CS 466 Notes
O(lg*n) Upper Bound on Amortized Cost of Union-Find

Theorem: A sequence of n union and find operations, performed as discussed in class with path compression, uses O(lg*n) operations on an amortized basis.

Proof sketch:
Let *n(v)* denote the maximum size of the subtree of which *v* is root. (i.e. the number of nodes in the tree, including *v*, when *v* becomes the child of another node (or when the computation ends).
Define the *rank* of *v*, r(v) as $\lfloor \lg n(v) \rfloor$.

Lemma 1: If *w* is the parent of v, r(v) < r(w).
Because $n(w) \geq n(v)$, before *v* became a child of *w*, when that occurred, *n(w)* increased by *n(v)*.

Lemma 2: There are at most $n/2^s$ nodes of rank *s*, for $0 \leq s \leq \lfloor \lg n \rfloor$.
Ranks are monotonically increasing as we move up a tree, so if the ranks of two nodes are equal, their subtrees must be disjoint. If *v* is of rank *s*, $n(v) \geq 2^s$. The lemma follows as there are at most *n* nodes in total.

Define "a tower of 2's": $t(i) = \begin{cases} 1 & if\ i = 0 \\ 2^{t(i-1)} & if\ i \geq 1 \end{cases}$
and $lg^*(n) = min\ \{i: t(i) \geq n\}$

So, for example $lg^*(2^{65536}) = 5$.

Next we define rank groups. *v* and *u* are in rank group *g* if $g = lg^*(r(v)) = lg^*(r(v))$. The largest rank is $\lfloor \lg n \rfloor$, so the largest rank group is $lg^*(lg\ n) = lg^*(n) - 1$.

With these definitions we charge each step in a find operation to either the operation itself or to the node. First note that unions cost only 1 operation. In performing a find we "simply" follow the path to the root (and make each node a child the root, but this does not alter the order of runtime).
The key trick is to charge the step from node *v* to its parent *w*:

- the find operation 1 unit each time we go to a node from a different rank group (so each find is charged at most lg^*n).
- otherwise the step is charged to node *v*, the one we are leaving.

The issue is to show that the total charge to all nodes is not too high, indeed at most $n\ lg^*\ n$.
Ranks increase monotonically as we move up a tree, so any node, *v*, can be charged at most the number of ranks there are in its rank group. As any node of rank 0 has a parent in a different group so we need only consider rank groups > 1. If *v* is in group *g*, it can be charged at most $t(g) - t(g-1)$ times before it has a parent in a higher group and so is never charged again. So the total cost charged to all nodes is at most

$$c \leq \sum_{g=1}^{lg^*n - 1} n(g)(t(g) - t(g-1))$$

($n(g)$ denotes the number of nodes in rank group *g*)
The rest is manipulation.
First: $n(g) \leq n/t(g)$ as

$$n(g) \leq \sum_{s=t(g-1)+1}^{t(g)} \frac{n}{2^s}$$

$$= n/2^{t(g)-t(g-1)-1} \sum_{s=0}^{t(g)-t(g-1)-1} 1/2^s$$

$$< \frac{2n}{2^{t(g-1)+1}} = \frac{n}{2^{t(g-1)}} = n/t(g)$$

Now put this bound into the bound for *c*:

$$c < \sum_{g=1}^{lg^*n-1} \frac{n}{t(g)} (t(g) - t(g-1)) \leq \sum_{g=1}^{lg^*n-1} \frac{n}{t(g)} t(g) = n\, lg^*n.$$

□

This analysis is due to Hopcroft and Ullman (though notation and details follow Goodrich and Tamassia). Tarjan proved a tighter bound, which is given in CLRS. This is based on Ackermann's function. Ackermann's function is a very quickly growing function, much faster (once it gets going) than our tower of 2's. Actually it is a function that is not "primitive recursive". Primitive recursive functions essentially are functions defined with a restricted form of recursion that is equivalent to iteration in which we have only "for loops" with previously computed upper bound on the number of times we will go around the loop. This is in contrast with the iterative construction of a "while loop" which has no bound on the number of iterations. This clearly gets into issues whether a function it total (i.e. that the procedure halts). Ackermann's function, defined below, if "clearly" total, but indeed "grows to fast" to be primitive recursive. We define Ackermann's function by:

$A_0(n) = 2n$	for $n \geq 0$
$A_i(1) = A_{i-1}(2)$	for $i \geq 1$
$A_i(n) = A_{i-1}(A_i(n-1)$	for $i \geq 1$ and $n \geq 2$.

So $A_0(n) = 2n$, which is innocent enough.
$A_1(n) = 2^n$, which is starting to move along.
$A_2(n)$ = a tower of 2's of height n.
etc.

We get an unbounded, but very slowly growing, function in the same way we did with the tower of 2's.

$$\alpha(n) = \min(i : A_i(i) \geq n\}$$

Tarjan showed that the union-find algorithm has amortized cost $O(\alpha(n))$, but that it is not amortized constant time.