Union-Find

Lecture 15
October 15, 2015
15.1: Union Find
15.2: Kruskal’s algorithm – a quick reminder
Compute minimum spanning tree

1. **G**: Undirected graph with weights on edges.
2. **Q**: Compute **MST** (minimum spanning tree) of **G**.
3. **Kruskal’s Algorithm**:
   1. Sort edges by increasing weight.
   2. Start with a copy of **G** with no edges.
   3. Add edges by increasing weight, and insert into graph \( \Leftarrow \Rightarrow \) do not form a cycle.
      (i.e., connect two different things together.)
Compute minimum spanning tree

2. Q: Compute **MST** (minimum spanning tree) of G.
3. Kruskal’s Algorithm:
   1. Sort edges by increasing weight.
   2. Start with a copy of G with no edges.
   3. Add edges by increasing weight, and insert into graph ⇐⇒ do not form a cycle.
      (i.e., connect two different things together.)
Compute minimum spanning tree

1. **G**: Undirected graph with weights on edges.
2. **Q**: Compute **MST** (minimum spanning tree) of **G**.
3. **Kruskal’s Algorithm**:
   1. Sort edges by increasing weight.
   2. Start with a copy of **G** with no edges.
   3. Add edges by increasing weight, and insert into graph $\iff$ do not form a cycle.
      (i.e., connect two different things together.)
Kruskal’s Algorithm

Process edges in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Figure: Graph $G$

Figure: MST of $G$
Kruskal’s Algorithm

Process edges in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Figure: Graph $G$

Figure: MST of $G$
Kruskal’s Algorithm

Process edges in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Figure: Graph $G$

Figure: MST of $G$
Kruskal’s Algorithm

Process edges in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Figure: Graph $G$

Figure: MST of $G$
Kruskal’s Algorithm

Process edges in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Figure: Graph $G$

Figure: MST of $G$
Kruskal’s Algorithm

Process edges in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Figure: Graph $G$

Figure: MST of $G$
Kruskal’s Algorithm

Process edges in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Figure: Graph $G$

Figure: MST of $G$
Kruskal’s Algorithm

Process edges in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Figure: Graph $G$

Figure: MST of $G$
Kruskal’s Algorithm

Process edges in the order of their costs (starting from the least) and add edges to $T$ as long as they don’t form a cycle.

Figure: Graph $G$

Figure: MST of $G$
15.2.1: Requirements from the data-structure
Requirements from the data-structure

1. Maintain a collection of sets.
2. $\text{makeSet}(x)$ - creates a set that contains the single element $x$.
3. $\text{find}(x)$ - returns the set that contains $x$.
4. $\text{union}(A, B)$ - returns set = union of $A$ and $B$. That is $A \cup B$.
   ... merges the two sets $A$ and $B$ and return the merged set.
Requirements from the data-structure

1. Maintain a collection of sets.
2. \texttt{makeSet}(x) - creates a set that contains the single element \(x\).
3. \texttt{find}(x) - returns the set that contains \(x\).
4. \texttt{union}(A, B) - returns set = union of \(A\) and \(B\). That is \(A \cup B\).
   ... merges the two sets \(A\) and \(B\) and return the merged set.
Requirements from the data-structure

1. Maintain a collection of sets.
2. \texttt{makeSet}(x) - creates a set that contains the single element \( x \).
3. \texttt{find}(x) - returns the set that contains \( x \).
4. \texttt{union}(A, B) - returns set = union of \( A \) and \( B \). That is \( A \cup B \).
   ... merges the two sets \( A \) and \( B \) and return the merged set.
Requirements from the data-structure

1. Maintain a collection of sets.
2. \texttt{makeSet}(x) - creates a set that contains the single element \(x\).
3. \texttt{find}(x) - returns the set that contains \(x\).
4. \texttt{union}(A, B) - returns set = union of \(A\) and \(B\). That is \(A \cup B\).

... merges the two sets \(A\) and \(B\) and return the merged set.
Requirements from the data-structure

1. Maintain a collection of sets.
2. \texttt{makeSet}(x) - creates a set that contains the single element \(x\).
3. \texttt{find}(x) - returns the set that contains \(x\).
4. \texttt{union}(A, B) - returns set = union of \(A\) and \(B\). That is \(A \cup B\).

... merges the two sets \(A\) and \(B\) and return the merged set.
Requirements from the data-structure

1. Maintain a collection of sets.
2. \texttt{makeSet}(x) - creates a set that contains the single element \( x \).
3. \texttt{find}(x) - returns the set that contains \( x \).
4. \texttt{union}(A, B) - returns set = union of \( A \) and \( B \). That is \( A \cup B \).
   ... merges the two sets \( A \) and \( B \) and return the merged set.
15.2.2: Amortized analysis
1. Use data-structure as a black-box inside algorithm.  
   ... Union-Find in Kruskal algorithm for computing MST.

2. Bounded worst case time per operation.

3. Care: *overall* running time spend in data-structure.

4. **amortized running-time** of operation 
   \[= \text{average time to perform an operation on data-structure}.\]

5. Amortized time per operation \[= \frac{\text{overall running time}}{\text{number of operations}}.\]
Amortized Analysis

1. Use data-structure as a black-box inside algorithm. 
   ... Union-Find in Kruskal algorithm for computing MST.

2. Bounded worst case time per operation.

3. Care: overall running time spend in data-structure.

4. Amortized running-time of operation
   \[= \text{average time to perform an operation on data-structure.} \]

5. Amortized time per operation
   \[= \frac{\text{overall running time}}{\text{number of operations}}.\]
Amortized Analysis

1. Use data-structure as a black-box inside algorithm. ... Union-Find in Kruskal algorithm for computing MST.

2. Bounded worst case time per operation.

3. Care: overall running time spend in data-structure.

4. Amortized running-time of operation
   \[ = \text{average time to perform an operation on data-structure.} \]

5. Amortized time per operation
   \[ = \frac{\text{overall running time}}{\text{number of operations}}. \]
Amortized Analysis

1. Use data-structure as a black-box inside algorithm.  
   ... Union-Find in Kruskal algorithm for computing MST.

2. Bounded worst case time per operation.

3. Care: overall running time spend in data-structure.

4. **Amortized running-time** of operation  
   := average time to perform an operation on data-structure.

5. Amortized time per operation = \( \frac{\text{overall running time}}{\text{number of operations}} \).
Amortized Analysis

1. Use data-structure as a black-box inside algorithm. ... Union-Find in Kruskal algorithm for computing MST.
2. Bounded worst case time per operation.
3. Care: overall running time spend in data-structure.
4. *amortized running-time* of operation
   \[ = \text{average time to perform an operation on data-structure}. \]
5. Amortized time per operation \[ = \frac{\text{overall running time}}{\text{number of operations}}. \]
15.2.3: The data-structure
The Union-Find representation of the sets $A = \{a, b, c, d, e\}$ and $B = \{f, g, h, i, j, k\}$. The set $A$ is uniquely identified by a pointer to the root of $A$, which is the node containing $a$. 
Reversed Trees:

1. Initially: Every element is its own node.
2. Node $v$: $\overline{p}(v)$ pointer to its parent.
3. Set uniquely identified by root node/element.
Reversed Trees:

1. Initially: Every element is its own node.
2. Node $v$: $\overline{p}(v)$ pointer to its parent.
3. Set uniquely identified by root node/element.

makeSet: Create a singleton pointing to itself: $\overline{a}$
Reversed Trees

1. Reversed Trees:
   1. Initially: Every element is its own node.
   2. Node $v$: $p(v)$ pointer to its parent.
   3. Set uniquely identified by root node/element.

2. **makeSet**: Create a singleton pointing to itself:

3. **find**($x$):
   1. Start from node containing $x$, traverse up tree, till arriving to root.
   2. **find**($x$):
      
      $x \to b \to a$
   3. $a$: returned as set.
Reversed Trees

1. Reversed Trees:
   1. Initially: Every element is its own node.
   2. Node $v$: $\overline{p}(v)$ pointer to its parent.
   3. Set uniquely identified by root node/element.

2. **makeSet**: Create a singleton pointing to itself:

3. **find($x$)**:
   1. Start from node containing $x$, traverse up tree, till arriving to root.
   2. **find($x$)**:
      
      \[
      x \rightarrow b \rightarrow a
      \]
   3. $a$: returned as set.
Reversed Trees:

1. Initially: Every element is its own node.
2. Node $v$: $\overline{p}(v)$ pointer to its parent.
3. Set uniquely identified by root node/element.

**makeSet:** Create a singleton pointing to itself:

**find($x$):**

1. Start from node containing $x$, traverse up tree, till arriving to root.

2. **find($x$):**

   $x \rightarrow b \rightarrow a$

3. $a$: returned as set.
Reversed Trees:

1. Initially: Every element is its own node.
2. Node $v$: $\overline{P}(v)$ pointer to its parent.
3. Set uniquely identified by root node/element.

**makeSet**: Create a singleton pointing to itself:

**find**($x$):

1. Start from node containing $x$, traverse up tree, till arriving to root.

**find**($x$):

$x \rightarrow b \rightarrow a$

$a$: returned as set.
union\((a, p)\): Merge two sets.

1. Hanging the root of one tree, on the root of the other.
2. A destructive operation, and the two original sets no longer exist.
Pseudo-code of naive version...

\[ \text{makeSet}(x) \]
\[ \overline{p}(x) \leftarrow x \]

\[ \text{find}(x) \]
\[ \text{if } x = \overline{p}(x) \text{ then} \]
\[ \text{return } x \]
\[ \text{return } \text{find}(\overline{p}(x)) \]

\[ \text{union}(x, y) \]
\[ A \leftarrow \text{find}(x) \]
\[ B \leftarrow \text{find}(y) \]
\[ \overline{p}(B) \leftarrow A \]
Example…
The long chain

After: \texttt{makeSet(a)}, \texttt{makeSet(b)}, \texttt{makeSet(c)}, \texttt{makeSet(d)}, \\
\texttt{makeSet(e)}, \texttt{makeSet(f)}, \texttt{makeSet(g)}, \texttt{makeSet(h)}
Example…

The long chain

After:  \texttt{makeSet}(a), \texttt{makeSet}(b), \texttt{makeSet}(c), \texttt{makeSet}(d), \texttt{makeSet}(e), \texttt{makeSet}(f), \texttt{makeSet}(g), \texttt{makeSet}(h), \texttt{union}(g, h)
Example...
The long chain

After: \texttt{makeSet}(a), \texttt{makeSet}(b), \texttt{makeSet}(c), \texttt{makeSet}(d), \texttt{makeSet}(e), \texttt{makeSet}(f), \texttt{makeSet}(g), \texttt{makeSet}(h)
union(g, h)
union(f, g)
Example...
The long chain

After:  \(\text{makeSet}(a), \text{makeSet}(b), \text{makeSet}(c), \text{makeSet}(d), \text{makeSet}(e), \text{makeSet}(f), \text{makeSet}(g), \text{makeSet}(h)\)
\(\text{union}(g, h)\)
\(\text{union}(f, g)\)
\(\text{union}(e, f)\)
Example...
The long chain

After:  
\textit{makeSet}(a), \textit{makeSet}(b), \textit{makeSet}(c), \textit{makeSet}(d), \textit{makeSet}(e), \textit{makeSet}(f), \textit{makeSet}(g), \textit{makeSet}(h)
\textit{union}(g, h)
\textit{union}(f, g)
\textit{union}(e, f)
\textit{union}(d, e)
Example...
The long chain

After:  
\texttt{makeSet(a)}, \texttt{makeSet(b)}, \texttt{makeSet(c)}, \texttt{makeSet(d)}, \texttt{makeSet(e)}, \texttt{makeSet(f)}, \texttt{makeSet(g)}, \texttt{makeSet(h)}
\texttt{union(g, h)}
\texttt{union(f, g)}
\texttt{union(e, f)}
\texttt{union(d, e)}
\texttt{union(c, d)}
Example...

