Basic info

- Dr. Maraist
  - My office: 209 Wing Tech. Center
  - Class web site: cs.uwlax.edu/~jmaraist/353-spring-18

Today

- Some problems
- Introduction

Next time

- Stable matching

Friday

- Asymptotic measure
- Problems due: (Ch. 1) 1, 2, 4

Contact information

- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Chapter 1

Introduction:
Some Representative Problems
Matching Residents to Hospitals

**Goal.** Given a set of preferences among hospitals and medical school students, design a *self-reinforcing* admissions process.

**Unstable pair:** applicant $x$ and hospital $y$ are **unstable** if:
- $x$ prefers $y$ to its assigned hospital.
- $y$ prefers $x$ to one of its admitted students.

**Stable assignment.** Assignment with no unstable pairs.
- Natural and desirable condition.
- Individual self-interest will prevent any applicant/hospital deal from being made.
Stable Matching Problem

Goal. Given n men and n women, find a "suitable" matching.

- Participants rate members of opposite sex.
- Each man lists women in order of preference from best to worst.
- Each woman lists men in order of preference from best to worst.

**Men's Preference Profile**

<table>
<thead>
<tr>
<th>Favorite</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xavier</td>
<td>Amy</td>
<td>Bertha</td>
<td>Clare</td>
</tr>
<tr>
<td>Yancey</td>
<td>Bertha</td>
<td>Amy</td>
<td>Clare</td>
</tr>
<tr>
<td>Zeus</td>
<td>Amy</td>
<td>Bertha</td>
<td>Clare</td>
</tr>
</tbody>
</table>

**Women's Preference Profile**

<table>
<thead>
<tr>
<th>Favorite</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>Yancey</td>
<td>Xavier</td>
<td>Zeus</td>
</tr>
<tr>
<td>Bertha</td>
<td>Xavier</td>
<td>Yancey</td>
<td>Zeus</td>
</tr>
<tr>
<td>Clare</td>
<td>Xavier</td>
<td>Yancey</td>
<td>Zeus</td>
</tr>
</tbody>
</table>
Perfect matching: everyone is matched monogamously.
- Each man gets exactly one woman.
- Each woman gets exactly one man.

Stability: no incentive for some pair of participants to undermine assignment by joint action.
- In matching $M$, an unmatched pair $m-w$ is unstable if man $m$ and woman $w$ prefer each other to current partners.
- Unstable pair $m-w$ could each improve by eloping.

Stable matching: perfect matching with no unstable pairs.

Stable matching problem. Given the preference lists of $n$ men and $n$ women, find a stable matching if one exists.
Stable Matching Problem

Q. Is assignment X-C, Y-B, Z-A stable?

<table>
<thead>
<tr>
<th></th>
<th>Men's Preference Profile</th>
<th>Women's Preference Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>favorite</strong></td>
<td>1st</td>
<td>2nd</td>
</tr>
<tr>
<td>Xavier</td>
<td>Amy</td>
<td>Bertha</td>
</tr>
<tr>
<td>Yancey</td>
<td>Bertha</td>
<td>Amy</td>
</tr>
<tr>
<td>Zeus</td>
<td>Amy</td>
<td>Bertha</td>
</tr>
</tbody>
</table>
**Stable Matching Problem**

**Q.** Is assignment X-C, Y-B, Z-A stable?

**A.** No. Bertha and Xavier will hook up.

---

**Men's Preference Profile**

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xavier</td>
<td>Amy</td>
<td>Bertha</td>
<td>Clare</td>
</tr>
<tr>
<td>Yancey</td>
<td>Bertha</td>
<td>Amy</td>
<td>Clare</td>
</tr>
<tr>
<td>Zeus</td>
<td>Amy</td>
<td>Bertha</td>
<td>Clare</td>
</tr>
</tbody>
</table>

