CIS 121

DATA STRUCTURES AND ALGORITHMS

ERIC EATON

http://www.seas.upenn.edu/~cis121/
CIS 121 course overview

What is CIS 121?

- Third course in the intro sequence CIS 120, 160, 121
- Programming and problem solving, with applications.
- **Algorithm**: method for solving a problem.
- **Data structure**: method to store information.

<table>
<thead>
<tr>
<th>topic</th>
<th>data structures and algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data types</strong></td>
<td>stack, queue, bag, union-find, priority queue</td>
</tr>
<tr>
<td><strong>sorting</strong></td>
<td>quicksort, mergesort, heapsort, radix sorts</td>
</tr>
<tr>
<td><strong>searching</strong></td>
<td>BST, red-black BST, hash table</td>
</tr>
<tr>
<td><strong>graphs</strong></td>
<td>BFS, DFS, Prim, Kruskal, Dijkstra</td>
</tr>
<tr>
<td><strong>strings</strong></td>
<td>KMP, regular expressions, tries, data compression</td>
</tr>
<tr>
<td><strong>advanced</strong></td>
<td>B-tree, k-d tree, suffix array, maxflow</td>
</tr>
</tbody>
</table>
Why study algorithms?

Their impact is broad and far-reaching.

Internet. Web search, packet routing, distributed file sharing, ...

Biology. Human genome project, protein folding, ...

Computers. Circuit layout, file system, compilers, ...

Computer graphics. Movies, video games, virtual reality, ...

Security. Cell phones, e-commerce, voting machines, ...

Multimedia. MP3, JPG, DivX, HDTV, face recognition, ...

Social networks. Recommendations, news feeds, advertisements, ...

Physics. N-body simulation, particle collision simulation, ...

⋮
Why study algorithms?

For intellectual stimulation.

“For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing.” — Francis Sullivan

“An algorithm must be seen to be believed.” — Donald Knuth
Why study algorithms?

To become a proficient programmer.

“I will, in fact, claim that the difference between a bad programmer and a good one is whether whether the programer considers code or data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.” — Linus Torvalds (creator of Linux)

“Algorithms + Data Structures = Programs.” — Niklaus Wirth
Why study algorithms?

For the interview.
Why study algorithms?

- Their impact is broad and far-reaching.
- For intellectual stimulation.
- To become a proficient programmer.
- To pass your job interviews.
- They may unlock the secrets of life and of the universe.

Why study anything else?
Coursework and Grading

Weekly assignments: 45%

- (Roughly) Weekly programming/written assignments
- Collaboration/lateness policies: see website

Exams: 10% + 10% + 10% + 20%

- Exam 1, 2, 3 (in-class, see website schedule)
- Final exam (finals week)

Attendance of Lecture and Recitations: 5%

- Attendance of both is mandatory

Required Textbook:

INTRODUCTION TO ALGORITHMS
THIRD EDITION
Where to get help?

Piazza

• Low latency, low bandwidth
• Post publicly as much as possible
• Mark solution-revealing questions as private
• Please don’t email Dr. Eaton — use Piazza to message me

Office hours

• High bandwidth, high latency

Tutoring

• The Tutoring Center

Study groups

• Can discuss problems in general
• But, you must write-up solutions yourself
  – Recommendation: Throw away any notes made in a group setting!
• Cannot work together on programming

http://www.seas.upenn.edu/~cis121/

http://www.vpul.upenn.edu/tutoring/
Where not to get help?

http://world.edu/academic-plagiarism
http://www.upenn.edu/academicintegrity/ai_codeofacademicintegrity.html

Cheating policy

- All homework submissions are checked by plagiarism detection software
- Any assignments that are flagged will be automatically referred to the Office of Student Conduct, which will adjudicate whether the course collaboration policy was violated
How to succeed in CIS 121

Make the class part of your routine

• Come to lectures
• Engage in recitations
• Read the assigned chapters immediately before or after lecture
• Start homework assignments early
• Do not wait until the last day

Make use of your TAs and instructors

• Attend recitations
• Go to office hours (fine to come even if you don’t have a specific question)

Do not violate the academic integrity policy (i.e., don’t cheat)
What's ahead?

Lecture 1.  [today] Union-Find
Lecture 2. Analysis of algorithms - Big Oh!

Recitations start in two weeks

Assistance this Week:
  • Review sessions for CIS 120, CIS 160
  • Help in office hours on the Getting Started assignment

Homework 0 (Getting Started) will be posted shortly
1.5 **Union-Find**

- **dynamic connectivity**
- **quick find**
- **quick union**
- **improvements**
- **applications**
Theme 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 not.
- Find a way to address the problem.
- Iterate until satisfied.

