CS 226 Lecture 1: Introduction

R. Sedgewick
rs@cs.princeton.edu

Intermediate-level survey course
- prerequisite: COS 126
- programming/problem solving

Algorithm: method for solving a problem
Data structure: a way to store information

Efficient algorithms use good data structures

Types of algorithms considered
- Sorting
- Searching
- Strings
- Geometric
- Graph
- Mathematical
- Other

Why study algorithms?

Using a computer?

- want it to go faster
- want it to process more data
- want to do something that would otherwise be impossible

Technology improves things by a constant factor

Good algorithm design can do better

- Abacus
- Slide Rule
- Calculator
- Computer
- Supercomputer

A bad algorithm on supercomputer can be worse than a good algorithm on abacus

Algorithms as a field of study

- old enough that basics are known
- new enough that new discoveries arise
- burgeoning application areas
  - science
  - engineering
  - commercial
- philosophical implications

.

.

.

Analysis of algorithms

Compare algorithms by comparing estimated costs
Refine and verify estimates for the best ones

N: size of the input
Running time usually proportional to

- 1
- log N
- N
- N log N
- N^2
- 2^N

Other functions sometimes arise

- loglog N [log(log N)]
- log* N number of logs until 1 reached

Often can write down a formula for running time

* worst case (guarantee)
* average case (prediction)

Analysis depends on detailed information

* costs of basic operations
* properties of input

Suppress details in approximate analysis

- "Within a constant factor" O-notation
- Predict time for 2N from time for N

Example: Online connectivity

Input:

    sequence of pairs of integers (p, q)
    p "is connected to" q

Output:

    nothing if p and q are already connected
    (p, q) otherwise

in	out	evidence
3 4	3 4	
4 9	4 9	
8 0	8 0	
2 3	2 3	
5 6	5 6	
2 9		(2--3--4--9)
5 9	5 9	
7 3	7 3	
4 8	4 8	
5 6		(5--6)
0 2		(2--3--4--8--0)
6 1	6 1	

Example of application

    integers represent computers
    pairs represent network connections
    can p and q communicate through network?

