http://algs4.cs.princeton.edu

UNION-FIND 并查集

- *dynamic connectivity*
- *quick find*
- *quick union*
- *improvements*
- *applications*

Subtext of today's lecture (and this course)

Steps to developing a usable algorithm.

- Model the problem.
- Find an algorithm to solve it.
- Fast enough? Fits in memory?
- If not, figure out why.
- Find a way to address the problem.
- Iterate until satisfied.

The scientific method.

Mathematical analysis.

http://algs4.cs.princeton.edu

1.5 Union-Find

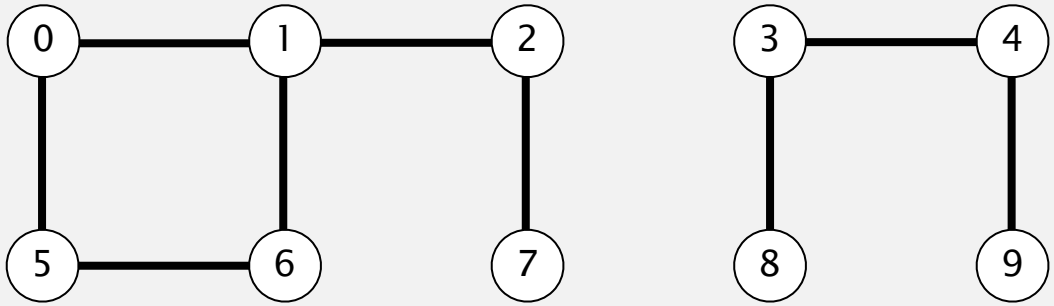
- *dynamic connectivity*
- *quick find*
- *quick union*
- *improvements*
- *applications*

Dynamic connectivity

Given a set of N objects.

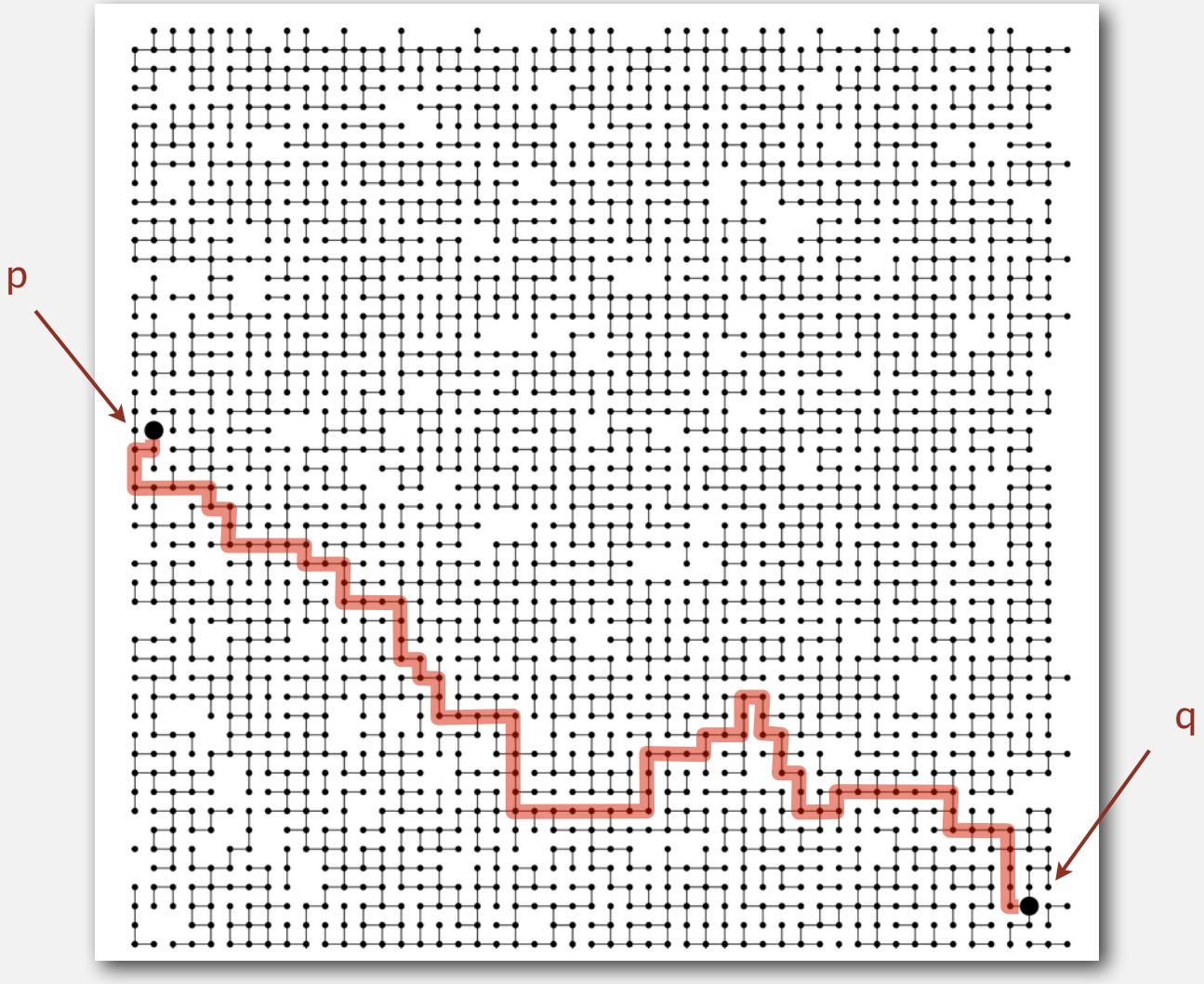
- Union command: connect two objects.
- Find/connected query: is there a path connecting the two objects?

```
union(4, 3)
union(3, 8)
union(6, 5)
union(9, 4)
union(2, 1)
connected(0, 7)   ×
connected(8, 9)   ✔
union(5, 0)
union(7, 2)
union(6, 1)
union(1, 0)
connected(0, 7)   ✔
```



Connectivity example

Q. Is there a path connecting p and q ?



A. Yes.

Modeling the objects

Applications involve manipulating objects of all types.

- Pixels in a digital photo.
- Computers in a network.
- Friends in a social network.
- Transistors in a computer chip.
- Elements in a mathematical set.
- Variable names in Fortran program.
- Metallic sites in a composite system.

When programming, convenient to name objects 0 to N –1.

- Use integers as array index.
- Suppress details not relevant to union-find.

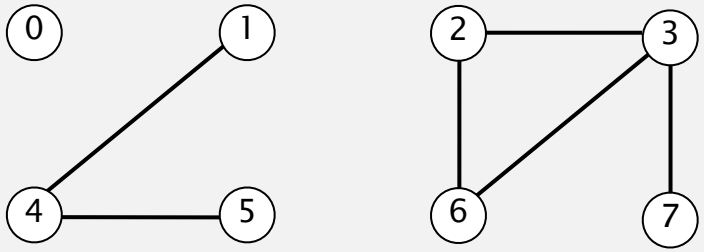
can use symbol table to translate from site names to integers: stay tuned (Chapter 3)

Modeling the connections

We assume "is connected to" is an equivalence relation:

- Reflexive: p is connected to p.
- Symmetric: if p is connected to q, then q is connected to p.
- Transitive: if p is connected to q and q is connected to r, then p is connected to r.

Connected components. Maximal set of objects that are mutually connected.



