Lowest Common Ancestor(LCA)
a.k.a
Nearest Common Ancestor(NCA)

Fayssal El Moufatich

Technische Universität München
St. Petersburg

JASS 2008

Outline

- Introduction
- Definitions
- Applications
- Algorithms
  1. Harel-Tarjan's Algorithm and its variants
  2. LCA and DRS
  3. RMQ
  4. Bender-Farach's Algorithm
  5. Space-economic algorithm for Bender-Farach's Algorithm

Introduction

- One of the most fundamental algorithmic problems on trees is how to find the **Least Common Ancestor** of a pair of nodes.
- Studied intensively because:
  - It is inherently algorithmically beautifull.
  - Fast algorithms for the LCA problem can be used to solve other algorithmic problems.

Definitions

- Let there be a rooted tree T(E,V).

Definitions

- Let there be a rooted tree T(E,V).
- A node $x \in T$ is an ancestor of a node $y \in T$ if the path from the root of T to y goes through x.

Definitions

- Let there be a rooted tree T(E,V).
- A node $x \in T$ is an ancestor of a node $y \in T$ if the path from the root of T to y goes through x.
- A node $v \in T$ is a common ancestor of x and y if it is an ancestor of both x and y.

Definitions

- Let there be a rooted tree T(E,V).
- A node $x \in T$ is an ancestor of a node $y \in T$ if the path from the root of T to y goes through x.
- A node $v \in T$ is a common ancestor of x and y if it is an ancestor of both x and y.
- The Nearest/Lowest Common Ancestor, **NCA or LCA**, of two nodes x, y is the common ancestor of x and y whose distance to x (and to y) smaller than the distance to x of any common ancestor of x and y.

Definitions

- Let there be a rooted tree T(E,V).
- A node $x \in T$ is an ancestor of a node $y \in T$ if the path from the root of T to y goes through x.
- A node $v \in T$ is a common ancestor of x and y if it is an ancestor of both x and y.
- The Nearest/Lowest Common Ancestor, **NCA or LCA**, of two nodes x, y is the common ancestor of x and y whose distance to x (and to y) smaller than the distance to x of any common ancestor of x and y.
- We denote the NCA of x and y as $nca(x, y)$.

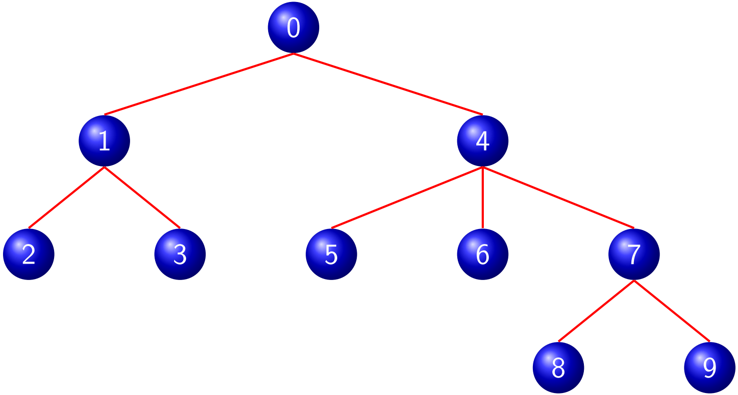
Definitions

- Let there be a rooted tree T(E,V).
- A node x $\in$ T is an ancestor of a node y $\in$ T if the path from the root of T to y goes through x.
- A node v $\in$ T is a common ancestor of x and y if it is an ancestor of both x and y.
- The Nearest/Lowest Common Ancestor, **NCA or LCA**, of two nodes x, y is the common ancestor of x and y whose distance to x (and to y) smaller than the distance to x of any common ancestor of x and y.
- We denote the NCA of x and y as $nca(x, y)$.
- Efficiently computing NCAs has been studied extensively for the last 3 decades in *online* and offline settings.

Definitions

- Let there be a rooted tree T(E,V).
- A node x $\in$ T is an ancestor of a node y $\in$ T if the path from the root of T to y goes through x.
- A node v $\in$ T is a common ancestor of x and y if it is an ancestor of both x and y.
- The Nearest/Lowest Common Ancestor, **NCA or LCA**, of two nodes x, y is the common ancestor of x and y whose distance to x (and to y) smaller than the distance to x of any common ancestor of x and y.
- We denote the NCA of x and y as $nca(x, y)$.
- Efficiently computing NCAs has been studied extensively for the last 3 decades in *online* and offline settings.

Example



Applications

- A procedure solving the NCA problem is used by algorithms for:
  - Finding the *maximum weighted matching* in a graph.
  - Finding a *minimum spanning tree* in a graph.
  - Finding a *dominator tree* in a graph in a *directed flow-graph*.
  - Several string algorithms.
  - *Dynamic planarity testing*.
  - In *network routing*.
  - Solving various geometric problems including *range searching*.
  - Finding *evolutionary trees*.
  - And in *bounded tree-width* algorithms.
  - ....

Survey of Algorithms

- One of the most fundamental results on computing NCAS is that of Harel and Tarjan [Harel, 1980], [Harel & Tarjan, 1984].

Survey of Algorithms

- One of the most fundamental results on computing NCAS is that of Harel and Tarjan [Harel, 1980], [Harel & Tarjan, 1984].
- They describe a linear time algorithm to preprocess a tree and build a data structure that allows subsequent NCA queries to be answered in **constant time!**.

Survey of Algorithms

- One of the most fundamental results on computing NCAS is that of Harel and Tarjan [Harel, 1980], [Harel & Tarjan, 1984].
- They describe a linear time algorithm to preprocess a tree and build a data structure that allows subsequent NCA queries to be answered in **constant time!**.
- Several simpler algorithms with essentially the same properties but better constant factors were proposed afterwards.

Survey of Algorithms

- One of the most fundamental results on computing NCAS is that of Harel and Tarjan [Harel, 1980], [Harel & Tarjan, 1984].
- They describe a linear time algorithm to preprocess a tree and build a data structure that allows subsequent NCA queries to be answered in **constant time!**.
- Several simpler algorithms with essentially the same properties but better constant factors were proposed afterwards.
- They all use the observation that it is rather easy to solve the problem when the input tree is a complete binary tree.

Survey of Algorithms

- One of the most fundamental results on computing NCAS is that of Harel and Tarjan [Harel, 1980], [Harel & Tarjan, 1984].
- They describe a linear time algorithm to preprocess a tree and build a data structure that allows subsequent NCA queries to be answered in **constant time!**.
- Several simpler algorithms with essentially the same properties but better constant factors were proposed afterwards.
- They all use the observation that it is rather easy to solve the problem when the input tree is a complete binary tree.

How do we do it?

- Label the nodes by their index in an *inorder* traversal of the complete binary tree.

How do we do it?

- Label the nodes by their index in an *inorder* traversal of the complete binary tree.
- If the tree has n nodes, each such number occupies $\ell = \lfloor \log(n) \rfloor$ bits.

How do we do it?

- Label the nodes by their index in an *inorder* traversal of the complete binary tree.
- If the tree has n nodes, each such number occupies $\ell = \lfloor \log(n) \rfloor$ bits.
- Assume the LSB is the rightmost and its index is 0.

How do we do it?

