Union Find

Version of October 31, 2014
A disjoint set Union-Find data structure supports three operations on collections of disjoint sets over some universe $U$. For any $x, y \in U$: 

1. **Create-Set($x$)**: Create a set containing a single item $x$.
2. **Find-Set($x$)**: Find the set that contains $x$.
3. **Union($x, y$)**: Merge the set containing $x$, and another set containing $y$ to a single set. After this operation, we have $\text{Find-Set}(x) = \text{Find-Set}(y)$. 


A disjoint set Union-Find data structure supports three operations on collections of disjoint sets over some universe $U$. For any $x, y \in U$:

1. **Create-Set($x$)**
   - Create a set containing a single item $x$. 

2. **Find-Set($x$)**
   - Find the set that contains $x$.

3. **Union($x$, $y$)**
   - Merge the set containing $x$, and another set containing $y$ to a single set. After this operation, we have $\text{Find-Set}(x) = \text{Find-Set}(y)$. 

A disjoint set Union-Find data structure supports three operations on collections of disjoint sets over some universe $U$. For any $x, y \in U$:

1. **Create-Set($x$)**
   - Create a set containing a single item $x$.

2. **Find-Set($x$)**
   - Find the set that contains $x$. 
A disjoint set Union-Find data structure supports three operations on collections of disjoint sets over some universe $U$. For any $x, y \in U$:

1. **Create-Set($x$)**
   - Create a set containing a single item $x$.

2. **Find-Set($x$)**
   - Find the set that contains $x$.

3. **Union($x, y$)**
   - Merge the set containing $x$, and another set containing $y$ to a single set.
A disjoint set Union-Find data structure supports three operations on collections of disjoint sets over some universe $U$. For any $x, y \in U$:

1. **Create-Set($x$)**
   - Create a set containing a single item $x$.

2. **Find-Set($x$)**
   - Find the set that contains $x$.

3. **Union($x, y$)**
   - Merge the set containing $x$, and another set containing $y$ to a single set.
   - After this operation, we have Find-Set($x$) = Find-Set($y$).
The Disjoint Set Union-Find data structure
- The basic implementation
- An improvement
Every item is in a tree.
Every item is in a tree. (Do not confuse these with the subtrees formed by Kruskal’s algorithm.)
Every item is in a tree. (Do not confuse these with the subtrees formed by Kruskal’s algorithm.)

The root of the tree is the representative item of all items in that tree
Every item is in a tree. (Do not confuse these with the subtrees formed by Kruskal’s algorithm.)

The root of the tree is the representative item of all items in that tree

i.e., the root of the tree represents the whole items.
Every item is in a tree. (Do not confuse these with the subtrees formed by Kruskal’s algorithm.)

The root of the tree is the representative item of all items in that tree

i.e., the root of the tree represents the whole items.

use the root’s ID as the unique ID of the set.
Every item is in a tree. (Do not confuse these with the subtrees formed by Kruskal’s algorithm.)

The root of the tree is the representative item of all items in that tree

- i.e., the root of the tree represents the whole items.
- use the root’s ID as the unique ID of the set.

In this up-tree implementation, every node (except the root) has a pointer pointing to its parent.
Every item is in a tree. (Do not confuse these with the subtrees formed by Kruskal’s algorithm.)
The root of the tree is the representative item of all items in that tree
  i.e., the root of the tree represents the whole items.
  use the root’s ID as the unique ID of the set.
In this up-tree implementation, every node (except the root) has a pointer pointing to its parent.
  The root element has a pointer pointing to itself.
Create-Set(x) and Find-Set(x)

Create-Set(x):

parent = x;

Find-Set(x):

Also easy simply trace the parent point until we hit the root, then return the root element.

while x \neq x.parent do
    x = x.parent;
end

return x
Create-Set($x$) and Find-Set($x$)

Create-Set($x$): easy

Find-Set($x$):

```
while $x \neq x$.parent do
    x = x.parent;
end
return x
```
Create-Set(\(x\)) and Find-Set(\(x\))

Create-Set(\(x\)): easy

\[ x.parent \leftarrow x; \]

Find-Set(\(x\)): also easy

simply trace the parent point until we hit the root, then return the root element.