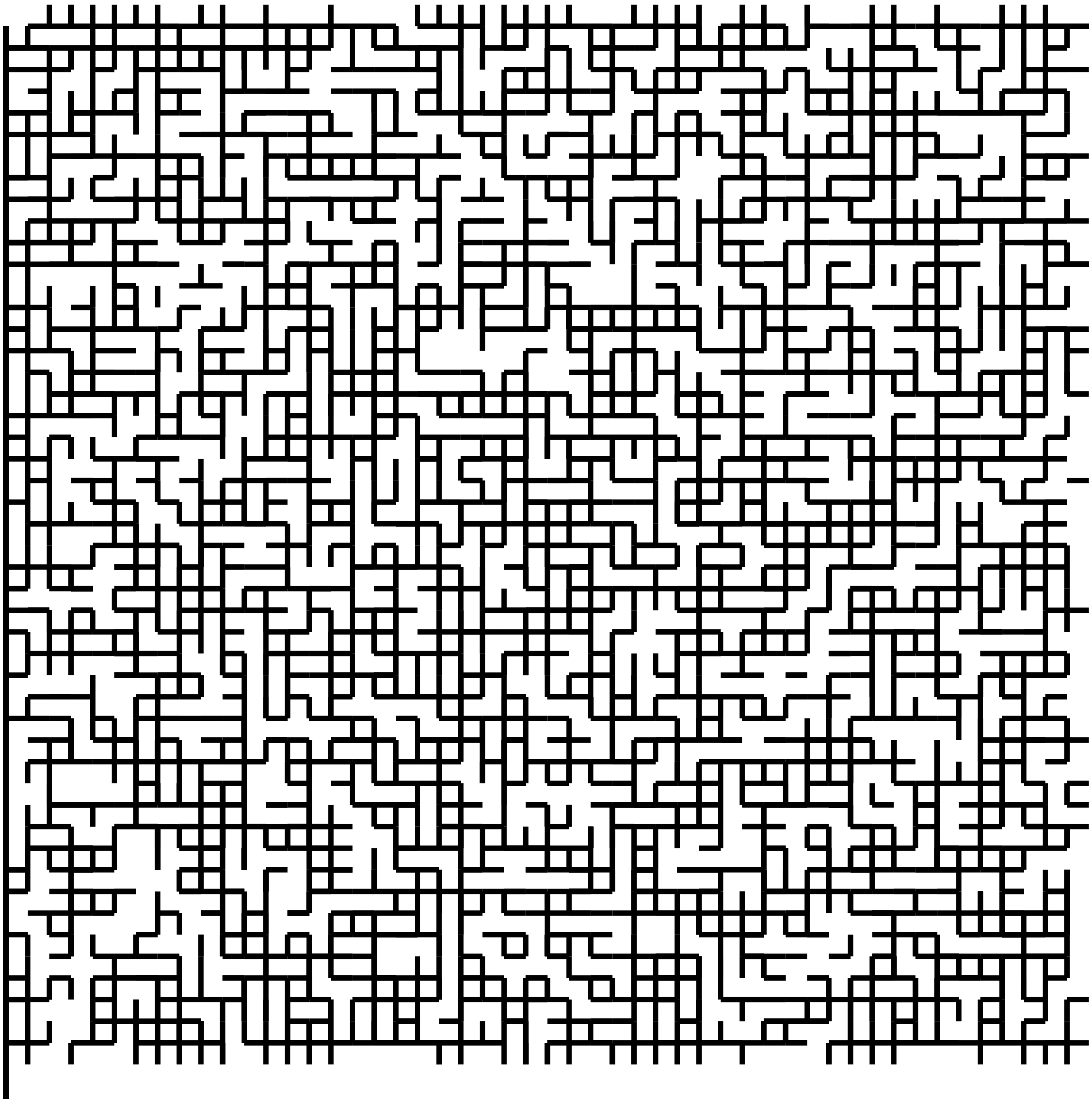
Network connection example

Number of nodes and edges can be huge
for practical problems

Ex:

- telephone network
- computer chip

Disconnected piece may be hard to spot
particularly for a computer!



Quick-find algorithm

Maintain array with names for components
    if i and j are connected,
    id[i] and id[j] are the same

To maintain this property for p-q connection
    ignore if id[p] = id[q]
    change all entries with p's id to q's id

```
main(int argc, char *argv[])
  { int i, p, q, t, N = atoi(argv[1]);
     int *id = malloc(N*sizeof(int));
     for (i = 0; i < N; i++) id[i] = i;
     while (scanf("%d %d ", &p, &q) != EOF)
        {
           if (id[p] == id[q]) continue;
           t = id[p];
           for (i = 0; i < N; i++)
              if (id[i] == t) id[i] = id[q];
           printf(" %d %d ", p, q);
        }
  }
```

QUICK-FIND name due to constant-time test
    to find out if edge makes a new connection
SLOW-UNION?

Quick-find example

Trace contents of id array
after each connection

Ex:

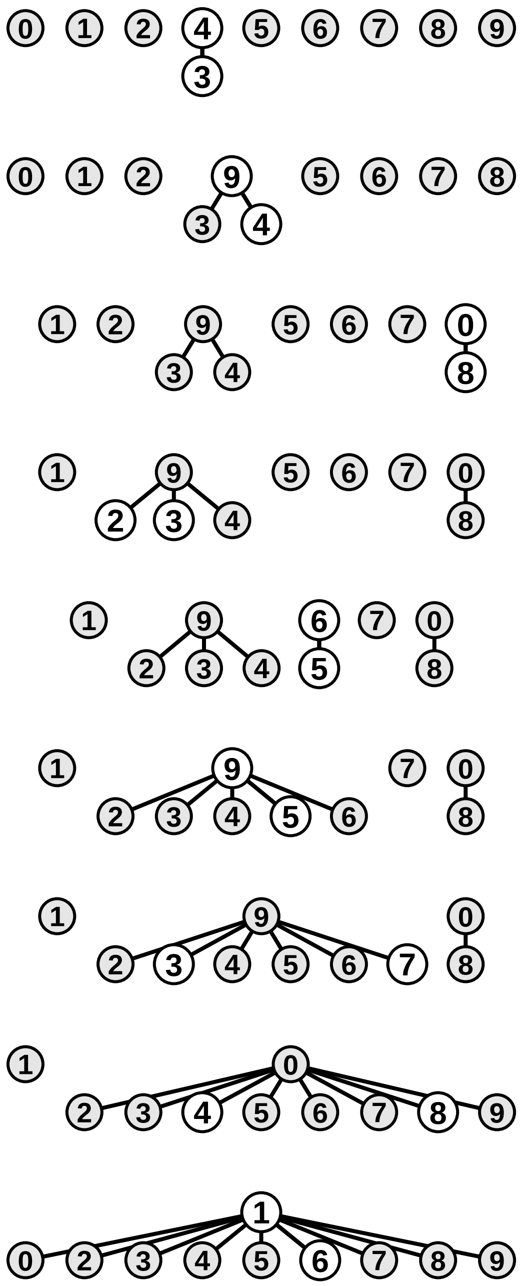
3	**4**	**0**	**1**	**2**	**4**	**4**	**5**	**6**	**7**	**8**	**9**
4	**9**	**0**	**1**	**2**	**9**	**9**	**5**	**6**	**7**	**8**	**9**
8	**0**	**0**	**1**	**2**	**9**	**9**	**5**	**6**	**7**	**0**	**9**
2	**3**	**0**	**1**	**9**	**9**	**9**	**5**	**6**	**7**	**0**	**9**
5	**6**	**0**	**1**	**9**	**9**	**9**	**6**	**6**	**7**	**0**	**9**
5	**9**	**0**	**1**	**9**	**9**	**9**	**9**	**9**	**7**	**0**	**9**
7	**3**	**0**	**1**	**9**	**9**	**9**	**9**	**9**	**9**	**0**	**9**
4	**8**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
6	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Problem with quick-find:
- change too many entries for union
- too slow for huge problems

Quick-find example

Use graphical representation

- numbers in nodes
- node i points up to id[i]
- top nodes represent components



Problem size and computation time

```
Rough standard for 2000
     10^9 operations per second
     10^9 words of memory
     touch each word in approximately 1 second
          (roughly unchanged since at least 1950)

Bigger memories --> bigger problems

Ex: huge problem for quick-find
     10^10 edges connecting 10^9 nodes
          (edges need not fit in memory)
     examples: telephone network, computer chip

Quick-find might take 10^20 operations
     relabel every node (10 ops) for every edge
     3000 years of computer time (too much)

Use rough estimate
     number of edges and nodes both O(N)
     running time of quick find O(N^2)

Gap grows as scale increases
          new computer may be 10 times faster
          and have 10 times as much memory
          but (with quadratic algorithm)
               takes 10 times as long to finish!
```

Quick-union algorithm

Maintain array with names for components
    if i and j are connected,
    (id[i])* and (id[j])* are the same
    where (id[i])* = id[id[id[...id[i]]]]
        (go until it doesn't change)

To maintain this property for p-q connection
    ignore if (id[p])* = (id[q])*
    set id[i] to j

```
main(int argc, char *argv[])
  { int i, j, p, q, t, N = atoi(argv[1]);
    int *id = malloc(N*sizeof(int));
    for (i = 0; i < N; i++) id[i] = i;
    while (scanf("%d %d ", &p, &q) != EOF)
      {
        i = p; j = q;
        while (i != id[i]) i = id[i];
        while (j != id[j]) j = id[j];
        if (i == j) continue;
        id[i] = j;
      }
  }
```

QUICK-UNION: constant-time to make a new connectio
SLOW-FIND?

Quick-union example

Graphical representation exposes tree structure
FIND operation is traveling up tree to root
(check if two nodes hit the same root)
UNION operation links one tree to another

