COS 226

Algorithms and Data Structures
Princeton University
Spring 2011

Robert Sedgewick

COS 226 course overview

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>advanced</td>
<td>B-tree, suffix arrays, maxflow, simplex</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, JPEG, 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!

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." — D. E. Knuth

Why study algorithms?

To solve problems that could not otherwise be addressed.

Ex. Network connectivity. [stay tuned]

They may unlock the secrets of life and of the universe.

Computational models are replacing mathematical models in scientific inquiry.

"Algorithms: a common language for nature, human, and computer." — Avi Wigderson
Why study algorithms?

For fun and profit.

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?

The usual suspects

Lectures. Introduce new material.
Precepts. Discussion, problem-solving, background for programming assignment.

First precept meets this week

<table>
<thead>
<tr>
<th>What</th>
<th>When</th>
<th>Where</th>
<th>Who</th>
<th>Office</th>
<th>Office Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>L01</td>
<td>MW 11-12:20</td>
<td>Friend 101</td>
<td>Prof. Sedgewick</td>
<td>CS 319</td>
<td>W 1:30-2:30</td>
</tr>
<tr>
<td>P01</td>
<td>Th 12:30</td>
<td>CS 102</td>
<td>Mos Ginzburg (lead)</td>
<td>CS 205</td>
<td>*</td>
</tr>
<tr>
<td>P01A</td>
<td>Th 12:30</td>
<td>Friend 108</td>
<td>Josh Kroll</td>
<td>Sherrard 320</td>
<td>*</td>
</tr>
<tr>
<td>P01B</td>
<td>Th 12:30</td>
<td>Friend 109</td>
<td>Dave Walker</td>
<td>CS 211</td>
<td>*</td>
</tr>
<tr>
<td>P03</td>
<td>Th 3:30</td>
<td>Friend 109</td>
<td>Mos Ginzburg</td>
<td>CS 205</td>
<td>*</td>
</tr>
<tr>
<td>P02</td>
<td>F 11</td>
<td>CS 102</td>
<td>Rob Schapira</td>
<td>CS 407</td>
<td>*</td>
</tr>
<tr>
<td>P02A</td>
<td>F 11</td>
<td>Friend 109</td>
<td>Aman Dhesi</td>
<td>CS 103B</td>
<td>*</td>
</tr>
</tbody>
</table>

* UTAs in Friend 106/107 (check web page for hours).

All questions answered

90-minute lecture? Soporiphic!

Each lecture will include a 5-minute break for a brief AQA session.
- ask a question!
- email a question
- preferably not about course content

All students must meet RS at office hours
- W 1:30 to 2:30 (more after break)
Coursework and grading

8 programming assignments. 40%
• Electronic submission.
• Due 11pm, starting Tuesday 2/9.

Exercises. 15%
• Due at beginning of lecture, starting Monday 2/8.

Exams.
• Closed-book with cheatsheet.
• Midterm. 15%
• Final. 30%

Staff discretion. To adjust borderline cases.

Course content.
• Course info.
• Exercises.
• Lecture slides.
• Programming assignments.
• Submit assignments.

Exams.
• Closed-book with cheatsheet.
• Midterm.
• Final.

Resources (web)

Course content.
• Course info.
• Exercises.
• Lecture slides.
• Programming assignments.
• Submit assignments.

Booksite (under construction).
• Brief summary of content.
• Download code from lecture, book, exercises, examples
• Links to references

Resources (books)

Required readings.
• to be published in early March

Preliminary edition (Fall 2010)
• on reserve in the library
• or, borrow a copy from a friend
• see the web for some Algs4 excerpts

Recommended Java reference.
• Introduction to Programming in Java.

What’s ahead?

Lecture 1. Union find.
Precept 1. Meets Thursday or Friday.
Lecture 2. Analysis of algorithms.

Exercise 1. Due in precept Thursday or Friday.
Assignment 1. Due via electronic submission at 11pm on Tuesday.

Right course? See Maia after lecture.
Placed out of COS 126? Review the following in Intro to Programming in Java
• Section 1.5: I/O and command-line interface.
• Section 4.1: Analysis of algorithms.
• Section 4.3: Stacks and queues.

Not registered? Go to any precept this week.
Change precept? Use SCORE. see Coleen Kenny-McGinley in CS 210
if the only precept you can attend is closed
1.5 Case Study

- 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.

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

More difficult problem: find the path.
Network connectivity: larger example

Q. Is there a path from p to q?

A. Yes.

but finding the path is more difficult: stay tuned (Chapter 4)

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.
- Terrorists communicating to develop a plot.

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)

Modeling the connections

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.