- Label the nodes by their index in an *inorder* traversal of the complete binary tree.
- If the tree has n nodes, each such number occupies $\ell = \lfloor \log(n) \rfloor$ bits.
- Assume the LSB is the rightmost and its index is 0.
- Let $\mathit{inorder}(x)$ and $\mathit{inorder}(y)$ be the inorder indexes of x and y.

How do we do it?

- Label the nodes by their index in an *inorder* traversal of the complete binary tree.
- If the tree has n nodes, each such number occupies $\ell = \lfloor \log(n) \rfloor$ bits.
- Assume the LSB is the rightmost and its index is 0.
- Let $inorder(x)$ and $inorder(y)$ be the inorder indexes of x and y.
- Let $i = max((1), (2), (3))$ where:

How do we do it?

- Label the nodes by their index in an *inorder* traversal of the complete binary tree.
- If the tree has n nodes, each such number occupies $\ell = \lfloor \log(n) \rfloor$ bits.
- Assume the LSB is the rightmost and its index is 0.
- Let $inorder(x)$ and $inorder(y)$ be the inorder indexes of x and y.
- Let $i = max((1), (2), (3))$ where:
  1. index of the leftmost bit in which $inorder(x)$ and $inorder(y)$ differ.

How do we do it?

- Label the nodes by their index in an *inorder* traversal of the complete binary tree.
- If the tree has n nodes, each such number occupies $\ell = \lfloor \log(n) \rfloor$ bits.
- Assume the LSB is the rightmost and its index is 0.
- Let $inorder(x)$ and $inorder(y)$ be the inorder indexes of x and y.
- Let $i = max((1), (2), (3))$ where:
  1. index of the leftmost bit in which $inorder(x)$ and $inorder(y)$ differ.
  2. index of the rightmost 1 in $inorder(x)$.

How do we do it?

- Label the nodes by their index in an *inorder* traversal of the complete binary tree.
- If the tree has n nodes, each such number occupies $\ell = \lfloor \log(n) \rfloor$ bits.
- Assume the LSB is the rightmost and its index is 0.
- Let $inorder(x)$ and $inorder(y)$ be the inorder indexes of x and y.
- Let $i = max((1), (2), (3))$ where:
  1. index of the leftmost bit in which $inorder(x)$ and $inorder(y)$ differ.
  2. index of the rightmost 1 in $inorder(x)$.
  3. index of the rightmost 1 in $inorder(y)$.

How do we do it?

- Label the nodes by their index in an *inorder* traversal of the complete binary tree.
- If the tree has n nodes, each such number occupies $\ell = \lfloor \log(n) \rfloor$ bits.
- Assume the LSB is the rightmost and its index is 0.
- Let $inorder(x)$ and $inorder(y)$ be the inorder indexes of x and y.
- Let $i = max((1), (2), (3))$ where:
  1. index of the leftmost bit in which $inorder(x)$ and $inorder(y)$ differ.
  2. index of the rightmost 1 in $inorder(x)$.
  3. index of the rightmost 1 in $inorder(y)$.
- It can be proved by induction that:

Lemma:[Shieber & Vishkin, 1987]

the $inorder(nca(x, y))$ consists of the leftmost $\ell - i$ bits of $inorder(x)$ (or $inorder(y)$ if the max was (3)) followed by a 1 and i zeros.

How do we do it?

- Label the nodes by their index in an *inorder* traversal of the complete binary tree.
- If the tree has n nodes, each such number occupies $\ell = \lfloor \log(n) \rfloor$ bits.
- Assume the LSB is the rightmost and its index is 0.
- Let $inorder(x)$ and $inorder(y)$ be the inorder indexes of x and y.
- Let $i = max((1), (2), (3))$ where:
  1. index of the leftmost bit in which $inorder(x)$ and $inorder(y)$ differ.
  2. index of the rightmost 1 in $inorder(x)$.
  3. index of the rightmost 1 in $inorder(y)$.
- It can be proved by induction that:

Lemma:[Shieber & Vishkin, 1987]

the $inorder(nca(x, y))$ consists of the leftmost $\ell - i$ bits of $inorder(x)$ (or $inorder(y)$ if the max was (3)) followed by a 1 and i zeros.

Example

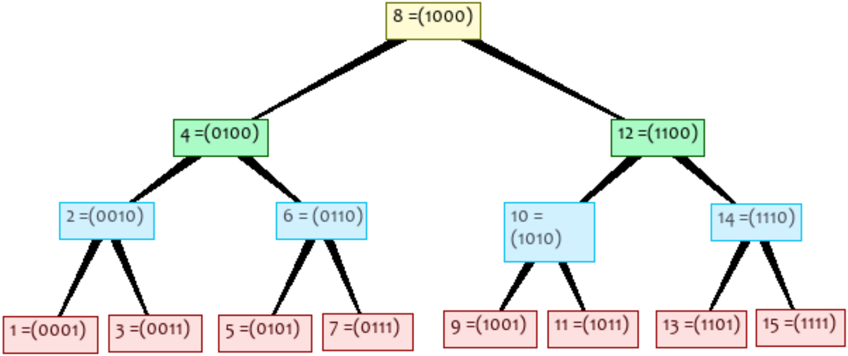
Basic idea

construct the $inorder(nca(x, y))$ from $inorder(x)$ and $inorder(y)$ alone and without accessing the original tree or any other global data structure
$\Rightarrow$constant time!

Example

Basic idea

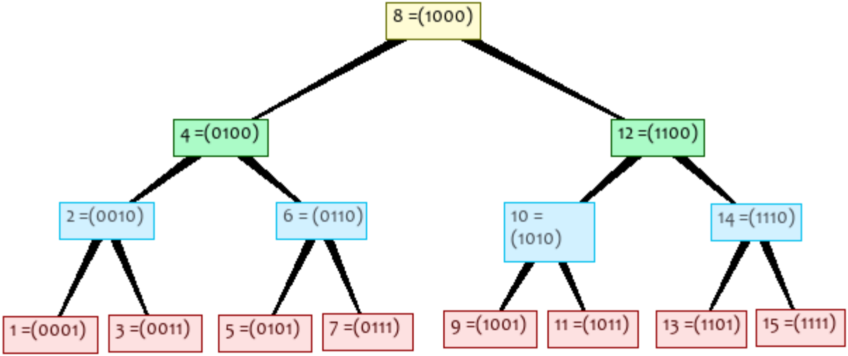
construct the $inorder(nca(x, y))$ from $inorder(x)$ and $inorder(y)$ alone and without accessing the original tree or any other global data structure
$\Rightarrow$constant time!



Example

Basic idea

construct the $inorder(nca(x, y))$ from $inorder(x)$ and $inorder(y)$ alone and without accessing the original tree or any other global data structure
$\Rightarrow$constant time!



So what if the input tree is not a completely balanced binary tree?

- Simply do a mapping to a completely binary balanced tree!

So what if the input tree is not a completely balanced binary tree?

- Simply do a mapping to a completely binary balanced tree!
- Different algorithms differ by the way they do the mapping.

So what if the input tree is not a completely balanced binary tree?

- Simply do a mapping to a completely binary balanced tree!
- Different algorithms differ by the way they do the mapping.
- All algorithms have to use some precomputed auxilliary data structures and the labels of the nodes to compute the NCAS :(.

So what if the input tree is not a completely balanced binary tree?