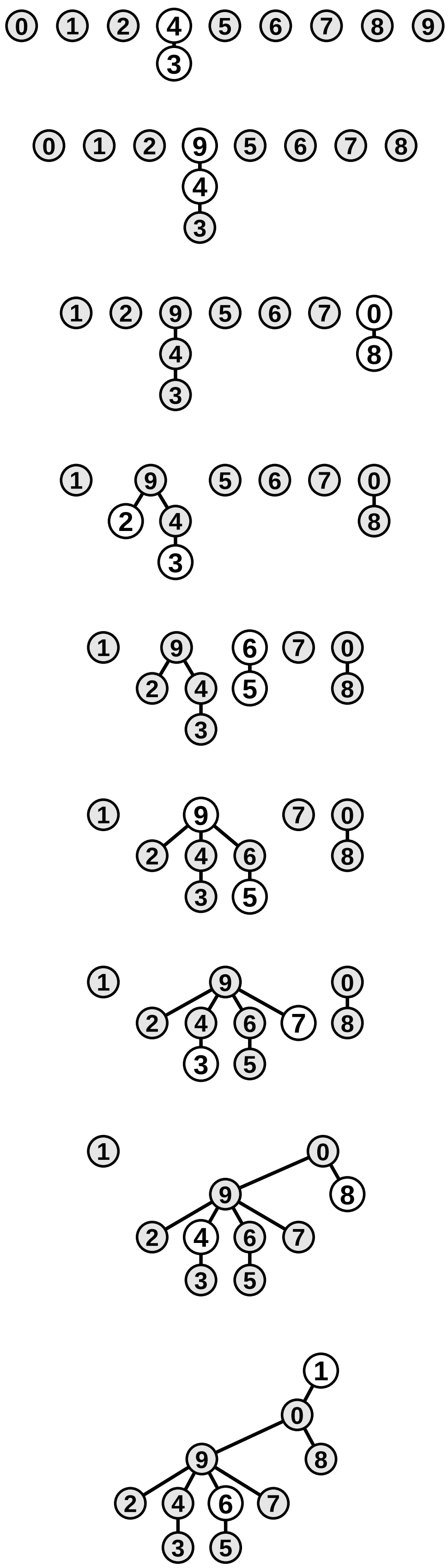
Ex:

```
3 4  0 1 2 4 4 5 6 7 8 9
4 9  0 1 2 4 9 5 6 7 8 9
8 0  0 1 2 4 9 5 6 7 0 9
2 3  0 1 9 4 9 5 6 7 0 9
5 6  0 1 9 4 9 6 6 7 0 9
5 9  0 1 9 4 9 6 9 7 0 9
7 3  0 1 9 4 9 6 9 9 0 9
4 8  0 1 9 4 9 6 9 9 0 0
6 1  1 1 9 4 9 6 9 9 0 0
```

Faster than quick-find for "random" input
(is real input "random"?)

Problem with quick-union:
- paths to root can grow too long
- no guarantee for huge problems

Quick-union example



Weighted quick-union algorithm

Modify quick-union to avoid imbalance

keep track of size of each component

balance by linking small one below large one

```
main(int argc, char *argv[])
  { int i, j, p, q, t, N = atoi(argv[1]);
    int *id = malloc(N*sizeof(int));
    int *sz = malloc(N*sizeof(int));
    for (i = 0; i < N; i++) id[i] = i;
    for (i = 0; i < N; i++) sz[i] = 1;
    while (scanf("%d %d ", &p, &q) != EOF)
      {
        for (i = p; i != id[i]; i = id[i]) ;
        for (j = q; j != id[j]; j = id[j]) ;
        if (i == j) continue;
        if (sz[i] < sz[j])
             { id[i] = j; sz[j] += sz[i]; }
        else { id[j] = i; sz[i] += sz[j]; }
        printf(" %d %d ", p, q);
      }
  }
```

Fast enough?

Weighted quick-union example

Avoids long paths in trees

Ex:

```
3 4  0 1 2 3 3 5 6 7 8 9
4 9  0 1 2 3 3 5 6 7 8 3
8 0  8 1 2 3 3 5 6 7 8 3
2 3  8 1 3 3 3 5 6 7 8 3
5 6  8 1 3 3 3 5 5 7 8 3
5 9  8 1 3 3 3 3 5 7 8 3
7 3  8 1 3 3 3 3 5 3 8 3
4 8  8 1 3 3 3 3 5 3 3 3
6 1  8 3 3 3 3 3 5 3 3 3
```

Is performance improved?