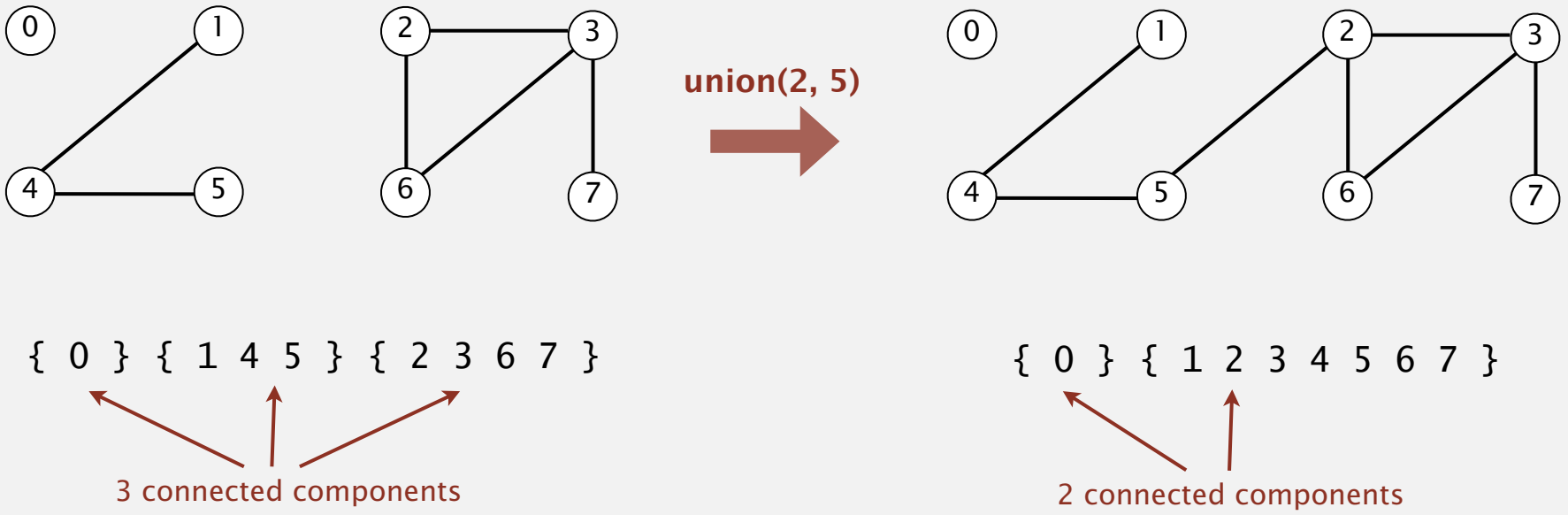
{ 0 } { 1 4 5 } { 2 3 6 7 }

3 connected components

Implementing the operations

Find query. Check if two objects are in the same component.

Union command. Replace components containing two objects with their union.



Union-find data type (API)

Goal. Design efficient data structure for union-find.

- Number of objects N can be huge.
- Number of operations M can be huge.
- Find queries and union commands may be intermixed.

public class UF	
UF(int N)	*initialize union-find data structure with N objects (0 to N – 1)*
void union(int p, int q)	*add connection between p and q*
boolean connected(int p, int q)	*are p and q in the same component?*
int find(int p)	*component identifier for p (0 to N – 1)*
int count()	*number of components*

Dynamic-connectivity client

- Read in number of objects N from standard input.
- Repeat:
  - read in pair of integers from standard input
  - if they are not yet connected, connect them and print out pair

```
public static void main(String[] args)
{
   int N = StdIn.readInt();
   UF uf = new UF(N);
   while (!StdIn.isEmpty())
   {
      int p = StdIn.readInt();
      int q = StdIn.readInt();
      if (!uf.connected(p, q))
      {
         uf.union(p, q);
         StdOut.println(p + " " + q);
      }
   }
}
```

```
% more tinyUF.txt
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7
```

http://algs4.cs.princeton.edu

1.5 Union-Find

- *dynamic connectivity*
- *quick find*
- *quick union*
- *improvements*
- *applications*

http://algs4.cs.princeton.edu

1.5 Union-Find

- *dynamic connectivity*
- *quick find*
- *quick union*
- *improvements*
- *applications*

Quick-find [eager approach]

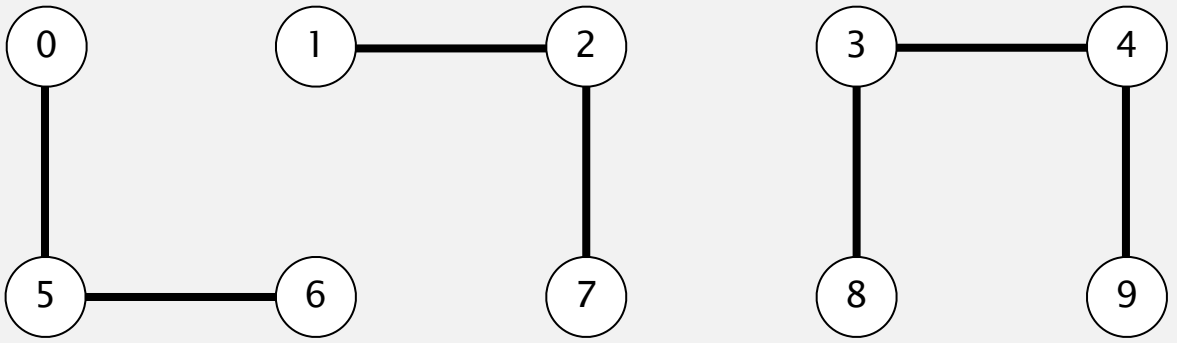
Data structure.

- Integer array `id[]` of length `N`.
- Interpretation: `p` and `q` are connected iff they have the same id.

if and only if

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	1	8	8	0	0	1	8	8

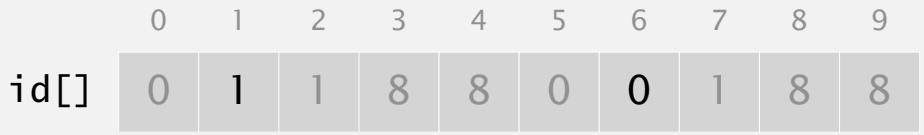
0, 5 and 6 are connected
1, 2, and 7 are connected
3, 4, 8, and 9 are connected



Quick-find [eager approach]

Data structure.

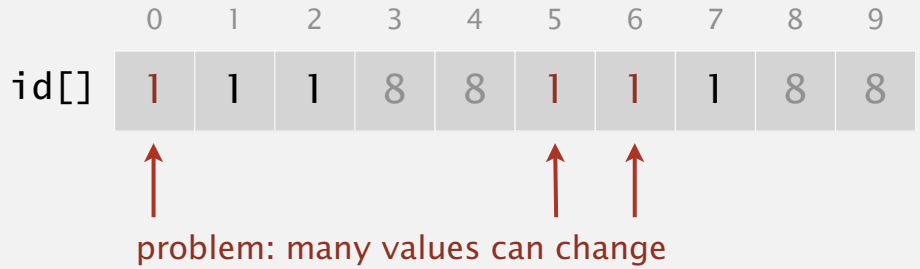
- Integer array `id[]` of length `N`.
- Interpretation: `p` and `q` are connected iff they have the same id.



Find. Check if `p` and `q` have the same id.

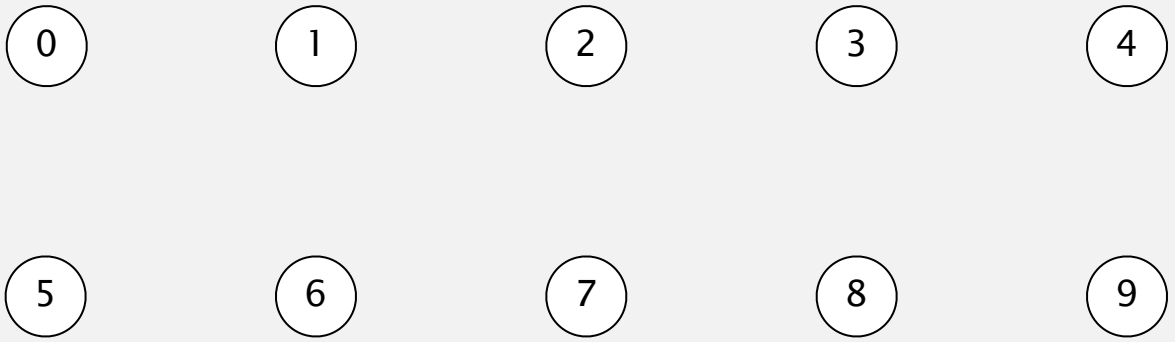
`id[6] = 0; id[1] = 1`
6 and 1 are not connected

Union. To merge components containing `p` and `q`, change all entries whose id equals `id[p]` to `id[q]`.