- Simply do a mapping to a completely binary balanced tree!
- Different algorithms differ by the way they do the mapping.
- All algorithms have to use some precomputed auxilliary data structures and the labels of the nodes to compute the NCAS :(.
- $\Rightarrow$Most of algorithms for general trees do not allow to compute a unique identifier of nca(x,y) from short labels associated with x and y alone.

So what if the input tree is not a completely balanced binary tree?

- Simply do a mapping to a completely binary balanced tree!
- Different algorithms differ by the way they do the mapping.
- All algorithms have to use some precomputed auxilliary data structures and the labels of the nodes to compute the NCAS :(.
- $\Rightarrow$Most of algorithms for general trees do not allow to compute a unique identifier of nca(x,y) from short labels associated with x and y alone.
- However, ...

So what if the input tree is not a completely balanced binary tree?

- Simply do a mapping to a completely binary balanced tree!
- Different algorithms differ by the way they do the mapping.
- All algorithms have to use some precomputed auxilliary data structures and the labels of the nodes to compute the NCAS :(.
- ⇒Most of algorithms for general trees do not allow to compute a unique identifier of nca(x,y) from short labels associated with x and y alone.
- However, ...

Cont.

- One can prove the following:

Cont.

- One can prove the following:

Theorem

There is a linear time algorithm that labels the n nodes of a rooted tree T with labels of length $\mathcal{O}(\log n)$ bits such that from the labels of nodes x, y in T alone, one can compute the label of $nca(x, y)$ in constant time.

Cont.

- One can prove the following:

Theorem

There is a linear time algorithm that labels the n nodes of a rooted tree T with labels of length $\mathcal{O}(\log n)$ bits such that from the labels of nodes x, y in T alone, one can compute the label of $nca(x, y)$ in constant time.

Proof:[Kaplan et al., 2002]

- Use lexigraphic sorting the sequence of intergers or binary strings.
- Use results from Gilbert and Moore on alphabetic coding of sequences of integers $\langle b \rangle_k (|b_i| < \log n - \log y_i + \mathcal{O}(1)$ for all i).
- use labeling along **HPs, Heavy Paths**.

Cont.

- One can prove the following:

Theorem

There is a linear time algorithm that labels the n nodes of a rooted tree T with labels of length $\mathcal{O}(\log n)$ bits such that from the labels of nodes x, y in T alone, one can compute the label of $nca(x, y)$ in constant time.

Proof:[Kaplan et al., 2002]

- Use lexigraphic sorting the sequence of intergers or binary strings.
- Use results from Gilbert and Moore on alphabetic coding of sequences of integers $\langle b \rangle_k (|b_i| < \log n - \log y_i + \mathcal{O}(1)$ for all i).
- use labeling along **HPs, Heavy Paths**.

NCA and Discrete Range Searching (DRS)

- Gabow, Bentley and Tarjan observed that one-dimensional DRS problem is equivalent to NCA problem.
- DRS used by most of simple NCA algorithms.

DRS Problem

Given a sequence of real numbers x_1, x_2, ...x_n, preprocess the sequence so that one can answer efficiently subsequent queries of the form:
given a pair of indices (i, j), what is the maximum element among x_i,...,x_j or $max(i, j)$.

NCA and Discrete Range Searching (DRS)

- Gabow, Bentley and Tarjan observed that one-dimensional DRS problem is equivalent to NCA problem.
- DRS used by most of simple NCA algorithms.

DRS Problem

Given a sequence of real numbers x_1, x_2, ...x_n, preprocess the sequence so that one can answer efficiently subsequent queries of the form:
given a pair of indices (i, j), what is the maximum element among x_i,...,x_j or $max(i, j)$.

- DRS problem is a fundamental geometric searching problem.

NCA and Discrete Range Searching (DRS)

- Gabow, Bentley and Tarjan observed that one-dimensional DRS problem is equivalent to NCA problem.
- DRS used by most of simple NCA algorithms.

DRS Problem

Given a sequence of real numbers x_1, x_2, ...x_n, preprocess the sequence so that one can answer efficiently subsequent queries of the form:
given a pair of indices (i, j), what is the maximum element among x_i,...,x_j or $max(i, j)$.

- DRS problem is a fundamental geometric searching problem.
- DRS can be reduced to NCA by constructing a **Cartesian tree** for the sequence x_1, ..., x_n [Gabow et al., 1984].

NCA and Discrete Range Searching (DRS)

- Gabow, Bentley and Tarjan observed that one-dimensional DRS problem is equivalent to NCA problem.
- DRS used by most of simple NCA algorithms.

DRS Problem

Given a sequence of real numbers x_1, x_2, ...x_n, preprocess the sequence so that one can answer efficiently subsequent queries of the form:
given a pair of indices (i, j), what is the maximum element among x_i,...,x_j or $max(i, j)$.

- DRS problem is a fundamental geometric searching problem.
- DRS can be reduced to NCA by constructing a **Cartesian tree** for the sequence x_1, ..., x_n [Gabow et al., 1984].

What is a Cartesian tree?

Cartesian Tree

The Cartesian tree of the sequence $x_1, ..., x_n$ is a **binary tree** with n nodes each containing a number x_i and the following properties:

What is a Cartesian tree?

Cartesian Tree

The Cartesian tree of the sequence $x_1, ..., x_n$ is a **binary tree** with n nodes each containing a number x_i and the following properties:
Let $x_j = max(x_1, ..., x_n)$

What is a Cartesian tree?

Cartesian Tree

The Cartesian tree of the sequence $x_1, ..., x_n$ is a **binary tree** with n nodes each containing a number x_i and the following properties:
Let $x_j = max(x_1, ..., x_n)$

1. The root of the Cartesian tree contains x_j.

What is a Cartesian tree?

Cartesian Tree

The Cartesian tree of the sequence $x_1, ..., x_n$ is a **binary tree** with n nodes each containing a number x_i and the following properties:
Let $x_j = max(x_1, ..., x_n)$

1. The root of the Cartesian tree contains x_j.
2. The left subtree of the root is a Cartesian tree for $x_1, ..., x_{j-1}$.

What is a Cartesian tree?

Cartesian Tree

The Cartesian tree of the sequence $x_1, \ldots, x_n$ is a **binary tree** with n nodes each containing a number x_i and the following properties:

Let $x_j = max(x_1, \ldots, x_n)$

1. The root of the Cartesian tree contains x_j.
2. The left subtree of the root is a Cartesian tree for $x_1, \ldots, x_{j-1}$.
3. The right subtree of the root is a Cartesian tree for $x_{j+1}, \ldots, x_n$.

What is a Cartesian tree?

Cartesian Tree

The Cartesian tree of the sequence $x_1, ..., x_n$ is a **binary tree** with n nodes each containing a number x_i and the following properties:

Let $x_j = max(x_1, ..., x_n)$

1. The root of the Cartesian tree contains x_j.
2. The left subtree of the root is a Cartesian tree for $x_1, ..., x_{j-1}$.
3. The right subtree of the root is a Cartesian tree for $x_{j+1}, ..., x_n$.

Remarks

- The Cartesian tree for $x_1, ..., x_n$ can be constructed in $\mathcal{O}(n)$ [Vuillemin, 1980].

What is a Cartesian tree?