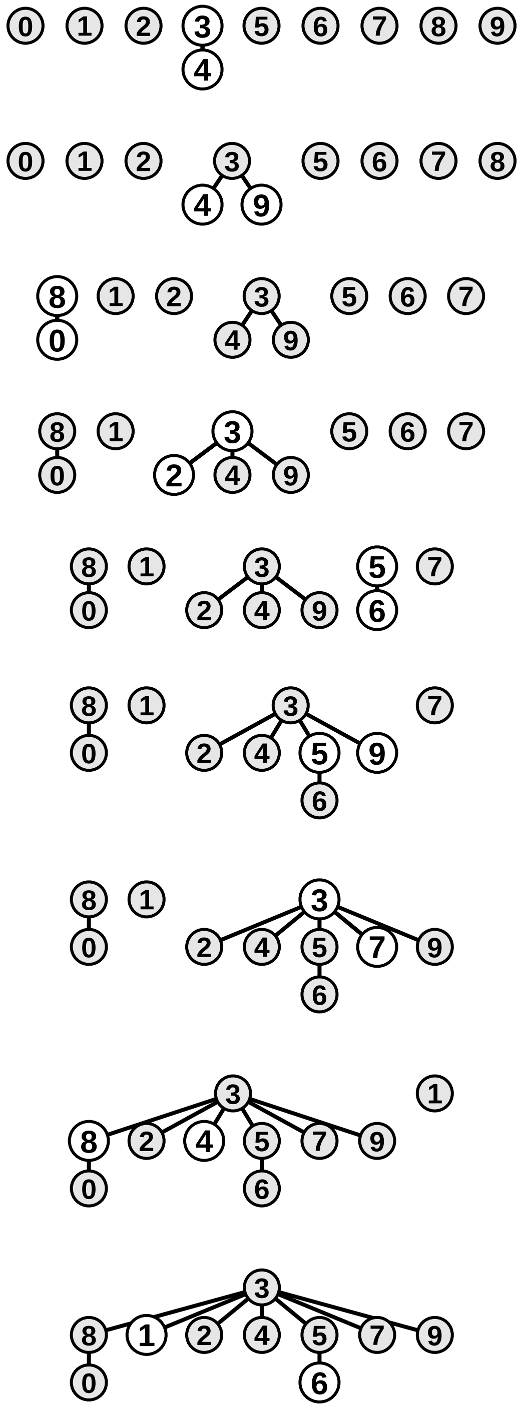
To answer this question

run empirical studies

analyze the algorithm

[need to do both]

Weighted quick-union example



Weighted quick-union worst case

Height of biggest tree increases only
when it links to an equally big tree

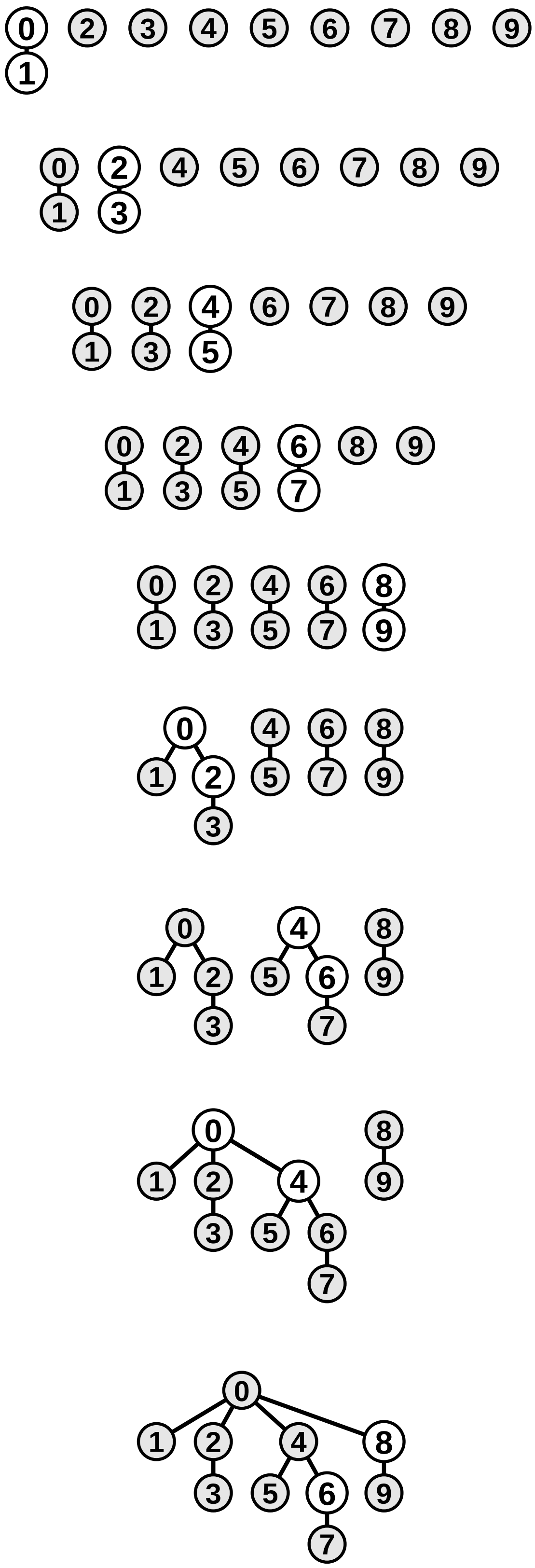
Ex:

```
0 1  0 0 2 3 4 5 6 7 8 9
2 3  0 0 2 2 4 5 6 7 8 9
4 5  0 0 2 2 4 4 6 7 8 9
6 7  0 0 2 2 4 4 6 6 8 9
8 9  0 0 2 2 4 4 6 6 8 8
0 2  0 0 0 2 4 4 6 6 8 8
4 6  0 0 0 2 4 4 4 6 8 8
0 4  0 0 0 2 0 4 4 6 8 8
6 8  0 0 0 2 0 4 4 6 0 8
```