The long chain

After:  \texttt{makeSet}(a), \texttt{makeSet}(b), \texttt{makeSet}(c), \texttt{makeSet}(d), \texttt{makeSet}(e), \texttt{makeSet}(f), \texttt{makeSet}(g), \texttt{makeSet}(h) \\
\texttt{union}(g, h) \\
\texttt{union}(f, g) \\
\texttt{union}(e, f) \\
\texttt{union}(d, e) \\
\texttt{union}(c, d) \\
\texttt{union}(b, c)
Example...
The long chain

After: makeSet(a), makeSet(b), makeSet(c), makeSet(d), makeSet(e), makeSet(f), makeSet(g), makeSet(h)
union(g, h)
union(f, g)
union(e, f)
union(d, e)
union(c, d)
union(b, c)
union(a, b)
Find is slow, hack it!

1. \texttt{find} might require $\Omega(n)$ time.

2. Q: How improve performance?

3. Two “hacks”:
   
   (i) **Union by rank**:
   Maintain in root of tree, a bound on its depth (\texttt{rank}).
   **Rule**: Hang the smaller tree on the larger tree in \texttt{union}.

   (ii) **Path compression**:
   During find, make all pointers on path point to root.
Find is slow, hack it!

1. **find** might require $\Omega(n)$ time.

2. **Q**: How improve performance?

3. Two “hacks”:
   (i) **Union by rank**:
   Maintain in root of tree, a bound on its depth (**rank**).
   **Rule**: Hang the smaller tree on the larger tree in **union**.

   (ii) **Path compression**:
   During find, make all pointers on path point to root.
Find is slow, hack it!

1. **find** might require $\Omega(n)$ time.

2. **Q**: How improve performance?

3. Two “hacks”:

   (i) **Union by rank**: Maintain in root of tree, a bound on its depth (**rank**). **Rule**: Hang the smaller tree on the larger tree in **union**.

   (ii) **Path compression**: During find, make all pointers on path point to root.
Find is slow, hack it!

1. **find** might require $\Omega(n)$ time.

2. **Q**: How improve performance?

3. Two “hacks”:
   
   (i) **Union by rank**:
   Maintain in root of tree, a bound on its depth (**rank**). **Rule**: Hang the smaller tree on the larger tree in **union**.

   (ii) **Path compression**:
   During find, make all pointers on path point to root.
Find is slow, hack it!

1. \textbf{find} might require $\Omega(n)$ time.
2. \textbf{Q:} How improve performance?
3. Two “hacks”:
   
   (i) \textbf{Union by rank:}
   Maintain in root of tree, a bound on its depth (\textit{rank}).
   \textbf{Rule:} Hang the smaller tree on the larger tree in \textit{union}.

   (ii) \textbf{Path compression:}
   During find, make all pointers on path point to root.
Find is slow, hack it!

1. **find** might require $\Omega(n)$ time.

2. **Q**: How improve performance?

3. Two “hacks”:

   (i) **Union by rank**: Maintain in root of tree, a bound on its depth (rank). **Rule**: Hang the smaller tree on the larger tree in **union**.

   (ii) **Path compression**: During find, make all pointers on path point to root.
Find is slow, hack it!

1. **find** might require $\Omega(n)$ time.

2. **Q**: How improve performance?

3. Two “hacks”:
   
   (i) **Union by rank**: Maintain in root of tree, a bound on its depth (rank).
   
   **Rule**: Hang the smaller tree on the larger tree in union.

   (ii) **Path compression**: During find, make all pointers on path point to root.
Path compression in action...

(a) The tree before performing $\text{find}(z)$, and (b) The reversed tree after performing $\text{find}(z)$ that uses path compression.
Pseudo-code of improved version...

**makeSet**($x$)
- $\overline{p}(x) \leftarrow x$
- $\text{rank}(x) \leftarrow 0$

**find**($x$)
- if $x \neq \overline{p}(x)$ then
  - $\overline{p}(x) \leftarrow \text{find}(\overline{p}(x))$
- return $\overline{p}(x)$

**union**($x$, $y$)
- $A \leftarrow \text{find}(x)$
- $B \leftarrow \text{find}(y)$
- if $\text{rank}(A) > \text{rank}(B)$ then
  - $\overline{p}(B) \leftarrow A$
- else
  - $\overline{p}(A) \leftarrow B$
  - if $\text{rank}(A) = \text{rank}(B)$ then
    - $\text{rank}(B) \leftarrow \text{rank}(B) + 1$
15.3: Analyzing the Union-Find Data-Structure
Definition

\( \nu \): Node \textbf{UnionFind} data-structure \( \mathcal{D} \)

\( \nu \) is \textbf{leader} \iff \nu \text{ root of a (reversed) tree in } \mathcal{D}.

“\text{When you’re not leader, you’re little people.}”

“You know the score pal. If you’re not cop, you’re little people.” - Blade Runner (movie).
Definition

\( v \): Node **UnionFind** data-structure \( D \)
\( v \) is **leader** ⇐⇒ \( v \) root of a (reversed) tree in \( D \).

“When you’re not leader, you’re little people.”
“You know the score pal. If you’re not cop, you’re little people.” - Blade Runner (movie).
Once node $v$ stop being a leader, can never become leader again.

Proof.

1. $x$ stopped being leader because union operation hanged $x$ on $y$.
2. From this point on...
3. $x$ might change only its parent pointer (find).
4. $x$ parent pointer will never become equal to $x$ again.
5. $x$ never a leader again.
Lemma

Once node \( v \) stop being a leader, can never become leader again.

Proof.

1. \( x \) stopped being leader because \textit{union} operation hanged \( x \) on \( y \).
2. From this point on...
3. \( x \) might change only its parent pointer (\textit{find}).
4. \( x \) parent pointer will never become equal to \( x \) again.
5. \( x \) never a leader again.
Lemma

Once node $v$ stop being a leader, can never become leader again.

Proof.

1. $x$ stopped being leader because union operation hanged $x$ on $y$.
2. From this point on...
3. $x$ might change only its parent pointer (find).
4. $x$ parent pointer will never become equal to $x$ again.
5. $x$ never a leader again.
Once node $v$ stop being a leader, can never become leader again.

Proof.
1. $x$ stopped being leader because union operation hanged $x$ on $y$.
2. From this point on...
3. $x$ might change only its parent pointer (find).
4. $x$ parent pointer will never become equal to $x$ again.
5. $x$ never a leader again.
Lemma

Once node $v$ stop being a leader, can never become leader again.

Proof.

1. $x$ stopped being leader because `union` operation hanged $x$ on $y$.
2. From this point on...
3. $x$ might change only its parent pointer (find).
4. $x$ parent pointer will never become equal to $x$ again.
5. $x$ never a leader again.
Lemma

Once node $v$ stop being a leader, can never become leader again.

Proof.

1. $x$ stopped being leader because union operation hanged $x$ on $y$.
2. From this point on...
3. $x$ might change only its parent pointer (find).
4. $x$ parent pointer will never become equal to $x$ again.
5. $x$ never a leader again.
Another Lemma

**Lemma**

*Once a node stop being a leader then its rank is fixed.*

**Proof.**

1. Rank of element changes only by \texttt{union} operation.
2. \texttt{union} operation changes rank only for... the “new” leader of the new set.
3. If an element is no longer a leader, than its rank is fixed.
Another Lemma

Lemma

Once a node stop being a leader then its rank is fixed.

Proof.

1. rank of element changes only by union operation.
2. union operation changes rank only for the “new” leader of the new set.
3. if an element is no longer a leader, than its rank is fixed.
Lemma

Once a node stop being a leader then its rank is fixed.

Proof.

1. rank of element changes only by \textit{union} operation.

2. \textit{union} operation changes rank only for... the “new” leader of the new set.

3. if an element is no longer a leader, than its rank is fixed.
Another Lemma

Lemma

Once a node stop being a leader then its rank is fixed.

Proof.

1. rank of element changes only by \textit{union} operation.
2. \textit{union} operation changes rank only for... the “new” leader of the new set.
3. if an element is no longer a leader, than its rank is fixed.
Another Lemma

**Lemma**

*Once a node stop being a leader then its rank is fixed.*

**Proof.**

1. rank of element changes only by **union** operation.
2. **union** operation changes rank only for... the “new” leader of the new set.
3. if an element is no longer a leader, than its rank is fixed.
Ranks are strictly monotonically increasing

**Lemma**

*Ranks are monotonically increasing in the reversed trees...*  
...along a path from node to root of the tree.
Proof...

1. Claim: \( \forall u \rightarrow v \) in DS: \( \text{rank}(u) < \text{rank}(v) \).


3. Assume claim holds at time \( t \), before an operation.

4. If operation is \( \text{union}(A, B) \), and assume that we hanged \( \text{root}(A) \) on \( \text{root}(B) \).
   Must be that \( \text{rank}(\text{root}(B)) \) is now larger than \( \text{rank}(\text{root}(A)) \) (verify!).
   Claim true after operation!

5. If operation \( \text{find} \): traverse path \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \).
   By induction, \( \text{rank}(v) > \) rank of all other nodes of \( \pi \).
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Claim: $\forall u \rightarrow v$ in DS: $\text{rank}(u) < \text{rank}(v)$.


Assume claim holds at time $t$, before an operation.

If operation is $\text{union}(A, B)$, and assume that we hanged $\text{root}(A)$ on $\text{root}(B)$. Must be that $\text{rank}(\text{root}(B))$ is now larger than $\text{rank}(\text{root}(A))$ (verify!).

Claim true after operation!

If operation $\text{find}$: traverse path $\pi$, then all the nodes of $\pi$ are made to point to the last node $v$ of $\pi$.

By induction, $\text{rank}(v) > \text{rank}$ of all other nodes of $\pi$.

All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Proof...

1. Claim: $\forall u \rightarrow v$ in DS: $\text{rank}(u) < \text{rank}(v)$.


3. Assume claim holds at time $t$, before an operation.

4. If operation is $\text{union}(A, B)$, and assume that we hanged $\text{root}(A)$ on $\text{root}(B)$. Must be that $\text{rank}(\text{root}(B))$ is now larger than $\text{rank}(\text{root}(A))$ (verify!). Claim true after operation!

5. If operation $\text{find}$: traverse path $\pi$, then all the nodes of $\pi$ are made to point to the last node $v$ of $\pi$. By induction, $\text{rank}(v) > \text{rank}$ of all other nodes of $\pi$. All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Claim: \( \forall u \rightarrow v \) in DS: \( \text{rank}(u) < \text{rank}(v) \).


Assume claim holds at time \( t \), before an operation.

If operation is \texttt{union}(A, B), and assume that we hanged \root(A) on \root(B).

\textbf{Must be that} \( \text{rank}(\root(B)) \) is now larger than \( \text{rank}(\root(A)) \) (verify!).

Claim true after operation!

If operation \texttt{find}: traverse path \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \).

By induction, \( \text{rank}(v) > \text{rank} \) of all other nodes of \( \pi \).

All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Proof...