\[ \text{while } x \neq x.\text{parent} \text{ do} \]
\[ \quad x = x.\text{parent}; \]
\[ \text{end} \]

return \(x\).
Create-Set(x): easy

\texttt{x.parent=x;}

Find-Set(x):
Create-Set(x): easy

\[
x.\text{parent} = x;
\]

Find-Set(x): also easy
Create-Set($x$): easy

\[ x\.parent \leftarrow x; \]

Find-Set($x$): also easy

- simply trace the parent point until
Create-Set($x$): easy

\[ x\text{.parent} = x; \]

Find-Set($x$): also easy

- simply trace the parent point until we hit the root, then return
Create-Set($x$): easy

\[
x \text{.parent} = x;
\]

Find-Set($x$): also easy

- simply trace the parent point until we hit the root, then return the root element.
Create-Set(x): easy

\[ x\text{.parent} = x; \]

Find-Set(x): also easy

- simply trace the parent point until we hit the root, then return the root element.

\[
\text{while } x \neq x\text{.parent do}
\]

Create-Set($x$): easy

```plaintext
x.parent = x;
```

Find-Set($x$): also easy

- simply trace the parent point until we hit the root, then return the root element.

```plaintext
while $x \neq x.parent$ do
  $x = x.parent$;
end
return
```
Create-Set\( (x) \): easy

\[
x\. \text{parent} = x;
\]

Find-Set\( (x) \): also easy

- simply trace the parent point until we hit the root, then return the root element.

\[
\text{while } x \neq x\. \text{parent} \text{ do}
\]
\[
| \quad x = x\. \text{parent};
\]
\[
\text{end}
\]
\[
\text{return } x
\]
Naive solution:

- put the parent pointer of the representation of $x$ pointing to the representation of $y$. 
**Naive** solution:

- put the parent pointer of the representation of $x$ pointing to the representation of $y$. 

[Diagram showing the union operation]
Naive solution:

- put the parent pointer of the representation of $x$ pointing to the representation of $y$.

Question

Is this a good idea?
Problem

May become a linked-list at the end! Hence it is not efficient.

Question

Can we do better?

Simple trick (Union by height):

when we union two trees together, we always make the root of
the taller tree the parent of shorter tree.
May become a **linked-list** at the end! Hence it is not efficient.
May become a **linked-list** at the end! Hence it is not efficient.

**Question**

Can we do better?
Problem

May become a **linked-list** at the end! Hence it is not efficient.

**Question**

Can we do better?

**Simple trick** (*Union by height)*:
May become a **linked-list** at the end! Hence it is not efficient.

**Question**

Can we do better?

**Simple trick (Union by height):**

- when we union two trees together, we always make the root of the **taller** tree the parent of shorter tree.
The root of every tree also holds the **height** of the tree.
The root of every tree also holds the height of the tree.
In case two trees have the same height, we choose the root of the first tree point to the root of the second.
The root of every tree also holds the **height** of the tree.

In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is **increased by 1**.

```
Union(x, y)
    a = Find-Set(x);
    b = Find-Set(y);
    if a.height ≤ b.height then
        if a.height == b.height then
            b.height++;
        end
        a.parent = b;
    else
        b.parent = a;
    end
```
The root of every tree also holds the height of the tree.

In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

```
Union(x, y)
  a=Find-Set(x);
b=Find-Set(y);
  if a.height ≤ b.height then
    if a.height == b.height then
      b.height++;
    end
    a.parent=b;
  else
    b.parent=a;
  end
```
The root of every tree also holds the **height** of the tree.

In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

Union($x, y$)
The root of every tree also holds the **height** of the tree.
In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

\[
\text{Union}(x, y)
\]

\[
a = \text{Find-Set}(x);
b = \text{Find-Set}(y);
\]
The root of every tree also holds the **height** of the tree.

In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

**Union(x, y)**

```plaintext
a = Find-Set(x);
b = Find-Set(y);
if a.height ≤ b.height then
    if a.height == b.height then
        b.height++;
    end
    a.parent = b;
else
    b.parent = a;
end
```
The root of every tree also holds the **height** of the tree.

In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

**Union**($x, y$)

\[
a = \text{Find-Set}(x); \\
b = \text{Find-Set}(y); \\
\textbf{if } a.\text{height} \leq b.\text{height} \textbf{ then} \\
\quad \textbf{if } a.\text{height} == b.\text{height} \textbf{ then} \\
\quad \quad b.\text{height}++; \\
\textbf{end}
\]
The root of every tree also holds the **height** of the tree.