Cartesian Tree

The Cartesian tree of the sequence $x_1, ..., x_n$ is a **binary tree** with n nodes each containing a number x_i and the following properties:
Let $x_j = max(x_1, ..., x_n)$

1. The root of the Cartesian tree contains x_j.
2. The left subtree of the root is a Cartesian tree for $x_1, ..., x_{j-1}$.
3. The right subtree of the root is a Cartesian tree for $x_{j+1}, ..., x_n$.

Remarks

- The Cartesian tree for $x_1, ..., x_n$ can be constructed in $\mathcal{O}(n)$ [Vuillemin, 1980].
- The maximum among $x_i, ..., x_j$ corresponds to just the NCA of the node containing x_i and the node containing x_j.

What is a Cartesian tree?

Cartesian Tree

The Cartesian tree of the sequence $x_1, ..., x_n$ is a **binary tree** with n nodes each containing a number x_i and the following properties:
Let $x_j = max(x_1, ..., x_n)$

1. The root of the Cartesian tree contains x_j.
2. The left subtree of the root is a Cartesian tree for $x_1, ..., x_{j-1}$.
3. The right subtree of the root is a Cartesian tree for $x_{j+1}, ..., x_n$.

Remarks

- The Cartesian tree for $x_1, ..., x_n$ can be constructed in $\mathcal{O}(n)$ [Vuillemin, 1980].
- The maximum among $x_i, ..., x_j$ corresponds to just the NCA of the node containing x_i and the node containing x_j.

What about NCA as DRS?

- Gabow et al. also show how to reduce the NCA problem to the DRS problem.
- Given a tree, we first construct a sequence of its nodes by doing a depth first traversal.

What about NCA as DRS?

- Gabow et al. also show how to reduce the NCA problem to the DRS problem.
- Given a tree, we first construct a sequence of its nodes by doing a depth first traversal.
- Each time we visit a node, we add it to the end of the sequence so that each node appears in the sequence as many times as its degree.[a prefix of the Euler tour of the tree]

What about NCA as DRS?

- Gabow et al. also show how to reduce the NCA problem to the DRS problem.
- Given a tree, we first construct a sequence of its nodes by doing a depth first traversal.
- Each time we visit a node, we add it to the end of the sequence so that each node appears in the sequence as many times as its degree.[a prefix of the Euler tour of the tree]
- Let $depth(x)$ be the depth of a node x.

What about NCA as DRS?

- Gabow et al. also show how to reduce the NCA problem to the DRS problem.
- Given a tree, we first construct a sequence of its nodes by doing a depth first traversal.
- Each time we visit a node, we add it to the end of the sequence so that each node appears in the sequence as many times as its degree.[a prefix of the Euler tour of the tree]
- Let $depth(x)$ be the depth of a node x.
- Replace each node x in the sequence by -$depth(x)$.

What about NCA as DRS?

- Gabow et al. also show how to reduce the NCA problem to the DRS problem.
- Given a tree, we first construct a sequence of its nodes by doing a depth first traversal.
- Each time we visit a node, we add it to the end of the sequence so that each node appears in the sequence as many times as its degree.[a prefix of the Euler tour of the tree]
- Let $depth(x)$ be the depth of a node x.
- Replace each node x in the sequence by -$depth(x)$.
- To compute $nca(x, y)$, we pick arbirary 2 elements x_i and x_j representing x and y, and compute the maximum among $x_i, ..., x_j$.

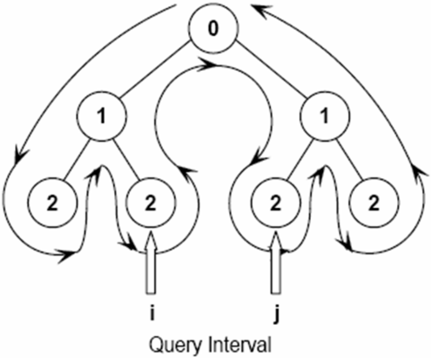
What about NCA as DRS?

- Gabow et al. also show how to reduce the NCA problem to the DRS problem.
- Given a tree, we first construct a sequence of its nodes by doing a depth first traversal.
- Each time we visit a node, we add it to the end of the sequence so that each node appears in the sequence as many times as its degree.[a prefix of the Euler tour of the tree]
- Let $depth(x)$ be the depth of a node x.
- Replace each node x in the sequence by -$depth(x)$.
- To compute $nca(x, y)$, we pick arbirary 2 elements x_i and x_j representing x and y, and compute the maximum among $x_i, ..., x_j$.
- The node corresponding to the maximum element is $nca(x, y)$!

What about NCA as DRS?

- Gabow et al. also show how to reduce the NCA problem to the DRS problem.
- Given a tree, we first construct a sequence of its nodes by doing a depth first traversal.
- Each time we visit a node, we add it to the end of the sequence so that each node appears in the sequence as many times as its degree.[a prefix of the Euler tour of the tree]
- Let $depth(x)$ be the depth of a node x.
- Replace each node x in the sequence by -$depth(x)$.
- To compute $nca(x, y)$, we pick arbirary 2 elements x_i and x_j representing x and y, and compute the maximum among $x_i, ..., x_j$.
- The node corresponding to the maximum element is $nca(x, y)$!

Euler tour of a tree



Euler tour of the tree on the left

0121210121210

i j

Query Interval

What is the LCA of given two nodes then?

- Simply the node of the least depth (i.e. Closest to the root) that lies between the nodes in the Euler tour.

What is the LCA of given two nodes then?

- Simply the node of the least depth (i.e. Closest to the root) that lies between the nodes in the Euler tour.
- Hence, finding specific node in the tree $\Leftrightarrow$ finding minimum element in the proper interval in the array of numbers.

What is the LCA of given two nodes then?

- Simply the node of the least depth (i.e. Closest to the root) that lies between the nodes in the Euler tour.
- Hence, finding specific node in the tree $\Leftrightarrow$ finding minimum element in the proper interval in the array of numbers.
- Latter problem can be solved by **min-range queries**.

What is the LCA of given two nodes then?

- Simply the node of the least depth (i.e. Closest to the root) that lies between the nodes in the Euler tour.
- Hence, finding specific node in the tree $\Leftrightarrow$ finding minimum element in the proper interval in the array of numbers.
- Latter problem can be solved by **min-range queries**.

Range Minimum Query (RMQ) Problem

Definition of RMQ Problem

- Same as the DSR problem but outputs the minimum instead.
- **Structure to Preprocess:** an array of numbers of length n.

Range Minimum Query (RMQ) Problem

Definition of RMQ Problem

- Same as the DSR problem but outputs the minimum instead.
- **Structure to Preprocess:** an array of numbers of length n.
- **Query:** for indices i and j and n, query **RMQ(x,y)** returns the index of the smallest element in the subarray A[i...j].

Range Minimum Query (RMQ) Problem

Definition of RMQ Problem

- Same as the DSR problem but outputs the minimum instead.
- **Structure to Preprocess:** an array of numbers of length n.
- **Query:** for indices i and j and n, query **RMQ(x,y)** returns the index of the smallest element in the subarray A[i...j].