The **scientific method.**

**Mathematical analysis.**
Dynamic connectivity problem

Given a set of N objects, support two operations:

- Connect two objects.
- Is there a path connecting the two objects?

connect 4 and 3
connect 3 and 8
connect 6 and 5
connect 9 and 4
connect 2 and 1
are 0 and 7 connected? ✗
are 8 and 9 connected? ✔
connect 5 and 0
connect 7 and 2
connect 6 and 1
connect 1 and 0
are 0 and 7 connected? ✔
A larger 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 a 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!
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 component. Maximal set of objects that are mutually connected.

\[
\begin{array}{ccc}
0 & 1 & 2 \\
4 & 5 & 6 \\
\end{array}
\begin{array}{ccc}
3 & 7 \\
\end{array}
\]

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

3 connected components
Implementing the operations

**Find.** In which component is object $p$?

**Connected.** Are objects $p$ and $q$ in the same component?

**Union.** Replace components containing objects $p$ and $q$ with their union.

---

0 1 2 3
4 5 6 7

union(2, 5)

0 1 2 3
4 5 6 7

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

3 connected components

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

2 connected components
**Goal.** Design efficient data structure for union-find.

- Number of objects $N$ can be huge.
- Number of operations $M$ can be huge.
- Union and find operations may be intermixed.

```java
public class UF
{
    UF(int N) { initialize union-find data structure with N singleton objects (0 to N – 1) }
    void union(int p, int q) { add connection between p and q }
    int find(int p) { component identifier for p (0 to N – 1) }
    boolean connected(int p, int q) { are p and q in the same component? }
}
```

1-line implementation of connected():

```java
public boolean connected(int p, int q) {
    return find(p) == find(q);
}
```
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: $id[p]$ is the id of the component containing $p$.

<table>
<thead>
<tr>
<th>id[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

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: \( id[p] \) is the id of the component containing \( p \).

Find. What is the id of \( p \)?

Connected. Do \( p \) and \( q \) have the same id?

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

<table>
<thead>
<tr>
<th>id[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

\( id[6] = 0; id[1] = 1 \)

6 and 1 are not connected

<table>
<thead>
<tr>
<th>id[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

after union of 6 and 1

problem: many values can change
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

union(4, 3)

id[]

0 1 2 3 4 5 6 7 8 9
Quick-find demo

union(3, 8)

id[]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td></td>
<td></td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Quick-find demo

union(6, 5)

id[]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Quick-find demo

union(9, 4)

```
id[]  0  1  2  3  4  5  6  7  8  9
0  1  2  8  8  5  5  7  8  8
```
Quick-find demo

union(2, 1)

id[]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id[]</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>8</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Quick-find demo

connected(8, 9)

id[]

\[
\begin{array}{cccccccccccc}
0 & 1 & 1 & 8 & 8 & 5 & 5 & 7 & 8 & 8 \\
\end{array}
\]

already connected
Quick-find demo

connected(5, 0)

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

\[
\begin{array}{c|cccccccc}
\text{id[]} & 0 & 1 & 1 & 8 & 8 & 5 & 5 & 7 & 8 & 8 \\
\end{array}
\]

not connected
Quick-find demo

union(5, 0)
Quick-find demo

union(7, 2)
Quick-find demo

union(6, 1)
Quick-find demo

id[]

0 1 2 3 4 5 6 7 8 9
1 1 1 8 8 1 1 1 8 8
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 int find(int p)   {  return id[p];  }

    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;
    }
}

Quick-find: Java implementation

- Set id of each object to itself (N array accesses)
- Return the id of p (1 array access)
- Change all entries with id[p] to id[q] (at most 2N + 2 array accesses)
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 int find(int p) {  return id[p];  }

    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)

return the id of p
(1 array access)

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).

<table>
<thead>
<tr>
<th>algorithm</th>
<th>initialize</th>
<th>union</th>
<th>find</th>
<th>connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

order of growth of number of array accesses

**Union is too expensive.** A sequence of $N$ union operations on $N$ objects will take $N^2$ array accesses to process
Quadratic algorithms do not scale

Huge problem for quick-find.
- $10^9$ union commands on $10^9$ objects.
- Quick-find takes more than $10^{18}$ operations.
- 30+ years of computer time!

Quadratic algorithms don't scale well
1.5 UNION-FIND

- dynamic connectivity
- quick find
- quick union
- improvements
- applications
Quick-union  [lazy approach - delay work until we have to do it]

Data structure - represents trees.

