CSE 326: Data Structures
Disjoint Sets – Union/Find

Hal Perkins
Spring 2007
Lectures 19-21

Disjoint Union - Find

- Maintain a set of pairwise disjoint sets.
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Required operations
  - Union – merge two sets to create their union (original sets need not be preserved)
  - Find – determine which set a given item appears in (in particular, be able to quickly tell whether two items are in the same set)

Set Representation

- Maintain a set of pairwise disjoint sets.
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Each set has a unique name, one of its members
  - {3,5,7} , {4,2,8}, {9}, {1,6}

Union

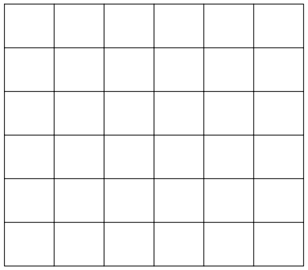
- Union(x,y) – take the union of two sets named x and y
  - {3,5,7} , {4,2,8}, {9}, {1,6}
  - Union(5,1)
    {3,5,7,1,6}, {4,2,8}, {9},

Find

- Find(x) – return the name of the set containing x.
  - {3,5,7,1,6}, {4,2,8}, {9},
  - Find(1) = 5
  - Find(4) = 8

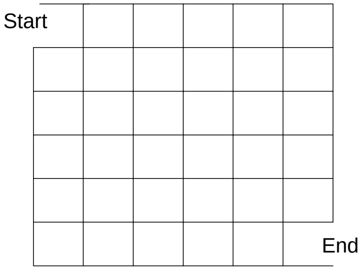
An Example: Building Mazes

- Build a random maze by erasing edges.



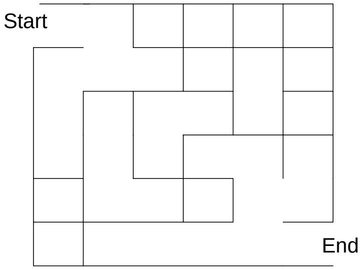
Building Mazes (2)

- Pick Start and End



Building Mazes (3)

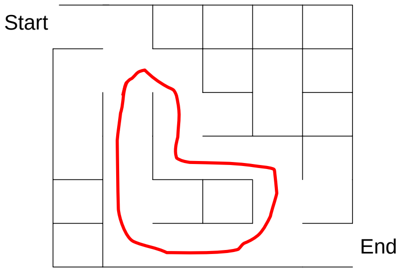
- Repeatedly pick random edges to delete.



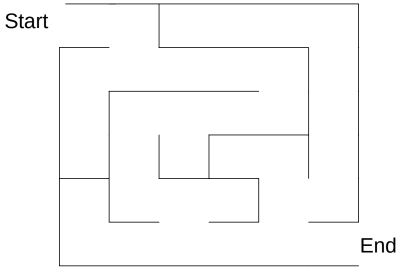
Desired Properties

- None of the boundary is deleted
- Every cell is reachable from every other cell.
- Only one path from any one cell to another (There are no cycles – no cell can reach itself by a path unless it retraces some part of the path.)

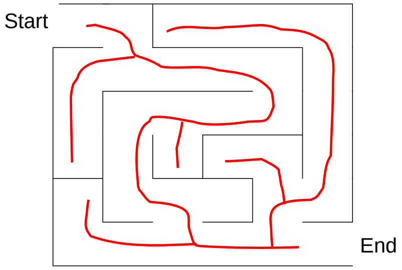
A Cycle



A Good Solution



A Hidden Tree



Number the Cells

We have disjoint sets S ={ {1}, {2}, {3}, {4},… {36} } each cell is a set by itself.
Also a set of all possible edges E ={ (1,2), (1,7), (2,8), (2,3), … } 60 edges total.

