Impact of Great Algorithms

- Internet. Packet routing, Google, Akamai.
- Biology. Human genome project, protein folding.
- Computers. Circuit layout, file system, compilers.
- Multimedia. CD player, DVD, MP3, JPEG, DivX, HDTV.
- Transportation. Airline crew scheduling, map routing.
- Physics. N-body simulation, particle collision simulation.
- Information processing. Database search, data compression.

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

What is COS 226?
- Intermediate-level survey course.
- Programming and problem solving with applications.
- Data structure: method to store information.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Data Structures and Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>data types</td>
<td>stack, queue, list, union-find, priority queue</td>
</tr>
<tr>
<td>sorting</td>
<td>quicksort, mergesort, heapsort, radix sorts</td>
</tr>
<tr>
<td>searching</td>
<td>hash table, BST, red-black tree, B-tree</td>
</tr>
<tr>
<td>graphs</td>
<td>DFS, Prim, Kruskal, Dijkstra, Ford-Fulkerson</td>
</tr>
<tr>
<td>strings</td>
<td>KMP, Rabin-Karp, TST, Huffman, LZW</td>
</tr>
<tr>
<td>geometry</td>
<td>Graham scan, k-d tree, Voronoi diagram</td>
</tr>
</tbody>
</table>

A misperception: algor (painful) + arithmos (number).

Why Study Algorithms?

Using a computer?
- Want it to go faster? Process more data?
- Want it to do something that would otherwise be impossible?

Algorithms as a field of study.
- Old enough that basics are known.
- New enough that new discoveries arise.
- Burgeoning application areas.
- Philosophical implications.
The Usual Suspects

Lectures. Robert Sedgewick
• MW 11-12:20, Friend 006.

Precepts. Jordan Boyd-Grabner, Ananya Misra, Matthew Hoffman
• T 12:30, Friend 108.
• T 1:30, Friend 108.
• T 3:30, Friend 108.
• Clarify programming assignments, exercises, lecture material.
• First precept meets 2/7.

Coursework and Grading

Regular programming assignments: 45%
• Due 11:55pm, starting 2/16.
• More details next lecture.

Weekly written exercises: 15%
• Due at beginning of Wednesday lecture, starting 2/8.

Exams:
• Closed book with cheatsheet.
  • Midterm. 15%
  • Final. 25%

Staff discretion. Adjust borderline cases.

Course Materials

• Syllabus.
• Exercises.
• Lecture slides.
• Programming assignments.

Algorithms in Java, 3rd edition.
• Parts 1-4. [sorting, searching]
• Part 5. [graph algorithms]

• Strings and geometry handouts.

Questionnaire

Please fill out questionnaire so that we can adapt course as needed.
• Who are you?
• What is your programming experience?
• Why are you taking COS 226?
• What do you hope to get out of it?
**Union Find**


---

**An Example Problem: Network Connectivity**

Network connectivity.
- Nodes at grid points.
- Add connections between pairs of nodes.
- Is there a path from node A to node B?

---

**Network Connectivity**

<table>
<thead>
<tr>
<th>in</th>
<th>out</th>
<th>evidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 4</td>
<td>3 4</td>
<td></td>
</tr>
<tr>
<td>4 9</td>
<td>4 9</td>
<td></td>
</tr>
<tr>
<td>8 0</td>
<td>8 0</td>
<td></td>
</tr>
<tr>
<td>2 3</td>
<td>2 3</td>
<td></td>
</tr>
<tr>
<td>5 6</td>
<td>5 6</td>
<td></td>
</tr>
<tr>
<td>2 9</td>
<td></td>
<td>(2-3-4-9)</td>
</tr>
<tr>
<td>5 9</td>
<td>5 9</td>
<td></td>
</tr>
<tr>
<td>7 3</td>
<td>7 3</td>
<td></td>
</tr>
<tr>
<td>4 8</td>
<td>4 8</td>
<td></td>
</tr>
<tr>
<td>5 6</td>
<td></td>
<td>(5-6)</td>
</tr>
<tr>
<td>0 2</td>
<td></td>
<td>(2-3-4-8-0)</td>
</tr>
<tr>
<td>6 1</td>
<td>6 1</td>
<td></td>
</tr>
</tbody>
</table>