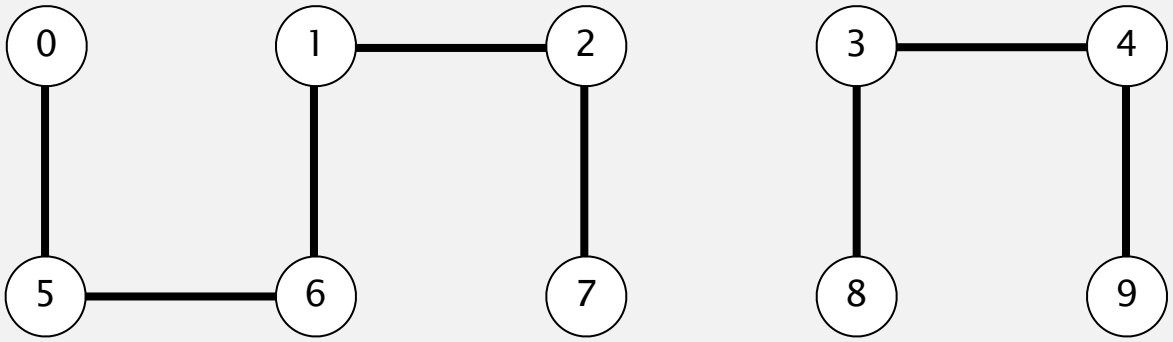
after union of 6 and 1

Quick-find demo



	0	1	2	3	4	5	6	7	8	9
id[]	0	1	2	3	4	5	6	7	8	9

Quick-find demo



	0	1	2	3	4	5	6	7	8	9
id[]	1	1	1	8	8	1	1	1	8	8

Quick-find: Java implementation

```
public class QuickFindUF
{
   private int[] id;

   public QuickFindUF(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++)
         id[i] = i;
   }

   public boolean connected(int p, int q)
   {  return id[p] == id[q];  }

   public void union(int p, int q)
   {
      int pid = id[p];
      int qid = id[q];
      for (int i = 0; i < id.length; i++)
         if (id[i] == pid) id[i] = qid;
   }
}
```

set id of each object to itself
(N array accesses)

check whether p and q
are in the same component
(2 array accesses)

change all entries with id[p] to id[q]
(at most 2N + 2 array accesses)

Quick-find is too slow

Cost model. Number of array accesses (for read or write).

algorithm	initialize	union	find
quick-find	N	N	1

order of growth of number of array accesses

quadratic

Union is too expensive. It takes N^2 array accesses to process a sequence of N union commands on N objects.

http://algs4.cs.princeton.edu

1.5 Union-Find

- *dynamic connectivity*
- *quick find*
- *quick union*
- *improvements*
- *applications*

http://algs4.cs.princeton.edu

1.5 Union-Find

- *dynamic connectivity*
- *quick find*
- *quick union*
- *improvements*
- *applications*

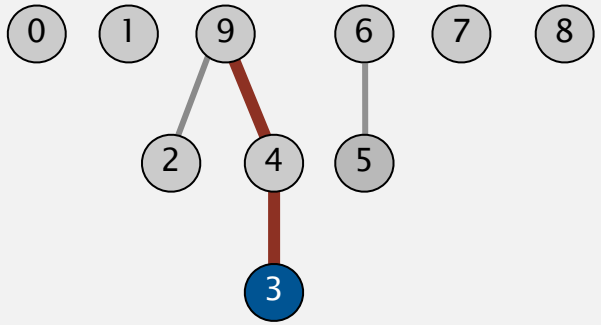
Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of length `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change
(algorithm ensures no cycles)

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9



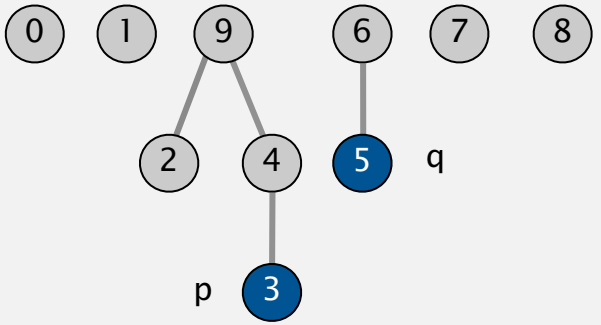
root of 3 is 9

Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of length `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	9



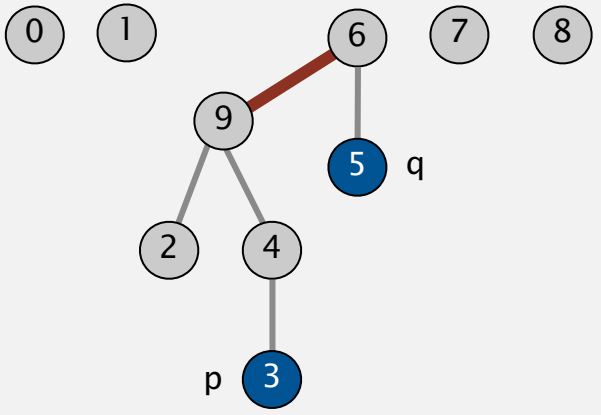
Find. Check if `p` and `q` have the same root.

root of 3 is 9
root of 5 is 6
3 and 5 are not connected

Union. To merge components containing `p` and `q`, set the id of `p`'s root to the id of `q`'s root.

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	9	4	9	6	6	7	8	6

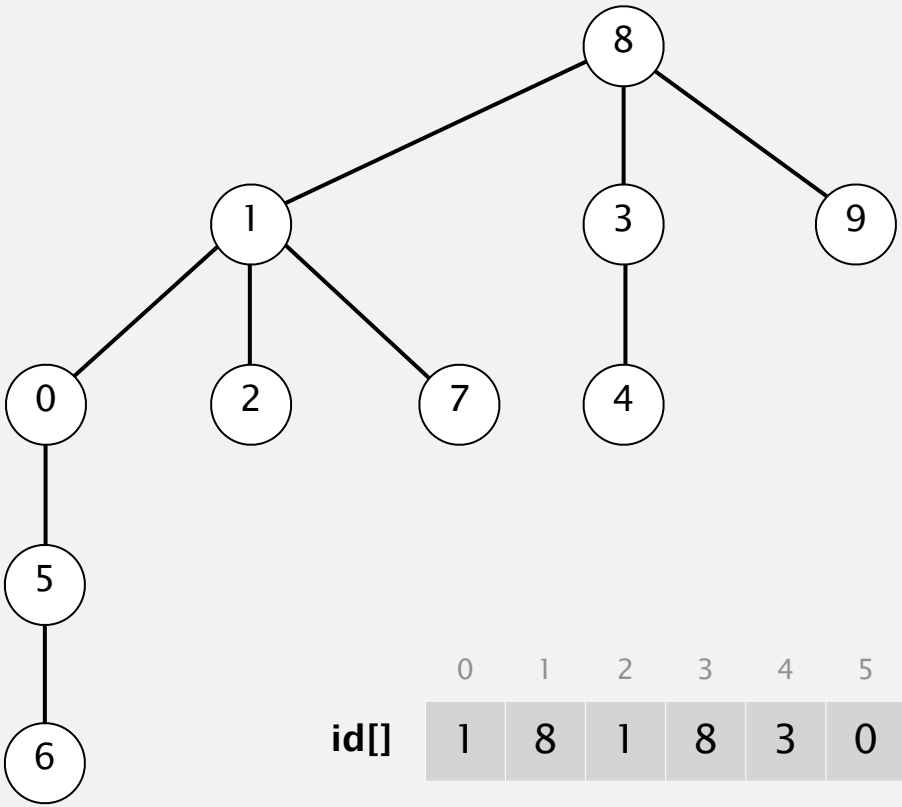
only one value changes



Quick-union demo

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	2	3	4	5	6	7	8	9

Quick-union demo



	0	1	2	3	4	5	6	7	8	9
id[]	1	8	1	8	3	0	5	1	8	8

Quick-union: Java implementation

```
public class QuickUnionUF
{
   private int[] id;

   public QuickUnionUF(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
   }

   private int root(int i)
   {
      while (i != id[i]) i = id[i];
      return i;
   }

   public boolean connected(int p, int q)
   {
      return root(p) == root(q);
   }

   public void union(int p, int q)
   {
      int i = root(p);
      int j = root(q);
      id[i] = j;
   }
}
```

set id of each object to itself
(N array accesses)

chase parent pointers until reach root
(depth of i array accesses)

check if p and q have same root
(depth of p and q array accesses)

change root of p to point to root of q
(depth of p and q array accesses)

Quick-union is also too slow

Cost model. Number of array accesses (for read or write).

algorithm	initialize	union	find
quick-find	N	N	1
quick-union	N	N †	N

← worst case

† includes cost of finding roots

Quick-find defect.

- Union too expensive (N array accesses).
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.