Start							
	1	2	3	4	5	6	
	7	8	9	10	11	12	
	13	14	15	16	17	18	
	19	20	21	22	23	24	
	25	26	27	28	29	30	
	31	32	33	34	35	36	End

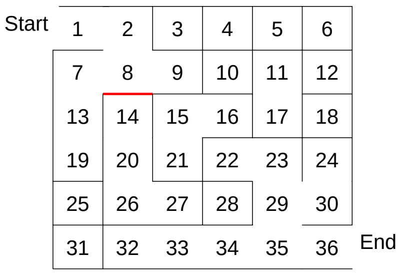
Basic Algorithm

- S = set of sets of connected cells
- E = set of edges not yet examined
- Maze = set of maze edges (initially empty)

```
While there is more than one set in S {
  pick a random edge (x,y) and remove from E
  u := Find(x);
  v := Find(y);
  if u ≠ v then    // removing edge (x,y) connects previously non-
                   // connected cells x and y - leave this edge removed!
     Union(u,v)
  else             // cells x and y were already connected, add this
                   // edge to set of edges that will make up final maze.
     add edge (x,y) to Maze
}
All remaining members of E together with Maze form the maze
```

Example Step

Pick (8,14)



S
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
.
.
{22,23,24,29,30,32
33,34,35,36}

Example

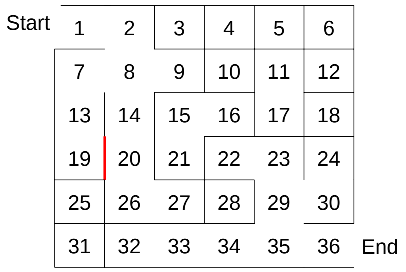
S
{1,2,7,8,9,13,19}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{14,20,26,27}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

Find(8) = 7
Find(14) = 20
→
Union(7,20)

S
{1,2,7,8,9,13,19,14,20 26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

Example

Pick (19,20)

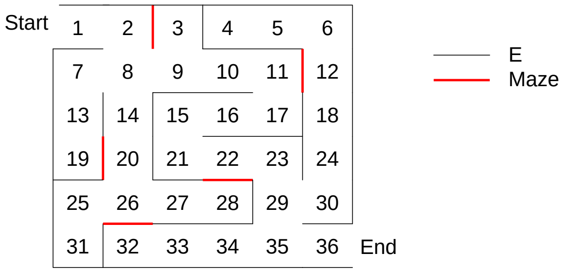


Start	1	2	3	4	5	6
	7	8	9	10	11	12
	13	14	15	16	17	18
	19	20	21	22	23	24
	25	26	27	28	29	30
	31	32	33	34	35	36	End

S
{1,2,7,8,9,13,19
14,20,26,27}
{3}
{4}
{5}
{6}
{10}
{11,17}
{12}
{15,16,21}
.
.
{22,23,24,29,39,32
33,34,35,36}

Example at the End

S
{1,2,3,4,5,6,7,… 36}



Start	1	2	3	4	5	6
	7	8	9	10	11	12
	13	14	15	16	17	18
	19	20	21	22	23	24
	25	26	27	28	29	30
	31	32	33	34	35	36	End

Implementing the DS ADT

- *n* elements,
  Total Cost of: *m* finds, ≤ *n*-1 unions

can there be more unions?

- Target complexity: $O(m+n)$
  i.e. $O(1)$ amortized
- $O(1)$ worst-case for find as well as union would be great, but…
  Known result: both find and union *cannot* be done in worst-case $O(1)$ time

Attempt #1

- Hash elements to a hashtable
- Store set identifier for each element as data

runtime for find:

runtime for union:

runtime for m finds, n-1 unions:

Attempt #2

- Hash elements to a hashtable
- Store set identifier for each element as data
- *Link* all elements in the same set together

runtime for find:

runtime for union:

runtime for m finds, n-1 unions:

Attempt #3

- Hash elements to a hashtable
- Store set identifier for each element as data
- *Link* all elements in the same set together
- Always update identifiers of *smaller* set

runtime for find:

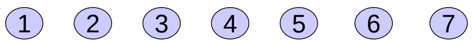
runtime for union:

runtime for m finds, n-1 unions:

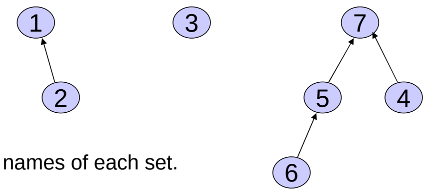
[Read section 8.2]

Up-Tree for Disjoint Union/Find

Initial state:



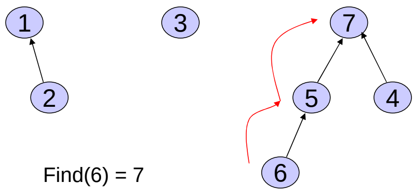
After several Unions:



Roots are the names of each set.