1. Claim: \( \forall u \rightarrow v \) in DS: rank\((u) < \text{rank}\((v)\).
3. Assume claim holds at time \( t \), before an operation.
4. If operation is \texttt{union}(A, B), and assume that we hanged \texttt{root}(A) on \texttt{root}(B).
   Must be that rank(root(B)) is now larger than rank(root(A)) (verify!).
   Claim true after operation!
5. If operation \texttt{find}: traverse path \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \).
   By induction, rank\((v) > \text{rank} \) of all other nodes of \( \pi \).
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Proof...

1. Claim: \( \forall u \rightarrow v \) in DS: \( \text{rank}(u) < \text{rank}(v) \).
3. Assume claim holds at time \( t \), before an operation.
4. If operation is \text{union}(A, B)\), and assume that we hanged \text{root}(A)\ on \text{root}(B). Must be that \( \text{rank}(	ext{root}(B)) \) is now larger than \( \text{rank}(	ext{root}(A)) \) (verify!). Claim true after operation!
5. If operation \text{find}: \) traverse path \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \). By induction, \( \text{rank}(v) > \) rank of all other nodes of \( \pi \). All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
1. Claim: $\forall u \rightarrow v$ in DS: $\text{rank}(u) < \text{rank}(v)$.


3. Assume claim holds at time $t$, before an operation.

4. If operation is $\text{union}(A, B)$, and assume that we hanged $\text{root}(A)$ on $\text{root}(B)$.
   Must be that $\text{rank}(\text{root}(B))$ is now larger than $\text{rank}(\text{root}(A))$ (verify!).
   Claim true after operation!

5. If operation $\text{find}$: traverse path $\pi$, then all the nodes of $\pi$ are made to point to the last node $v$ of $\pi$.
   By induction, $\text{rank}(v) > \text{rank}$ of all other nodes of $\pi$.
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Claim: \( \forall u \rightarrow v \) in DS: \( \text{rank}(u) < \text{rank}(v) \).


Assume claim holds at time \( t \), before an operation.

If operation is \( \text{union}(A, B) \), and assume that we hanged \( \text{root}(A) \) on \( \text{root}(B) \).

Must be that \( \text{rank}(\text{root}(B)) \) is now larger than \( \text{rank}(\text{root}(A)) \) (verify!).

Claim true after operation!

If operation \( \text{find} \): traverse path \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \).

By induction, \( \text{rank}(v) > \) rank of all other nodes of \( \pi \).

All the nodes that get compressed, the rank of their new parent, is larger than their own rank.
Proof...

1. Claim: \( \forall u \rightarrow v \) in DS: \( \text{rank}(u) < \text{rank}(v) \).
3. Assume claim holds at time \( t \), before an operation.
4. If operation is \texttt{union}(A, B), and assume that we hanged \texttt{root}(A) on \texttt{root}(B).
   Must be that \( \text{rank}(\text{root}(B)) \) is now larger than \( \text{rank}(\text{root}(A)) \) (verify!).
   Claim true after operation!
5. If operation \texttt{find}: traverse path \( \pi \), then all the nodes of \( \pi \) are made to point to the last node \( v \) of \( \pi \).
   By induction, \( \text{rank}(v) > \) rank of all other nodes of \( \pi \).
   All the nodes that get compressed, the rank of their new parent, is larger than their own rank.

\[\blacksquare\]
Trees grow exponentially in size with rank

**Lemma**

*When node gets rank* $k \implies$ *at least* $\geq 2^k$ *elements in its subtree.*

**Proof.**

1. Proof is by induction.
2. For $k = 0$: obvious since a singleton has a rank zero, and a single element in the set.
3. Node $u$ gets rank $k$ only if the merged two roots $u, v$ has rank $k - 1$.
4. By induction, $u$ and $v$ have $\geq 2^{k-1}$ nodes before merge.
5. Merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.
Trees grow exponentially in size with rank

Lemma

When node gets rank $k \implies$ at least $\geq 2^k$ elements in its subtree.

Proof.

1. Proof is by induction.
2. For $k = 0$: obvious since a singleton has a rank zero, and a single element in the set.
3. Node $u$ gets rank $k$ only if the merged two roots $u, v$ has rank $k - 1$.
4. By induction, $u$ and $v$ have $\geq 2^{k-1}$ nodes before merge.
5. Merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.
Trees grow exponentially in size with rank

Lemma
When node gets rank $k \Rightarrow$ at least $\geq 2^k$ elements in its subtree.

Proof.
1. Proof is by induction.
2. For $k = 0$: obvious since a singleton has a rank zero, and a single element in the set.
3. Node $u$ gets rank $k$ only if the merged two roots $u, v$ has rank $k - 1$.
4. By induction, $u$ and $v$ have $\geq 2^{k-1}$ nodes before merge.
5. Merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.
Trees grow exponentially in size with rank

Lemma

When node gets rank \( k \) \( \implies \) at least \( \geq 2^k \) elements in its subtree.

Proof.

1. Proof is by induction.

2. For \( k = 0 \): obvious since a singleton has a rank zero, and a single element in the set.

3. Node \( u \) gets rank \( k \) only if the merged two roots \( u, v \) has rank \( k - 1 \).

4. By induction, \( u \) and \( v \) have \( \geq 2^{k-1} \) nodes before merge.

5. Merged tree has \( \geq 2^{k-1} + 2^{k-1} = 2^k \) nodes.
Lemma

When node gets rank $k \implies$ at least $\geq 2^k$ elements in its subtree.

Proof.

1. Proof is by induction.
2. For $k = 0$: obvious since a singleton has a rank zero, and a single element in the set.
3. Node $u$ gets rank $k$ only if the merged two roots $u, v$ has rank $k - 1$.
4. By induction, $u$ and $v$ have $\geq 2^{k-1}$ nodes before merge.
5. Merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.
Trees grow exponentially in size with rank

Lemma

When node gets rank $k \implies$ at least $\geq 2^k$ elements in its subtree.

Proof.

1. Proof is by induction.
2. For $k = 0$: obvious since a singleton has a rank zero, and a single element in the set.
3. Node $u$ gets rank $k$ only if the merged two roots $u, v$ has rank $k - 1$.
4. By induction, $u$ and $v$ have $\geq 2^{k-1}$ nodes before merge.
5. Merged tree has $\geq 2^{k-1} + 2^{k-1} = 2^k$ nodes.
Lemma

# nodes that get assigned rank $k$ throughout execution of Union-Find DS is at most $n/2^k$.

Proof.

1. By induction. For $k = 0$ it is obvious.
2. when $v$ become of rank $k$. Charge to roots merged: $u$ and $v$.
3. Before union: $u$ and $v$ of rank $k - 1$
4. After merge: $\text{rank}(v) = k$ and $\text{rank}(u) = k - 1$.
5. $u$ no longer leader. Its rank is now fixed.
6. $u, v$ leave rank $k - 1$ $\implies$ $v$ enters rank $k$.
7. By induction: at most $n/2^{k-1}$ nodes of rank $k - 1$ created.
   $\implies$ # nodes rank $k$: $\leq (n/2^{k-1})/2 = n/2^k$. 

Sariel (UIUC)
New CS473 26
Fall 2015 26 / 38
Lemma

# nodes that get assigned rank $k$ throughout execution of Union-Find DS is at most $n/2^k$.

Proof.

1. By induction. For $k = 0$ it is obvious.
2. when $v$ become of rank $k$. Charge to roots merged: $u$ and $v$.
3. Before union: $u$ and $v$ of rank $k - 1$
4. After merge: $\text{rank}(v) = k$ and $\text{rank}(u) = k - 1$.
5. $u$ no longer leader. Its rank is now fixed.
6. $u, v$ leave rank $k - 1 \implies v$ enters rank $k$.
7. By induction: at most $n/2^{k-1}$ nodes of rank $k - 1$ created. 
   \[ \implies \# \text{nodes rank } k : \leq \left(\frac{n}{2^{k-1}}\right)/2 = n/2^k. \]
Having higher rank is rare

Lemma

# nodes that get assigned rank $k$ throughout execution of Union-Find DS is at most $n/2^k$.

Proof.

1. By induction. For $k = 0$ it is obvious.
2. when $v$ become of rank $k$. Charge to roots merged: $u$ and $v$.
3. Before union: $u$ and $v$ of rank $k - 1$
4. After merge: $\text{rank}(v) = k$ and $\text{rank}(u) = k - 1$.
5. $u$ no longer leader. Its rank is now fixed.
6. $u, v$ leave rank $k - 1$ $\implies$ $v$ enters rank $k$.
7. By induction: at most $n/2^{k-1}$ nodes of rank $k - 1$ created.
   $\implies$ # nodes rank $k$: $\leq (n/2^{k-1})/2 = n/2^k$. 
Having higher rank is rare

Lemma

# nodes that get assigned rank $k$ throughout execution of Union-Find DS is at most $n/2^k$.

Proof.

1. By induction. For $k = 0$ it is obvious.
2. when $v$ become of rank $k$. Charge to roots merged: $u$ and $v$.
3. Before union: $u$ and $v$ of rank $k - 1$
4. After merge: $\text{rank}(v) = k$ and $\text{rank}(u) = k - 1$.
5. $u$ no longer leader. Its rank is now fixed.
6. $u, v$ leave rank $k - 1 \implies v$ enters rank $k$.
7. By induction: at most $n/2^{k-1}$ nodes of rank $k - 1$ created.
   \[ \implies \# \text{nodes rank } k: \leq \left(\frac{n}{2^{k-1}}\right)/2 = \frac{n}{2^k}. \]
Having higher rank is rare

Lemma

\# nodes that get assigned rank \( k \) throughout execution of Union-Find DS is at most \( n/2^k \).

Proof.

1. By induction. For \( k = 0 \) it is obvious.
2. When \( v \) become of rank \( k \). Charge to roots merged: \( u \) and \( v \).
3. Before union: \( u \) and \( v \) of rank \( k - 1 \)
4. After merge: \( \text{rank}(v) = k \) and \( \text{rank}(u) = k - 1 \).
5. \( u \) no longer leader. Its rank is now fixed.
6. \( u, v \) leave rank \( k - 1 \) \(\implies\) \( v \) enters rank \( k \).
7. By induction: at most \( n/2^{k-1} \) nodes of rank \( k - 1 \) created. \(\implies\) \# nodes rank \( k \): \( \leq (n/2^{k-1})/2 = n/2^k \).
Lemma

# nodes that get assigned rank $k$ throughout execution of Union-Find DS is at most $n/2^k$.

Proof.

1. By induction. For $k = 0$ it is obvious.
2. when $v$ become of rank $k$. Charge to roots merged: $u$ and $v$.
3. Before union: $u$ and $v$ of rank $k − 1$
4. After merge: $\text{rank}(v) = k$ and $\text{rank}(u) = k − 1$.
5. $u$ no longer leader. Its rank is now fixed.
6. $u, v$ leave rank $k − 1 \implies v$ enters rank $k$.
7. By induction: at most $n/2^{k-1}$ nodes of rank $k − 1$ created.

$\implies$ # nodes rank $k$: $\leq (n/2^{k-1})/2 = n/2^k$. 
Having higher rank is rare

Lemma

\# nodes that get assigned rank \(k\) throughout execution of Union-Find DS is at most \(n/2^k\).

Proof.

1. By induction. For \(k = 0\) it is obvious.
2. when \(v\) become of rank \(k\). Charge to roots merged: \(u\) and \(v\).
3. Before union: \(u\) and \(v\) of rank \(k - 1\)
4. After merge: \(\text{rank}(v) = k\) and \(\text{rank}(u) = k - 1\).
5. \(u\) no longer leader. Its rank is now fixed.
6. \(u, v\) leave rank \(k - 1\) \(\implies\) \(v\) enters rank \(k\).
7. By induction: at most \(n/2^{k-1}\) nodes of rank \(k - 1\) created.
\(\implies\) \# nodes rank \(k\): \(\leq (n/2^{k-1})/2 = n/2^{k}\).
Find takes logarithmic time

Lemma

The time to perform a single find operation when we perform union by rank and path compression is $O(\log n)$ time.

