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, 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</td>
</tr>
<tr>
<td>graphs</td>
<td>BFS, DFS, Prim, Kruskal, Dijkstra</td>
</tr>
<tr>
<td>strings</td>
<td>KMP, Regular expressions, TST, Huffman, LZW</td>
</tr>
<tr>
<td>geometry</td>
<td>Graham scan, k-d tree, Voronoi diagram</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. CD player, DVD, MP3, JPG, DivX, HDTV, ...
Transportation. Airline crew scheduling, map routing, ...
Physics. N-body simulation, particle collision simulation, ...
...

Old roots, new opportunities.
- Study of algorithms dates at least to Euclid
- Some important algorithms were discovered by undergraduates!
To solve problems that could not otherwise be addressed.

*Ex.* Network connectivity. [stay tuned]

Why study algorithms? To solve problems that could not otherwise be addressed. For intellectual stimulation. They may unlock the secrets of life and of the universe.

Computational models are replacing mathematical models in scientific enquiry:

\[ E = mc^2 \]
\[ F = ma \]
\[ F = \frac{Gm_1m_2}{r^2} \]
\[ \Psi(r) = E \Psi(r) \]

20th century science (formula based)

21st century science (algorithm based)

“Algorithms: a common language for nature, human, and computer.” — Avi Wigderson

“*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.*” — D. E. Knuth

For fun and profit.

Why study algorithms?
Why study algorithms?

- Their impact is broad and far-reaching.
- Old roots, new opportunities.
- To solve problems that could not otherwise be addressed.
- For intellectual stimulation.
- They may unlock the secrets of life and of the universe.
- For fun and profit.

Why study anything else?

Coursework and grading

8 programming assignments. 45%
- Electronic submission.
- Due 11:55pm, starting Wednesday 9/17.

Exercises. 15%
- Due in lecture, starting Tuesday 9/16.

Exams.
- Closed-book with cheatsheet.
- Midterm. 15%
- Final. 25%

Staff discretion. To adjust borderline cases.

everyone needs to meet me (at least) once!

The usual suspects

Lectures. Introduce new material, answer questions.

Precepts. Answer questions, solve problems, discuss programming assignment.

Resources (web)

Course content.
- Course info.
- Exercises.
- Lecture slides.
- Programming assignments.

Course administration.
- Check grades.
- Submit assignments.

Booksites.
- Brief summary of content.
- Download code from lecture.

Course content.
- Course info.
- Exercises.
- Lecture slides.
- Programming assignments.

Course administration.
- Check grades.
- Submit assignments.

Booksites.
- Brief summary of content.
- Download code from lecture.

Resources (web)

Course content.
- Course info.
- Exercises.
- Lecture slides.
- Programming assignments.

Course administration.
- Check grades.
- Submit assignments.

Booksites.
- Brief summary of content.
- Download code from lecture.

Resources (web)

Course content.
- Course info.
- Exercises.
- Lecture slides.
- Programming assignments.

Course administration.
- Check grades.
- Submit assignments.

Booksites.
- Brief summary of content.
- Download code from lecture.
Union-Find Algorithms

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

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.
Dynamic connectivity

Given a set of objects
- **Union**: connect two objects.
- **Find**: is there a path connecting the two objects?

```
union(3, 4)
union(8, 0)
union(2, 3)
union(5, 6)
find(0, 2) no
find(2, 4) yes
union(5, 1)
union(7, 3)
union(1, 6)
union(4, 8)
find(0, 2) yes
find(2, 4) yes
```

Network connectivity: larger example

Modeling the objects

Dynamic connectivity 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 objects 0 to N-1.
- Use integers as array index.
- Suppress details not relevant to union-find.

can use symbol table to translate from object names to integers (stay tuned)
**Transitivity.** 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.

```
( 1 5 6 ) ( 2 3 4 7 ) ( 0 8 )
```

**Find query.** Check if two objects are in the same set.

**Union command.** Replace sets containing two objects with their union.

```
public class UnionFind

    UnionFind(int N) {
        create union-find data structure with \( N \) objects and no connections
    }

    boolean find(int p, int q) {
        are \( p \) and \( q \) in the same set?
    }

    void unite(int p, int q) {
        replace sets containing \( p \) and \( q \) with their union
    }
```
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 sets containing $p$ and $q$, change all entries with $id[p]$ to $id[q]$.

