Amortized Analysis: DISJOINT SETS
(CLRS chpts 17 & 21)

Problem: Maintain a dynamic collection of pairwise disjoint sets \( S = \{ S_1, S_2, S_3, ..., S_r \} \) in which each set \( S_i \) has one representative element, \( \text{rep}[S_i] \).

Supported Operations

Make-Set (x): Creates a new set containing the single element \( x \). \( x \) must not be a member of any pre-existing set; \( x \) is the representative.

Union (x, y): Replace \( S_x, S_y \) with \( S_x \cup S_y \) in \( S \) for any \( x, y \) in distinct sets \( S_x, S_y \). Update representative.

Find-Set (x): Returns representative \( \text{rep}[S_x] \) of set \( S_x \) containing \( x \).
Solution 0: Doubly-linked list (unordered)

rep [Si]

Si: [X1] → [X2] → [X3] → ... → [Xk]

Ops:

Make-Set (x) initializes x as lone node: \( \Theta(n) \)
Find-Set (x) walks "left" from x to rep at head: \( \Theta(n) \)
Union (x,y) concatenates lists containing x and y, leaving front as rep: \( \Theta(n) \)

Walk to tail of one end and head of other

-> We can improve on this

Solution 1: Simple Balanced Tree -> Forest

rep [Bi] [X]

Si: [X1]  
   |   
   v   
X2   X3
   
X4   X5
   
X6   X7
   
X8   X9

Make-Set (x): initializes new tree with root node x: \( \Theta(n) \)
Find-Set (x): walks up tree from root to root: \( \Theta(h) = \Theta(g(n)) \)
Union (x,y): concatenates trees containing x and y, with overall root as rep: \( \Theta(h) = \Theta(g(n)) \)

This data structure supports so few ops
-no search, no delete, no comparisons
that these running times are much poorer than necessary.
We will make improvements and find we can do substantially
better than \( \Theta(g(n)) \) or even \( \Theta(gg(n) \ldots g(n)) \), but not quite \( \Theta(n) \)
Improvements

1. Linked-List Improvements
   - Problem here is that need to walk along linked list to find ends
   - Augment the linked list to include pointer from each element to rep. (and from rep. to tail)

Time Analysis

**Find-Set(x)**: \( \Theta(n) \) had been \( \Theta(1) \)

**Union(x, y)**: Concatenates lists and unions rep pointers for list containing \( y \): \( \Theta(n) \) same as before

Amortized Analysis: Find cost per operation by considering cost of large number of operations and divide cost to find cost per operation. Especially useful when some operations (infrequent) occur large overhead costs that wish to spread over other operations (e.g., allocating extra space when table grows

For \( m \) union operations

- Initial sets \( \{e_0, e_1, \ldots, e_{n-1}\} \)
- \( \text{Union}(n-1, n), \text{Union}(n-2, n-1), \ldots, \text{Union}(1, 2) \)
- \( \text{Union}(m-i, m-i+1) \) modifies a list of length \( i \)
- Total cost = \( 2 \leq i \) \( i = \Theta(n^{2}) \) order \( n \) per operation
Further improvement: always add smaller list onto larger (fewer pointers to move)

[Must augment data structure to include weights (# elements)]

New amortized cost:
- Bound on cost of $m$ Union operations and $n$ Make-Set ops
- For each list node $x$, need to bound the number of times its pointer to rep. is updated
- Each time $x$ has pointer updated, length of list containing $x$ grows by a factor $\geq 2$
- Thus, the update can happen at most $\lg n$ times
- Additionally, each union takes $\Theta(1)$ time

Total cost is $O(n \lg n + m)$
3. Forest of trees improvements

- unordered
- possibly unbalanced
- not necessarily binary

- `rep[sc]` is root
- tree only stores elements and parent pointers

```
UNION (x, y)
Step 1: FIND-SET (x)
Step 2: FIND-SET (y)
Step 3: Connect root of y to root of x
```

4. Always attach smaller tree to larger tree (size is # of nodes)
   - depth = $O(\log n)$
   - proof is as before: each time the depth of x increases (by at most 1) the size of the tree grows by $\geq 2$

5. Path Compression

   - After finding the root of $T_y$, make all traversed nodes (on path from $y$ to root) point to root

   - Example: A very shallow tree can become very shallow

   - Can prove that $N$ Make-Set ops and $M$ `FIND-SET` ops and arbitrary number of `UNION`s
     - $O(n + M(1 + \log 2 + \log n))$ using this improvement above.
**Really Impressive Result**

Both improvements at once (small onto big & path compression) lead to spectacular time behavior.

Define $A_k(y) = \begin{cases} j+1 & \text{for } k=0 \\ A_{k-1}(A_{k-1}(y)) & \text{for } k \geq 1 \end{cases}$

$A_k(1)$ grows very fast! ($A_4(1) > 2^{2048}$)

This is Ackermann's function $A$

Now define $\alpha(n) = \min \{ k : A_k(1) \geq n \}$

This function $\alpha$ grows extremely slowly with $n$.

**Theorem:** Any sequence of $n$ operations of the forest data structure with both improvements costs $\Theta(n \alpha(n))$.

**Proof:** CLRS §21.4

Just barely superlinear.

Any given operation heavily depends on size of data structure.
Why is this important?

Maintaining Dynamic Connectivity Information

Suppose a graph is given incrementally as

\text{add-vertex}(v)
\text{add-edge}(u,v)

We need to support connectivity queries

\text{Connected}(u,v) \rightarrow \text{are } u \text{ and } v \text{ in the same connected component?}

\text{add-vertex}(v) \rightarrow \text{make-set}(v)
\text{add-edge}(u,v) \rightarrow \text{union}(u,v)

\text{Connected}(u,v)

\text{if } \text{find-set}(u) = \text{find-set}(v)
\text{then return TRUE}
\text{else return FALSE}

// FALSE!!