Proof.

1. rank of leader $v$ of reversed tree $T$, bounds depth of $T$.
2. By previous lemma: $\max$ rank $\leq \lg n$.
3. Depth of tree is $O(\log n)$.
4. Time to perform find bounded by depth of tree.
Find takes logarithmic time

**Lemma**

*The time to perform a single **find** operation when we perform union by rank and path compression is $O(\log n)$ time.*

**Proof.**

1. rank of leader $v$ of reversed tree $T$, bounds depth of $T$.
2. By previous lemma: $\max \text{ rank} \leq \lg n$.
3. Depth of tree is $O(\log n)$.
4. Time to perform **find** bounded by depth of tree.
Find takes logarithmic time

**Lemma**

The time to perform a single find operation when we perform union by rank and path compression is $O(\log n)$ time.

**Proof.**

1. Rank of leader $v$ of reversed tree $T$, bounds depth of $T$.
2. By previous lemma: $\max \text{ rank } \leq \lg n$.
3. Depth of tree is $O(\log n)$.
4. Time to perform find bounded by depth of tree.
Find takes logarithmic time

Lemma

The time to perform a single \textbf{find} operation when we perform union by rank and path compression is $O(\log n)$ time.

Proof.

1. rank of leader $v$ of reversed tree $T$, bounds depth of $T$.
2. By previous lemma: \textbf{max} rank $\leq \lg n$.
3. Depth of tree is $O(\log n)$.
4. Time to perform \textbf{find} bounded by depth of tree.
Find takes logarithmic time

Lemma

The time to perform a single find operation when we perform union by rank and path compression is $O(\log n)$ time.

Proof.

1. rank of leader $v$ of reversed tree $T$, bounds depth of $T$.
2. By previous lemma: max rank $\leq \lg n$.
3. Depth of tree is $O(\log n)$.
4. Time to perform find bounded by depth of tree.
**log**\(^*\) in detail

1. \(\text{log}^*(n)\): number of times to take \(\log\) of number to get number smaller than two.

2. \(\log^* 2 = 1\)

3. \(\log^* 2^2 = 2\).

4. \(\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3\).

5. \(\log^* 2^{2^{2^2}} = \log^*(65536) = 4\).

6. \(\log^* 2^{2^{2^{2^2}}} = \log^* 2^{65536} = 5\).

7. \(\log^*\) is a monotone increasing function.

8. \(\beta = 2^{2^{2^2}} = 2^{65536}\): huge number

   For practical purposes, \(\log^*\) returns value \(\leq 5\).
**log\(^*\) in detail**

1. \(\log^*(n)\): number of times to take \(\log\) of number to get number smaller than two.

2. \(\log^* 2 = 1\)

3. \(\log^* 2^2 = 2\).

4. \(\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3\).

5. \(\log^* 2^{2^{2^2}} = \log^*(65536) = 4\).

6. \(\log^* 2^{2^{2^{2^2}}} = \log^* 2^{65536} = 5\).

7. \(\log^*\) is a monotone increasing function.

8. \(\beta = 2^{2^{2^{2^2}}} = 2^{65536}: \) huge number
   
   For practical purposes, \(\log^*\) returns value \(\leq 5\).
**log** in detail

1. \( \log^*(n) \): number of times to take \( \lg \) of number to get number smaller than two.

2. \( \log^* 2 = 1 \)

3. \( \log^* 2^2 = 2 \).

4. \( \log^* 2^{2^2} = 1 + \log^* (2^2) = 2 + \log^* 2 = 3 \).

5. \( \log^* 2^{2^{2^2}} = \log^*(65536) = 4 \).

6. \( \log^* 2^{2^{2^{2^2}}} = \log^* 2^{65536} = 5 \).

7. \( \log^* \) is a monotone increasing function.

8. \( \beta = 2^{2^{2^{2^2}}} = 2^{65536} \): huge number
   For practical purposes, \( \log^* \) returns value \( \leq 5 \).

For clarity, \( \log^* \) is sometimes called the "fastest growing inverse function" or "epsilon order".
**log**\(^*\) in detail

1. \(\log^*(n)\): number of times to take \(\log\) of number to get number smaller than two.

2. \(\log^* 2 = 1\)

3. \(\log^* 2^2 = 2\).

4. \(\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3\).

5. \(\log^* 2^{2^{2^2}} = \log^*(65536) = 4\).

6. \(\log^* 2^{2^{2^{2^2}}} = \log^* 2^{65536} = 5\).

7. \(\log^*\) is a monotone increasing function.

8. \(\beta = 2^{2^{2^{2^2}}} = 2^{65536}\): huge number

   For practical purposes, \(\log^*\) returns value \(\leq 5\).
**log\* in detail**

1. $\log^*(n)$: number of times to take $\log_2$ of number to get number smaller than two.

2. $\log^* 2 = 1$

3. $\log^* 2^2 = 2$.

4. $\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3$.

5. $\log^* 2^{2^{2^2}} = \log^*(65536) = 4$.

6. $\log^* 2^{2^{2^{2^2}}} = \log^* 2^{65536} = 5$.

7. $\log^*$ is a monotone increasing function.

8. $\beta = 2^{2^{2^{2^2}}} = 2^{65536}$: huge number
   For practical purposes, $\log^*$ returns value $\leq 5$. 
\[ \log^* (n) \]: number of times to take \( \log \) of number to get number smaller than two.

1. \( \log^* 2 = 1 \)
2. \( \log^* 2^2 = 2 \).
3. \( \log^* 2^{2^2} = 1 + \log^* (2^2) = 2 + \log^* 2 = 3 \).
4. \( \log^* 2^{2^{2^2}} = \log^* (65536) = 4 \).
5. \( \log^* 2^{2^{2^{2^2}}} = \log^* (2^{65536}) = 5 \).
6. \( \log^* \) is a monotone increasing function.
7. \( \beta = 2^{2^{2^{2^2}}} = 2^{65536} \): huge number
   For practical purposes, \( \log^* \) returns value \( \leq 5 \).
**log**\(^*\) in detail

1. **log**\(^*(n)\): number of times to take **lg** of number to get number smaller than two.

2. \(\log^* 2 = 1\)

3. \(\log^* 2^2 = 2\).

4. \(\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3\).

5. \(\log^* 2^{2^{2^2}} = \log^*(65536) = 4\).

6. \(\log^* 2^{2^{2^{2^2}}} = \log^* 2^{65536} = 5\).

7. **log**\(^*\) is a monotone increasing function.

8. \(\beta = 2^{2^{2^2}} = 2^{65536}\): huge number

For practical purposes, **log**\(^*\) returns value \(\leq 5\).
**log**\(^*\) in detail

1. **log**\(^*\)(n): number of times to take **lg** of number to get number smaller than two.

2. \(\log^* 2 = 1\)

3. \(\log^* 2^2 = 2\).

4. \(\log^* 2^{2^2} = 1 + \log^*(2^2) = 2 + \log^* 2 = 3\).

5. \(\log^* 2^{2^{2^2}} = \log^*(65536) = 4\).

6. \(\log^* 2^{2^{2^{2^2}}} = \log^* 2^{65536} = 5\).

7. **log**\(^*\) is a monotone increasing function.

8. \(\beta = 2^{2^{2^2}} = 2^{65536}\): huge number

   For practical purposes, **log**\(^*\) returns value \(\leq 5\).
Theorem

For a sequence of \( m \) operations over \( n \) elements, the overall running time of the **UnionFind** data-structure is \( O((n + m) \log^* n) \).

1. Intuitively: **UnionFind** data-structure takes constant time per operation...
   (unless \( n \) is larger than \( \beta \) which is unlikely).
2. Not quite correct if \( n \) sufficiently large...
Can do much better!

**Theorem**

For a sequence of $m$ operations over $n$ elements, the overall running time of the **UnionFind** data-structure is $O((n + m) \log^* n)$.

1. Intuitively: **UnionFind** data-structure takes constant time per operation...
   (unless $n$ is larger than $\beta$ which is unlikely).

2. Not quite correct if $n$ sufficiently large...
Theorem

For a sequence of \( m \) operations over \( n \) elements, the overall running time of the UnionFind data-structure is \( O((n + m) \log^* n) \).

1. Intuitively: UnionFind data-structure takes constant time per operation...
   (unless \( n \) is larger than \( \beta \) which is unlikely).
2. Not quite correct if \( n \) sufficiently large...
The tower function...

**Definition**

\[ \text{Tower}(b) = 2^{\text{Tower}(b-1)} \text{ and } \text{Tower}(0) = 1. \]

\[ \text{Tower}(i) \text{: a tower of } 2^{2^{\ldots^{2}}} \text{ of height } i. \]

Observe that \( \log^*(\text{Tower}(i)) = i. \)

**Definition**

For \( i \geq 0 \), let \( \text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)] \); that is

\[ \text{Block}(i) = [z, 2^z - 1] \text{ for } z = \text{Tower}(i - 1) + 1. \]

Also \( \text{Block}(0) = [0, 1] \). As such,

\[ \text{Block}(0) = [0, 1], \text{ Block}(1) = [2, 2], \text{ Block}(2) = [3, 4], \]
\[ \text{Block}(3) = [5, 16], \text{ Block}(4) = [17, 65536], \]
\[ \text{Block}(5) = [65537, 2^{65536}] \ldots \]
The tower function...

**Definition**

$$\text{Tower}(b) = 2^{\text{Tower}(b-1)} \text{ and } \text{Tower}(0) = 1.$$  

\text{Tower}(i): a tower of \(2^{2^{\cdots^2}}\) of height \(i\).

Observe that \(\log^*(\text{Tower}(i)) = i\).

**Definition**

For \(i \geq 0\), let \(\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]\); that is

\(\text{Block}(i) = [z, 2^z-1]\) for \(z = \text{Tower}(i - 1) + 1\).

Also \(\text{Block}(0) = [0, 1]\). As such,

\(\text{Block}(0) = [0, 1], \text{ Block}(1) = [2, 2], \text{ Block}(2) = [3, 4], \)

\(\text{Block}(3) = [5, 16], \text{ Block}(4) = [17, 65536], \)

\(\text{Block}(5) = [65537, 2^{65536}]\) ...
The tower function...

**Definition**

\[
\text{Tower}(b) = 2^{\text{Tower}(b-1)} \quad \text{and} \quad \text{Tower}(0) = 1.
\]

**Tower** \((i)\): a tower of \(2^{2^{2^{\ldots^{2}}}}\) of height \(i\).

Observe that \(\log^*(\text{Tower}(i)) = i\).

**Definition**

For \(i \geq 0\), let \(\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]\); that is

\[
\text{Block}(i) = [z, 2^{z-1}] \quad \text{for} \quad z = \text{Tower}(i - 1) + 1.
\]

Also \(\text{Block}(0) = [0, 1]\). As such,

\(\text{Block}(0) = [0, 1]\), \(\text{Block}(1) = [2, 2]\), \(\text{Block}(2) = [3, 4]\),

\(\text{Block}(3) = [5, 16]\), \(\text{Block}(4) = [17, 65536]\),

\(\text{Block}(5) = [65537, 2^{65536}]\) ...
The tower function...