In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

**Union(x, y)**

\[
a = \text{Find-Set}(x);
b = \text{Find-Set}(y);
\]

\[
\text{if } a.\text{height} \leq b.\text{height} \text{ then}
\]

\[
\quad \text{if } a.\text{height} == b.\text{height} \text{ then}
\]

\[
\quad \quad \text{b.\text{height}}++;\n\]

\[
\quad \text{end}\n\]

\[
\text{a.parent} = b;\n\]
The root of every tree also holds the **height** of the tree.

In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

**Union(x, y)**

a = Find-Set(x);
b = Find-Set(y);

if \(a.\text{height} \leq b.\text{height}\) then
  if \(a.\text{height} == b.\text{height}\) then
    b.\text{height}++;
  else
    a.\text{parent}=b;

else
  b.\text{parent}=a;

The root of every tree also holds the **height** of the tree.

In case two trees have the same height, we choose the root of the first tree point to the root of the second. And the tree height is increased by 1.

**Union**($x$, $y$)

```java
a = Find-Set(x);
b = Find-Set(y);
if $a$.height $\leq$ $b$.height then
    if $a$.height == $b$.height then
        $b$.height++;
    end
    $a$.parent = $b$;
else
    $b$.parent = $a$;
end
```
Lemma

For the root $x$ of any tree, let $\text{size}(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $\text{size}(x) \geq 2^{h(x)}$. 
Lemma

For the root $x$ of any tree, let $\text{size}(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $\text{size}(x) \geq 2^{h(x)}$.

Proof.

(By induction)
Lemma

For the root $x$ of any tree, let $\text{size}(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $\text{size}(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $\text{size}(x) = 1$. We have
Lemma

For the root $x$ of any tree, let $\text{size}(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $\text{size}(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $\text{size}(x) = 1$. We have $1 \geq 2^0 = 1$. 


Lemma

For the root $x$ of any tree, let $\text{size}(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $\text{size}(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $\text{size}(x) = 1$. We have $1 \geq 2^0 = 1$.

2. Suppose the assumption is true for any $x$ and $y$ before $\text{Union}(x, y)$.
Lemma

For the root $x$ of any tree, let $size(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $size(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.

2. Suppose the assumption is true for any $x$ and $y$ before $\text{Union}(x, y)$. Let the size and height of the resulting tree be $size(x')$, and $h(x')$. 
Lemma

For the root $x$ of any tree, let $\text{size}(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $\text{size}(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $\text{size}(x) = 1$. We have $1 \geq 2^0 = 1$.

2. Suppose the assumption is true for any $x$ and $y$ before Union($x, y$). Let the size and height of the resulting tree be $\text{size}(x')$, and $h(x')$.
   
   - $h(x) < h(y)$, we have
     
     $$\text{size}(x') = \text{size}(x) + \text{size}(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} =$$
Lemma

For the root $x$ of any tree, let $\text{size}(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $\text{size}(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $\text{size}(x) = 1$. We have $1 \geq 2^0 = 1$.

2. Suppose the assumption is true for any $x$ and $y$ before $\text{Union}(x, y)$. Let the size and height of the resulting tree be $\text{size}(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have
     \[
     \text{size}(x') = \text{size}(x) + \text{size}(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}.\]
Lemma

For the root $x$ of any tree, let $\text{size}(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $\text{size}(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $\text{size}(x) = 1$. We have $1 \geq 2^0 = 1$.

2. Suppose the assumption is true for any $x$ and $y$ before Union($x, y$). Let the size and height of the resulting tree be $\text{size}(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have
     \[
     \text{size}(x') = \text{size}(x) + \text{size}(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}. 
     \]
   - $h(x) = h(y)$, we have
     \[
     \text{size}(x') = \text{size}(x) + \text{size}(y) \geq 2^{h(x)} + 2^{h(y)} = 2^{h(y)+1} = 2^{h(x')}. 
     \]
Lemma

For the root $x$ of any tree, let $\text{size}(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $\text{size}(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $\text{size}(x) = 1$. We have $1 \geq 2^0 = 1$.