Remark

As with the DSR algorithm, LCA can be reduced to an RMQ problem.[Bender-Farach, 2000]

Range Minimum Query (RMQ) Problem

Definition of RMQ Problem

- Same as the DSR problem but outputs the minimum instead.
- **Structure to Preprocess:** an array of numbers of length n.
- **Query:** for indices i and j and n, query **RMQ(x,y)** returns the index of the smallest element in the subarray A[i...j].

Remark

As with the DSR algorithm, LCA can be reduced to an RMQ problem.[Bender-Farach, 2000]

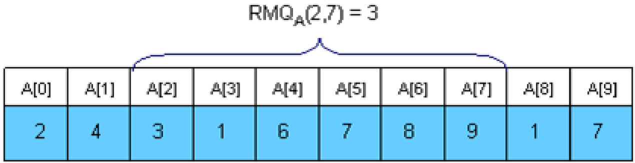
Range Minimum Query (RMQ) Problem

Definition of RMQ Problem

- Same as the DSR problem but outputs the minimum instead.
- **Structure to Preprocess:** an array of numbers of length n.
- **Query:** for indices i and j and n, query **RMQ(x,y)** returns the index of the smallest element in the subarray A[i...j].

Remark

As with the DSR algorithm, LCA can be reduced to an RMQ problem.[Bender-Farach, 2000]



Isn't that a loop in our reduction?

- We started by reducing the range-min/DSR problem to an LCA problem

Isn't that a loop in our reduction?

- We started by reducing the range-min/DSR problem to an LCA problem
- Answer is no!

Isn't that a loop in our reduction?

- We started by reducing the range-min/DSR problem to an LCA problem
- Answer is no!
- The constructed array of numbers has a special property known as ∓ 1 property:

Isn't that a loop in our reduction?

- We started by reducing the range-min/DSR problem to an LCA problem
- Answer is no!
- The constructed array of numbers has a special property known as ∓ 1 property:

∓ 1 property

Each number differs by exactly one from its preceding number.

Isn't that a loop in our reduction?

- We started by reducing the range-min/DSR problem to an LCA problem
- Answer is no!
- The constructed array of numbers has a special property known as ∓ 1 property:

∓ 1 property

Each number differs by exactly one from its preceding number.

- Hence, our reduction is a special case of the range-min query problem that can be solved without further reductions.

Isn't that a loop in our reduction?

- We started by reducing the range-min/DSR problem to an LCA problem
- Answer is no!
- The constructed array of numbers has a special property known as ∓ 1 property:

∓ 1 property

Each number differs by exactly one from its preceding number.

- Hence, our reduction is a special case of the range-min query problem that can be solved without further reductions.

Cont.

- Our goal then is to solve the following problem:

Cont.

- Our goal then is to solve the following problem:

Problem

Preprocess an array of n numbers satisfying the ∓ 1 property such that given two indices i and j in the array, determine the index of the minimum element within the given range $[i, j]$, $\mathcal{O}(1)$ time and $\mathcal{O}(n)$ space.

Cont.

- Our goal then is to solve the following problem:

Problem

Preprocess an array of n numbers satisfying the ∓ 1 property such that given two indices i and j in the array, determine the index of the minimum element within the given range $[i, j]$, $\mathcal{O}(1)$ time and $\mathcal{O}(n)$ space.

Bender-Farach Algorithm for LCA

- Reengineered from existing complicated LCA algorithms. (PRAM from Berkman et al.)

Bender-Farach Algorithm for LCA

- Reengineered from existing complicated LCA algorithms. (PRAM from Berkman et al.)
- Reduces the LCA problem to an RMQ problem and considers RMQ solutions rather.

Bender-Farach Algorithm for LCA

- Reengineered from existing complicated LCA algorithms. (PRAM from Berkman et al.)
- Reduces the LCA problem to an RMQ problem and considers RMQ solutions rather.

Naïve Attempt

- RMQ has a simple solution with complexity $\langle \mathcal{O}(n^2), \mathcal{O}(1) \rangle$:

Naïve Attempt

- RMQ has a simple solution with complexity $\langle \mathcal{O}(n^2), \mathcal{O}(1) \rangle$:
- Build a lookup table storing answers to all the n^2 possible queries.

Naïve Attempt

- RMQ has a simple solution with complexity $\langle \mathcal{O}(n^2), \mathcal{O}(1) \rangle$:
- Build a lookup table storing answers to all the n^2 possible queries.
- To achieve $\mathcal{O}(n^2)$ preprocessing rather than $\mathcal{O}(n^3)$, we use a trivial dynamic program.

Naïve Attempt

- RMQ has a simple solution with complexity $\langle \mathcal{O}(n^2), \mathcal{O}(1) \rangle$:
- Build a lookup table storing answers to all the n^2 possible queries.
- To achieve $\mathcal{O}(n^2)$ preprocessing rather than $\mathcal{O}(n^3)$, we use a trivial dynamic program.

A Faster RMQ Algorithm

- Idea: precompute each query whose length is a power of 2.

A Faster RMQ Algorithm

- Idea: precompute each query whose length is a power of 2.
- i.e. for each i in $[1, n]$ and every j in $[1, \log n]$, find the minimum of the block starting at i and has length 2^j

A Faster RMQ Algorithm

- Idea: precompute each query whose length is a power of 2.
- i.e. for each i in $[1, n]$ and every j in $[1, \log n]$, find the minimum of the block starting at i and has length 2^j
- i.e.

$$M[i,j] = argmin_{k=i \ldots i+2^j-1} A[k] \tag{1}$$

A Faster RMQ Algorithm

- Idea: precompute each query whose length is a power of 2.
- i.e. for each i in $[1, n]$ and every j in $[1, \log n]$, find the minimum of the block starting at i and has length 2^j
- i.e.

$$M[i,j] = argmin_{k=i \ldots i+2^j-1} A[k] \quad (1)$$

- Table M has size $\mathcal{O}(n \log n)$

A Faster RMQ Algorithm

- Idea: precompute each query whose length is a power of 2.
- i.e. for each i in $[1, n]$ and every j in $[1, \log n]$, find the minimum of the block starting at i and has length 2^j
- i.e.

$$M[i, j] = argmin_{k=i \dots i+2^j-1} A[k] \quad (1)$$

- Table M has size $\mathcal{O}(n \log n)$
- We fill it in using *dynamic programming*.

A Faster RMQ Algorithm

- Idea: precompute each query whose length is a power of 2.
- i.e. for each i in $[1, n]$ and every j in $[1, \log n]$, find the minimum of the block starting at i and has length 2^j
- i.e.

$$M[i, j] = argmin_{k=i \dots i+2^j-1} A[k] \tag{1}$$

- Table M has size $\mathcal{O}(n \log n)$
- We fill it in using *dynamic programming*.
- Find the minimum in a block of size 2^j by comparing the two minima of its constituent blocks of size 2^{j-1}.