**Definition**

\[ \text{Tower}(b) = 2^{\text{Tower}(b-1)} \text{ and } \text{Tower}(0) = 1. \]

**Tower(i)**: a tower of \( 2^{2^{\ldots^{2}}} \) of height \( i \).

Observe that \( \log^*(\text{Tower}(i)) = i \).

**Definition**

For \( i \geq 0 \), let \( \text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)] \); that is

\[ \text{Block}(i) = [z, 2^z - 1] \text{ for } z = \text{Tower}(i - 1) + 1. \]

Also \( \text{Block}(0) = [0, 1] \). As such,

\begin{align*}
\text{Block}(0) &= [0, 1], & \text{Block}(1) &= [2, 2], & \text{Block}(2) &= [3, 4], \\
\text{Block}(3) &= [5, 16], & \text{Block}(4) &= [17, 65536], \\
\text{Block}(5) &= [65537, 2^{65536}] & \ldots
\end{align*}
The tower function...

**Definition**

\[
\text{Tower}(b) = 2^{\text{Tower}(b-1)} \text{ and } \text{Tower}(0) = 1.
\]

\(\text{Tower}(i)\): a tower of \(2^{2^{\cdots^{2}}}\) of height \(i\).

Observe that \(\log^*(\text{Tower}(i)) = i\).

**Definition**

For \(i \geq 0\), let \(\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]\); that is

\[
\text{Block}(i) = [z, 2^z - 1]
\]

for \(z = \text{Tower}(i - 1) + 1\).

Also \(\text{Block}(0) = [0, 1]\). As such,

\(
\text{Block}(0) = [0, 1], \text{Block}(1) = [2, 2], \text{Block}(2) = [3, 4], \\
\text{Block}(3) = [5, 16], \text{Block}(4) = [17, 65536], \\
\text{Block}(5) = [65537, 2^{65536}] \ldots
\)
The tower function...

**Definition**

\[ \text{Tower}(b) = 2^{\text{Tower}(b-1)} \text{ and } \text{Tower}(0) = 1. \]

**Tower**\((i)\): a tower of \(2^{2^{\cdots^{2}}}\) of height \(i\).

Observe that \(\log^*(\text{Tower}(i)) = i\).

**Definition**

For \(i \geq 0\), let \(\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]\); that is \(\text{Block}(i) = [z, 2^z - 1]\) for \(z = \text{Tower}(i - 1) + 1\).

Also, \(\text{Block}(0) = [0, 1]\). As such,

\(\text{Block}(0) = [0, 1], \text{Block}(1) = [2, 2], \text{Block}(2) = [3, 4], \text{Block}(3) = [5, 16], \text{Block}(4) = [17, 65536], \text{Block}(5) = [65537, 2^{65536}] \ldots\)
The tower function...

**Definition**

\[ \text{Tower}(b) = 2^{\text{Tower}(b-1)} \text{ and } \text{Tower}(0) = 1. \]

\[ \text{Tower}(i) : \text{a tower of } 2^{2^{2^\ldots^2}} \text{ of height } i. \]

Observe that \( \log^*(\text{Tower}(i)) = i. \)

**Definition**

For \( i \geq 0 \), let \( \text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)] \); that is \[ \text{Block}(i) = [z, 2^z] \] for \( z = \text{Tower}(i - 1) + 1. \)

Also \( \text{Block}(0) = [0, 1] \). As such, \[ \text{Block}(0) = [0, 1], \text{ Block}(1) = [2, 2], \text{ Block}(2) = [3, 4], \text{ Block}(3) = [5, 16], \text{ Block}(4) = [17, 65536], \text{ Block}(5) = [65537, 2^{65536}] \ldots \]
The tower function...

**Definition**

\[ \text{Tower}(b) = 2^{\text{Tower}(b-1)} \text{ and } \text{Tower}(0) = 1. \]

\[ \text{Tower}(i) \]: a tower of \( 2^{2^{\cdots^2}} \) of height \( i \).

Observe that \( \log^*(\text{Tower}(i)) = i \).

**Definition**

For \( i \geq 0 \), let \( \text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)] \); that is

\[ \text{Block}(i) = [z, 2^z - 1] \text{ for } z = \text{Tower}(i - 1) + 1. \]

Also \( \text{Block}(0) = [0, 1] \). As such,

\[ \text{Block}(0) = [0, 1], \text{ Block}(1) = [2, 2], \text{ Block}(2) = [3, 4], \]

\[ \text{Block}(3) = [5, 16], \text{ Block}(4) = [17, 65536], \]

\[ \text{Block}(5) = [65537, 2^{65536}] \ldots \]
The tower function...

**Definition**

\[ \text{Tower}(b) = 2^{\text{Tower}(b-1)} \text{ and } \text{Tower}(0) = 1. \]

\[ \text{Tower}(i) : \text{ a tower of } 2^{2^{2^{\ldots^2}}} \text{ of height } i. \]

Observe that \( \log^*(\text{Tower}(i)) = i. \)

**Definition**

For \( i \geq 0 \), let \( \text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)] \); that is

\[ \text{Block}(i) = [z, 2^z-1] \text{ for } z = \text{Tower}(i - 1) + 1. \]

Also \( \text{Block}(0) = [0, 1] \). As such,

\[ \text{Block}(0) = [0, 1], \text{ Block}(1) = [2, 2], \text{ Block}(2) = [3, 4], \]
\[ \text{Block}(3) = [5, 16], \text{ Block}(4) = [17, 65536], \]
\[ \text{Block}(5) = [65537, 2^{65536}] \ldots \]
Running time of find...

1. RT of \textbf{find}(x) proportional to length of the path from \( x \) to the root of its tree.

2. ...start from \( x \) and we visit the sequence:
   \[ x_1 = x, \ x_2 = \overline{p}(x_1), \ x_3 = \overline{p}(x_2), \ldots, \ x_i = \overline{p}(x_{i-1}), \ldots, \ x_m = \overline{p}(x_{m-1}) = \text{root of tree}. \]

3. \( \text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m). \)

4. RT of \textbf{find}(x) is \( O(m) \).

\begin{definition}
A node \( x \) is \textbf{in the \textit{i}th block} if \( \text{rank}(x) \in \text{Block}(i) \).
\end{definition}

5. Looking for ways to pay for the \textbf{find} operation.

6. Since other two operations take constant time...
Running time of find...

1. RT of $\text{find}(x)$ proportional to length of the path from $x$ to the root of its tree.

2. ...start from $x$ and we visit the sequence:
   
   \[
   x_1 = x, \quad x_2 = \overline{p}(x_1), \quad x_3 = \overline{p}(x_2), \ldots, \quad x_i = \overline{p}(x_{i-1}),
   \]
   
   
   \[
   \ldots, \quad x_m = \overline{p}(x_{m-1}) = \text{root of tree}.
   \]

3. $\text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m)$.

4. RT of $\text{find}(x)$ is $O(m)$.

Definition

A node $x$ is in the $i$th block if $\text{rank}(x) \in \text{Block}(i)$.

5. Looking for ways to pay for the $\text{find}$ operation.

6. Since other two operations take constant time...
Running time of \texttt{find}...

1. RT of \texttt{find}(x) proportional to length of the path from \textit{x} to the root of its tree.

2. ...start from \textit{x} and we visit the sequence:

\[ x_1 = x, \quad x_2 = \overline{p}(x_1), \quad x_3 = \overline{p}(x_2), \ldots, \quad x_i = \overline{p}(x_{i-1}), \]

\[ \ldots, \quad x_m = \overline{p}(x_{m-1}) = \text{root of tree}. \]

3. \( \text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m). \)

4. RT of \texttt{find}(x) is \( O(m) \).

\textbf{Definition}

A node \textit{x} is \textbf{in the \textit{i}th block} if \( \text{rank}(x) \in \text{Block}(i) \).

5. Looking for ways to pay for the \texttt{find} operation.

6. Since other two operations take constant time...
Running time of find...

1. RT of $\textbf{find}(x)$ proportional to length of the path from $x$ to the root of its tree.

2. ...start from $x$ and we visit the sequence:
   \[ x_1 = x, \ x_2 = \overline{p}(x_1), \ x_3 = \overline{p}(x_2), \ldots, \ x_i = \overline{p}(x_{i-1}), \ldots, \ x_m = \overline{p}(x_{m-1}) = \text{root of tree}. \]

3. rank($x_1$) $<$ rank($x_2$) $<$ rank($x_3$) $<$ $\ldots$ $<$ rank($x_m$).

4. RT of $\textbf{find}(x)$ is $O(m)$.

**Definition**

A node $x$ is in the $i$th block if rank($x$) $\in$ Block($i$).

5. Looking for ways to pay for the $\textbf{find}$ operation.

6. Since other two operations take constant time...
1. RT of \textbf{find}(x) proportional to length of the path from \textit{x} to the root of its tree.

2. ...start from \textit{x} and we visit the sequence:
   \[x_1 = x, \ x_2 = \overline{p}(x_1), \ x_3 = \overline{p}(x_2), \ldots, \ x_i = \overline{p}(x_{i-1}), \]
   \[\ldots, \ x_m = \overline{p}(x_{m-1}) = \text{root of tree}.\]

3. \text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m).

4. RT of \textbf{find}(x) is \(O(m)\).

\textbf{Definition}

A node \(x\) is \underline{in the \textit{i}th block} if \(\text{rank}(x) \in \text{Block}(i)\).

5. Looking for ways to pay for the \textbf{find} operation.

6. Since other two operations take constant time...
1. RT of \textbf{find}(x) proportional to length of the path from \textit{x} to the root of its tree.

2. ...start from \textit{x} and we visit the sequence:
\[ x_1 = x, \ x_2 = \overline{p}(x_1), \ x_3 = \overline{p}(x_2), \ldots, \ x_i = \overline{p}(x_{i-1}), \]
\[ \ldots, \ x_m = \overline{p}(x_{m-1}) = \text{root of tree}. \]

3. \(\text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m)\).

4. RT of \textbf{find}(x) is \(O(m)\).

**Definition**

A node \textit{x} is \textbf{in the \textit{i}th block} if \text{rank}(x) \in \text{Block}(i).

5. Looking for ways to pay for the \textbf{find} operation.

6. Since other two operations take constant time...
1. RT of $\text{find}(x)$ proportional to length of the path from $x$ to the root of its tree.

2. ...start from $x$ and we visit the sequence:
   
   $x_1 = x$, $x_2 = \overline{p}(x_1)$, $x_3 = \overline{p}(x_2)$, $\ldots$, $x_i = \overline{p}(x_{i-1})$, $\ldots$, $x_m = \overline{p}(x_{m-1}) = \text{root of tree}$.

3. $\text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m)$.

4. RT of $\text{find}(x)$ is $O(m)$.

**Definition**

A node $x$ is **in the $i$th block** if $\text{rank}(x) \in \text{Block}(i)$.

5. Looking for ways to pay for the $\text{find}$ operation.

6. Since other two operations take constant time...
Running time of find...

1. RT of $\text{find}(x)$ proportional to length of the path from $x$ to the root of its tree.

2. ...start from $x$ and we visit the sequence:
   
   $x_1 = x$, $x_2 = \overrightarrow{p}(x_1)$, $x_3 = \overrightarrow{p}(x_2)$, \ldots, $x_i = \overrightarrow{p}(x_{i-1})$,
   
   \ldots, $x_m = \overrightarrow{p}(x_{m-1}) = \text{root of tree}$.

3. $\text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m)$.

4. RT of $\text{find}(x)$ is $O(m)$.

Definition

A node $x$ is in the $i$th block if $\text{rank}(x) \in \text{Block}(i)$.