Find Operation

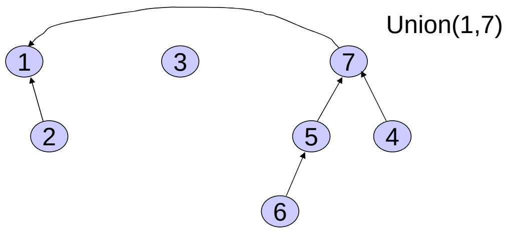
Find(x) - follow x to the root and return the root



Find(6) = 7

Union Operation

Union(x,y) - assuming x and y are roots, point y to x.

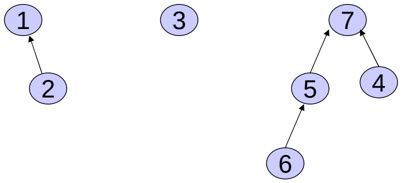


Simple Implementation

- Array of indices

Up[x] = 0 means x is a root.

	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0



Implementation

```
int Find(int x) {

  while(up[x] != 0) {
    x = up[x];
  }

  return x;
}
```

```
void Union(int x, int y) {
  up[y] = x;
}
```

runtime for Union():

runtime for Find():

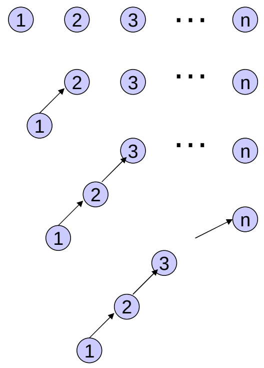
runtime for m Finds and n-1 Unions:

Now this doesn’t look good ☹

Can we do better? *Yes!*

1. Improve union so that *find* only takes $\Theta(\log n)$
   - Union-by-size
   - Reduces complexity to $\Theta(m \log n + n)$
2. Improve find so that it becomes even better!
   - Path compression
   - Reduces complexity to almost $\Theta(m + n)$

A Bad Case



Union(2,1)

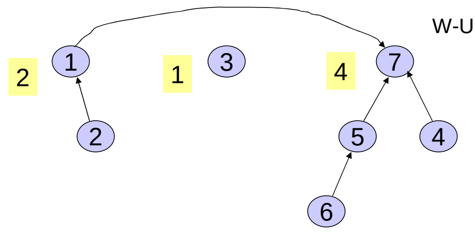
Union(3,2)
:
:

Union(n,n-1)

Find(1) n steps!!

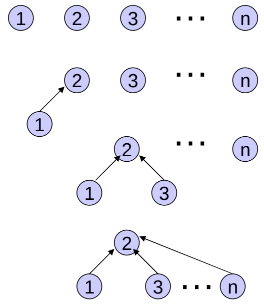
Weighted Union

- Weighted Union
  - Always point the *smaller* (total # of nodes) tree to the root of the larger tree



W-Union(1,7)

Example Again



W-Union(2,1)

W-Union(3,2)
:
:

W-Union(n,2)

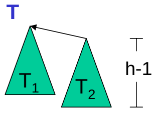
Find(1) constant time

Analysis of Weighted Union

With weighted union an up-tree of height h has weight *at least* 2^h.

- Proof by induction
  - **Basis**: h = 0. The up-tree has one node, $2^0 = 1$
  - **Inductive step**: Assume true for all h' < h.

