of size $K = 0.5 \log N$ can be normalized into $O(\sqrt{N})$ unique types, and all possible queries within such blocks can be precomputed in $O(N)$.

Proof:

1. **Block Representation:**
   - A block is represented as a binary number by replacing:
     - -1 with 0,
     - $+1$ with 1.
   - For a block of size $K = 0.5 \log N$, the total number of binary sequences is:

$$2^{K-1} = 2^{0.5 \log N - 1} = \sqrt{N}/2 = O(\sqrt{N}).$$

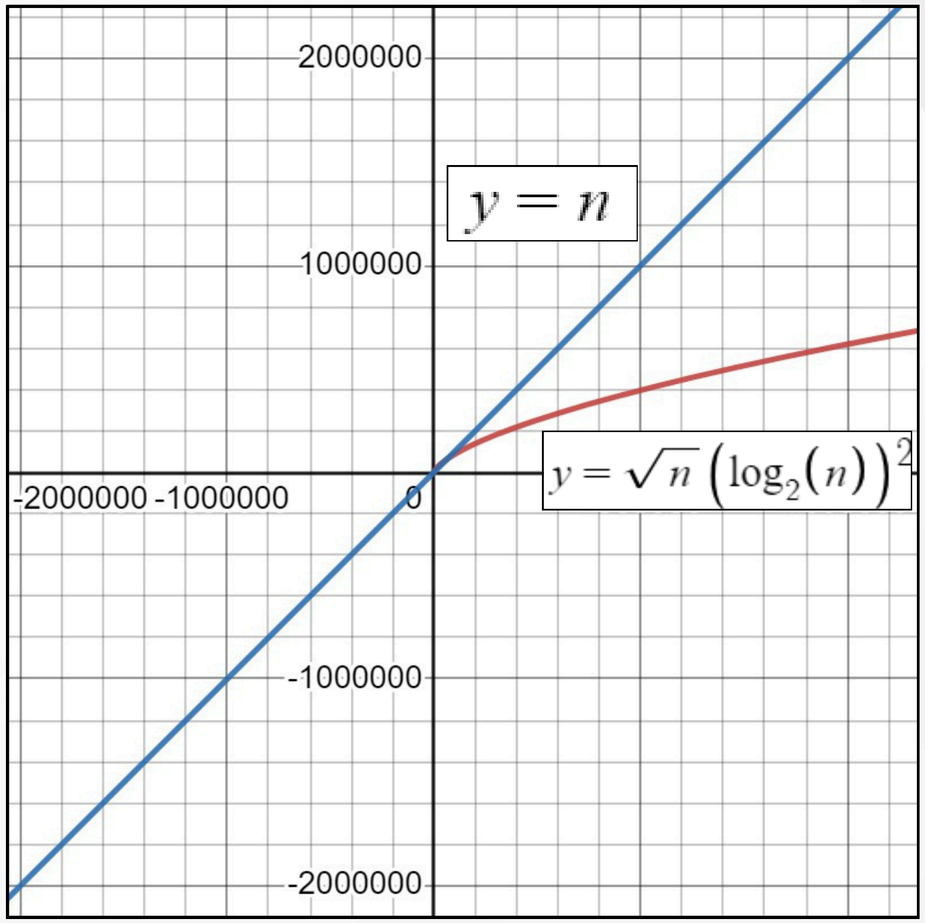
2. **Precomputing Queries:**
   - For each block type, precompute all range minimum queries (RMQ) for subranges within the block.
   - Total preprocessing time:

$$O(\sqrt{N} \cdot K^2) = O(\sqrt{N} \cdot (\log N)^2) = O(N).$$

3 **Block Storage:**

- Each block is characterized by a bitmask of length $K-1$, and the index of the minimum value is stored in a 3D array:

$$block[mask][l][r], of size O(\sqrt{N} \cdot \log^2 N).$$



Query Decomposition Lemma:

Statement: For any range $[l, r]$ in the array, the range minimum can be computed by decomposing it into:

1. The **suffix** of the block containing l, starting at l.
2. The **prefix** of the block containing r, ending at r.
3. The result of an **RMQ query** on the sparse table for the blocks strictly between l's and r's blocks.