Good news:
Worst case is O(lg N) steps per edge
Better news:
Average case is O(1) steps per edge

Ex:
10^10 edges connecting 10^9 nodes
reduces time from 3000 years to 1 minute

Weighted quick-union worst case



Path compression

Modify weighted quick-union to compress tree
make every node touched point to the new root
(no reason not to!)

```
main(int argc, char *argv[])
{ int i, j, p, q, t, N = atoi(argv[1]);
  int *id = malloc(N*sizeof(int));
  int *sz = malloc(N*sizeof(int));
  for (i = 0; i < N; i++) id[i] = i;
  for (i = 0; i < N; i++) sz[i] = 1;
  while (scanf("%d %d ", &p, &q) != EOF)
  {
    for (i = p; i != id[i]; i = id[i]) ;
    for (j = q; j != id[j]; j = id[j]) ;
    if (i == j) continue;
    if (sz[i] < sz[j])
      { id[i] = j; sz[j] += sz[i]; t = j; }
    else
      { id[j] = i; sz[i] += sz[j]; t = i; }
    for (i = p; i != id[i]; i = id[i]) id[i] = t;
    for (j = q; j != id[j]; j = id[j]) id[j] = t;
    printf(" %d %d ", p, q);
  }
}
```

Worth doing?

Weighted quick-union with path compression

Avoids long paths in trees

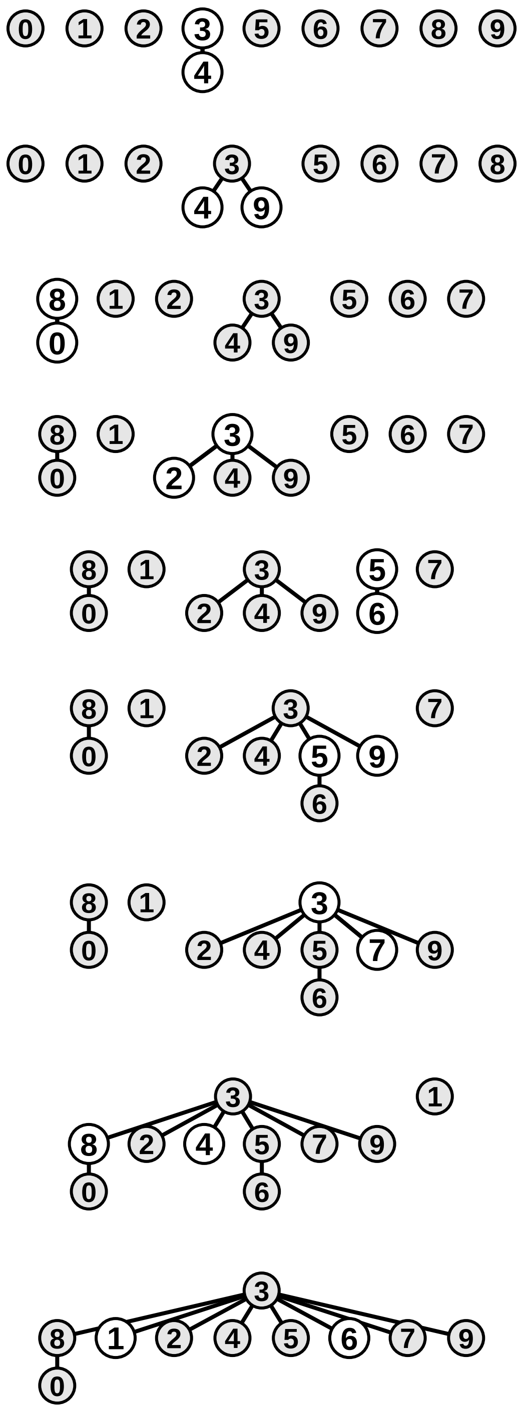
```
3 4  0 1 2 3 3 5 6 7 8 9
4 9  0 1 2 3 3 5 6 7 8 3
8 0  8 1 2 3 3 5 6 7 8 3
2 3  8 1 3 3 3 5 6 7 8 3
5 6  8 1 3 3 3 5 5 7 8 3
5 9  8 1 3 3 3 3 5 7 8 3
7 3  8 1 3 3 3 3 5 3 8 3
4 8  8 1 3 3 3 3 5 3 3 3
6 1  8 3 3 3 3 3 3 3 3 3
```

THM: Worst-case tree height is O(lg* N)
Proof: Extremely difficult
(but the *algorithm* is still simple!)
Note: lg* N is constant in this world

N	lg* N
2	1
4	2
16	3
65536	4
any practical value	5

Reduces cost to within a constant factor of gathering the data

Path compression example



Union-find summary

Worst-case cost per edge is proportional to

quick-find	**N**
quick-union	**N**
weighted	**lg N**
path compression	**5**

Online algorithm: can solve problem while
collecting the data, for "free"

Set-merging abstraction