---

**Union-Find Abstraction**

What are critical operations we need to support?
- **Objects.**
  
  | grid points |
  | 0 1 2 3 4 5 6 7 8 9 |

- **Disjoint sets of objects.**
  
  | subsets of connected grid points |
  | 0 1 2-3-9 5-6 7 4-8 |

- **Find:** are objects 2 and 9 in the same set?
  
  | are two grid points connected? |
  | 0 1 2-3-9 5-6 7 4-8 |

- **Union:** merge sets containing 3 and 8.
  
  | add a connection between two grid points |
  | 0 1 2-3-4-8-9 7 8-4 |
Union-Find Abstraction

What are critical operations we need to support?
- Objects.
- Disjoint sets of objects.
- Find: are two objects in the same set?
- Union: replace sets containing two items by their union.

Goal. Design efficient data structure for union and find.
- Number of operations $M$ can be huge.
- Number of objects $N$ can be huge.

Applications involve manipulating objects of all types.
- Variable name aliases.
- Pixels in a digital photo.
- Computers in a network.
- Web pages on the Internet.
- Transistors in a computer chip.
- Metallic sites in a composite system.

When programming, convenient to name them 0 to $N-1$.
- Details not relevant to union-find.
- Integers allow quick-access to object-related info (array indices).

Quick-Find [eager approach]

Data structure.
- Integer array $id[]$ of size $N$.
- Interpretation: $p$ and $q$ are connected if they have the same $id$.

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

Union. To merge components containing $p$ and $q$,
change all entries with $id[p]$ to $id[q]$.

Quick-Find

Applications involve manipulating objects of all types.
- Variable name aliases.
- Pixels in a digital photo.
- Computers in a network.
- Web pages on the Internet.
- Transistors in a computer chip.
- Metallic sites in a composite system.

When programming, convenient to name them 0 to $N-1$.
- Details not relevant to union-find.
- Integers allow quick-access to object-related info (array indices).
Quick-Union:

Data structure.
- Integer array \( id[] \) of size \( N \).
- Interpretation: \( id[x] \) is parent of \( x \).
- Root of \( x \) is \( id[id[\ldots id[x] \ldots]] \).

Find. Check if \( p \) and \( q \) have the same root.

Union. Set the id of \( q \)'s root to the id of \( p \)'s root.

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

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

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

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

Problem Size and Computation Time

Rough standard for 2000.
- \( 10^9 \) operations per second.
- \( 10^9 \) words of main memory.
- Touch all words in approximately 1 second. [unchanged since 1950]

Ex. Huge problem for quick find.
- \( 10^{10} \) edges connecting \( 10^9 \) nodes.
- Quick-find might take \( 10^{20} \) operations. [10 ops per query]
- 3,000 years of computer time!

Paradoxically, quadratic algorithms get worse with newer equipment.
- New computer may be 10x as fast.
- But, has 10x as much memory so problem may be 10x bigger.
- With quadratic algorithm, takes 10x as long!
Weighted Quick-Union

- Modify quick-union to avoid tall trees.
- Keep track of size of each component.
- Balance by linking small tree below large one.

**Ex:** Union of 5 and 3.
- Quick union: Link 9 to 6.
- Weighted quick union: Link 6 to 9.

---

Quick-Union: Java Implementation

```java
class QuickUnion {
    private int[] id;
    public QuickUnion(int N) {
        id = new int[N];
        for (int i = 0; i < N; i++) id[i] = i;
    }
    private int root(int x) {
        while (x != id[x]) x = id[x];
        return x;
    }
    public boolean find(int p, int q) {
        return root(p) == root(q);
    }
    public void unite(int p, int q) {
        int i = root(p), j = root(q);
        if (i == j) return;
        id[i] = j;
    }
}
```

**Quick-find defect.**
- Union too expensive.
- Trees are flat, but too hard to keep them flat.

**Quick-union defect.**
- Finding the root can be expensive.
- Trees can get tall.

---

Summary

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Union</th>
<th>Find</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick-find</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>Quick-union</td>
<td>1†</td>
<td>N</td>
</tr>
</tbody>
</table>