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

<table>
<thead>
<tr>
<th>id[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

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

parent of 3 is 4
root of 3 is 9
Quick-union  [lazy approach - delay work until we have to do it]

Data structure - represents trees.

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

Find.  What is the root of p?
Connected.  Do p and q have the same root?

Union.  To merge components containing p and q, set the id of p's root to the id of q's root.
Quick-union demo
Quick-union demo

union(4, 3)

0 1 2 3 4 5 6 7 8 9
Quick-union demo

union(4, 3)
Quick-union demo
Quick-union demo

union(3, 8)

id[]  0  1  2  3  4  5  6  7  8  9
      0  1  2  3  3  5  6  7  8  9
Quick-union demo

union(3, 8)

id[]

0 1 2 3 4 5 6 7 8 9
Quick-union demo

0 1 2 5 6 7 8 9

id[]

0 1 2 8 3 5 6 7 8 9
Quick-union demo

union(6, 5)

0  1  2  5  6  7  8  9

id[]

0  1  2  8  3  5  6  7  8  9
Quick-union demo

union(6, 5)
Quick-union demo

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

<table>
<thead>
<tr>
<th>id[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Quick-union demo

union(9, 4)

```
id[]  0 1 2 3 4 5 6 7 8 9
0 1 2 8 3 5 5 7 8 9
```
Quick-union demo

union(9, 4)

0 1 2 5 6 3 4 8 7 9

id[]

0 1 2 8 3 5 5 7 8 8
Quick-union demo
Quick-union demo

union(2, 1)

id[]

0 1 2 8 3 5 5 7 8 8
Quick-union demo

union(2, 1)

0 1
2

5 7
6

8
3
4

id[]
0 1 1 8 3 5 5 7 8 8
Quick-union demo
Quick-union demo

connected(8, 9)

0

1

2

5

6

7

8

3

4

5

6

7

8

9

id[]

0 1 1 8 3 5 5 7 8 8
Quick-union demo

connected(5, 4)

0
1
2
3
4
5
6
7
8
9

id[]

0 1 1 8 3 5 5 7 8 8
Quick-union demo

union(5, 0)
Quick-union demo

union(5, 0)

id[] array:

```
0 1 1 8 3 0 5 7 8 8
```
Quick-union demo

```
id[] 0 1 2 3 4 5 6 7 8 9
0 1 1 8 3 0 5 7 8 8
```
Quick-union demo

union(7, 2)

<table>
<thead>
<tr>
<th>id[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Quick-union demo

union(7, 2)

id[]

0 1 1 8 3 0 5 1 8 8
Quick-union demo

\[
\begin{array}{c}
\text{id[]} \\
0 & 1 & 1 & 8 & 3 & 0 & 5 & 1 & 8 & 8
\end{array}
\]
Quick-union demo

union(6, 1)

id[]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>id[]</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
Quick-union demo

union(6, 1)
Quick-union demo

```
id[]
0 1 2 3 4 5 6 7 8 9
1 1 1 8 3 0 5 1 8 8
```
Quick-union demo

union(7, 3)
Quick-union demo

union(7, 3)
Quick-union demo

```
<table>
<thead>
<tr>
<th>id[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
```
public class QuickUnionUF {
    private int[] id;

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

    public int find(int i) {
        while (i != id[i]) i = id[i];
        return i;
    }

    public void union(int p, int q) {
        int i = find(p);
        int j = find(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)

change root of p to point to root of q
(depth of p and q array accesses)
public class QuickUnionUF {
    private int[] id;

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

    public int find(int i) {
        while (i != id[i]) i = id[i];
        return i;
    }

    public void union(int p, int q) {
        int i = find(p);
        int j = find(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)
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).

<table>
<thead>
<tr>
<th>algorithm</th>
<th>initialize</th>
<th>union</th>
<th>find</th>
<th>connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>N</td>
<td>N †</td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

† 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/connected too expensive (could be $N$ array accesses).
1.5 Union-Find

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

**Weighted quick-union**

reasonable alternatives: union by height or "rank"

- **quick-union**
  - might put the larger tree lower

- **weighted**
  - always chooses the better alternative

**Improvement 1: weighting**

Weighted quick-union always chooses the better alternative might put the larger tree lower
Weighted quick-union demo
Weighted quick-union demo

<table>
<thead>
<tr>
<th>id[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>
Weighted quick-union demo

union(4, 3)
Weighted quick-union demo

union(4, 3)

id[]  0 1 2 4 4 5 6 7 8 9
Weighted quick-union demo
Weighted quick-union demo

union(3, 8)
Weighted quick-union demo

union(3, 8)

weighting: make 8 point to 4 (instead of 4 to 8)
Weighted quick-union demo

```
id[]   0 1 2 3 4 5 6 7 8 9
0 1 2 4 4 5 6 7 4 9
```
Weighted quick-union demo

union(6, 5)