Proof:

1 **Range Partitioning:**

- The range $[l, r]$ spans multiple blocks, so it can be divided into three disjoint parts:
  - **Part 1:** The suffix of the block containing l, starting at index l.
  - **Part 2:** The prefix of the block containing r, ending at index r.
  - **Part 3:** All complete blocks between the block of l and the block of r.

2 **Correctness of Decomposition:**

- The minimum value of the entire range $[l, r]$ must be the minimum of the three disjoint parts, as they fully cover the range.
- Each part is queried independently, ensuring no overlap or omissions.

3 **Efficient Query Handling:**

- **Part 1 and Part 2:**
  - Queries within a single block can be answered in $O(1)$ using precomputed intra-block RMQs.
- **Part 3:**
  - A sparse table is precomputed for the minimums of all blocks, allowing RMQ queries between blocks in $O(1)$.

4 **Combining Results:**

- The global minimum for $[l, r]$ is obtained as:

$$RMQ(l, r) = \min(suffix\_min, prefix\_min, inter\_block\_min),$$

where each component is queried independently in $O(1)$.

Applications

The Farach-Colton and Bender algorithm has numerous practical applications across various domains. Its efficient solutions to the Lowest Common Ancestor (LCA) and Range Minimum Query (RMQ) problems make it highly relevant in the following areas:

- **Genome Sequencing and Bioinformatics**: The LCA algorithm is essential for analyzing phylogenetic trees to determine evolutionary relationships between species. The RMQ aids in sequence alignment and identifying minimal subsequences, which are critical in DNA analysis.
- **Artificial Intelligence**: Facilitates hierarchical clustering and decision-making in tree-based machine learning algorithms. It improves the efficiency of search and decision processes in AI systems involving hierarchical data.

- **Network and Hierarchical Systems**: Used in communication networks to optimize routing paths by determining the closest common ancestor in network hierarchies. Helps in shared directory detection and resource allocation in hierarchical file systems.
- **Text Processing**: Integral to constructing suffix trees for fast string matching and pattern searching. RMQ is used in text compression algorithms and enhancing search engine functionality.
- **Graph Theory and Computational Geometry**: Solves subtree queries efficiently in dynamic tree data structures. Used in divide-and-conquer strategies for solving geometric problems like point location and range searching.
- **Gaming and Virtual Reality**: Collision detection in complex environments utilizes LCA for hierarchical object representation.

Conclusion

- The Farach-Colton and Bender algorithm presents an efficient and elegant solution to the Lowest Common Ancestor (LCA) and Range Minimum Query (RMQ) problems, leveraging advanced data structures and preprocessing techniques to achieve optimal time complexity.
- Through our exploration, we demonstrated the algorithm's utility in diverse domains such as genome sequencing, computer graphics, network optimization, and more. The reduction of the LCA problem to the RMQ problem and the subsequent efficient preprocessing via Euler tours and sparse tables underscore the power of combining mathematical insights with algorithmic ingenuity.
- By achieving preprocessing complexity of O(N) and query complexity of O(1), the algorithm sets a benchmark for solving hierarchical queries with unparalleled efficiency. This makes it particularly valuable for applications involving large-scale data and real-time query requirements.

References

1.Harel, D., & Tarjan, R. E. (1984). Fast Algorithms for Finding Nearest Common Ancestors. SIAM Journal on Computing, 13(2), 338–355.

2.Bender, M. A., & Farach-Colton, M. (2000). The LCA Problem Revisited. Proceedings of LATIN 2000: Theoretical Informatics, 88–94.

3.Schieber, B., & Vishkin, U. (1988). On Finding Lowest Common Ancestors: Simplification and Parallelization. SIAM Journal on Computing, 17(6), 1253–1262.

4.MIT OpenCourseWare (2017). Efficient Data Structures for RMQ. Lecture 15, Course 851 Advanced Data Structures.

5.Sparse Table and RMQ Implementation. CP-Algorithms