A Faster RMQ Algorithm

- Idea: precompute each query whose length is a power of 2.
- i.e. for each i in $[1, n]$ and every j in $[1, \log n]$, find the minimum of the block starting at i and has length 2^j
- i.e.

$$(1) \qquad M[i,j] = argmin_{k=i \ldots i+2^j-1} A[k]$$

- Table M has size $\mathcal{O}(n \log n)$
- We fill it in using *dynamic programming*.
- Find the minimum in a block of size 2^j by comparing the two minima of its constituent blocks of size 2^{j-1}.
- Formally speaking,

$$(2) \qquad M[i,j] = M[i,j-1] \, if \, A[M[i,j-1]] \leq A[M[i+2^{j-1}, j-1]]$$

and

$$(3) \qquad M[i,j] = M[i+2^{j-1}, j-1] \, otherwise$$

A Faster RMQ Algorithm

- Idea: precompute each query whose length is a power of 2.
- i.e. for each i in $[1, n]$ and every j in $[1, \log n]$, find the minimum of the block starting at i and has length 2^j
- i.e.

$$(1) \qquad M[i,j] = argmin_{k=i \ldots i+2^j-1} A[k]$$

- Table M has size $\mathcal{O}(n \log n)$
- We fill it in using *dynamic programming*.
- Find the minimum in a block of size 2^j by comparing the two minima of its constituent blocks of size 2^{j-1}.
- Formally speaking,

$$(2) \qquad M[i,j] = M[i,j-1] \, if \, A[M[i,j-1]] \leq A[M[i+2^{j-1},j-1]]$$

and

$$(3) \qquad M[i,j] = M[i+2^{j-1},j-1] \, otherwise$$

How do we use blocks to compute an arbitrary RMQ(i,j)?

- Select 2 overlapping blocks that entirely cover the subrange.

How do we use blocks to compute an arbitrary RMQ(i,j)?

- Select 2 overlapping blocks that entirely cover the subrange.
- Let 2^k be the size of the largest block that fits into the range from i to j, i.e. $k = \lfloor \log(j - i) \rfloor$.

How do we use blocks to compute an arbitrary RMQ(i,j)?

- Select 2 overlapping blocks that entirely cover the subrange.
- Let 2^k be the size of the largest block that fits into the range from i to j, i.e. $k = \lfloor \log(j - i) \rfloor$.
- $RMQ(i, j)$ can be computed by comparing the minima of the 2 blocks: i to $i + 2^k - 1$ (M(i,k)) and $j - 2^k + 1$ to j $(M(j - 2^k + 1, k))$.

How do we use blocks to compute an arbitrary RMQ(i,j)?

- Select 2 overlapping blocks that entirely cover the subrange.
- Let 2^k be the size of the largest block that fits into the range from i to j, i.e. $k = \lfloor \log(j - i) \rfloor$.
- $RMQ(i, j)$ can be computed by comparing the minima of the 2 blocks: i to $i + 2^k - 1$ (M(i,k)) and $j - 2^k + 1$ to j ($M(j - 2^k + 1, k)$).
- Already computed values $\Rightarrow$ we can find RMQ in constant time!

How do we use blocks to compute an arbitrary RMQ(i,j)?

- Select 2 overlapping blocks that entirely cover the subrange.
- Let 2^k be the size of the largest block that fits into the range from i to j, i.e. $k = \lfloor \log(j-i) \rfloor$.
- $RMQ(i,j)$ can be computed by comparing the minima of the 2 blocks: i to $i + 2^k - 1$ (M(i,k)) and $j - 2^k + 1$ to j ($M(j - 2^k + 1, k)$).
- Already computed values $\Rightarrow$ we can find RMQ in constant time!

Remarks

- This gives the **Sparse Table(TS)** algorithm for RMQ with complexity $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$.

Remarks

- This gives the **Sparse Table(TS)** algorithm for RMQ with complexity $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$.
- Total computation to answer an RMQ query is 3 additions, 4 array reference and a minimum, and 2 ops: log and floor.

Remarks

- This gives the **Sparse Table(TS)** algorithm for RMQ with complexity $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$.
- Total computation to answer an RMQ query is 3 additions, 4 array reference and a minimum, and 2 ops: log and floor.
- Can be seen as problem of finding the MSB of a word.

Remarks

- This gives the **Sparse Table(TS)** algorithm for RMQ with complexity $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$.
- Total computation to answer an RMQ query is 3 additions, 4 array reference and a minimum, and 2 ops: log and floor.
- Can be seen as problem of finding the MSB of a word.
- LCA problem shown to have $\Omega(\log \log n)$on a pointer machine by Harel and Tarjan.

Remarks

- This gives the **Sparse Table(TS)** algorithm for RMQ with complexity $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$.
- Total computation to answer an RMQ query is 3 additions, 4 array reference and a minimum, and 2 ops: log and floor.
- Can be seen as problem of finding the MSB of a word.
- LCA problem shown to have $\Omega(\log \log n)$on a pointer machine by Harel and Tarjan.

An $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ algorithm for $\mp$RMQ

- Faster algorithm for $\mp$RMQ!

An $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ algorithm for $\mp$RMQ

- Faster algorithm for $\mp$RMQ!
- Suppose we have array A with $\mp$ **restriction**.

An $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ algorithm for $\mp$RMQ

- Faster algorithm for $\mp$RMQ!
- Suppose we have array A with $\mp$ **restriction**.
- Use lookup-table to precompute answers for small subarrays? $\Rightarrow$ remove log factor from preprocessing!

An $\langle\mathcal{O}(n), \mathcal{O}(1)\rangle$ algorithm for $\mp$RMQ

- Faster algorithm for $\mp$RMQ!
- Suppose we have array A with $\mp$ **restriction**.
- Use lookup-table to precompute answers for small subarrays? $\Rightarrow$ remove log factor from preprocessing!
- Partition A into blocks of size $\frac{\log n}{2}$.

An $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ algorithm for $\mp$RMQ

- Faster algorithm for $\mp$RMQ!
- Suppose we have array A with $\mp$ **restriction**.
- Use lookup-table to precompute answers for small subarrays? $\Rightarrow$ remove log factor from preprocessing!
- Partition A into blocks of size $\frac{\log n}{2}$.
- Define an arrayA'[1,..., $\frac{2n}{\log n}$] where A'[i] is the minimum of the ith block of A.

An $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ algorithm for $\mp$RMQ

- Faster algorithm for $\mp$RMQ!
- Suppose we have array A with $\mp$ **restriction**.
- Use lookup-table to precompute answers for small subarrays? $\Rightarrow$ remove log factor from preprocessing!
- Partition A into blocks of size $\frac{\log n}{2}$.
- Define an arrayA'[1,...,$\frac{2n}{\log n}$] where A'[i] is the minimum of the ith block of A.
- Define an equal size array B where B[i] is a position in the ith block in which A[i] occurs.

An $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ algorithm for $\mp$RMQ

- Faster algorithm for $\mp$RMQ!
- Suppose we have array A with $\mp$ **restriction**.
- Use lookup-table to precompute answers for small subarrays? $\Rightarrow$ remove log factor from preprocessing!
- Partition A into blocks of size $\frac{\log n}{2}$.
- Define an arrayA'[1,..., $\frac{2n}{\log n}$] where A'[i] is the minimum of the ith block of A.
- Define an equal size array B where B[i] is a position in the ith block in which A[i] occurs.
- B used to keep track of where the minima of A came from.

An $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ algorithm for $\mp$RMQ

- Faster algorithm for $\mp$RMQ!
- Suppose we have array A with $\mp$ **restriction**.
- Use lookup-table to precompute answers for small subarrays? $\Rightarrow$ remove log factor from preprocessing!
- Partition A into blocks of size $\frac{\log n}{2}$.
- Define an arrayA'[1,...,$\frac{2n}{\log n}$] where A'[i] is the minimum of the ith block of A.
- Define an equal size array B where B[i] is a position in the ith block in which A[i] occurs.
- B used to keep track of where the minima of A came from.