5. Looking for ways to pay for the $\text{find}$ operation.

6. Since other two operations take constant time...
Running time of `find`...

1. RT of `find(x)` proportional to length of the path from `x` to the root of its tree.

2. ...start from `x` and we visit the sequence:
   \[ x_1 = x, x_2 = \overline{p}(x_1), x_3 = \overline{p}(x_2), \ldots, x_i = \overline{p}(x_{i-1}), \ldots, x_m = \overline{p}(x_{m-1}) = \text{root of tree}. \]

3. \( \text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m). \)

4. RT of `find(x)` is \( O(m) \).

**Definition**

A node `x` is **in the `i`th block** if \( \text{rank}(x) \in \text{Block}(i) \).

5. Looking for ways to pay for the `find` operation.

6. Since other two operations take constant time...
Running time of find...

1. RT of \texttt{find}(x) proportional to length of the path from \textit{x} to the root of its tree.
2. ...start from \textit{x} and we visit the sequence:
   
   \[x_1 = x, \ x_2 = \overline{p}(x_1), \ x_3 = \overline{p}(x_2), \ldots, \ x_i = \overline{p}(x_{i-1}), \]
   
   \[\ldots, \ x_m = \overline{p}(x_{m-1}) = \text{root of tree}.\]
3. \(\text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m)\).
4. RT of \texttt{find}(x) is \(O(m)\).

**Definition**

A node \(x\) is \textbf{in the \textit{i}th block} if \(\text{rank}(x) \in \text{Block}(i)\).

5. Looking for ways to pay for the \texttt{find} operation.
6. Since other two operations take constant time...
Running time of find...

1. RT of $\text{find}(x)$ proportional to length of the path from $x$ to the root of its tree.

2. ...start from $x$ and we visit the sequence:
   
   \[ x_1 = x, \ x_2 = \text{p}(x_1), \ x_3 = \text{p}(x_2), \ldots, \ x_i = \text{p}(x_{i-1}), \ldots, \ x_m = \text{p}(x_{m-1}) = \text{root of tree}. \]

3. $\text{rank}(x_1) < \text{rank}(x_2) < \text{rank}(x_3) < \ldots < \text{rank}(x_m)$.

4. RT of $\text{find}(x)$ is $O(m)$.

Definition

A node $x$ is **in the $i$th block** if $\text{rank}(x) \in \text{Block}(i)$.

5. Looking for ways to pay for the $\text{find}$ operation.

6. Since other two operations take constant time...
Blocks and jumping pointers

1. maximum rank of node $v$ is $O(\log n)$.
2. # of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant, say 2).
3. $\text{find}(x)$: $\pi$ path used.
4. partition $\pi$ into each by rank.
5. Price of $\text{find}$ length $\pi$.
6. node $x$: $\nu = \text{index}_B(x)$ index block containing $\text{rank}(x)$.
7. $\text{rank}(x) \in \text{Block}(\text{index}_B(x))$.
8. $\text{index}_B(x)$: block of $x$
Blocks and jumping pointers

1. maximum rank of node $v$ is $O(\log n)$.

2. The number of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant, say 2).

3. **find** $(x)$: $\pi$ path used.

4. Partition $\pi$ into each by rank.

5. Price of **find** length $\pi$.

6. Node $x$: $v = \text{index}_B(x)$ index block containing $\text{rank}(x)$.

7. $\text{rank}(x) \in \text{Block}\left(\text{index}_B(x)\right)$.

8. $\text{index}_B(x)$: **block of** $x$.
Blocks and jumping pointers

1. maximum rank of node $v$ is $O(\log n)$.
2. # of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant, say 2).
3. **find** ($x$): $\pi$ path used.
4. partition $\pi$ into each by rank.
5. Price of **find** length $\pi$.
6. node $x$: $v = \text{index}_B(x)$ index block containing $\text{rank}(x)$.
7. $\text{rank}(x) \in \text{Block}(\text{index}_B(x))$.
8. $\text{index}_B(x)$: **block of** $x$
Blocks and jumping pointers

1. maximum rank of node $v$ is $O(\log n)$.

2. # of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant, say 2).

3. **find** ($x$): $\pi$ path used.

4. partition $\pi$ into each by rank.

5. Price of **find** length $\pi$.

6. node $x$: $v = \text{index}_B(x)$ index block containing $\text{rank}(x)$.

7. $\text{rank}(x) \in \text{Block}(\text{index}_B(x))$.

8. $\text{index}_B(x)$: block of $x$
Blocks and jumping pointers

1. maximum rank of node $v$ is $O(\log n)$.

2. $\#$ of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant, say 2).

3. **find** ($x$): $\pi$ path used.

4. partition $\pi$ into each by rank.

5. Price of **find** length $\pi$.

6. node $x$: $\nu = \text{index}_B(x)$ index block containing $\text{rank}(x)$.

7. $\text{rank}(x) \in \text{Block}(\text{index}_B(x))$.

8. $\text{index}_B(x)$: block of $x$
maximum rank of node $v$ is $O(\log n)$.

2. # of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant, say 2).

3. **find** $(x)$: $\pi$ path used.

4. partition $\pi$ into each by rank.

5. Price of **find** length $\pi$.

6. node $x$: $\nu = \text{index}_B(x)$ index block containing $\text{rank}(x)$.

7. $\text{rank}(x) \in \text{Block}\left(\text{index}_B(x)\right)$.

8. $\text{index}_B(x)$: block of $x$
Blocks and jumping pointers

1. maximum rank of node $v$ is $O(\log n)$.
2. # of blocks is $O(\log^* n)$, as $O(\log n) \in \text{Block}(c \log^* n)$, ($c$: constant, say 2).
3. find $(x)$: $\pi$ path used.
4. partition $\pi$ into each by rank.
5. Price of find length $\pi$.
6. node $x$: $\nu = \text{index}_B(x)$ index block containing rank$(x)$.
7. rank$(x) \in \text{Block}(\text{index}_B(x))$.
8. index$_B(x)$: block of $x$
The path of find operation, and its pointers

Block(0) → Block(1) → Block(1...4) → Block(5) → Block(6...7) → Block(8) → Block(9) → Block(10)
During a **find** operation...

$\pi$: path traversed.

Ranks of the nodes visited in $\pi$ monotone increasing.

Once leave block $i$th, never go back!

charge visit to nodes in $\pi$ next to element in a different block...

to total number of blocks $\leq O(\log^* n)$. 
The pointers between blocks...

1. During a **find** operation...
2. $\pi$: path traversed.
3. Ranks of the nodes visited in $\pi$ monotone increasing.
4. Once leave block $i$th, never go back!
5. charge visit to nodes in $\pi$ next to element in a different block...
6. to total number of blocks $\leq O(\log^* n)$. 
The pointers between blocks...

1. During a **find** operation...
2. \( \pi \): path traversed.
3. Ranks of the nodes visited in \( \pi \) monotone increasing.
4. Once leave block \( i \)th, never go back!
5. charge visit to nodes in \( \pi \) next to element in a different block...
6. to total number of blocks \( \leq O(\log^* n) \).
The pointers between blocks...

1. During a **find** operation...
2. \(\pi\): path traversed.
3. Ranks of the nodes visited in \(\pi\) monotone increasing.
4. Once leave block \(i\)th, never go back!
5. charge visit to nodes in \(\pi\) next to element in a different block...
6. to total number of blocks \(\leq O(\log^* n)\).
Jumping pointers

**Definition**

- $\pi$: path traversed by `find`.

1. If for $x \in \pi$, the node $p(x)$ is in a different block than $x$, then $x \rightarrow p(x)$ is a **jump between blocks**.

2. Jump inside a block is an **internal jump** (i.e., $x$ and $p(x)$ are in same block).
Not too many jumps between blocks

**Lemma**

During a single \textbf{find}(x) operation, the number of jumps between blocks along the search path is \(O(\log^* n)\).

**Proof.**

1. \(\pi = x_1, \ldots, x_m\): path followed by \textbf{find}.
2. \(x_i = \overline{p}(x_i^-)\), for all \(i\).
3. \(0 \leq \text{index}_B(x_1) \leq \text{index}_B(x_2) \leq \ldots \leq \text{index}_B(x_m)\).
4. \(\text{index}_B(x_m) = O(\log^* n)\).
5. Number of elements in \(\pi\) such that \(\text{index}_B(x) \neq \text{index}_B(\overline{p}(x))\)...
6. ... at most \(O(\log^* n)\).
Not too many jumps between blocks

Lemma

*During a single* \( \text{find}(x) \) *operation, the number of jumps between blocks along the search path is* \( O(\log^* n) \).*

Proof.

1. \( \pi = x_1, \ldots, x_m \): path followed by \( \text{find} \).
2. \( x_i = \overline{p}(x_{i-}) \), for all \( i \).
3. \( 0 \leq \text{index}_B(x_1) \leq \text{index}_B(x_2) \leq \ldots \leq \text{index}_B(x_m) \).
4. \( \text{index}_B(x_m) = O(\log^* n) \).
5. Number of elements in \( \pi \) such that \( \text{index}_B(x) \neq \text{index}_B(\overline{p}(x)) \)
6. \( \ldots \) at most \( O(\log^* n) \).
Lemma

During a single `find(x)` operation, the number of jumps between blocks along the search path is $O(\log^* n)$.

Proof.

1. $\pi = x_1, \ldots, x_m$: path followed by `find`.
2. $x_i = \overline{p}(x_{i-1})$, for all $i$.
3. $0 \leq \text{index}_B(x_1) \leq \text{index}_B(x_2) \leq \ldots \leq \text{index}_B(x_m)$.
4. $\text{index}_B(x_m) = O(\log^* n)$.
5. Number of elements in $\pi$ such that $\text{index}_B(x) \neq \text{index}_B(\overline{p}(x))$...
6. ... at most $O(\log^* n)$. 
Not too many jumps between blocks

Lemma
During a single \texttt{find}(x) operation, the number of jumps between blocks along the search path is $O(\log^* n)$.

Proof.

1. $\pi = x_1, \ldots, x_m$: path followed by \texttt{find}.
2. $x_i = \overline{p}(x_i-)$, for all $i$.
3. $0 \leq \text{index}_B(x_1) \leq \text{index}_B(x_2) \leq \ldots \leq \text{index}_B(x_m)$.
4. $\text{index}_B(x_m) = O(\log^* n)$.
5. Number of elements in $\pi$ such that $\text{index}_B(x) \neq \text{index}_B(\overline{p}(x))$...
6. ... at most $O(\log^* n)$. 
Not too many jumps between blocks

Lemma

During a single \texttt{find}(x) operation, the number of jumps between blocks along the search path is $O(\log^* n)$.

Proof.

1. $\pi = x_1, \ldots, x_m$: path followed by \texttt{find}.
2. $x_i = \overline{p}(x_{i-})$, for all $i$.
3. $0 \leq \text{index}_B(x_1) \leq \text{index}_B(x_2) \leq \ldots \leq \text{index}_B(x_m)$.
4. $\text{index}_B(x_m) = O(\log^* n)$.
5. Number of elements in $\pi$ such that $\text{index}_B(x) \neq \text{index}_B(\overline{p}(x))$...
6. ... at most $O(\log^* n)$. 
Not too many jumps between blocks

Lemma

During a single $\text{find}(x)$ operation, the number of jumps between blocks along the search path is $O(\log^* n)$.

Proof.