Implementing the operations

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

Union command. Replace sets 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.

```java
public class UF

UF(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 component?

void union(int p, int q) add connection between \( p \) and \( q \)

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
  - write out pair if they are not already connected

```java
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.find(p, q)) continue;
        uf.union(p, q);
        StdOut.println(p + " " + q);
    }
}
```

% more tiny.txt

```
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7
```

Quick-find [eager approach]

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

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

5 and 6 are connected
2, 3, 4, and 9 are connected

\[ \begin{array}{c}
\begin{array}{cccccccccc}
1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9
\end{array}
\end{array} \]

\[ \begin{array}{c}
\begin{array}{cccccccccc}
id[] & 0 & 1 & 9 & 9 & 9 & 6 & 6 & 7 & 8 & 9
\end{array}
\end{array} \]

dynamic connectivity
quick find
quick union
improvements
applications
Quick-find [eager approach]

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

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

The table shows:
- 5 and 6 are connected.
- 2, 3, 4, and 9 are connected.

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

<table>
<thead>
<tr>
<th><code>p</code></th>
<th><code>q</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

**Union.** To merge sets containing `p` and `q`, change all entries with `id[p]` to `id[q].`

<table>
<thead>
<tr>
<th><code>p</code></th>
<th><code>q</code></th>
</tr>
</thead>
</table>

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

The table shows:
- 3 and 6 not connected.

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

**Quick-find: Java implementation**

```java
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 find(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;
  }
}
```

**Quick-find example**

```java
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 find(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;
  }
}
```
Quick-find is too slow

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

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

Quick-find defect.
- Union too expensive.
- Trees are flat, but too expensive to keep them flat.
- Ex. Takes $N^2$ array accesses to process sequence of $N$ union commands on $N$ objects.

Quadratic algorithms do not scale

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

Ex. Huge problem for quick-find.
- 10⁹ union commands on 10⁹ objects.
- Quick-find takes more than 10¹⁸ 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 $id[id[id[...id[i]...]]]$.

Quick-find is too slow

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

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

Quick-find defect.
- Union too expensive.
- Trees are flat, but too expensive to keep them flat.
- Ex. Takes $N^2$ array accesses to process sequence of $N$ union commands on $N$ objects.
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]...]]].

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

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

Quick-union example
Quick-union: Java implementation

```java
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 find(int p, int q) {
        return root(p) == root(q);
    }

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

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

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

† includes cost of finding root

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).

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 small tree below large one.

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

- **Reference input**
  - $p$ $q$
  - 4 3
  - 3 8
  - 6 5
  - 9 4
  - 2 1
  - 8 9
  - 5 0
  - 7 2
  - 6 1
  - 1 0
  - 6 7

- **Worst-case input**
  - $p$ $q$
  - 0 1
  - 2 3
  - 4 5
  - 6 7
  - 0 2
  - 4 6
  - 0 4

Quick-union and weighted quick-union example

- **Quick-union**
- **Weighted**

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.

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

**Union.** Modify quick-union to:
- Merge smaller tree into larger tree.
- Update the $sz[]$ array.

```java
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

**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 $\lg N$. 

$$N = 10$$

$$\text{depth}(x) = 3 \times \lg N$$
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?

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

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

**Simpler one-pass variant:** halve the path length by making every other node in path point to its grandparent.

In practice. No reason not to! Keeps tree almost completely flat.
Weighted quick-union with path compression example

Weighted quick-union with path compression: amortized analysis

**Proposition.** Starting from an empty data structure, any sequence of $M$ union–find operations on $N$ objects makes at most proportional to $N + M \lg^* N$ array accesses.

- Proof is very difficult.
- Can be improved to $N + M \alpha(M, N)$.
- But the algorithm is still simple!

**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.** No linear-time algorithm 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>$M$</td>
</tr>
<tr>
<td>quick-union</td>
<td>$M$</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>

$M$ 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.
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.

<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>

Percolation

Likelihood of percolation

Depends on site vacancy probability $p$. 

$N = 8$

$p_{low}$ ($0.4$)
does not percolate

$p_{medium}$ ($0.6$)
percolates?

$p_{high}$ ($0.8$)
percolates
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 \( N \)-by-\( N \) whole grid to be blocked.
- Declare random sites open until top connected to bottom.
- Vacancy percentage estimates \( p^* \).

**UF solution to find percolation threshold**

**How to check whether system percolates?**

- Create an object for each site.
- Sites are in same set if connected by open sites.
- Percolates if any site in top row is in same set as any site in bottom row.

**How to declare a new site open?**

Brute force algorithm: check all \( N^2 \) pairs.
Q. How to declare a new site open?
A. Take union of new site and all adjacent open sites.

UF solution to find percolation threshold

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.

UF solution: a critical optimization

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.

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.

UF solution: a critical optimization

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.

UF solution: a critical optimization

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.

UF solution: a critical optimization

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.

UF solution: a critical optimization

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.

UF solution: a critical optimization

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.

UF solution: a critical optimization

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.

UF solution: a critical optimization

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.

Subtext of today's lecture (and this course)