- Trees can get tall.
- Find too expensive (could be N array accesses).

http://algs4.cs.princeton.edu

1.5 Union-Find

- *dynamic connectivity*
- *quick find*
- *quick union*
- *improvements*
- *applications*

http://algs4.cs.princeton.edu

1.5 Union-Find

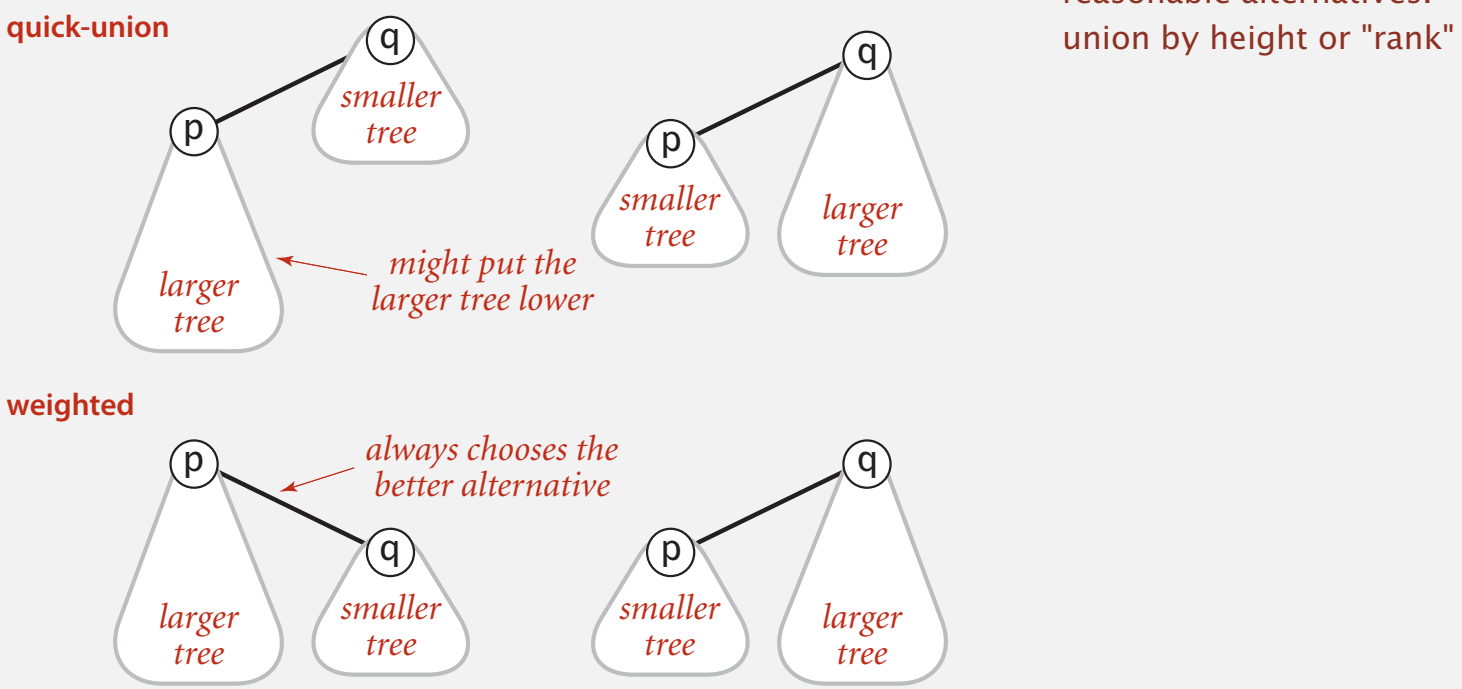
- *dynamic connectivity*
- *quick find*
- *quick union*
- *improvements*
- *applications*

Improvement 1: weighting

Weighted quick-union.

- Modify quick-union to avoid tall trees.
- Keep track of size of each tree (number of objects).
- Balance by linking root of smaller tree to root of larger tree.

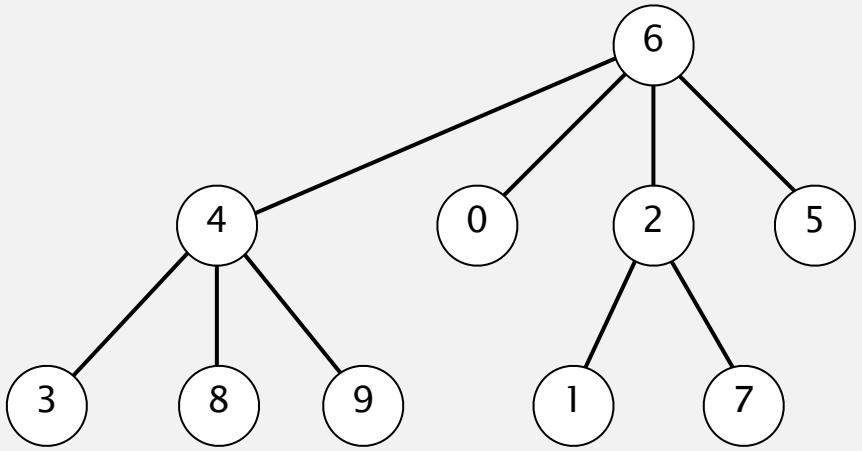
reasonable alternatives:
union by height or "rank"



Weighted quick-union demo

	0	1	2	3	4	5	6	7	8	9
id[]	0	1	2	3	4	5	6	7	8	9

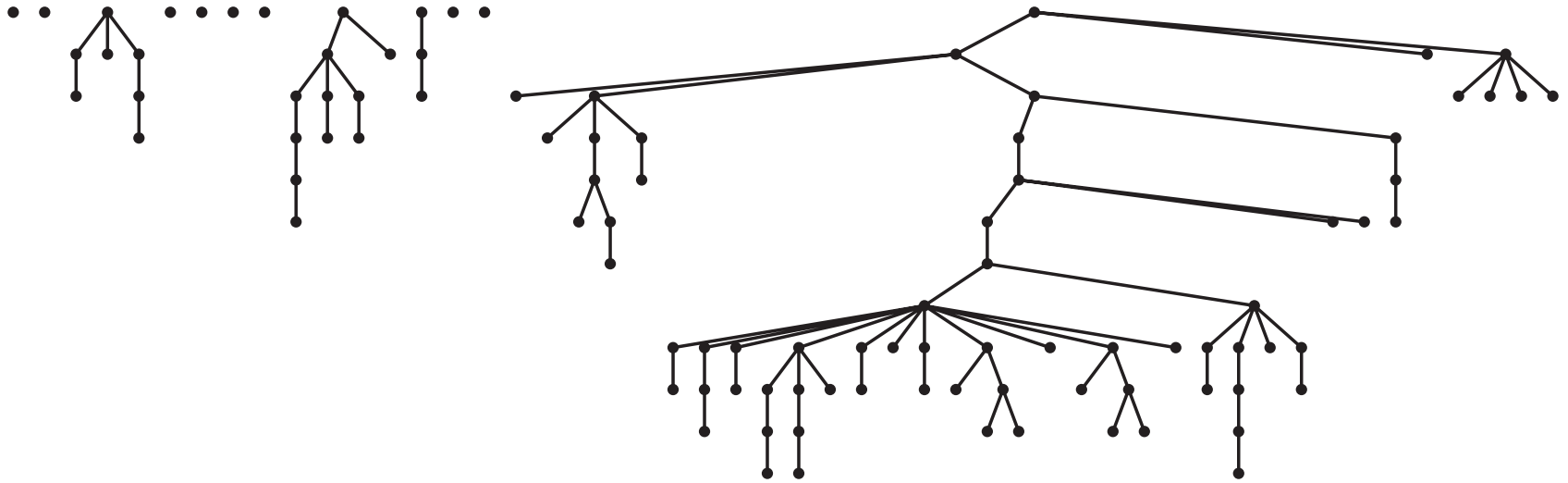
Weighted quick-union demo



	0	1	2	3	4	5	6	7	8	9
id[]	6	2	6	4	6	6	6	2	4	4

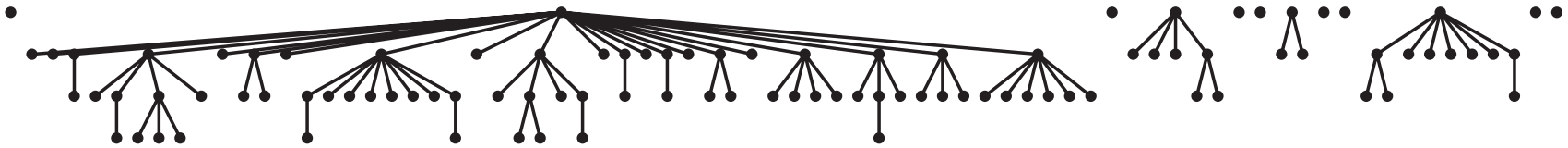
Quick-union and weighted quick-union example

quick-union



average distance to root: 5.11

weighted



average distance to root: 1.52

Quick-union and weighted quick-union (100 sites, 88 `union()` operations)

Weighted quick-union: Java implementation

Data structure. Same as quick-union, but maintain extra array `sz[i]` to count number of objects in the tree rooted at `i`.