1. $\pi = x_1, \ldots, x_m$: path followed by $\text{find}$.
2. $x_i = \bar{p}(x_{i-})$, for all $i$.
3. $0 \leq \text{index}_B(x_1) \leq \text{index}_B(x_2) \leq \ldots \leq \text{index}_B(x_m)$.
4. $\text{index}_B(x_m) = O(\log^* n)$.
5. Number of elements in $\pi$ such that $\text{index}_B(x) \neq \text{index}_B(\bar{p}(x))$...
6. ... at most $O(\log^* n)$. 
Not too many jumps between blocks

**Lemma**

*During a single find\( (x)\) operation, the number of jumps between blocks along the search path is \(O(\log^* n)\).*

**Proof.**

1. \(\pi = x_1, \ldots, x_m\): path followed by find.
2. \(x_i = \bar{p}(x_{i-})\), for all \(i\).
3. \(0 \leq \text{index}_B(x_1) \leq \text{index}_B(x_2) \leq \ldots \leq \text{index}_B(x_m)\).
4. \(\text{index}_B(x_m) = O(\log^* n)\).
5. Number of elements in \(\pi\) such that \(\text{index}_B(x) \neq \text{index}_B(\bar{p}(x))\)...
6. ... at most \(O(\log^* n)\).
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in same block.
2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.
3. \textit{find} passes through $x$.
4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before \textit{find} operation.
5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after \textit{find} operation.
6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
7. $\implies$ parent pointer $x$ jumped forward...
8. ...and new parent has higher rank.
9. Charge internal block jumps to this “progress”.
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in same block.
2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.
3. $\textbf{find}$ passes through $x$.
4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before $\textbf{find}$ operation.
5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after $\textbf{find}$ operation.
6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
7. $\implies$ parent pointer $x$ jumped forward...
8. ...and new parent has higher rank.
9. Charge internal block jumps to this “progress”.
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in same block.
2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.
3. `find` passes through $x$.
4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before `find` operation.
5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after `find` operation.
6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
7. $\rightarrow$ parent pointer $x$ jumped forward...
8. ...and new parent has higher rank.
9. Charge internal block jumps to this “progress”.


Benefits of an internal jump

1. \( x \) and \( \overline{p}(x) \) are in same block.
2. \( \text{index}_B(x) = \text{index}_B(\overline{p}(x)) \).
3. \textbf{find} passes through \( x \).
4. \( r_{\text{bef}} = \text{rank}(\overline{p}(x)) \) before \textbf{find} operation.
5. \( r_{\text{aft}} = \text{rank}(\overline{p}(x)) \) after \textbf{find} operation.
6. By path compression: \( r_{\text{aft}} > r_{\text{bef}} \).
7. \( \rightarrow \) parent pointer \( x \) jumped forward...
8. ...and new parent has higher rank.
9. Charge internal block jumps to this “progress”. 

Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in same block.
2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.
3. \textbf{find} passes through $x$.
4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before \textbf{find} operation.
5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after \textbf{find} operation.
6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
7. $\rightarrow$ parent pointer $x$ jumped forward...
8. ...and new parent has higher rank.
9. Charge internal block jumps to this “progress”.
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in the same block.
2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.
3. \texttt{find} passes through $x$.
4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before \texttt{find} operation.
5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after \texttt{find} operation.
6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
7. $\implies$ parent pointer $x$ jumped forward...
8. ...and new parent has a higher rank.
9. Charge internal block jumps to this “progress”.
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in same block.
2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.
3. \textbf{find} passes through $x$.
4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before \textbf{find} operation.
5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after \textbf{find} operation.
6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
7. $\implies$ parent pointer $x$ jumped forward...
8. ...and new parent has higher rank.
9. Charge internal block jumps to this “progress”.

Sariel (UIUC) New CS473 Fall 2015
Benefits of an internal jump

1. $x$ and $\overline{p}(x)$ are in same block.
2. $\text{index}_B(x) = \text{index}_B(\overline{p}(x))$.
3. find passes through $x$.
4. $r_{\text{bef}} = \text{rank}(\overline{p}(x))$ before find operation.
5. $r_{\text{aft}} = \text{rank}(\overline{p}(x))$ after find operation.
6. By path compression: $r_{\text{aft}} > r_{\text{bef}}$.
7. $\implies$ parent pointer $x$ jumped forward...
8. ...and new parent has higher rank.
9. Charge internal block jumps to this “progress”.
Changing parents...

Your parent can be promoted only a few times before leaving block

Lemma

At most $|\text{Block}(i)| \leq \text{Tower}(i)$ find operations can pass through an element $x$, which is in the $i$th block (i.e., $\text{index}_B(x) = i$) before $p(x)$ is no longer in the $i$th block. That is $\text{index}_B(p(x)) > i$.

Proof.

1. parent of $x$ incr rank every-time internal jump goes through $x$.
2. At most $|\text{Block}(i)|$ different values in the $i$th block.
3. $\text{Block}(i) = \lfloor \text{Tower}(i-1) + 1, \text{Tower}(i) \rfloor$
4. Claim follows, as: $|\text{Block}(i)| \leq \text{Tower}(i)$. 
Changing parents...
Your parent can be promoted only a few times before leaving block

Lemma

At most $|\text{Block}(i)| \leq \text{Tower}(i)$ find operations can pass through an element $x$, which is in the $i$th block (i.e., $\text{index}_B(x) = i$) before $\overline{p}(x)$ is no longer in the $i$th block. That is $\text{index}_B(\overline{p}(x)) > i$.

Proof.

1. parent of $x$ incr rank every-time internal jump goes through $x$.
2. At most $|\text{Block}(i)|$ different values in the $i$th block.
3. $\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]$
4. Claim follows, as: $|\text{Block}(i)| \leq \text{Tower}(i)$. 

}\]
Changing parents...

Your parent can be promoted only a few times before leaving block

Lemma

At most $|\text{Block}(i)| \leq \text{Tower}(i)$ find operations can pass through an element $x$, which is in the $i$th block (i.e., $\text{index}_B(x) = i$) before $\overline{p}(x)$ is no longer in the $i$th block. That is $\text{index}_B(\overline{p}(x)) > i$.

Proof.

1. parent of $x$ incr rank every-time internal jump goes through $x$.
2. At most $|\text{Block}(i)|$ different values in the $i$th block.
3. $\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]$
4. Claim follows, as: $|\text{Block}(i)| \leq \text{Tower}(i)$. 
Changing parents...
Your parent can be promoted only a few times before leaving block

Lemma

At most $|\text{Block}(i)| \leq \text{Tower}(i)$ find operations can pass through an element $x$, which is in the $i$th block (i.e., $\text{index}_B(x) = i$) before $\overline{p}(x)$ is no longer in the $i$th block. That is $\text{index}_B(\overline{p}(x)) > i$.

Proof.

1. parent of $x$ incr rank every-time internal jump goes through $x$.
2. At most $|\text{Block}(i)|$ different values in the $i$th block.
3. $\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]$
4. Claim follows, as: $|\text{Block}(i)| \leq \text{Tower}(i)$. 
Changing parents...
Your parent can be promoted only a few times before leaving block

Lemma

At most \(|\text{Block}(i)| \leq \text{Tower}(i)\) find operations can pass through an element \(x\), which is in the \(i\)th block (i.e., \(\text{index}_B(x) = i\)) before \(p(x)\) is no longer in the \(i\)th block. That is \(\text{index}_B(p(x)) > i\).

Proof.

1. parent of \(x\) incr rank every-time internal jump goes through \(x\).
2. At most \(|\text{Block}(i)|\) different values in the \(i\)th block.
3. \(\text{Block}(i) = [\text{Tower}(i - 1) + 1, \text{Tower}(i)]\)
4. Claim follows, as: \(|\text{Block}(i)| \leq \text{Tower}(i)|

Sariel (UIUC)  New CS473  38  Fall 2015  38 / 38
Few elements are in the bigger blocks

Lemma

At most $\frac{n}{\text{Tower}(i)}$ nodes are assigned ranks in the $i$th block throughout the algorithm execution.

Proof.

By lemma, the number of elements with rank in the $i$th block
Few elements are in the bigger blocks

Lemma

At most $n / \text{Tower}(i)$ nodes are assigned ranks in the $i$th block throughout the algorithm execution.

Proof.

By lemma, the number of elements with rank in the $i$th block

$$\leq \sum_{k \in \text{Block}(i)} \frac{n}{2^k}$$
Few elements are in the bigger blocks

Lemma

At most $n / \text{Tower}(i)$ nodes are assigned ranks in the $i$th block throughout the algorithm execution.

Proof.

By lemma, the number of elements with rank in the $i$th block

$$\leq \sum_{k \in \text{Block}(i)} \frac{n}{2^k} = \sum_{k = \text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{n}{2^k}$$
Few elements are in the bigger blocks

Lemma
At most \( \frac{n}{\text{Tower}(i)} \) nodes are assigned ranks in the \( i \)th block throughout the algorithm execution.

Proof.
By lemma, the number of elements with rank in the \( i \)th block

\[
\leq \sum_{k \in \text{Block}(i)} \frac{n}{2^k} = \sum_{k=\text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{n}{2^k} = n \cdot \sum_{k=\text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{1}{2^k}
\]
Few elements are in the bigger blocks

**Lemma**

At most \( \frac{n}{\text{Tower}(i)} \) nodes are assigned ranks in the \( i \)th block throughout the algorithm execution.

**Proof.**

By lemma, the number of elements with rank in the \( i \)th block

\[
\leq \sum_{k \in \text{Block}(i)} \frac{n}{2^k} = \sum_{k = \text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{n}{2^k} \\
= n \cdot \sum_{k = \text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{1}{2^k} \leq \frac{n}{2^{\text{Tower}(i-1)}}
\]
Few elements are in the bigger blocks

**Lemma**

At most $\frac{n}{\text{Tower}(i)}$ nodes are assigned ranks in the $i$th block throughout the algorithm execution.

**Proof.**

By lemma, the number of elements with rank in the $i$th block

$$\leq \sum_{k=\text{Block}(i)} \frac{n}{2^k} = \sum_{k=\text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{n}{2^k}$$

$$= n \cdot \sum_{k=\text{Tower}(i-1)+1}^{\text{Tower}(i)} \frac{1}{2^k} \leq \frac{n}{2^{\text{Tower}(i-1)}} = \frac{n}{\text{Tower}(i)}.$$
Total number of internal jumps is $O(n)$

**Lemma**

The number of internal jumps performed, inside the $i$th block, during the lifetime of the union-find data-structure is $O(n)$.

**Proof.**

1. $x$ in $i$th block, have at most $|\text{Block}(i)|$ internal jumps...
2. ... after that all jumps through $x$ are between blocks, by lemma...
3. $\leq n/\text{Tower}(i)$ elements assigned ranks in the $i$th block, throughout algorithm execution.
4. total number of internal jumps is $|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n.$
Total number of internal jumps is $O(n)$

Lemma

The number of internal jumps performed, inside the $i$th block, during the lifetime of the union-find data-structure is $O(n)$.

Proof.

1. $x$ in $i$th block, have at most $|\text{Block}(i)|$ internal jumps...
2. ... after that all jumps through $x$ are between blocks, by lemma...
3. $\leq \frac{n}{\text{Tower}(i)}$ elements assigned ranks in the $i$th block, throughout algorithm execution.
4. total number of internal jumps is $|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n.$
Total number of internal jumps is $O(n)$

Lemma

The number of internal jumps performed, inside the $i$th block, during the lifetime of the union-find data-structure is $O(n)$.

Proof.

1. $x$ in $i$th block, have at most $|\text{Block}(i)|$ internal jumps...

2. ... after that all jumps through $x$ are between blocks, by lemma...