Quick-find example
Quick-find: Java implementation

```java
public class QuickFind {
    private int[] id;

    public QuickFind(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];
    }
}
```

Quick-find is too slow

Quick-find defect.
- Union too expensive (N operations).
- Trees are flat, but too expensive to keep them flat.

Ex. May take $N^2$ operations to process $N$ union commands on $N$ objects.

Quick-find is too slow

Algorithm | Union | Find
---|---|---
quick-find | $N$ | $I$

Quadratic algorithms do not scale

Rough standard (for now).
- $10^9$ operations per second.
- $10^9$ words of main memory.
- Touch all words in approximately 1 second.

Ex. 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!

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!
Quick-union [lazy approach]

Data structure.
• Integer array id[] of size N.
• Interpretation: id[i] is parent of i.
• Root of i is id[id[id[...id[i]...]]] (keep going until it doesn’t change).

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

Union. To merge subsets containing p and q, set the id of q’s root to the id of p’s root.

problem: trees can get tall
Quick-union: Java implementation

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

Quick-find defect.
- Union too expensive (N operations).
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.
- Trees can get tall.
- Find too expensive (could be N operations).

<table>
<thead>
<tr>
<th>algorithm</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>N * N</td>
<td>N</td>
</tr>
</tbody>
</table>

* includes cost of finding root

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

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

```
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 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
```

Weighted quick-union analysis

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

**Q.** How does depth of \( x \) increase by 1?

**A.** 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.

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:

- Merge smaller tree into larger tree.
- Update the \( sz[] \) array.

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

Weighted quick-union analysis

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

```
<table>
<thead>
<tr>
<th>algorithm</th>
<th>union</th>
<th>find</th>
</tr>
</thead>
<tbody>
<tr>
<td>quick-find</td>
<td>( N )</td>
<td>( I )</td>
</tr>
<tr>
<td>quick-union</td>
<td>( N^* )</td>
<td>( N )</td>
</tr>
<tr>
<td>weighted QU</td>
<td>( \lg N^* )</td>
<td>( \lg N )</td>
</tr>
</tbody>
</table>
```

* includes cost of finding root

**Q.** Stop at guaranteed acceptable performance?

**A.** No, easy to improve further.
Quick union with path compression. Just after computing the root of $p$, set the id of each examined node to $\text{root}(p)$.

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

Weighted quick-union with path compression example

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

WQUPC performance

Theorem. [Tarjan 1975] Starting from an empty data structure, any sequence of $M$ union and find operations on $N$ objects takes $O(N + M \log^* N)$ time.

• Proof is very difficult.
• But the algorithm is still simple!

Linear algorithm?
• Cost within constant factor of reading in the data.
• In theory, WQUPC is not quite linear.
• In practice, WQUPC is linear.

Amazing fact. No linear-time linking strategy exists.
**Summary**

**Bottom line.** WQUPC 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 \times N )</td>
</tr>
<tr>
<td>quick-union</td>
<td>( M \times N )</td>
</tr>
<tr>
<td>weighted QU</td>
<td>( N + M \times \log N )</td>
</tr>
<tr>
<td>QU + path compression</td>
<td>( N + M \times \log N )</td>
</tr>
<tr>
<td>weighted QU + path compression</td>
<td>( N + M \times \lg^* N )</td>
</tr>
</tbody>
</table>

\( M \) union-find operations on a set of \( N \) objects

**Ex.** \([10^9 \text{ unions and finds with } 10^9 \text{ objects}]\)
- WQUPC reduces time from 30 years to 6 seconds.
- Supercomputer won’t help much; good algorithm enables solution.

**Union-find applications**
- Percolation.
- Games (Go, Hex).
- Network 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.

**Percolation**

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

Percolation examples:
- Does not percolate:
  - No open site connected to top
- Percolates:
  - Site connected to top

<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 site vacancy probability $p$.

Percolation phase transition

Theory guarantees a sharp threshold $p^*$ (when $N$ is large).
• $p > p^*$: almost certainly percolates.
• $p < p^*$: almost certainly does not percolate.

Q: What is the value of $p^*$?

Q: What is the value of $p^*$?
Monte Carlo simulation

- Initialize N-by-N whole grid to be blocked.
- Make random sites open until top connected to bottom.
- Vacancy percentage estimates p*.

Q. How to declare a new site open?
A. Take union of new site and all adjacent open sites.
UF solution: a critical optimization

Q. How to avoid checking all pairs of top and bottom sites?
A. Create a virtual top and bottom objects:
    system percolates when virtual top and bottom objects are in same set.

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.