Amortized Cost

- Not an algorithm design mechanism per se
- For analyzing cost of algorithms, especially for a sequence of operations
  - when worst case is not typical of average case

Example: Exercise 17.3-6
Implement a queue with two stacks

Why? less blocking on concurrent ops

Queue Operations

enqueue(x, Q)
\[ L.push(x, SI) \]
dequeue(Q)
\[ \text{return} \left( \text{top(SO)} \right) \]
\[ \text{pop(SO)} \]

\[
\begin{array}{c}
\text{Q} \\
G \\
F \\
E \\
D \\
\text{SI} \\
\end{array}
\begin{array}{c}
\text{SO} \\
A \\
B \\
C \\
\end{array}
\]

Queue Content
A Complication

dequeue(Q)
   if empty(SO) then
      while not empty(SI) do
         push(top(SI), SO)
         pop(SI)
   return(top(SO))
   pop(SO)

Complexity

- operation enqueue is always $O(1)$
- in a sequence of $n$ queue ops, one dequeue could be as bad as $O(n)$
- but total cost of sequence isn’t so bad as $O(n^2)$

Consider: each element has $\$4$

- pay $\$1$ for pushing it on SI
- pay $\$2$ for popping from SI and push in SO
- pay $\$1$ for popping from SO.

Claim: can do a sequence of $n$ enqueues +
degrees in $O(n^2)$ time
Algorithm Design & Analysis

Union-Find

Assumptions
1. Disjoint union (no duplicate removal)
2. Elements 1..n
3. Only UNION and FIND (Look-up)
4. Name each set by an elt. in set

Naming by an element is not a limitation
Usually want to know if FIND(i) = FIND(j)

Critical Assumption: the number of ops is proportional to the number of elements

First Hack

Array R[1..n]
R[i] = name of set that contains elt. i

Initially, R[i] = i

Union(i, j): Scan R. For every element k with R[k] = i, set R[k] = j

i 1 2 3 4 5
R[i] 1 2 3 4 5

UNION(2, 4) UNION(4, 3)

Each union costs possibly O(n)

n unions in O(n^2)
n FINDs in O(n)
Improving Efficiency

More efficiency
- find just the elements of the set
- always choose a smaller set to iterate through

R[i] as before

If j is a “name”, then LIST[j] points to
a linked list of elements in j’s set

and SIZE[j] = # elts. in j’s set

Union(i,j) {assume i=R[i], j=R[j] }
1. Assume SIZE[i] ≤ SIZE[j]
2. Traverse LIST[i] and change each element to set j
3. Add LIST[i] to LIST[j]
4. Adjust SIZE[j] ← SIZE[i] + SIZE[i]

Example

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>R[i]</td>
<td>4</td>
<td>24</td>
<td>4</td>
<td>4</td>
<td>24</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>SIZE</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>35</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>LIST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Union(2,4)
**Time Complexity**

**Theorem**: Can execute \( n-1 \) Union operations in \( O(n \cdot \log n) \) time.

- Time for Union is proportional to the number of elements moved when they go from one set to another.
- Charge the cost to the elements moved.

- How many times may a single element be moved? Whenever it changes sets, it goes from a set of size \( k \) to a set of size \( \geq 2k \). Can only move \( \log n \) times.

**Tree-Structured Union-Find**

Data structure: forest of trees, one per set. Almost linear time for \( n \) ops.

- **Union(i, j)** - Have \( i \) point to \( j \).
- **Find(i)** - Traverse up the tree to root.

Graph:

```
          7
         / \  
        6   3
         \   /  
          2  4   
            /  
           1   
```

Root element is "name" of set.
Balancing

Can get bad trees

Keep trees somewhat balanced:
always merge smaller into larger

Lemma: If for every Union, the smaller tree is made a subtree of the larger, then any tree of height $h$ has at least $2^h$ nodes.

Proof: If $h = 0$, then the tree must be a single node.

$2^0 = 1$

Proof, Continued

If $h > 0$, choose height $h$ tree formed by Union's that has fewest nodes has $2^{h-1}$ nodes last tree added.