0 1 2 4 4 5 6 7 8 9

id[]
Weighted quick-union demo

union(6, 5)
Weighted quick-union demo
Weighted quick-union demo

union(9, 4)

id[] = [0 1 2 4 4 6 6 7 4 9]
Weighted quick-union demo

union(9, 4)

weighting: make 9 point to 4
Weighted quick-union demo
Weighted quick-union demo

union(2, 1)

0  1  2  4  8  9
3  5

id[]
0 1 2 4 4 6 6 7 4 4
Weighted quick-union demo

union(2, 1)

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

id[]
0 1 2 2 4 4 6 6 7 4 4
```
Weighted quick-union demo
Weighted quick-union demo

union(5, 0)

id[]

0 1 2  3 4  5 6  7 8  9
0 | 2 | 2 | 4 | 4 | 6 | 6 | 7 | 4 | 4
Weighted quick-union demo

union(5, 0)

weighting: make 0 point to 6 (instead of 6 to 0)

```
id[]   0  1  2  3  4  5  6  7  8  9
    6 2 2 4 4 6 6 7 4 4
```
Weighted quick-union demo

id[]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Weighted quick-union demo

union(7, 2)

id[]

```
0 1 2 3 4 5 6 7 8 9
6 2 2 4 4 6 6 7 4 4
```
Weighted quick-union demo

union(7, 2)

weighting: make 7 point to 2

id[]

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
6 & 2 & 2 & 4 & 4 & 6 & 6 & 2 & 4 & 4 \\
\end{array}
\]
Weighted quick-union demo

id[]: 6 2 2 4 4 6 6 2 4 4
Weighted quick-union demo

union(6, 1)

id[] = [6, 2, 2, 4, 4, 6, 6, 2, 4, 4]
Weighted quick-union demo

union(6, 1)

```
id[]   0  1  2  3  4  5  6  7  8  9
       6 2 6 4 4 6 6 2 4 4
```
Weighted quick-union demo
Weighted quick-union demo

union(7, 3)
Weighted quick-union demo

union(7, 3)

weighting: make 4 point to 6 (instead of 6 to 4)
Weighted quick-union demo
Example trees in quick-union versus weighted quick-union

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

average distance to root: 5.11

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/connected.** Identical to quick-union.

**Union.** Modify quick-union to:
- Link root of smaller tree to root of larger tree.
- Update the \( sz[] \) array.

```java
int i = find(p);
int j = find(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 $p$.
- Union: takes constant time, given roots.

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

Weighted quick-union analysis

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

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

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

N = 11
depth(x) = 3 ≤ $\lg N$
Weighted quick-union analysis

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

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

**Pf.** What causes the depth of object $x$ to increase?

Increases by 1 when tree $T_1$ containing $x$ is merged into another tree $T_2$.
- The size of the new 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

$T_2$

$T_1$

$x$

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>$\vdots$</th>
<th>$N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lg N$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Weighted quick-union analysis

**Running time.**
- Find: takes time proportional to depth of $p$.
- Union: takes constant time, given roots.

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

<table>
<thead>
<tr>
<th>algorithm</th>
<th>initialize</th>
<th>union</th>
<th>find</th>
<th>connected</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>N</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>quick-union</td>
<td>N</td>
<td>N†</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>weighted QU</td>
<td>N</td>
<td>$\lg N$†</td>
<td>$\lg N$</td>
<td>$\lg N$</td>
</tr>
</tbody>
</table>

† 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.
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.

Bottom line. Now, `find()` has the side effect of compressing the tree.
Path compression: Java implementation

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

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

```java
public int find(int i) {
    while (i != id[i]) {
        id[i] = id[id[i]];
        i = id[i];
    }
    return i;
}
```

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.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\lg^* N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>65536</td>
<td>4</td>
</tr>
<tr>
<td>$2^{65536}$</td>
<td>5</td>
</tr>
</tbody>
</table>

**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] Proved that no linear-time algorithm exists.
Key point. Weighted quick union (and/or path compression) makes it possible to solve problems that could not otherwise be addressed.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>worst-case time</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>M N</td>
</tr>
<tr>
<td>quick-union</td>
<td>M N</td>
</tr>
<tr>
<td>weighted QU</td>
<td>N + M log N</td>
</tr>
<tr>
<td>QU + path compression</td>
<td>N + M log N</td>
</tr>
<tr>
<td>weighted QU + path compression</td>
<td>N + M lg* N</td>
</tr>
</tbody>
</table>

