The Union-Find Problem

We look at the problem maintaining a system of sets that are pairwise disjoint. It should support two operations: (1) Find (2) Union.

\[ C: \text{collection of subsets of } \{1, 2, \ldots, n\} \]
\[ \text{s.t. } \cup I = \{1, 2, \ldots, n\} \text{ and } I \cap J = \emptyset \]
\[ \text{if } I, J \in C. \]

Find(i): determines the set IEC within I.

Union(I, J): Join sets I and J in C.

Often in applications we need the above two operations in the following way:

\[ I := \text{Find}(i); \quad J := \text{Find}(j); \]
\[ \text{If } I \neq J \text{ then } \text{Union}(I, J) \text{ endif.} \]

Here it does not really matter what I and J really are, except that they need to be different iff they represent different sets.
A Simple Solution.

C: array [1...n] of integers

Each set is represented by one of its elements, and C[i] stores the name (the index of the representative) of the set containing i.

Finding a set takes O(1), but union takes Θ(n) since the entire array needs to be scanned in the worst-case.
The previous solution can be improved by storing

(i) the elements of a set in a linked list (next pointer)

(ii) the size of a set at its representative

```plaintext
function Find(i)
    return C[i].set

procedure Union(I, J)
    if C[I].size < C[J].size then I ← J endif
    C[I].size := C[I].size + C[J].size;
    Second := C[I].next; C[I].next := J;
    t := J; loop
        C[t].set := I;
        if C[t].next := 0 then
            C[t].next := Second
        exit loop
    endif
    t := t.next
endloop
```
The worst-case of a single union operation is still $\Theta(n)$, as before, but now we can show a logarithmic amortized bound.

Claim: n-1 union operations take time $O(n \log n)$

Proof: We consider the size of the set that contains the element $i$. So define

$$O(i) = C \cdot [\text{find}(i)].\text{size}.$$ 

$O(i)$ changes whenever $i$ is touched in the union operation; in this case the new $O(i)$ is at least twice as large as the old one. This is because $i$ is touched only if it belongs to the smaller of the two sets joined. Define $k$ as the number of times element $i$ is touched. Then $O(i) \geq 2^k \Rightarrow k \leq \log n$. 
Tree Representation.

We consider representing each set as a tree.

Idea - each set is represented by
- Find(i) traverses the path from i up to the root.
- Union(I, J) links the two trees.

Ex.
Union (2, 3)
  " (4, 7)
  " (2, 4)
  " (1, 2)
  " (4, 10)
  " (9, 12)
  " (12, 2)
  " (8, 11)
  " (8, 2)
  " (5, 6)
  " (6, 1)

Union takes O(1) time,
Find takes time proportional to the depth of the tree node.
Weighted Merging. The same idea as before improves time: instead of joining arbitrarily, join the smaller to the larger tree.

Assume: C has fields

\( p \) ... index of parents
\( h \) ... height of the tree

function Find(i)
if \( C[i].p = i \) then return i
else return Find(\( C[i].p \))
endif

procedure Union(I, J)
if \( C[I].h < C[J].h \) then
    \( C[I].p := J \)
else
    \( C[J].p := I \);
    if \( C[I].h = C[J].h \) then
        \( C[I].h := C[I].h + 1 \)
endif
endif
Claim. The height of a tree with n nodes is at most $\log n$.

So, Find takes $O(\log n)$ time.
Union takes $O(1)$ time.

Path Compression

The idea is to connect all nodes visited during a Find operation directly to the root.

function Find(i)
if $C[i].p \neq 0$ then $C[i].p := \text{Find}(C[i].p)$
Return $C[i].p$

Example $(i,j)$ stands for
$I := \text{Find}(i), J := \text{Find}(j)$
If $I \neq J$ then Union$(I,J)$

(2,3) \ (2,4) \ (1,6) \ (2,6) \ (5,7) \ (4,6)

\[ \begin{array}{ccccccc}
2 & 2 & 2 & 3 & 3 & 3 & 5 \\
3 & 3 & 6 & 6 & 6 & 6 & 7 \\
\end{array} \]

a path compression
Ackermann's Function

It can be shown that m find operations take $O(m \alpha(m))$ time where

$\alpha(m)$ is the slowly growing inverse Ackermann's function.

**Def.**

\[
\begin{align*}
A_k(1) &= 2 & \text{for } k \geq 1 \\
A_1(n) &= 2n & \text{for } n \geq 1 \\
A_k(n) &= A_{k-1}(A_k(n-1)) & \text{for } k, n \geq 2
\end{align*}
\]

<table>
<thead>
<tr>
<th>$n=1$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k=1$</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>$2^2$</td>
<td>$2^{2^2}$</td>
<td>$2^{2^{2^2}}$</td>
<td>$2^{2^{2^{2^2}}}$</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>$2^2$</td>
<td>$2^{2^2}$</td>
<td>$2^{2^{2^2}}$</td>
<td>$2^{2^{2^{2^2}}}$</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>$2^2$</td>
<td>$2^{2^2}$</td>
<td>$2^{2^{2^2}}$</td>
<td>$2^{2^{2^{2^2}}}$</td>
</tr>
</tbody>
</table>

$\alpha(m) = \min \{ n \mid A_n(n) \geq m \}$

For all practical purposes $\alpha(m) \leq 4$, but $\alpha(m)$ goes to infinity as $m$ goes to infinity.