3. $\leq n/\text{Tower}(i)$ elements assigned ranks in the $i$th block, throughout algorithm execution.

4. total number of internal jumps is $|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n$. 
Total number of internal jumps is $\mathcal{O}(n)$

**Lemma**

The number of internal jumps performed, inside the $i$th block, during the lifetime of the union-find data-structure is $\mathcal{O}(n)$.

**Proof.**

1. $x$ in $i$th block, have at most $|\text{Block}(i)|$ internal jumps...
2. ... after that all jumps through $x$ are between blocks, by lemma...
3. $\leq \frac{n}{\text{Tower}(i)}$ elements assigned ranks in the $i$th block, throughout algorithm execution.
4. total number of internal jumps is

$$|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n.$$
Total number of internal jumps is $O(n)$

**Lemma**

The number of internal jumps performed, inside the $i$th block, during the lifetime of the union-find data-structure is $O(n)$.

**Proof.**

1. $x$ in $i$th block, have at most $|\text{Block}(i)|$ internal jumps...
2. ... after that all jumps through $x$ are between blocks, by lemma...
3. $\leq n / \text{Tower}(i)$ elements assigned ranks in the $i$th block, throughout algorithm execution.
4. total number of internal jumps is 
$$|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n.$$
Total number of internal jumps is $O(n)$

**Lemma**

The number of internal jumps performed, inside the $i$th block, during the lifetime of the union-find data-structure is $O(n)$.

**Proof.**

1. $x$ in $i$th block, have at most $|\text{Block}(i)|$ internal jumps...
2. ... after that all jumps through $x$ are between blocks, by lemma...
3. $\leq n/Tower(i)$ elements assigned ranks in the $i$th block, throughout algorithm execution.
4. total number of internal jumps is $|\text{Block}(i)| \cdot \frac{n}{\text{Tower}(i)} \leq \text{Tower}(i) \cdot \frac{n}{\text{Tower}(i)} = n$. 
Total number of internal jumps

**Lemma**

*The number of internal jumps performed by the Union-Find data-structure overall is $O(n \log^* n)$.*

**Proof.**

1. Every internal jump associated with block it is in.
2. Every block contributes $O(n)$ internal jumps throughout time. (By previous lemma.)
3. There are $O(\log^* n)$ blocks.
4. There are at most $O(n \log^* n)$ internal jumps.
Total number of internal jumps

Lemma

The number of internal jumps performed by the Union-Find data-structure overall is $O(n \log^* n)$.

Proof.

1. Every internal jump associated with block it is in.
2. Every block contributes $O(n)$ internal jumps throughout time. (By previous lemma.)
3. There are $O(\log^* n)$ blocks.
4. There are at most $O(n \log^* n)$ internal jumps.
Total number of internal jumps

**Lemma**

The number of internal jumps performed by the Union-Find data-structure overall is $O(n \log^* n)$.

**Proof.**

1. Every internal jump associated with block it is in.
2. Every block contributes $O(n)$ internal jumps throughout time. (By previous lemma.)
3. There are $O(\log^* n)$ blocks.
4. There are at most $O(n \log^* n)$ internal jumps.
Total number of internal jumps

**Lemma**

The number of internal jumps performed by the Union-Find data-structure overall is $O(n \log^* n)$.

**Proof.**

1. Every internal jump associated with block it is in.
2. Every block contributes $O(n)$ internal jumps throughout time. (By previous lemma.)
3. There are $O(\log^* n)$ blocks.
4. There are at most $O(n \log^* n)$ internal jumps.
Total number of internal jumps

Lemma

The number of internal jumps performed by the Union-Find data-structure overall is $O(n \log^* n)$.

Proof.

1. Every internal jump associated with block it is in.
2. Every block contributes $O(n)$ internal jumps throughout time. (By previous lemma.)
3. There are $O(\log^* n)$ blocks.
4. There are at most $O(n \log^* n)$ internal jumps.
**Lemma**

*The overall time spent on $m$ find operations, throughout the lifetime of a union-find data-structure defined over $n$ elements, is $O((m + n) \log^* n)$.**

**Theorem**

*If we perform a sequence of $m$ operations over $n$ elements, the overall running time of the Union-Find data-structure is $O((n + m) \log^* n)$.***
Lemma

The overall time spent on \(m\) find operations, throughout the lifetime of a union-find data-structure defined over \(n\) elements, is \(O((m + n) \log^* n)\).

Theorem

If we perform a sequence of \(m\) operations over \(n\) elements, the overall running time of the Union-Find data-structure is \(O((n + m) \log^* n)\).
More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}(x)(0)$

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(x) = x + 2$</td>
<td>$g_1(y) = y - 2$</td>
</tr>
</tbody>
</table>
Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(x) = x + 2$</td>
<td>$g_1(y) = y - 2$</td>
</tr>
</tbody>
</table>

$f_2(x) = f_1(f_2(x - 1)) = 2x$
Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(x) = x + 2$</td>
<td>$g_1(y) = y - 2$</td>
</tr>
<tr>
<td>$f_2(x) = 2x$</td>
<td>$g_2(y) = y/2$</td>
</tr>
</tbody>
</table>

$f_3(x) = f_2(f_3(x - 1)) = 2^x$
More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}(0)$

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(x) = x + 2$</td>
<td>$g_1(y) = y - 2$</td>
</tr>
<tr>
<td>$f_2(x) = 2x$</td>
<td>$g_2(y) = y/2$</td>
</tr>
<tr>
<td>$f_3(x) = 2^x$</td>
<td>$g_3(y) = \log y$</td>
</tr>
</tbody>
</table>
More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(x) = x + 2$</td>
<td>$g_1(y) = y - 2$</td>
</tr>
<tr>
<td>$f_2(x) = 2x$</td>
<td>$g_2(y) = y/2$</td>
</tr>
<tr>
<td>$f_3(x) = 2^x$</td>
<td>$g_3(y) = \log y$</td>
</tr>
<tr>
<td>$f_4(x) = f_3(f_4(x - 1)) = \text{Tower}x$</td>
<td></td>
</tr>
</tbody>
</table>
More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(x) = x + 2$</td>
<td>$g_1(y) = y - 2$</td>
</tr>
<tr>
<td>$f_2(x) = 2x$</td>
<td>$g_2(y) = y/2$</td>
</tr>
<tr>
<td>$f_3(x) = 2^x$</td>
<td>$g_3(y) = \log y$</td>
</tr>
<tr>
<td>$f_4(x) = \text{Tower}(x)$</td>
<td></td>
</tr>
</tbody>
</table>
More on strange functions...

Idea: Define a sequence of functions \( f_i(x) = f_{i-1}^{(x)}(0) \)

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_1(x) = x + 2 )</td>
<td>( g_1(y) = y - 2 )</td>
</tr>
<tr>
<td>( f_2(x) = 2x )</td>
<td>( g_2(y) = y/2 )</td>
</tr>
<tr>
<td>( f_3(x) = 2^x )</td>
<td>( g_3(y) = \log y )</td>
</tr>
<tr>
<td>( f_4(x) = \text{Tower}(x) )</td>
<td>( g_4(x) = \log^* x )</td>
</tr>
</tbody>
</table>
More on strange functions...

Idea: Define a sequence of functions \( f_i(x) = f_{i-1}'(0) \)

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f_1(x) = x + 2 )</td>
<td>( g_1(y) = y - 2 )</td>
</tr>
<tr>
<td>( f_2(x) = 2x )</td>
<td>( g_2(y) = y/2 )</td>
</tr>
<tr>
<td>( f_3(x) = 2^x )</td>
<td>( g_3(y) = \lg y )</td>
</tr>
<tr>
<td>( f_4(x) = \text{Tower}(x) )</td>
<td>( g_4(x) = \log^* x )</td>
</tr>
<tr>
<td>( f_5(x) = ... )</td>
<td></td>
</tr>
</tbody>
</table>
More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(x) = x + 2$</td>
<td>$g_1(y) = y - 2$</td>
</tr>
<tr>
<td>$f_2(x) = 2x$</td>
<td>$g_2(y) = y/2$</td>
</tr>
<tr>
<td>$f_3(x) = 2^x$</td>
<td>$g_3(y) = \log y$</td>
</tr>
<tr>
<td>$f_4(x) = \text{Tower}(x)$</td>
<td>$g_4(x) = \log^* x$</td>
</tr>
<tr>
<td>$f_5(x) = \ldots$</td>
<td></td>
</tr>
</tbody>
</table>

$f_i(x) = f_{i-1}^{(x)}(1)$
More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}(0)$

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(x) = x + 2$</td>
<td>$g_1(y) = y - 2$</td>
</tr>
<tr>
<td>$f_2(x) = 2x$</td>
<td>$g_2(y) = y/2$</td>
</tr>
<tr>
<td>$f_3(x) = 2^x$</td>
<td>$g_3(y) = \lg y$</td>
</tr>
<tr>
<td>$f_4(x) = \text{Tower}(x)$</td>
<td>$g_4(x) = \log^* x$</td>
</tr>
<tr>
<td>$f_5(x) = ...$</td>
<td></td>
</tr>
</tbody>
</table>

$f_i(x) = f_{i-1}(1)$

$g_i(x) = \# \text{ of times one has to apply } g_{i-1}(\cdot) \text{ to } x \text{ before we get number smaller than } 2.$
More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}^{(x)}(0)$

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(x) = x + 2$</td>
<td>$g_1(y) = y - 2$</td>
</tr>
<tr>
<td>$f_2(x) = 2x$</td>
<td>$g_2(y) = y/2$</td>
</tr>
<tr>
<td>$f_3(x) = 2^x$</td>
<td>$g_3(y) = \log y$</td>
</tr>
<tr>
<td>$f_4(x) = \text{Tower}(x)$</td>
<td>$g_4(x) = \log^* x$</td>
</tr>
<tr>
<td>$f_5(x) = \ldots$</td>
<td></td>
</tr>
</tbody>
</table>

$f_i(x) = f_{i-1}^{(x)}(1)$

$g_i(x) = \# \text{ of times one has to apply } g_{i-1}(\cdot) \text{ to } x \text{ before we get number smaller than } 2.$

$A(n) = f_n(n): \text{ Ackerman function.}$
More on strange functions...

Idea: Define a sequence of functions $f_i(x) = f_{i-1}(0)$

<table>
<thead>
<tr>
<th>Function</th>
<th>Inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_1(x) = x + 2$</td>
<td>$g_1(y) = y - 2$</td>
</tr>
<tr>
<td>$f_2(x) = 2x$</td>
<td>$g_2(y) = y/2$</td>
</tr>
<tr>
<td>$f_3(x) = 2^x$</td>
<td>$g_3(y) = \log y$</td>
</tr>
<tr>
<td>$f_4(x) = \text{Tower}(x)$</td>
<td>$g_4(x) = \log^* x$</td>
</tr>
<tr>
<td>$f_5(x) = \ldots$</td>
<td></td>
</tr>
</tbody>
</table>

$f_i(x) = f_{i-1}(1)$
$g_i(x) = \# \text{ of times one has to apply } g_{i-1}(\cdot) \text{ to } x \text{ before we get number smaller than } 2.$

$A(n) = f_n(n)$: Ackerman function.

**Inverse Ackerman function:**

$\alpha(n) = A^{-1}(n) = \min i \text{ s.t. } g_i(n) \leq i.$
Theorem (Tarjan [1975])

If we perform a sequence of $m$ operations over $n$ elements, the overall running time of the Union-Find data-structure is $O((n + m)\alpha(n))$.

(The above is not quite correct, but close enough.)
R. E. Tarjan. Efficiency of a good but not linear set union algorithm. 