Minimum weight up-tree of height h formed by weighted unions



$W(T_1) \geq W(T_2) \geq 2^{h-1}$

Weighted union

Induction hypothesis

$W(T) \geq 2^{h-1} + 2^{h-1} = 2^h$

Analysis of Weighted Union (cont)

Let T be an up-tree of weight n formed by weighted union. Let h be its height.

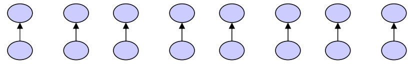
$$n \geq 2^h$$

$$\log_2 n \geq h$$

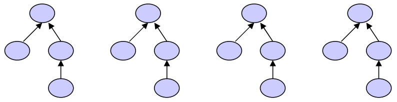
- Find(x) in tree T takes O(log n) time.
  - Can we do better?

Worst Case for Weighted Union

n/2 Weighted Unions

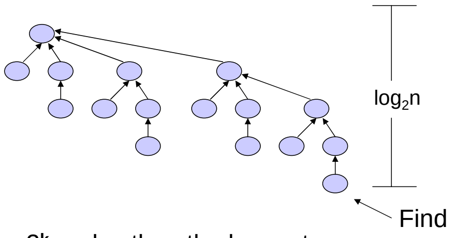


n/4 Weighted Unions



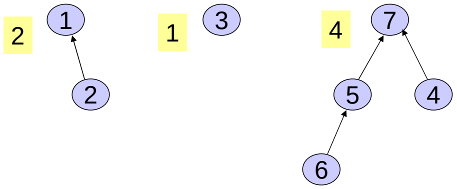
Example of Worst Cast (cont')

After n/2 + n/4 + …+ 1 Weighted Unions:



If there are n = 2^k nodes then the longest path from leaf to root has length k.

Array Implementation



	1	2	3	4	5	6	7
up	-1	1	-1	7	7	5	-1
weight	2		1				4

Weighted Union

```
W-Union(i,j : index){
  //i and j are roots
  wi := weight[i];
  wj := weight[j];
  if wi < wj then
    up[i] := j;
    weight[j] := wi + wj;
  else
    up[j] :=i;
    weight[i] := wi +wj;
}
```

new runtime for Union():

new runtime for Find():

runtime for m finds and n-1 unions =

Union-by-<u>size</u>: Find Analysis

- Complexity of Find: O(max node depth)

- All nodes start at depth 0
- Node depth increases:
  - Only when it is part of smaller tree in a union
  - Only by one level at a time

  Result: **tree size doubles when node depth increases by 1**

Find runtime = O(node depth) =

runtime for m finds and n-1 unions =

Nifty Storage Trick

- Use the same array representation as before

- Instead of storing **`-1`** for a root, simply store **`-size`**

[Read section 8.4, page 276]

How about Union-by-<u>height</u>?

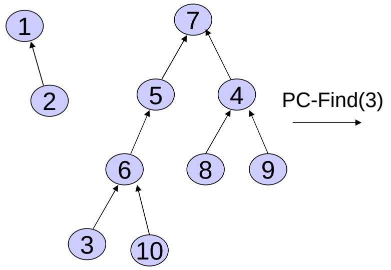
- Can still guarantee O(log *n*) worst case depth

  Left as an exercise!

- Problem: Union-by-height doesn't combine very well with the new find optimization technique we'll see next

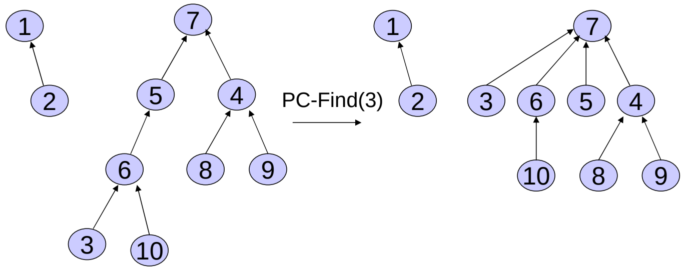
Path Compression

- On a Find operation point all the nodes on the search path directly to the root.