2. Suppose the assumption is true for any $x$ and $y$ before Union($x, y$). Let the size and height of the resulting tree be $\text{size}(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have
     
     \[
     \text{size}(x') = \text{size}(x) + \text{size}(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}. 
     \]
   - $h(x) = h(y)$, we have
     
     \[
     \text{size}(x') = \text{size}(x) + \text{size}(y) \geq 2^{h(x)} + 2^{h(y)} = 2^{h(y)+1} = 2^{h(x')}. 
     \]
   - $h(x) > h(y)$,
Lemma

For the root $x$ of any tree, let $\text{size}(x)$ denote the number of nodes and $h(x)$ be the height of the tree. Then $\text{size}(x) \geq 2^{h(x)}$.

Proof.

(By induction)

1. At beginning, $h(x) = 0$, and $\text{size}(x) = 1$. We have $1 \geq 2^0 = 1$.

2. Suppose the assumption is true for any $x$ and $y$ before Union($x, y$). Let the size and height of the resulting tree be $\text{size}(x')$, and $h(x')$.
   - $h(x) < h(y)$, we have
     \[
     \text{size}(x') = \text{size}(x) + \text{size}(y) \geq 2^{h(x)} + 2^{h(y)} \geq 2^{h(y)} = 2^{h(x')}.
     \]
   - $h(x) = h(y)$, we have
     \[
     \text{size}(x') = \text{size}(x) + \text{size}(y) \geq 2^{h(x)} + 2^{h(y)} = 2^{h(y)+1} = 2^{h(x')}.
     \]
   - $h(x) > h(y)$, is similar to the first case
Lemma

For $n$ items, the running time of

- Create-Set is $O(1)$,
Lemma

For $n$ items, the running time of

- Create-Set is $O(1)$,
- Find-Set is $O(\log n)$, and
Lemma

For $n$ items, the running time of

- Create-Set is $O(1)$,
- Find-Set is $O(\log n)$, and
- Union is $O(\log n)$

respectively.

Proof. Obviously, Create-Set($x$) is $O(1)$, and the running time of Union($x$, $y$) depends on Find-Set($x$). Since the running time of Find-Set($x$) depends on the height of the tree. From previous lemma, for any tree, we have $n \geq 2^h \Rightarrow h \leq \log n \Rightarrow h = O(\log n)$. Hence we have Find-Set($x$) = $O(\log n)$.
Lemma

For $n$ items, the running time of

- Create-Set is $O(1)$,
- Find-Set is $O(\log n)$, and
- Union is $O(\log n)$

respectively.

Proof.

- Obviously, Create-Set($x$) is $O(1)$,
Lemma

For n items, the running time of
- Create-Set is $O(1)$,
- Find-Set is $O(\log n)$, and
- Union is $O(\log n)$
respectively.

Proof.

- Obviously, Create-Set$(x)$ is $O(1)$, and the running time of Union$(x, y)$ depends on Find-Set$(x)$. 
Lemma

For $n$ items, the running time of
- Create-Set is $O(1)$,
- Find-Set is $O(\log n)$, and
- Union is $O(\log n)$

respectively.

Proof.

- Obviously, Create-Set($x$) is $O(1)$, and the running time of Union($x, y$) depends on Find-Set($x$).
- Since the running time of Find-Set($x$) depends on the height of the tree.
Lemma

For $n$ items, the running time of

- Create-Set is $O(1)$,
- Find-Set is $O(\log n)$, and
- Union is $O(\log n)$

respectively.

Proof.

- Obviously, Create-Set($x$) is $O(1)$, and the running time of Union($x, y$) depends on Find-Set($x$).
- Since the running time of Find-Set($x$) depends on the height of the tree. From previous lemma, for any tree, we have

$$ n \geq 2^h \implies h $$
Lemma

For $n$ items, the running time of

- $\text{Create-Set is } O(1)$,
- $\text{Find-Set is } O(\log n)$, and
- $\text{Union is } O(\log n)$

respectively.

Proof.

- Obviously, $\text{Create-Set}(x)$ is $O(1)$, and the running time of $\text{Union}(x, y)$ depends on $\text{Find-Set}(x)$.
- Since the running time of $\text{Find-Set}(x)$ depends on the height of the tree. From previous lemma, for any tree, we have
  \[ n \geq 2^h \implies h \leq \log n \]
  \[ \implies h = O(\log n) \]
Lemma

For n items, the running time of
- Create-Set is $O(1)$,
- Find-Set is $O(\log n)$, and
- Union is $O(\log n)$
respectively.