Cont.

- ST algorithm runs on A in time $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$.

Cont.

- ST algorithm runs on A in time $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$.
- Consider RMQ(i,j) in A:
  - i and j can be in same block? $\Rightarrow$ process each block to answer RMQ queries.

Cont.

- ST algorithm runs on A in time $\langle\mathcal{O}(n), \mathcal{O}(1)\rangle$.
- Consider RMQ(i,j) in A:
  - i and j can be in same block? $\Rightarrow$ process each block to answer RMQ queries.
  - $i < j$:

Cont.

- ST algorithm runs on A in time $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$.
- Consider RMQ(i,j) in A:
  - i and j can be in same block? $\Rightarrow$ process each block to answer RMQ queries.
  - $i < j$:
    - Minimum from i forward to end of its block.

Cont.

- ST algorithm runs on A in time $\langle\mathcal{O}(n),\mathcal{O}(1)\rangle$.
- Consider RMQ(i,j) in A:
  - i and j can be in same block? $\Rightarrow$ process each block to answer RMQ queries.
  - $i < j$:
    - Minimum from i forward to end of its block.
    - Minimum of all blocks btw. is block and js block.

Cont.

- ST algorithm runs on A in time $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$.
- Consider RMQ(i,j) in A:
  - i and j can be in same block? $\Rightarrow$ process each block to answer RMQ queries.
  - $i < j$:
    - Minimum from i forward to end of its block.
    - Minimum of all blocks btw. is block and js block.
    - Minimum from beginning of js block to j.

Cont.

- ST algorithm runs on A in time $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$.
- Consider RMQ(i,j) in A:
  - i and j can be in same block? $\Rightarrow$ process each block to answer RMQ queries.
  - $i < j$:
    - Minimum from i forward to end of its block.
    - Minimum of all blocks btw. is block and js block.
    - Minimum from beginning of js block to j.
- 2nd minimum is found in constant time by RMQ on A.

Cont.

- ST algorithm runs on A in time $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$.
- Consider RMQ(i,j) in A:
  - i and j can be in same block? $\Rightarrow$ process each block to answer RMQ queries.
  - $i < j$:
    - Minimum from i forward to end of its block.
    - Minimum of all blocks btw. is block and js block.
    - Minimum from beginning of js block to j.
- 2nd minimum is found in constant time by RMQ on A.

How to answer range RMQ queries inside blocks?

- In-block queries needed for 1st and 3rd values to complete algorithm.

How to answer range RMQ queries inside blocks?

- In-block queries needed for 1st and 3rd values to complete algorithm.
- RMQ processing on each block $\Rightarrow$ too much time in processing!

How to answer range RMQ queries inside blocks?

- In-block queries needed for 1st and 3rd values to complete algorithm.
- RMQ processing on each block $\Rightarrow$ too much time in processing!
- 2 blocks identical? $\Rightarrow$ share their processing!

How to answer range RMQ queries inside blocks?

- In-block queries needed for 1st and 3rd values to complete algorithm.
- RMQ processing on each block $\Rightarrow$ too much time in processing!
- 2 blocks identical? $\Rightarrow$ share their processing!
- Too much hope that blocks would be so repeated!:(

How to answer range RMQ queries inside blocks?

- In-block queries needed for 1st and 3rd values to complete algorithm.
- RMQ processing on each block $\Rightarrow$ too much time in processing!
- 2 blocks identical? $\Rightarrow$ share their processing!
- Too much hope that blocks would be so repeated!:(

Observation

If two arrays X[1,...,k] and Y[1,...,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 Xand Y.

How to answer range RMQ queries inside blocks?

- In-block queries needed for 1st and 3rd values to complete algorithm.
- RMQ processing on each block ⇒ too much time in processing!
- 2 blocks identical? ⇒ share their processing!
- Too much hope that blocks would be so repeated!:(

Observation

If two arrays X[1,...,k] and Y[1,...,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 Xand Y.

Cont.

- Normalize a block by subtracting its initial offset from every element.

Cont.

- Normalize a block by subtracting its initial offset from every element.
- Use the ∓ 1 property to show there very few kinds of normlized blocks:

Cont.

- Normalize a block by subtracting its initial offset from every element.
- Use the ∓ 1 property to show there very few kinds of normlized blocks:

Lemma

There are $\mathcal{O}(\sqrt{n})$ kinds of normalized blocks

Cont.

- Normalize a block by subtracting its initial offset from every element.
- Use the ∓ 1 property to show there very few kinds of normlized blocks:

Lemma

There are $\mathcal{O}(\sqrt{n})$ kinds of normalized blocks

Proof.

Adjacent elements in normalized blocks differ by $+1$ or -1. Thus, normalized blocks are specified by ∓ 1 vector of length $\frac{1}{2\log n} - 1$. There are $2^{\frac{1}{2\log n}-1}$ $=\mathcal{O}(\sqrt{n})$ such vectors ([Farach-Bender, 2000]) □

Cont.

- Normalize a block by subtracting its initial offset from every element.
- Use the ∓ 1 property to show there very few kinds of normlized blocks:

Lemma

There are $\mathcal{O}(\sqrt{n})$ kinds of normalized blocks

Proof.

Adjacent elements in normalized blocks differ by $+1$ or -1. Thus, normalized blocks are specified by ∓ 1 vector of length $\frac{1}{2\log n} - 1$. There are $2^{\frac{1}{2\log n}-1} = \mathcal{O}(\sqrt{n})$ such vectors ([Farach-Bender, 2000]) □

- We are basically done!

Cont.

- Normalize a block by subtracting its initial offset from every element.
- Use the ∓ 1 property to show there very few kinds of normlized blocks:

Lemma

There are $\mathcal{O}(\sqrt{n})$ kinds of normalized blocks

Proof.

Adjacent elements in normalized blocks differ by $+1$ or -1. Thus, normalized blocks are specified by ∓ 1 vector of length $\frac{1}{2\log n} - 1$. There are $2^{\frac{1}{2\log n}-1}$ $=\mathcal{O}(\sqrt{n})$ such vectors ([Farach-Bender, 2000]) □

- We are basically done!
- Create $\mathcal{O}(\sqrt{n})$ tables, one for each possible normalized block.

Cont.

- Normalize a block by subtracting its initial offset from every element.
- Use the ∓ 1 property to show there very few kinds of normlized blocks:

Lemma

There are $\mathcal{O}(\sqrt{n})$ kinds of normalized blocks

Proof.

Adjacent elements in normalized blocks differ by $+1$ or -1. Thus, normalized blocks are specified by ∓ 1 vector of length $\frac{1}{2 \log n} - 1$. There are $2^{\frac{1}{2 \log n} - 1} = \mathcal{O}(\sqrt{n})$ such vectors ([Farach-Bender, 2000]) □

- We are basically done!
- Create $\mathcal{O}(\sqrt{n})$ tables, one for each possible normalized block.
- A total of $\mathcal{O}(\sqrt{n}) \log^2 n$ total processing of normalized block tables and $O(1)$ query time.

Cont.