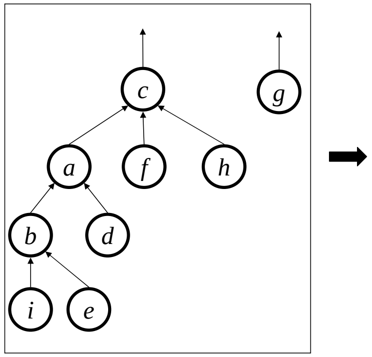
Path Compression

- On a Find operation point all the nodes on the search path directly to the root.

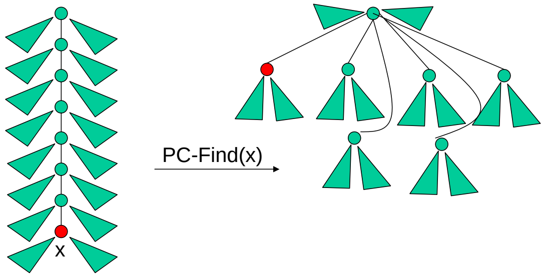


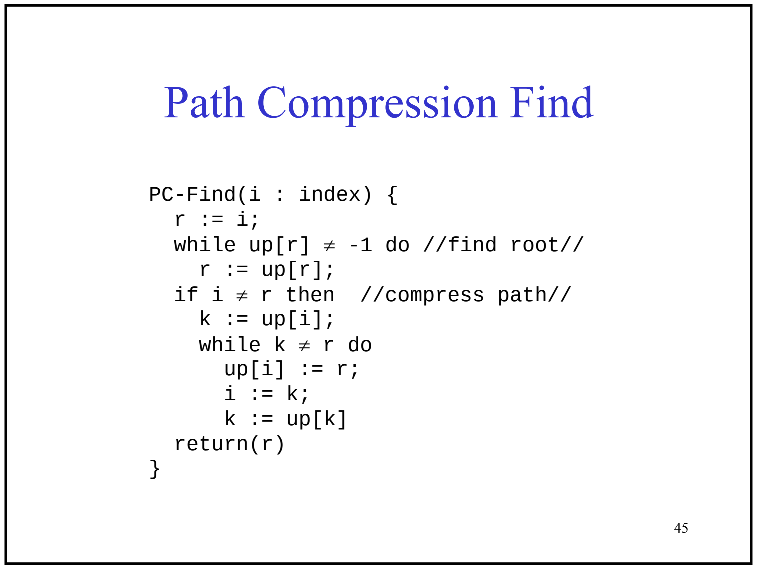
Your Turn

Draw the result of Find(e):



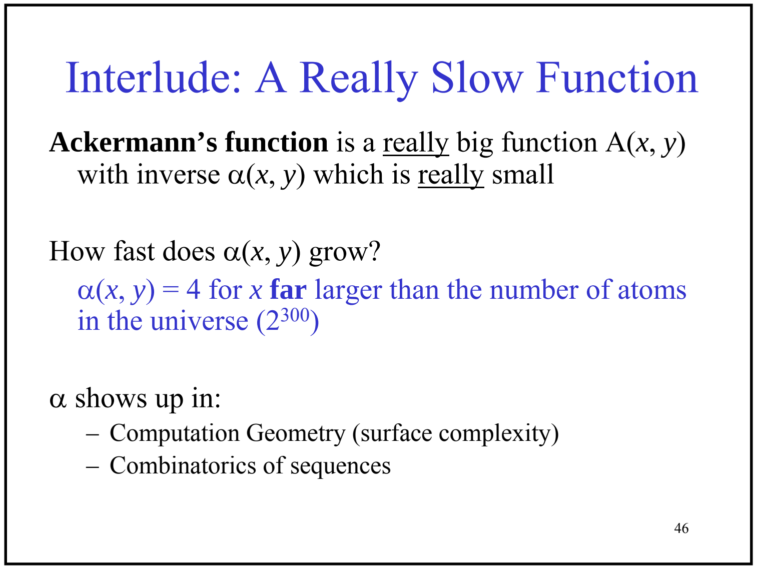
Self-Adjustment Works





Path Compression Find

```
PC-Find(i : index) {
  r := i;
  while up[r] ≠ -1 do //find root//
    r := up[r];
  if i ≠ r then  //compress path//
    k := up[i];
    while k ≠ r do
      up[i] := r;
      i := k;
      k := up[k]
  return(r)
}
```