FIND: is A in the same set as B?
UNION: merge A's set and B's set
useful for many other applications

Lessons

* start with simple algorithm
* don't use simple algorithm for large problems
* can't use simple algorithm for huge problems
* higher level of abstraction (tree) is helpful
* fast performance on real data OK, but
* strive for worst-case performance guarantees
* identify fundamental abstraction

A "trivial" algorithm can be
both useful and interesting

SORTING

Elementary algorithms, Shellsort

Quicksort

Mergesort

Priority queues

Radix sorts

* Sort an array that fills memory
* Make the union of M spelling dictionaries
* Priority queue abstract data type

SEARCHING

Tree searching

Hashing

Trie searching

* Oxford English Dictionary
* Internet search engines
* DNA subsequence library
* Dictionary abstract data type

STRINGS

String searching

Pattern matching

File compression

* file systems, audio and video
* cryptology

GEOMETRIC ALGORITHMS

Elementary algorithms

Convex hull

Multidimensional searching

* N-body simulation
* "Toy Story"

GRAPH ALGORITHMS

Properties of graphs

Searching in graphs

Advanced graph algorithms

* Cycle detection
* matching (e. g. medical residents to jobs)
* networks

OTHER TOPICS

Mathematical algorithms

Dynamic programming

Parallel algorithms

Randomized algorithms

Intractable problems

COURSEWORK

Text

* Algorithms, 3rd edition, in C
* new book, old book after midterm

Lecture notes

Programming Assignments

* weekly, eleven in all
* electronic submission Thursdays 11:59PM
  (first one due 2/12)
* writeup due following Monday at 11:00AM

Problem Sets

* weekly, nine in all
* due in precept
* first one due 2/9 or 2/10

Exams

* open book, notes
* midterm 3/11
* final at scheduled time

Online course information on homepage

READ (and do not lose) HANDOUT ONE

READ ONLINE INFORMATION