Look at last tree added, must have height $h-1$

Before the addition, $T_j$ has height $h-2$. Total nodes is $\geq 2^{h-1} + 2^{h-1} = 2^h$
Path Compression

Further improvement on Find time

Algorithm Design & Analysis

Complexity
Need funny functions to prove complexity

\begin{align*}
F(0) &= 1 \\
F(i) &= \frac{1}{2} F(i-2)
\end{align*}

\begin{tabular}{cccccc}
i & 0 & 1 & 2 & 3 & 4 & 5 \\
F(i) & 1 & 2 & 4 & 8 & 16 & 32 \\
\end{tabular}

G is kind of an inverse

\[ G(n) = \text{the least } k \text{ such that } F(k) \geq n \]

In this universe, \( G(n) \leq 5 \) for all practical \( n \)
Complexity 2

Show that a sequence \( \sigma \) of \( c \cdot n \) Union/Find's can be done in \( O(n \log(n)) \).

Assume: Union takes 1 "unit" of time.
Find takes "units" proportional to the depth of the element found.

Definition: The \( u \)-height (union height) of an element relative to an element.

1. Let \( \sigma' \) be \( \sigma \) with Find's removed (no path compression).
2. Construct a forest using \( \sigma' \), call that \( F \).
3. \( u \)-height(v) is height of v in \( F \) from a leaf to v.

Complexity 3

Lemma: There are at most \( \frac{n}{2^r} \) nodes of \( u \)-height \( r \).

Proof: Node of \( u \)-height = \( r \) has at least \( 2^r \) descendents in forest \( F \). (by last theorem)
If \( u \)-height(v) = \( u \)-height(w), sets of descendents are disjoint.
There are at most \( \frac{2^r}{r} \) distinct sets of \( 2^r \) elements, so at most that many nodes of \( u \)-height \( \frac{n}{2^r} \).

Corollary: no node has \( u \)-height greater than \( \log n \).
Lemma: If during execution of \( \sigma \) (not \( \sigma^{-1} \)), \( w \) is a proper descendent of \( v \), then \( u\text{-height}(w) < u\text{-height of } v \)

Proof: \( w \) a proper descendent of \( v \) under \( \sigma \) implies \( w \) is a proper descendent of \( v \) under \( \sigma^{-1} \) by the union that made \( w \) a descendent of \( v \) in \( \sigma \)

Put nodes into groups by \( u\text{-height} \)

\[
\begin{array}{c|c|c|c|c|c|c|c}
 u\text{-height} & 0 & 1 & 2 & 3 & 4 & 5 & 65536 \\hline
\text{group} & 0 & 1 & 2 & 3 & 4 & 5 & \ldots \\
\end{array}
\]

How to count cost of \( \text{Find}'s \) in \( c \cdot n \) operations.

Split cost of \( \text{Find} \) between
1. \( \text{to the Find itself} \)
2. certain nodes used in doing \( \text{the Find} \)

In the end have
\[
\sum \text{over Find} + \sum \text{over nodes}
\]

Suppose \( \text{Find}(i) \) traverses \( k \) nodes
- How to split up \( k \) “units”?
Consider each node \( v \) on path

1. If
   - \( v \) is root
   - \( w \) is root
   - \( v, w \) in different groups
   - \( \text{change } \text{Find one unit} \)

2. If \( v, w \) in same group and \( w \) not the root,
   - \( v \) one unit.

Along any upward path, \( u \)-heights increase

At most \( G(n) \) different groups
so case 1 applies to at most \( G(n) + 1 \) nodes per \( \text{Find} \)

So a \( \text{Find} \) operation gets charged at most \( O(G(n)) \) units
What happens when case 2 applies?

- \( v \) gets a new parent \( x \)
- and \( \text{u-height}(x) > \text{u-height}(w) \)

- How many times can \( v \) get a new parent before its parent is in a higher group, and case 1 applies?
  
  At most \( F(g) - F(g-1) \) times, where
  
  \( g \) is \( \text{u-height of } v \)

Once case 1 applies to \( v \), it always applies in the future.

