Disjoint-set data structure (Cormen et. al., Chapter 21)

A disjoint-set data structure keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. It supports three operations:

- *Find*: Determine which subset a particular element is in. *Find* typically returns an item from this set that serves as its "representative"; by comparing the result of two *Find* operations, one can determine whether two elements are in the same subset.
- *Union*: Join two subsets into a single subset.
- *MakeSet*: Make a set containing only a given element (a singleton). This is a trivial operation.

In order to define these operations more precisely, some way of representing the sets is needed. One common approach is to select a fixed element of each set, called its *representative*, to represent the set as a whole. Then, *Find*(x) returns the representative of the set that *x* belongs to, and *Union* takes two set representatives as its arguments.

Disjoint-set linked lists

A simple approach to creating a disjoint-set data structure is to create a linked list for each set. The element at the head of each list is chosen as its representative.

Naïve approach
MakeSet: creates a list of one element, O(1);
Union: appends the two lists, O(1).
Find: traverse the list backwards from a given element to the head of the list, O(n).

Weighted-union heuristic
Include in each linked list node a pointer to the head of the list, since this pointer refers directly to the set representative, O(1);
Union now has to update each element of the list being appended to make it point to the head of the new combined list, O(n). We can always append the shorter list to the longer.
Amortized cost: a sequence of *m MakeSet*, *Union*, and *Find* operations on *n* elements requires O($m + n\log n$) time.
Proof: (Aggregate analysis) Choose an arbitrary element x. We wish to count how many times in the worst case will x need to have its representative (i.e., the linked list it belongs to) updated. Note that the representative of x will be updated only when the list it belongs to is merged with another list of the same size or of greater. Each time that happens, the size of the list to which x belongs at least doubles. So equivalently, we ask "how many times can the size of a linked list can be doubled before it reaches the size of n?" It is O(log n). So for any given element of any given list, the will need to be updated O(log n) times in the worst case. Therefore updating a list of *n* elements by any number of operations takes O(n log n) time in the worst case. So any sequence of m operations takes O(m + nlogn) time.

Disjoint-set forests

Each set is represented by a tree, in which each node hold a reference to its parent. The representative of each set is the root of that set's tree.
Find: follows parent nodes until it reaches the root.
Union: combines two trees into one by attaching the root of one to the root of the other.

A naïve implementation takes O(n) for find (when the tree is extremely unbalanced).

Union by rank
Always attach the smaller tree (with smaller depth) to the root of the larger tree (with greater depth, which only increases the depth if the depths of these two trees were equal. Here, *rank* is used instead of *depth* since it stops being equal to the depth if *path compression* is also used. One-element trees are defined to have a rank of zero, and whenever two trees of the same rank *r* are united, the rank of the result is *r*+1. Just applying union by rank alone yields O(log n) amortized cost per *MakeSet*, *Union*, or *Find* operation.

Path compression,
To *flatten* (increasing the width while reducing the depth) the tree whenever *Find* is used. Each node visited on the way to a root node is attached directly to the root node; they all share the same representative. As *Find* recursively traverses up the tree, it changes each node's parent reference to point to the root that it found. The resulting tree is much flatter, speeding up future operations not only on these elements but on those referencing them, directly or indirectly.

```
MakeSet(x)
1 p[x] ← x
2 rank[x] ←  0

Union(x , y )
1  LINK(FIND-SET(x ), FIND-SET(y ))

Link(x , y )
1 if rank [x] > rank [y]
2          p[y] ← x
3 else
4          p[x] ← y
5 if rank[x] = rank[y]
6          rank[y] ← rank [y] + 1

Find (x)
1 if x ≠ p [x]
2          p [x] ← FIND-SET(p[x])
3 return p [x]
```

Combining these two techniques, the amortized cost per operation is only $O(\alpha(n))$, where $\alpha(n)$ is the inverse of the extremely fast-growing Ackermann function. $\alpha(n)$ is less than 5 for all practical values of n. Thus, the amortized running time per operation is effectively a small constant.

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0\,, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1\,, \end{cases} \qquad A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$$

For any integer j ≥ 1, we have $A_1(j) = 2j + 1$; $A_2(j) = 2^{j+1}(j + 1) – 1$; …

Example: merge 1 & 2