Find. Identical to quick-union.

```
return root(p) == root(q);
```

Union. Modify quick-union to:

- Link root of smaller tree to root of larger tree.
- Update the `sz[]` array.

```
int i = root(p);
int j = root(q);
if (i == j) return;
if  (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else                { id[j] = i; sz[i] += sz[j]; }
```

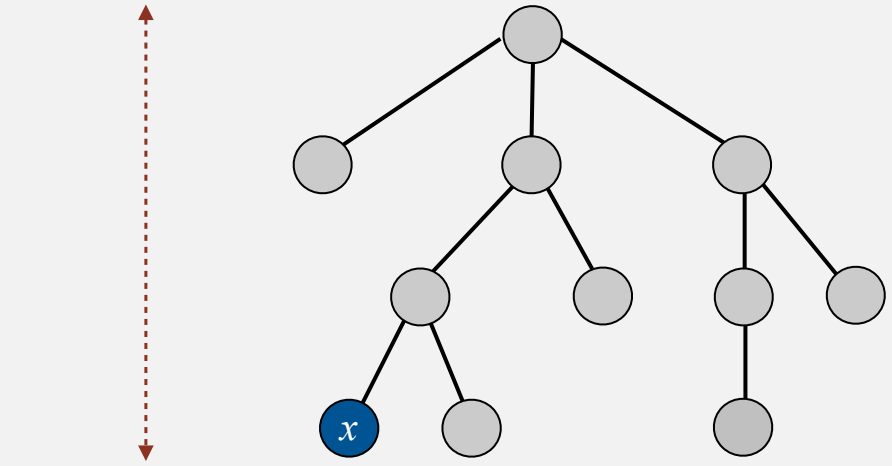
Weighted quick-union analysis

Running time.

- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.

lg = base-2 logarithm

Proposition. Depth of any node x is at most $\lg N$.



N = 10
depth(x) = 3 ≤ lg N

Weighted quick-union analysis

Running time.

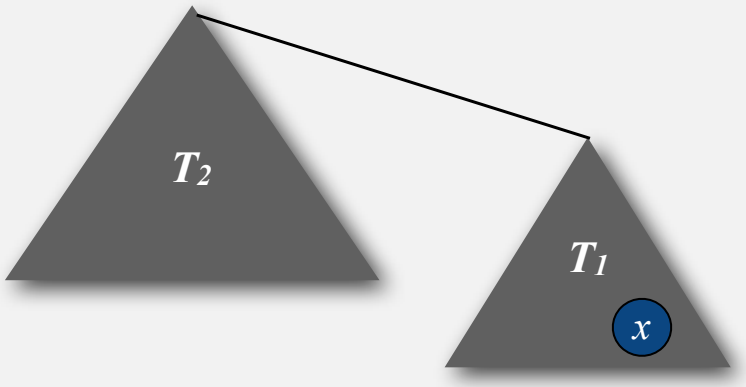
- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.

Proposition. Depth of any node x is at most $\lg N$.

Pf. When does depth of x increase?

Increases by 1 when tree T_1 containing x is merged into another tree T_2.

- The size of the tree containing x at least doubles since $|T_2| \geq |T_1|$.
- Size of tree containing x can double at most $\lg N$ times. Why?



Weighted quick-union analysis

Running time.

- Find: takes time proportional to depth of p and q.
- Union: takes constant time, given roots.

Proposition. Depth of any node x is at most $\lg N$.

algorithm	initialize	union	connected
quick-find	N	N	1
quick-union	N	N †	N
weighted QU	N	lg N †	lg N