Interlude: A Really Slow Function

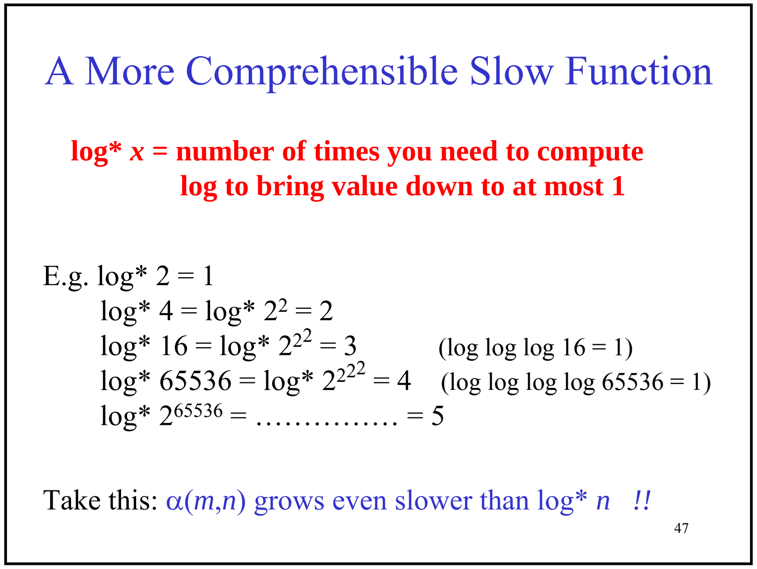
Ackermann's function is a really big function $A(x, y)$ with inverse $\alpha(x, y)$ which is really small

How fast does $\alpha(x, y)$ grow?

$\alpha(x, y) = 4$ for x **far** larger than the number of atoms in the universe (2^{300})

α shows up in:

- Computation Geometry (surface complexity)
- Combinatorics of sequences



A More Comprehensible Slow Function

log* x = number of times you need to compute log to bring value down to at most 1

E.g. $\log^* 2 = 1$

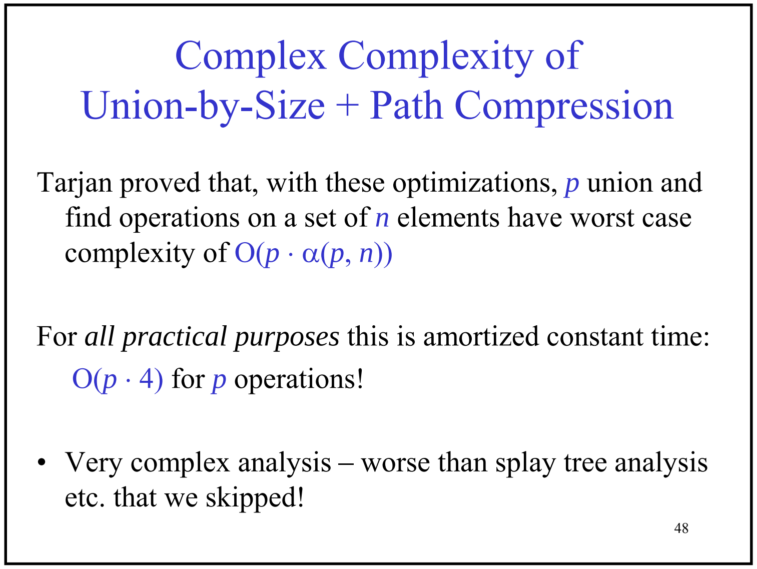
$\log^* 4 = \log^* 2^2 = 2$

$\log^* 16 = \log^* 2^{2^2} = 3$ (log log log 16 = 1)

$\log^* 65536 = \log^* 2^{2^{2^2}} = 4$ (log log log log 65536 = 1)

$\log^* 2^{65536} = \ldots\ldots\ldots\ldots\ldots = 5$

Take this: $\alpha(m,n)$ grows even slower than $\log^* n$!!