Proof.

- Obviously, Create-Set($x$) is $O(1)$, and the running time of Union($x, y$) depends on Find-Set($x$).
- Since the running time of Find-Set($x$) depends on the height of the tree. From previous lemma, for any tree, we have
\[
n \geq 2^h \implies h \leq \log n \implies h = O(\log n)
\]

Hence we have Find-Set($x$) = $O(\log n)$.
The Disjoint Set Union-Find data structure
  - The basic implementation
  - An improvement
We can make the running time even faster if we add another trick.
We can make the running time even faster if we add another trick.

In Find-Set(x), we trace the path from x to the root.
We can make the running time even faster if we add another trick.

In Find-Set(x), we trace the path from x to the root.

Let r be the root of the tree,
We can make the running time even faster if we add another trick.  

In \texttt{Find-Set}(x), we trace the \textit{path} from \texttt{x} to the root.  

Let \( r \) be the root of the tree, and the path from \texttt{x} to \( r \) is \( xa_1a_2 \ldots a_k r \).
We can make the running time even faster if we add another trick.

In `Find-Set(x)`, we trace the path from `x` to the root.

Let `r` be the root of the tree, and the path from `x` to `r` is `xa_1a_2...a_kr`.

As a by-product, we also make all the parent pointers of `x`, `a_1`, `a_2`, ..., `a_k` pointing to `r` directly.
We can make the running time even faster if we add another trick.

In `Find-Set(x)`, we trace the path from `x` to the root.

Let `r` be the root of the tree, and the path from `x` to `r` is `xa_1a_2...a_kr`.

As a by-product, we also make all the parent pointers of `x`, `a_1`, `a_2`, ..., `a_k` pointing to `r` directly.

- Shortens the time of some future calls to `Find-Set`.
- Does not increase height.
We can make the running time even faster if we add another trick.

In `Find-Set(x)`, we trace the path from `x` to the root.

Let `r` be the root of the tree, and the path from `x` to `r` is `xa_1a_2...a_kr`.

As a by-product, we also make all the parent pointers of `x`, `a_1`, `a_2`, ..., `a_k` pointing to `r` directly.

- Shortens the time of some future calls to `Find-Set`.
- Does not increase height.

This idea is called **path compression**.
Question

Does path compression improve the running time of union-find?

\[ \text{lg} (i^n) \text{ defined recursively for nonnegative integers } i \text{ as:} \]

\[ \begin{cases} n & \text{if } i = 0 \\ \text{lg}(\text{lg}(i - 1)^n) & \text{if } i > 0 \text{ and } \text{lg}(i - 1)^n > 0 \\ \text{undefined} & \text{if } i > 0 \text{ and } \text{lg}(i - 1)^n \leq 0 \\ \text{or } \text{lg}(i - 1)^n \text{ is undefined.} & \end{cases} \]

The iterated logarithm is defined as:

\[ \text{lg}^* n = \min \{ i \geq 0 : \text{lg} (i^n) \leq 1 \} \]

a very slow growing function. e.g.,

- \( \text{lg}^* 2 = 1 \)
- \( \text{lg}^* 4 = 2 \)
- \( \text{lg}^* 16 = 3 \)
- \( \text{lg}^* 65536 = 4 \)
- \( \text{lg}^* 2^{65536} = 5 \).
Question

Does path compression improves the running time of union-find?

\( \lg^{(i)} n \): defined recursively for nonnegative integers \( i \) as

\[
\lg^{(i)} n = \begin{cases} 
  n & \text{if } i = 0 \\
  \lg^{(i-1)} n & \text{if } i > 0 \text{ and } \lg^{(i-1)} n > 0,
\end{cases}
\]

or \( \lg^{(i)} n \) is undefined.

The iterated logarithm is defined as

\[ \lg^{*} n = \min \{ i \geq 0 : \lg^{(i)} n \leq 1 \} \]

a very slow growing function. e.g., \( \lg^{*} 2 = 1 \), \( \lg^{*} 4 = 2 \), \( \lg^{*} 16 = 3 \), \( \lg^{*} 65536 = 4 \), \( \lg^{*} 2^{65536} = 5 \).
Question

Does path compression improve the running time of union-find?