**Women's Preference Profile**

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>Yancey</td>
<td>Xavier</td>
<td>Zeus</td>
</tr>
<tr>
<td>Bertha</td>
<td>Xavier</td>
<td>Yancey</td>
<td>Zeus</td>
</tr>
<tr>
<td>Clare</td>
<td>Xavier</td>
<td>Yancey</td>
<td>Zeus</td>
</tr>
</tbody>
</table>
**Stable Matching Problem**

**Q.** Is assignment X-A, Y-B, Z-C stable?

**A.** Yes.

**Men's Preference Profile**

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xavier</td>
<td>Amy</td>
<td>Bertha</td>
<td>Clare</td>
</tr>
<tr>
<td>Yancey</td>
<td>Bertha</td>
<td>Amy</td>
<td>Clare</td>
</tr>
<tr>
<td>Zeus</td>
<td>Amy</td>
<td>Bertha</td>
<td>Clare</td>
</tr>
</tbody>
</table>

**Women's Preference Profile**

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>Yancey</td>
<td>Xavier</td>
<td>Zeus</td>
</tr>
<tr>
<td>Bertha</td>
<td>Xavier</td>
<td>Yancey</td>
<td>Zeus</td>
</tr>
<tr>
<td>Clare</td>
<td>Xavier</td>
<td>Yancey</td>
<td>Zeus</td>
</tr>
</tbody>
</table>
**Stable Roommate Problem**

**Q.** Do stable matchings always exist?  
**A.** Not obvious a priori.

**Stable roommate problem.**
- 2n people; each person ranks others from 1 to 2n-1.  
- Assign roommate pairs so that no unstable pairs.

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Adam</strong></td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td><strong>Bob</strong></td>
<td>C</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td><strong>Chris</strong></td>
<td>A</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td><strong>Doofus</strong></td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

**Observation.** Stable matchings do not always exist for stable roommate problem.
For us, what is a "solution" to a problem like Stable Matching?
What is a solution?

For us, what is a "solution" to a problem like Stable Matching?

- First an *algorithm*
  - Takes rankings tables
  - Returns a set of pairs
What is a solution?

For us, what is a "solution" to a problem like Stable Matching?

- First an *algorithm*
  - Takes rankings tables
  - Returns a set of pairs

- *Proof* that the algorithm finds a (perfect) matching
What is a solution?

For us, what is a "solution" to a problem like Stable Matching?

- First an *algorithm*
  - Takes rankings tables
  - Returns a set of pairs
- *Proof* that the algorithm finds a (perfect) matching
- Proof that the matching is stable
What is a solution?

For us, what is a "solution" to a problem like Stable Matching?

- First an *algorithm*
  - Takes rankings tables
  - Returns a set of pairs

- *Proof* that the algorithm finds a (perfect) matching
- Proof that the matching is stable
- An *accounting* of the cost of the algorithm
Input. Set of jobs with start times and finish times.

Goal. Find maximum cardinality subset of mutually compatible jobs.
Weighted Interval Scheduling

**Input.** Set of jobs with start times, finish times, and weights.

**Goal.** Find maximum weight subset of mutually compatible jobs.
Bipartite Matching

**Input.** Bipartite graph.

**Goal.** Find maximum cardinality matching.
Independent Set

**Input.** Graph.

**Goal.** Find **maximum cardinality** independent set.

subset of nodes such that no two joined by an edge

![Graph Diagram]
Competitive Facility Location

**Input.** Graph with weight on each node.

**Game.** Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

**Goal.** Select a maximum weight subset of nodes.

Second player can guarantee 20, but not 25.
Five Representative Problems

Variations on a theme: independent set.