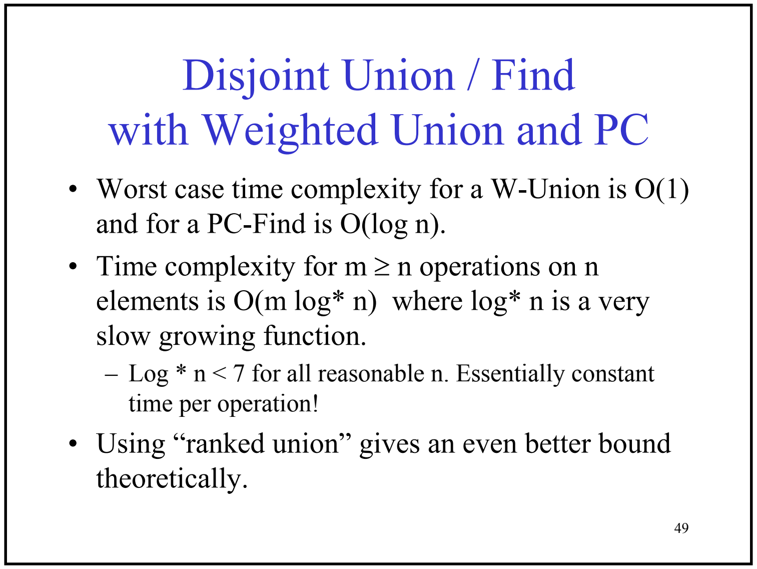
Complex Complexity of Union-by-Size + Path Compression

Tarjan proved that, with these optimizations, p union and find operations on a set of n elements have worst case complexity of $O(p \cdot \alpha(p, n))$

For *all practical purposes* this is amortized constant time:

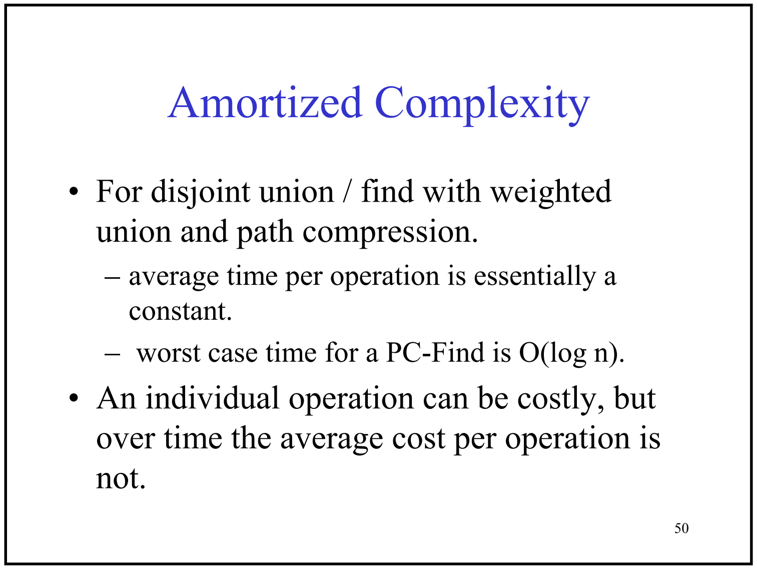
$O(p \cdot 4)$ for p operations!

- Very complex analysis – worse than splay tree analysis etc. that we skipped!



Disjoint Union / Find with Weighted Union and PC

- Worst case time complexity for a W-Union is O(1) and for a PC-Find is O(log n).
- Time complexity for m ≥ n operations on n elements is O(m log* n) where log* n is a very slow growing function.
  - Log * n < 7 for all reasonable n. Essentially constant time per operation!
- Using “ranked union” gives an even better bound theoretically.



Amortized Complexity

- For disjoint union / find with weighted union and path compression.
  - average time per operation is essentially a constant.
  - worst case time for a PC-Find is O(log n).
- An individual operation can be costly, but over time the average cost per operation is not.