\( \lg^{(i)} n \): defined recursively for nonnegative integers \( i \) as

\[
\lg^{(i)} n = \begin{cases} 
  n & \text{if } i = 0 \\
  \lg(\lg^{(i-1)} n) & \text{if } i > 0 \text{ and } \lg^{(i-1)} n > 0,
\end{cases}
\]
Question

Does path compression improves the running time of union-find?

\( \lg^{(i)} n \): defined recursively for nonnegative integers \( i \) as

\[
\lg^{(i)} n = \begin{cases} 
  n & \text{if } i = 0 \\
  \lg(\lg^{(i-1)} n) & \text{if } i > 0 \text{ and } \lg^{(i-1)} n > 0, \\
  \text{undefined} & \text{if } i > 0 \text{ and } \lg^{(i-1)} n \leq 0, \text{ or } \lg^{(i-1)} n \text{ is undefined.}
\end{cases}
\]
Question

Does path compression improve the running time of union-find?

$\lg^{(i)} n$: defined recursively for nonnegative integers $i$ as

$$\begin{align*}
\lg^{(i)} n &= \begin{cases} 
n & \text{if } i = 0 \\
\lg(\lg^{(i-1)} n) & \text{if } i > 0 \text{ and } \lg^{(i-1)} n > 0,
\text{undefined} & \text{if } i > 0 \text{ and } \lg^{(i-1)} n \leq 0, \text{ or } \lg^{(i-1)} n \text{ is undefined.}
\end{cases}
\end{align*}$$

The iterated logarithm is defined as

$$\lg^* n = \min \{i \geq 0 : \lg^{(i)} n \leq 1\}$$
Question

Does path compression improves the running time of union-find?

\(\lg^{(i)} n\): defined recursively for nonnegative integers \(i\) as

\[
\lg^{(i)} n = \begin{cases} 
n & \text{if } i = 0 \\
\lg(\lg^{(i-1)} n) & \text{if } i > 0 \text{ and } \lg^{(i-1)} n > 0, \\
\text{undefined} & \text{if } i > 0 \text{ and } \lg^{(i-1)} n \leq 0, \text{ or } \lg^{(i-1)} n \text{ is undefined.}
\end{cases}
\]

The iterated logarithm is defined as

\[
\lg^* n = \min \{ i \geq 0 : \lg^{(i)} n \leq 1 \}
\]

- a very slow growing function.
Question

Does path compression improves the running time of union-find?

\( \lg^{(i)} n \): defined recursively for nonnegative integers \( i \) as

\[
\begin{align*}
\lg^{(i)} n &= \begin{cases} 
  n & \text{if } i = 0 \\
  \lg(\lg^{(i-1)} n) & \text{if } i > 0 \text{ and } \lg^{(i-1)} n > 0, \\
  \text{undefined} & \text{if } i > 0 \text{ and } \lg^{(i-1)} n \leq 0, \text{ or } \lg^{(i-1)} n \text{ is undefined.}
\end{cases}
\end{align*}
\]

The iterated logarithm is defined as

\[
\lg^* n = \min \{ i \geq 0 : \lg^{(i)} n \leq 1 \}
\]

- a very slow growing function.
- e.g.,
  \[
  \lg^* 2 = 1, \lg^* 4 = 2, \lg^* 16 = 3, \lg^* 65536 = 4, \lg^* 2^{65536} = 5.
  \]
The following theorem is stated without proof.

**Theorem**

A sequence of $m$ Create-Set, Find-Set and Union operations,
The following theorem is stated without proof.

**Theorem**

A sequence of $m$ Create-Set, Find-Set and Union operations, $n$ of which are Create-Set operations,
The following theorem is stated without proof.

**Theorem**

A sequence of $m$ Create-Set, Find-Set and Union operations, $n$ of which are Create-Set operations, can be performed on a disjointed-set forest with union by height and path compression in worst-case time $O(m \lg^* n)$. 
The following theorem is stated without proof.

**Theorem**

A sequence of \( m \) Create-Set, Find-Set and Union operations, \( n \) of which are Create-Set operations, can be performed on a disjointed-set forest with union by height and path compression in worst-case time \( O(m \lg^* n) \).

**Question**

What is the running time of Kruskal’s algorithm if we employ this implementation of disjoint set Union-Find?