---

Charges to nodes:

\[
\sum \left( \max \text{ nodes in } g \right)\left( \max \text{ moves for node in } g \right)
\]

\[
N(g) \leq \sum \frac{n}{2^r} \leq \max \text{ nodes of u-height } r
\]

\[
= \frac{n}{2^F(g-1)+1} \cdot \left[ 1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots \right]
\]

\[
\leq 2 \cdot \frac{n}{2^F(g-1)+1} \leq \frac{n}{2^{F(g)}} \leq \frac{n}{F(g)}
\]

\[
\max \text{ moves for a node in a group } \leq F(g)
\]

\[
\leq \sum_{g} \frac{n}{2^{F(g)}} \cdot F(g) = \sum_{g} \frac{n}{2} \leq O(n \log(n))
\]
Graphs

An undirected graph \( G \) is a pair \( (N, E) \)
- \( N \) is a finite set of nodes
- \( E \) is a collection of edge sets called edges

Example
\( N = \{a, b, c, d\} \)
\( E = \{\{a,b\}, \{b,c\}, \{a,c\}, \{b,d\}, \{c,d\}\} \)

Paths

A path in graph \( G = (N, E) \) \( n_i \in N \)
- A sequence \( n_1, n_2, \ldots, n_k \) \( k \geq 2 \) of nodes
- \( \forall n_i, n_{i+1} \in E \) \( 1 \leq i \leq k \)

A path is simple if
- no duplicate nodes

A path is a cycle if
- \( k \geq 3 \)
- can be viewed as a path with \( n_k, n_1 \in E \)
Directed Graphs

In a directed graph (digraph) $E$ is a collection of ordered pairs of nodes.

Example

$E = \{(a,b), (b,c), (a,c), (b,d), (c,d), (d,c)\}$

representation

Time and space complexity depends on how we capture the graph.

Edge list - works if there no isolated node

$O(|V||E|)\{a,b\}, \{b,c\}, \{a,c\}, \{b,d\}, \{c,d\}$

Adjacency List

- $O(|V|)$
  - $a \rightarrow b, c$
  - $b \rightarrow a, c, d$
  - $c \rightarrow a, b, d$
  - $d \rightarrow b, c$
**Representation 2**

**Adjacency matrix**

\[
\begin{array}{cccc}
  & a & b & c & d \\
  a & 1 & 0 & 0 & 0 \\
  b & 1 & 0 & 0 & 0 \\
  c & 1 & 0 & 0 & 0 \\
  d & 0 & 0 & 0 & 0 \\
\end{array}
\]

- **O(1)**
- Undirected and symmetric
- Conventional to put 1’s on diagonal for an undirected graph

**Depth-First Search–Undirected Graphs**

A means to visit all nodes of an undirected graph

1. At node \( x \)
2. Pick edge \( \{x, y\} \)
3. If \( y \) visited, pick another edge
4. Else recursively visit \( y \)
5. If no more edges, we’re done
Algorithm Design & Analysis

Example DFS

1 2 3 7
4 5 6

DFS Algorithm

DFS(v)
mark v
for each edge \{v, w\} do
  if w not marked then
    save \(\langle v, w \rangle\)
    DFS(w)

"saved" edges form the depth-first spanning tree

\[ T = \text{tree edges} \]
\[ B = \mathcal{E} - T \quad \text{back edges} \]

DFS number

DFNum[v]

order in which nodes are initially visited
Time Complexity

Theorem: DFS requires $O(\max(n,e))$ steps on a graph with $n$ nodes and $e$ edges (given as an adjacency list).

Time spent looking for unmarked nodes

DFS(v) is called once for each node.

time spent, apart from recursive calls, is proportional to # of nodes adjacent to $v$

$$\sum (#\text{nodes adjacent to } v) = 2e$$

Back Edges

If \{v, w\} $\in$ B, then $v$ is an ancestor of $w$ in the spanning forest (or vice versa).

If $v$ an ancestor of $w$ in DFS tree, then

$\text{DFNum}[v] < \text{DFNum}[w]$. 