Order of growth for M union-find operations on a set of N objects

Ex. [10⁹ unions and finds with 10⁹ objects]
- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won't help much; good algorithm enables solution.
1.5 Union-Find

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

- Percolation.
- Games (Go, Hex).
- Dynamic connectivity.
  - Least common ancestor.
  - Equivalence of finite state automata.
  - Hoshen-Kopelman algorithm in physics.
  - Hinley-Milner polymorphic type inference.
  - Kruskal's minimum spanning tree algorithm.
  - Compiling equivalence statements in Fortran.
  - Morphological attribute openings and closings.
  - Matlab's `bwlabel()` function in image processing.
An abstract model for many physical systems:

- $N$-by-$N$ grid of sites.
- Each site is open with probability $p$ (and blocked with probability $1 - p$).
- System percolates iff top and bottom are connected by open sites.

$N = 8$
Percolation

An abstract model for many physical systems:
- $N$-by-$N$ grid of sites.
- Each site is open with probability $p$ (and blocked with probability $1 - p$).
- System **percolates** iff top and bottom are connected by open sites.

<table>
<thead>
<tr>
<th>model</th>
<th>system</th>
<th>vacant site</th>
<th>occupied site</th>
<th>percolates</th>
</tr>
</thead>
<tbody>
<tr>
<td>electricity</td>
<td>material</td>
<td>conductor</td>
<td>insulated</td>
<td>conducts</td>
</tr>
<tr>
<td>fluid flow</td>
<td>material</td>
<td>empty</td>
<td>blocked</td>
<td>porous</td>
</tr>
<tr>
<td>social interaction</td>
<td>population</td>
<td>person</td>
<td>empty</td>
<td>communicates</td>
</tr>
</tbody>
</table>
Likelihood of percolation

Depends on grid size $N$ and site vacancy probability $p$. 

- $p$ low (0.4) does not percolate
- $p$ medium (0.6) percolates?
- $p$ high (0.8) percolates
Percolation phase transition

When \( N \) is large, theory guarantees a sharp threshold \( p^* \).

- \( p > p^* \): almost certainly percolates.
- \( p < p^* \): almost certainly does not percolate.

Q. What is the value of \( p^* \)?
Monte Carlo simulation

- Initialize all sites in an $N$-by-$N$ grid to be blocked.
- Declare random sites open until top connected to bottom.
- Vacancy percentage estimates $p^*$. 

$N = 20$

135 open sites
Dynamic connectivity solution to estimate percolation threshold

Q. How to check whether an $N$-by-$N$ system percolates?
A. Model as a dynamic connectivity problem and use union-find.

\[
N = 5
\]

[Image of a $5 \times 5$ grid with open and blocked sites]

- open site
- blocked site
Dynamic connectivity solution to estimate percolation threshold

Q. How to check whether an $N$-by-$N$ system percolates?

- Create an object for each site and name them 0 to $N^2 - 1$. 

![Diagram of a 5x5 grid with sites numbered and some sites blocked.](image)
Dynamic connectivity solution to estimate percolation threshold

Q. How to check whether an $N$-by-$N$ system percolates?
   - Create an object for each site and name them 0 to $N^2 - 1$.
   - Sites are in same component iff connected by open sites.

\[ N = 5 \]

- open site
- blocked site
Q. How to check whether an $N$-by-$N$ system percolates?

- Create an object for each site and name them 0 to $N^2 - 1$.
- Sites are in same component iff connected by open sites.
- Percolates iff any site on bottom row is connected to any site on top row.

Dynamic connectivity solution to estimate percolation threshold

brute-force algorithm: $N^2$ calls to connected()
Dynamic connectivity solution to estimate percolation threshold

Clever trick. Introduce 2 virtual sites (and connections to top and bottom).
• Percolates iff virtual top site is connected to virtual bottom site.

More efficient algorithm: only 1 call to connected()

$N = 5$

- open site
- blocked site
Dynamic connectivity solution to estimate percolation threshold

Q. How to model opening a new site?

$N = 5$

open site

open this site

blocked site
Dynamic connectivity solution to estimate percolation threshold

Q. How to model opening a new site?
A. Mark new site as open; connect it to all of its adjacent open sites.

$N = 5$

open this site

open site

blocked site
Percolation threshold

Q. What is percolation threshold \( p^* \) ?

A. About 0.592746 for large square lattices.

Fast algorithm **enables** accurate answer to scientific question.
Theme 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.