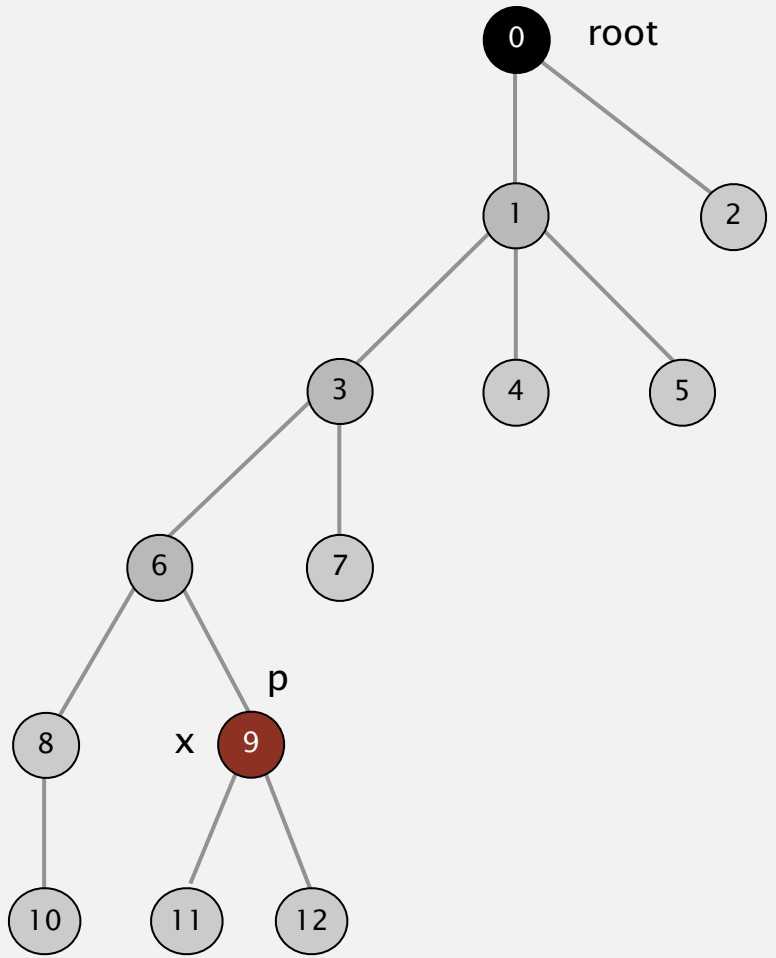
† includes cost of finding roots

Q. Stop at guaranteed acceptable performance?

A. No, easy to improve further.

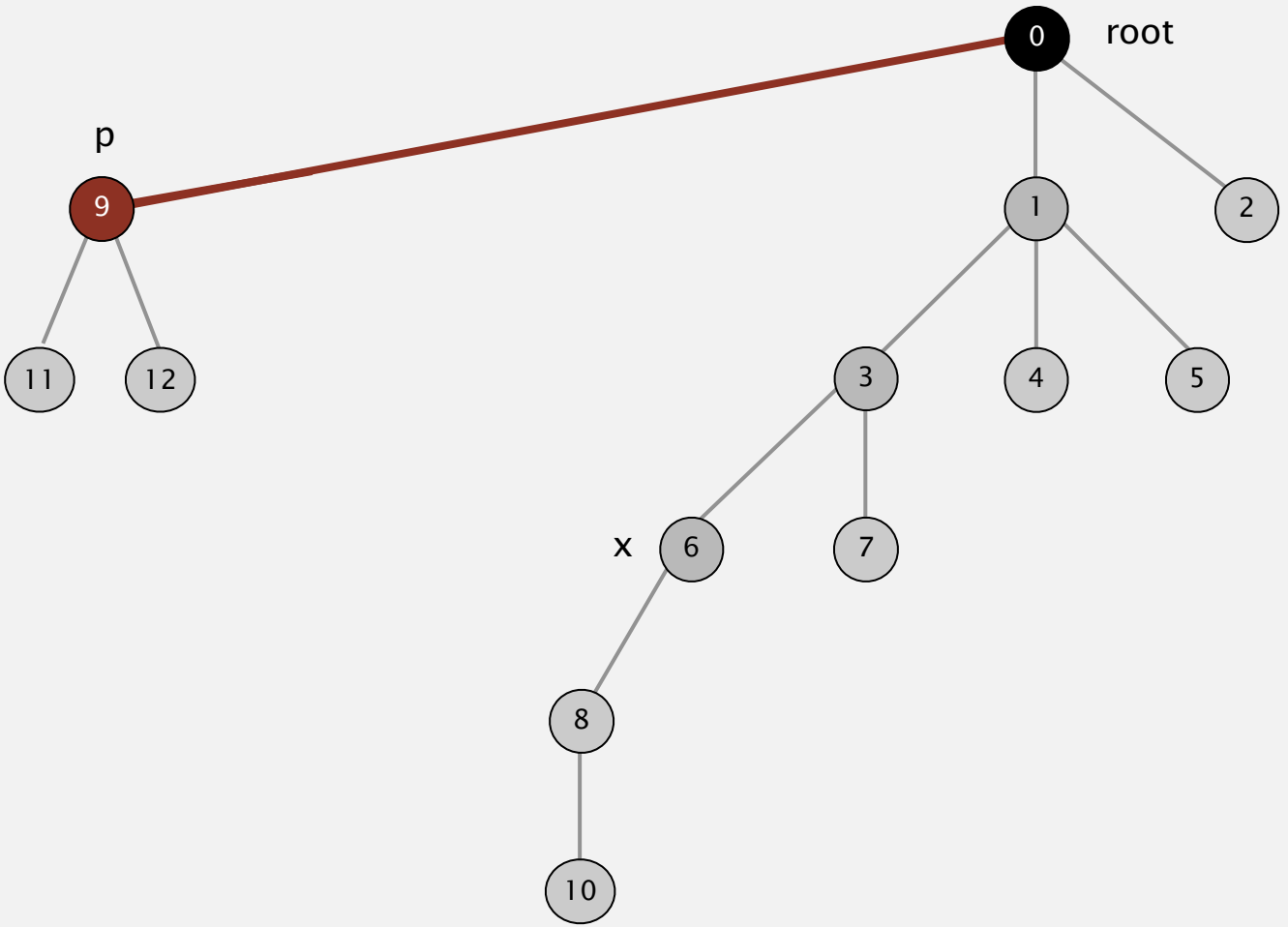
Improvement 2: path compression

Quick union with path compression. Just after computing the root of p, set the id of each examined node to point to that root.



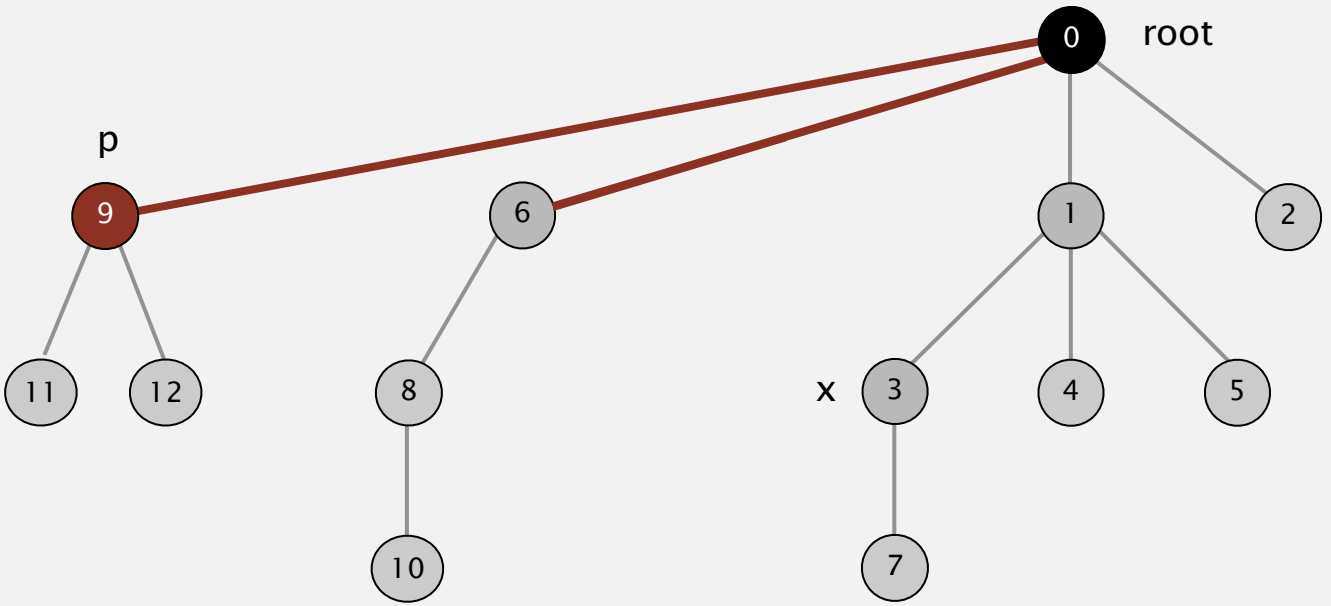
Improvement 2: path compression

Quick union with path compression. Just after computing the root of p, set the id of each examined node to point to that root.



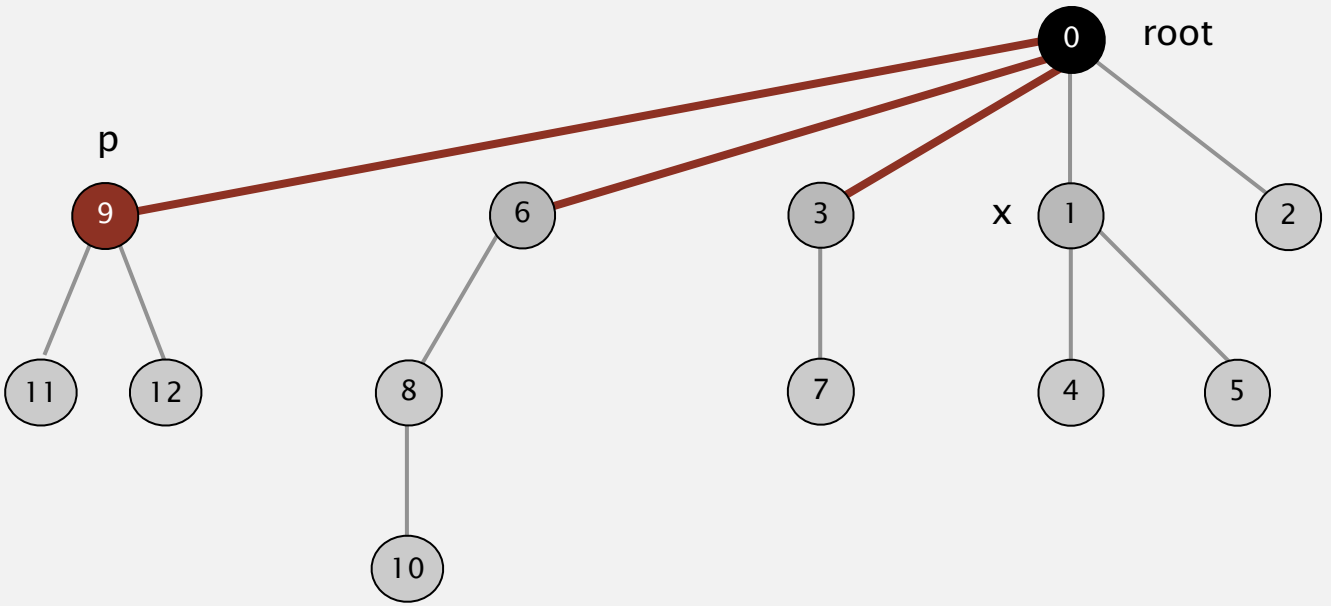
Improvement 2: path compression

Quick union with path compression. Just after computing the root of p, set the id of each examined node to point to that root.