† Union of two root nodes

---

Weighted Quick-Union

```
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 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 5 3 3 3
```
**Path Compression**

**Path compression.** Just after computing the root of $x$, set id of each examined node to $\text{root}(x)$.

```java
if (sz[i] < sz[j]) {
    id[i] = j; sz[j] += sz[i];
} else {
    id[j] = i; sz[i] += sz[j];
}
```

**Weighted Quick-Union: Java Implementation**

- Java implementation.
  - Almost identical to quick-union.
  - Maintain extra array $sz[]$ to count number of elements in the tree rooted at $i$.

**Find.** Identical to quick-union.

**Union.** Same as quick-union, but merge smaller tree into the larger tree, and update the $sz[]$ array.

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Union</th>
<th>Find</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quick-find</td>
<td>$N$</td>
<td>$1$</td>
</tr>
<tr>
<td>Quick-union</td>
<td>$1 + \lg N$</td>
<td>$N$</td>
</tr>
<tr>
<td>Weighted QU</td>
<td>$\lg N$</td>
<td>$\lg N$</td>
</tr>
</tbody>
</table>

**Weighted Quick-Union: Analysis**

- **Find.** takes time proportional to depth of $p$ and $q$.
- **Union.** takes constant time, given roots.
- **Fact:** depth is at most $1 + \lg N$. [needs proof]

**Stop at guaranteed acceptable performance?** No, can improve further.

**Weighted Quick-Union with Path Compression**

**Path compression.**
- Standard implementation: add second loop to $\text{root}$ 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.

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

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

Ex. Huge practical problem.
- \(10^{10}\) edges connecting \(10^9\) nodes.
- WQUPC reduces time from 3,000 years to 1 minute.
- Supercomputer wouldn’t help much.
- Good algorithm makes solution possible.

Bottom line. WQUPC on cell phone beats QF on supercomputer!

Applications

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>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>Path compression</td>
<td>(N + M \log N)</td>
</tr>
<tr>
<td>Weighted + path</td>
<td>(5 (M + N))</td>
</tr>
</tbody>
</table>

\(M\) union-find ops on a set of \(N\) elements

Theorem. A sequence of \(M\) union and find operations on \(N\) elements takes \(O(N + M \log^* N)\) time.
- Proof is very difficult.
- But the algorithm is still simple!

Remark. \(\log^* N\) is a constant in this universe.

Linear algorithm?
- Cost within constant factor of reading in the data.
- Theory: WQUPC is not quite linear.
- Practice: WQUPC is linear.
Other Applications

Union-find applications.
- Hex.
- Percolation.
- Connectivity.
- Image processing.
- Least common ancestor.
- Equivalence of finite state automata.
- Hinley-Milner polymorphic type inference.
- Kruskal’s minimum spanning tree algorithm.
- Compiling equivalence statements in Fortran.
- Scheduling unit-time tasks to P processors so that each job finishes between its release time and deadline.

Percolation

Percolation phase-transition.
- Two parallel conducting bars (top and bottom).
- Electricity flows from a site to one of its 4 neighbors if both are occupied by conductors.
- Suppose each site is randomly chosen to be a conductor with probability p or insulator with probability 1-p.

**Q.** What is percolation threshold $p^*$ at which charge carriers can percolate from top to bottom?

**A.** ~ 0.592746 for square lattices. [constant only known via simulation]

Hex

Hex. [Piet Hein 1942, John Nash 1948, Parker Brothers 1962]
- Two players alternate in picking a cell in a hex grid.
- Black: make a black path from upper left to lower right.
- White: make a white path from lower left to upper right.

Goal. Algorithm to detect when a player has won.


Percolation

top

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td>41</td>
<td>40</td>
<td>32</td>
<td>34</td>
<td>36</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
</tr>
</tbody>
</table>

bottom

| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

insulator
Summary

Lessons.

- Simple algorithms can be very useful.
- Start with brute force approach.
  - don’t use for large problems
  - can’t use for huge problems
- Strive for worst-case performance guarantees.
- Identify fundamental abstractions: union-find.
- Apply to many domains.