- Normalize a block by subtracting its initial offset from every element.
- Use the ∓ 1 property to show there very few kinds of normlized blocks:

Lemma

There are $\mathcal{O}(\sqrt{n})$ kinds of normalized blocks

Proof.

Adjacent elements in normalized blocks differ by $+1$ or -1. Thus, normalized blocks are specified by ∓ 1 vector of length $\frac{1}{2\log n} - 1$. There are $2^{\frac{1}{2\log n} - 1}$ $=\mathcal{O}(\sqrt{n})$ such vectors ([Farach-Bender, 2000]) □

- We are basically done!
- Create $\mathcal{O}(\sqrt{n})$ tables, one for each possible normalized block.
- A total of $\mathcal{O}(\sqrt{n})\log^2 n$ total processing of normalized block tables and $O(1)$ query time.
- Finally compute for each block in A which normalized block table it should use for its RMQ queries.

Cont.

- Normalize a block by subtracting its initial offset from every element.
- Use the ∓ 1 property to show there very few kinds of normlized blocks:

Lemma

There are $\mathcal{O}(\sqrt{n})$ kinds of normalized blocks

Proof.

Adjacent elements in normalized blocks differ by $+1$ or -1. Thus, normalized blocks are specified by ∓ 1 vector of length $\frac{1}{2\log n} - 1$. There are $2^{\frac{1}{2\log n}-1}$ $=\mathcal{O}(\sqrt{n})$ such vectors ([Farach-Bender, 2000]) □

- We are basically done!
- Create $\mathcal{O}(\sqrt{n})$ tables, one for each possible normalized block.
- A total of $\mathcal{O}(\sqrt{n})\log^2 n$ total processing of normalized block tables and $O(1)$ query time.
- Finally compute for each block in A which normalized block table it should use for its RMQ queries.

Wrapping up!

- Started by reducing from LCA problem to RMQ problem given reduction leads to ∓ 1RMQ problem.

Wrapping up!

- Started by reducing from LCA problem to RMQ problem given reduction leads to $\mp$1RMQ problem.
- Gave a trivial $\left\langle \mathcal{O}(n^2), \mathcal{O}(1) \right\rangle$ time table-lookup algorithm for RMQ and show how to sparsify the table to get $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$-time table-lookup algorithm.

Wrapping up!

- Started by reducing from LCA problem to RMQ problem given reduction leads to $\mp$1RMQ problem.
- Gave a trivial $\langle \mathcal{O}(n^2), \mathcal{O}(1) \rangle$ time table-lookup algorithm for RMQ and show how to sparsify the table to get $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$-time table-lookup algorithm.
- Used latter algorithm on a smaller summary array A and needed only to process small blocks to finish algorithm.

Wrapping up!

- Started by reducing from LCA problem to RMQ problem given reduction leads to $\mp$1RMQ problem.
- Gave a trivial $\langle \mathcal{O}(n^2), \mathcal{O}(1) \rangle$ time table-lookup algorithm for RMQ and show how to sparsify the table to get $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$-time table-lookup algorithm.
- Used latter algorithm on a smaller summary array A and needed only to process small blocks to finish algorithm.
- Finally, noticed most of these blocks are the same by using the $\mp$1 assumption from original reduction.(from RMQ problem point of view).

Wrapping up!

- Started by reducing from LCA problem to RMQ problem given reduction leads to $\mp$1RMQ problem.
- Gave a trivial $\langle \mathcal{O}(n^2), \mathcal{O}(1) \rangle$ time table-lookup algorithm for RMQ and show how to sparsify the table to get $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$-time table-lookup algorithm.
- Used latter algorithm on a smaller summary array A and needed only to process small blocks to finish algorithm.
- Finally, noticed most of these blocks are the same by using the $\mp$1 assumption from original reduction.(from RMQ problem point of view).

A Fast Algorithm for RMQ!

- We have $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle \mp$RMQ.

A Fast Algorithm for RMQ!

- We have $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle \mp$RMQ.
- General RMQ can be solved in the same complexity!

A Fast Algorithm for RMQ!

- We have $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle \mp$RMQ.
- General RMQ can be solved in the same complexity!
- By reducing RMQ problem to LCA problem again!

A Fast Algorithm for RMQ!

- We have $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ $\mp$RMQ.
- General RMQ can be solved in the same complexity!
- By reducing RMQ problem to LCA problem again!
- To solve a general RMQ problem, one would convert it to an LCA problem and then back to $\mp$1RMQ problem!

A Fast Algorithm for RMQ!

- We have $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle \mp$RMQ.
- General RMQ can be solved in the same complexity!
- By reducing RMQ problem to LCA problem again!
- To solve a general RMQ problem, one would convert it to an LCA problem and then back to $\mp$1RMQ problem!

How?

Lemma

If there is a $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ *solution for LCA, then there is a* $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ *solution for RMQ.*

How?

Lemma

If there is a $\langle\mathcal{O}(n), \mathcal{O}(1)\rangle$ *solution for LCA, then there is a* $\langle\mathcal{O}(n), \mathcal{O}(1)\rangle$ *solution for RMQ.*

- $\mathcal{O}(n)$ comes from time needed to build Cartesian Tree C of A and $\mathcal{O}(1)$ comes from time needed to convert LCA to an RMQ answer on A.

How?

Lemma

If there is a $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ solution for LCA, then there is a $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ solution for RMQ.

- $\mathcal{O}(n)$ comes from time needed to build Cartesian Tree C of A and $\mathcal{O}(1)$ comes from time needed to convert LCA to an RMQ answer on A.
- We can prove that:

$$RMQ_A(i,j) = LCA_C(i,j) \tag{4}$$

.

How?

Lemma

If there is a $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ solution for LCA, then there is a $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ solution for RMQ.

- $\mathcal{O}(n)$ comes from time needed to build Cartesian Tree C of A and $\mathcal{O}(1)$ comes from time needed to convert LCA to an RMQ answer on A.
- We can prove that:

$$RMQ_A(i,j) = LCA_C(i,j) \tag{4}$$

.

- **Reduction completed!**

How?

Lemma

If there is a $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ solution for LCA, then there is a $\langle \mathcal{O}(n), \mathcal{O}(1) \rangle$ solution for RMQ.

- $\mathcal{O}(n)$ comes from time needed to build Cartesian Tree C of A and $\mathcal{O}(1)$ comes from time needed to convert LCA to an RMQ answer on A.
- We can prove that:

$$RMQ_A(i,j) = LCA_C(i,j) \tag{4}$$

.

- **Reduction completed!**

Final Remarks

- We can solve the range-min query problem in an array of n numbers with ∓ 1 property in $\mathcal{O}(1)$ and $\mathcal{O}(n)$ space.

Final Remarks

- We can solve the range-min query problem in an array of n numbers with ∓ 1 property in $\mathcal{O}(1)$ and $\mathcal{O}(n)$ space.
- Divide array A into m=$\frac{2n}{\log n}$ buckets, each of size k=$\frac{\log n}{2}$.

Final Remarks

- We can solve the range-min query problem in an array of n numbers with ∓ 1 property in $\mathcal{O}(1)$ and $\mathcal{O}(n)$ space.
- Divide array A into m=$\frac{2n}{\log n}$ buckets, each of size k=$\frac{\log n}{2}$.
- Parallel and distributed versions for algorithm exist!

Thank you for your attention!