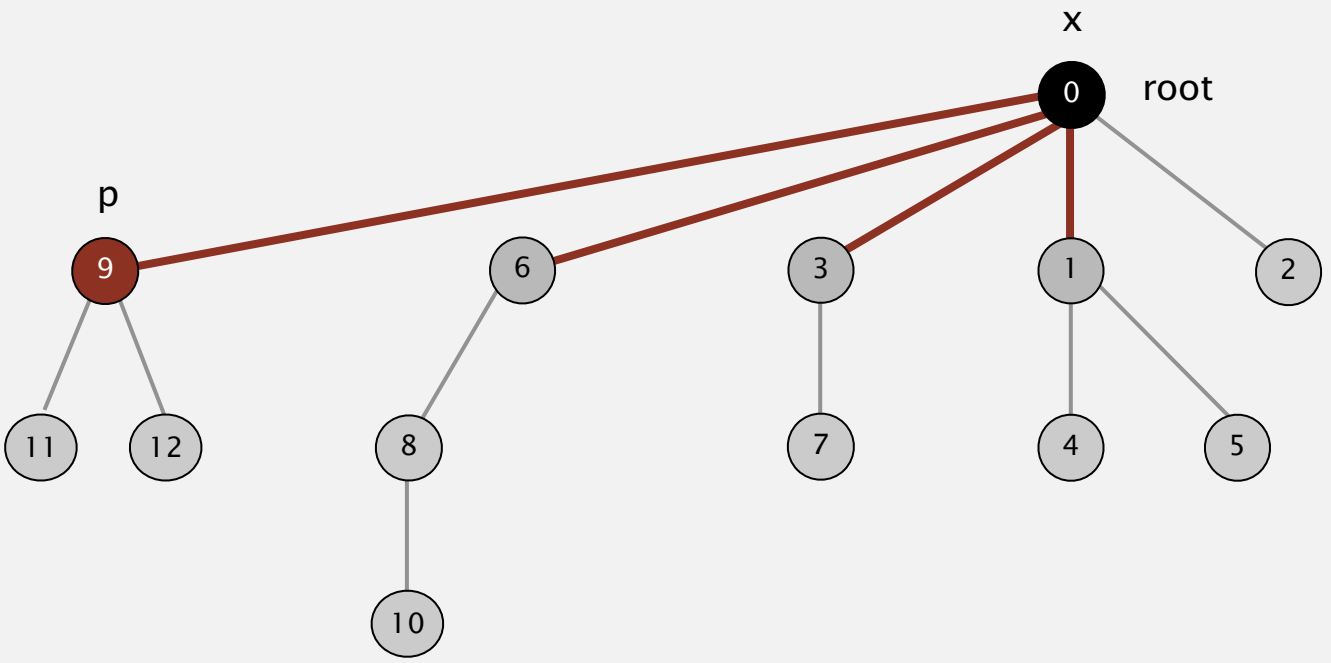
Improvement 2: path compression

Quick union with path compression. Just after computing the root of p, set the id of each examined node to point to that root.



Improvement 2: path compression

Quick union with path compression. Just after computing the root of p, set the id of each examined node to point to that root.



Path compression: Java implementation

Two-pass implementation: add second loop to `root()` to set the `id[]` of each examined node to the root.

Simpler one-pass variant: Make every other node in path point to its grandparent (thereby halving path length).

```
private int root(int i)
{
   while (i != id[i])
   {
      id[i] = id[id[i]];
      i = id[i];
   }
   return i;
}
```

only one extra line of code !

In practice. No reason not to! Keeps tree almost completely flat.

Weighted quick-union with path compression: amortized analysis

Proposition. [Hopcroft-Ulman, Tarjan] Starting from an empty data structure, any sequence of M union-find ops on N objects makes $\leq c(N + M \lg^* N)$ array accesses.

- Analysis can be improved to $N + M\,\alpha(M, N)$.
- Simple algorithm with fascinating mathematics.

N	lg* N
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

iterate log function

Linear-time algorithm for M union-find ops on N objects?

- Cost within constant factor of reading in the data.
- In theory, WQUPC is not quite linear.
- In practice, WQUPC is linear.

Amazing fact. [Fredman-Saks] No linear-time algorithm exists.

in "cell-probe" model of computation

Summary

Bottom line. Weighted quick union (with path compression) makes it possible to solve problems that could not otherwise be addressed.

algorithm	worst-case time
quick-find	M N
quick-union	M N
weighted QU	N + M log N
QU + path compression	N + M log N
weighted QU + path compression	N + M lg* N

M union-find operations on a set of N objects

Ex. [10^9 unions and finds with 10^9 objects]

- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.

http://algs4.cs.princeton.edu

1.5 Union-Find

- *dynamic connectivity*
- *quick find*
- *quick union*
- *improvements*
- *applications*

http://algs4.cs.princeton.edu

1.5 Union-Find

- *dynamic connectivity*
- *quick find*
- *quick union*
- *improvements*
- *applications*

```java
    private int count;
    private int[] parent;
    private int[] rank;
    public UnionFind5(int n) {
        count=n;
        parent=new int[n];
        rank=new int[n];
        for(int i=0;i<count;i++) {
            parent[i]=i;
            rank[i]=1;
        }
    }
    public boolean isConnected(int p,int q) {
        return find(p)==find(q);
    }
    int find(int p) {
        while(p!=parent[p]) {
            //路径压缩，让当前结点指向自己父亲的父亲
            parent[p]=parent[parent[p]];
            p=parent[p];
        }
        return p;
    }
    public void union(int p,int q) {
        int pRoot=find(p);
        int qRoot=find(q);
        if(pRoot==qRoot)
            return;
        if(rank[pRoot]>rank[qRoot]) {
            parent[qRoot]=pRoot;
        }else if(rank[pRoot]<rank[qRoot]) {
            parent[pRoot]=qRoot;
        }else {
            parent[qRoot]=pRoot;
            rank[pRoot]+=1;
        }
    }
```

```java
    private int count;
    private int[] parent;
    private int[] rank;
    public UnionFind6(int n) {
        count=n;
        parent=new int[n];
        rank=new int[n];
        for(int i=0;i<count;i++) {
            parent[i]=i;
            rank[i]=1;
        }
    }
    public boolean isConnected(int p,int q) {
        return find(p)==find(q);
    }
    int find(int p) {
        if(p!=parent[p])
            parent[p]=find(parent[p]);
        return parent[p];
    }
    public void union(int p,int q) {
        int pRoot=find(p);
        int qRoot=find(q);
        if(pRoot==qRoot)
            return;
        if(rank[pRoot]>rank[qRoot]) {
            parent[qRoot]=pRoot;
        }else if(rank[pRoot]<rank[qRoot]) {
            parent[pRoot]=qRoot;
        }else {
            parent[qRoot]=pRoot;
            rank[pRoot]+=1;
        }
    }
```

例1 亲戚

- 若某个家族人员过于庞大，要判断两个是否是亲戚，确实还很不容易，现在给出某个亲戚关系图，求任意给出的两个人是否具有亲戚关系。
- 规定：x和y是亲戚，y和z是亲戚，那么x和z也是亲戚。如果x,y是亲戚，那么x的亲戚都是y的亲戚，y的亲戚也都是x的亲戚。

例1 亲戚

- 数据输入：
- 第一行：三个整数n,m,p，（n<=5000,m<=5000,p<=5000），分别表示有n个人，m个亲戚关系，询问p对亲戚关系。
- 以下m行：每行两个数Mi，Mj，1<=Mi，Mj<=N，表示Ai和Bi具有亲戚关系。
- 接下来p行：每行两个数Pi，Pj，询问Pi和Pj是否具有亲戚关系。
- 数据输出：
- P行，每行一个’Yes’或’No’。表示第i个询问的答案为“具有”或“不具有”亲戚关系。

例1 亲戚

- 样例:
- input.txt
- 6 5 3
- 1 2
- 1 5
- 3 4
- 5 2
- 1 3
- 1 4
- 2 3
- 5 6
- output.txt
- Yes
- Yes
- No

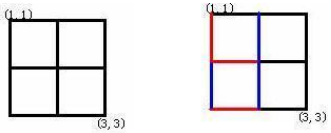
例1 亲戚

- 这个题目是最基础的并查集问题
- 运用基本的并查集工具就可以解决了

例2 格子游戏

【问题描述】

Alice和Bob玩了一个古老的游戏：首先画一个n * n的点阵（下图n = 3接着，他们两个轮流在相邻的点之间画上红边和蓝边：



直到围成一个封闭的圈（面积不必为1）为止，“封圈”的那个人就是赢家。因为棋盘实在是太大了(n <= 200)，他们的游戏实在是太长了！他们甚至在游戏中都不知道谁赢得了游戏。于是请你写一个程序，帮助他们计算他们是否结束了游戏？

【输入格式】

输入数据第一行为两个整数n和m。m表示一共画了m条线。以后m行，每行首先有两个数字(x, y)，代表了画线的起点坐标，接着用空格隔开一个字符，假如字符是"D "，则是向下连一条边，如果是"R "就是向右连一条边。输入数据不会有重复的边且保证正确。

【输出格式】

输出一行：在第几步的时候结束。假如m步之后也没有结束，则输出一行“draw”。

【输入样例】

```
3 5
1 1 D
1 1 R
1 2 D
2 1 R
2 2 D
```

【输出样例】

```
4
```

例2 格子游戏

算法思路：
该题实际上可以看作是一个图型结构，然后需要我们计算可以形成一个连通分量所需要的最少步数，题目中的每一步就相当于一条线段，我们就是要对线段的两个端点进行处理就好了。该端点又与常规的并查集操作中的点有所不同，**其是按坐标表示，常规的是一个数表示**，观察题目可以发现，这些点都是在一个矩阵中，那么这个矩阵中的点可以按照从左到右从上到下的方法排序，该顺序可以可以和左边建立起关系：*(x-1)矩阵大小+y，这样就可以将坐标转化为一个数做该点的索引**。剩下的就是进行并查集操作，检查每次走的步的两个端点是否有相同的祖先结点，如果没有就将两个变为一个集合，如果有那么就找到了一个连通分量，将当前所走过的步直接输出就可以得到结果。

算法描述：
1.判断每次输入的步（x,y,direction）的方向direction，如果向下(即D)，那么该步的两个端点为（x,y）和（x+1,y），如果向右（即R），那么两个端点为（x,y）和（x，y+1）；
2.查找两个端点的祖先结点，如果不相等，那么将其中一个端点的祖先结点改为另一个端点的祖先结点，如果相等，输出当前所走步数。

例3 可爱的猴子(POI2003)

- 树上挂着n只可爱的猴子，编号为1,…,n

  (2<=n<=200 000)。猴子1的尾巴挂在树上，每只猴子有两只手，每只手可以最多抓住一只猴子的尾巴。所有的猴子都是悬空的，因此如果一旦脱离了树，猴子会立刻掉到地上。第0,1,…,m(1<=m<=400000)秒钟每一秒都有某个猴子把它的某只手松开，因此常常有猴子掉到地上。
- 现在请你根据这些信息，计算出每个猴子掉在地上的时间。

例3 可爱的猴子(POI2003)

- 如果把连在一起的猴子看成一个集合，每次松手就是断开了集合之间的某些联系或者直接将一个集合分离成两个。
- 我们要求的是每只猴子第一次脱离猴子1所在集合的时间。
- “分查集”？

例二 可爱的猴子(POI2003)

- 我们不妨反过来想，如果时间从第m秒开始倒流，则出现的情形就是不断有某只猴子的手抓住另一只猴子。
- 则我们要求的就转化成了：每只猴子最开始在什么时候合并到猴子1所在的集合。
- 这样就可以应用并查集了。

例3 可爱的猴子(POI2003)

- 设在第t秒钟，猴子i抓住（实际上是放开）了猴子j，那么此时就将i所在的集合与j所在的集合合并。
- 如果需要合并，并且原先猴子i与猴子 j在同一个集合，那么就将猴子j所在集合的所有猴子掉落的时刻都是t
- 为了枚举某一个集合里的所有元素，我们还需要用一个链表结构与并查集共同维护猴子的集合。

例3 可爱的猴子(POI2003)

- 回顾我们的算法：
  - 并查集的操作时间复杂度为O(nα(n))
  - 每个猴子只有唯一的掉落时间，所以链表中每个元素只枚举一遍，复杂度为O(n)

- 所以算法的总时间复杂度是O(nα(n))

例4 食物链(NOI2001)

- 动物王国中有三类动物A,B,C，这三类动物的食物链构成了有趣的环形。A吃B， B吃C，C吃A。
- 现有N个动物，以1－N编号。每个动物都是A,B,C中的一种，但是我们并不知道它到底是哪一种。
- 有人用两种说法对这N个动物所构成的食物链关系进行描述：
- 第一种说法是“1 X Y”，表示X和Y是同类。
- 第二种说法是“2 X Y”，表示X吃Y。

例4 食物链(NOI2001)

- 此人对N个动物，用上述两种说法，一句接一句地说出K句话，这K句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。
- 1 – 当前的话与前面的某些真的话冲突，就是假话;
- 2 – 当前的话中X或Y比N大，就是假话;
- 3 – 当前的话表示X吃X，就是假话。
- 你的任务是根据给定的N（1<=N<=50,000）和K句话（0<=K<=100,000），输出假话的总数。

例4 食物链(NOI2001)

- 输入文件
- 第一行是两个整数N和K，以一个空格分隔。以下K行每行是三个正整 D，X，Y，两数之间用一个空格隔开，其中D表示说法的种类。
- 若D=1，则表示X和Y是同类。
- 若D=2，则表示X吃Y。
- 输出文件
- 只有一个整数，表示假话的数目。

例4 食物链(NOI2001)

输入文件	对7句话的分析
100 7	
1 101 1	假话
2 1 2	真话
2 2 3	真话
2 3 3	假话
1 1 3	假话
2 3 1	真话
1 5 5	真话

例4 食物链(NOI2001)

- 很显然，对假话条件2、3的处理十分简单，只要在读入数据时作两个条件判断即可解决，题目的主要任务在于处理条件1。
- 从表面上看，条件１的处理似乎也没有什么难度：一个动物无非就是A,B,C三类，而A,B,C之间的食物链关系是一对一单向环形的，也就是说如果已知动物X所属种类和X、Y之间的食物链关系，就一定可以确定出动物Y的种类，同时某个动物具体属于哪一类并不影响本题的结果，而只要求它与其他动物关系的相对位置正确即可。

例4 食物链(NOI2001)

- 于是，我们不妨开3个数组A,B,C，分别记录着三种类的成员，首先假设第一句有效话中的动物X为A类，将其放入数组A，倘若Y与X同类，则把Y也放入A；若Y被X吃，则将Y放入B，如此反复操作所有的有效话，就可以确定每个动物的种类，并容易统计出假话的个数。

例4 食物链(NOI2001)

- 问题似乎已经圆满地解决了，但是，稍稍认真思考就会发现，上面的这个算法存在着重大的错误，是十分片面的。
- 对于一个未知属性的生物我们都采取的是定义为A类型，这样子显然是错的。
- 可见，这个算法只能当每一句话都可直接与此前已知的食物链建立明确关系的时候才能使用。

例4 食物链(NOI2001)

- 通过上面的分析，并查集在本题中的运用已经呼之欲出。
- 一个集合有三类的元素，合并集合的时候，需要对三类元素进行合并。
- 直接开三倍空间，分别用来存储**同类，捕食，天敌**，然后这个题目的意思是，只要这句话不矛盾那就是对的，那怎么叫不矛盾呢，就是只要和前面的条件不冲突就是对的，我们默认前面的条件只要满足题目最基本的条件就是对的

```cpp
#include <iostream>
#include <algorithm>
#include <string.h>
#include <cstdio>
using namespace std;
const int maxn = 1e6 + 10;
int pre[maxn];
int find(int x) //路径压缩
{
    return x == pre[x] ? x : find(pre[x]); //三目运算符
}
void join(int x, int y) //合并操作
{
    pre[find(x)] = find(y);
}
int main()
{
    int n, k, d, x, y;
    int ans = 0;
    scanf("%d%d", &n, &k);
    for (int i = 1; i <= 3 * n; ++i) //开三倍集合，一倍用于存储同类，二倍用于存储捕食，三倍用于存储天敌
        pre[i] = i;
    for (int i = 1; i <= k; ++i)
    {
        scanf("%d%d%d", &d, &x, &y);
        {
            if (x > n || y > n || d == 2 && x == y) //如果不满足条件直接+1不用后面的判断
            {
                ++ans;
                continue;
            }
            if (d == 1) //同类
            {
                if (find(x) == find(y + n) || find(x) == find(y + 2 * n)) //如果x和y的捕食者是同类或者x与y的天敌是同类那
                    ++ans;
                else
                { //合并同类
                    join(x, y);
                    join(x + n, y + n);
                    join(x + 2 * n, y + 2 * n);
                }
            }
            else //捕食
            {
                if (find(x) == find(y + 2 * n) || find(x) == find(y)) //如果x与y是同类或者x与y的天敌是同类那就不满足捕食
                    ++ans;
                else //合并捕食
                {
                    join(x, y + n);
                    join(x + n, y + 2 * n);
                    join(x + 2 * n, y);
                }
            }
        }
    }
    printf("%d\n", ans);
}
```