Interval scheduling: \( n \log n \) greedy algorithm.
Weighted interval scheduling: \( n \log n \) dynamic programming algorithm.
Bipartite matching: \( n^k \) max-flow based algorithm.
Independent set: NP-complete.
Competitive facility location: PSPACE-complete.
Tentative schedule of topics

- **Weeks 1-2**  Introduction (Chapters 1-2)
- **Weeks 2-3**  Graphs (Chapter 3)
- **Weeks 4-5**  Greedy algorithms (Chapter 4)
- **Weeks 6-7**  Divide-and-conquer algorithms (Chapter 5)
- **Weeks 8-9**  Dynamic programming (Chapter 6)
- **Weeks 10-14**  To be decided (likely from Chapters 7-9)
Algorithms — January 26

Today
Asymptotic measure
Sec. 2.1-2.4

Monday
Stable matching
Read (re-read) Sec. 1.1

Wednesday
Priority queues and heaps
Read Sec. 2.5

Friday
Quiz on Sec. 2.1-2.4
Open-book (so bring the book)

Contact information

- Web: cs.uwlax.edu/~jmarais
- Email: jmarais@uwlax.edu
  - Expect replies within a business day (but often same-day)
- Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
How do we quantify efficiency for programs?

- By the clock time that passes during execution?
How do we quantify efficiency for programs?

- By the clock time that passes during execution?
- CPU time?
How do we quantify efficiency for programs?

- By the clock time that passes during execution?
- CPU time?
- Statements?
  - Machine code statements?
How do we quantify efficiency for programs?

- By the clock time that passes during execution?
- CPU time?
- Statements?
  - Machine code statements?
- Key operations
  - Could be comparisons, assignments, etc.
**Polynomial-Time**

**Brute force.** For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.
- Typically takes $2^N$ time or worse for inputs of size $N$.
- Unacceptable in practice.

**Desirable scaling property.** When the input size doubles, the algorithm should only slow down by some constant factor $C$.

There exists constants $c > 0$ and $d > 0$ such that on every input of size $N$, its running time is bounded by $cN^d$ steps.

**Def.** An algorithm is **poly-time** if the above scaling property holds.

$n!$ for stable matching with $n$ men and $n$ women

choose $C = 2^d$
Exercise 2.1

Suppose you have algorithms with these (exact) running times:

- $n^2$
- $n^3$
- $100n^2$
- $n \log n$
- $2^n$

How much slower do each get when you

- Double the input size?
- Increase the input size by one?
Worst-Case Analysis

Worst case running time. Obtain bound on largest possible running time of algorithm on input of a given size $N$.

- Generally captures efficiency in practice.
- Draconian view, but hard to find effective alternative.

Average case running time. Obtain bound on running time of algorithm on random input as a function of input size $N$.

- Hard (or impossible) to accurately model real instances by random distributions.
- Algorithm tuned for a certain distribution may perform poorly on other inputs.
**Worst-Case Polynomial-Time**

**Def.** An algorithm is **efficient** if its running time is polynomial.

**Justification:** It really works in practice!
- Although \(6.02 \times 10^{23} \times N^{20}\) is technically poly-time, it would be useless in practice.
- In practice, the poly-time algorithms that people develop almost always have low constants and low exponents.
- Breaking through the exponential barrier of brute force typically exposes some crucial structure of the problem.

**Exceptions.**
- Some poly-time algorithms do have high constants and/or exponents, and are useless in practice.
- Some exponential-time (or worse) algorithms are widely used because the worst-case instances seem to be rare.
What is the smallest integer value for which the first expression becomes greater than the second function?

1. $n^2$ and $10n$
2. $2^n$ and $2n^3$
3. $\frac{n^2}{\log n}$ and $n(\log n)^2$
4. $\frac{n^3}{2}$ and $n^{2.81}$
Why It Matters

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>30</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>50</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>100</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>1,000</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>10,000</td>
<td>$&lt; 1$ sec</td>
<td>$&lt; 1$ sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>100,000</td>
<td>$&lt; 1$ sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.
Combining time scales

What about these running times?

- $n^3 + n^2$
- $n^3 + 200n^2$
- $2^n + n^2$
- $2n^2$
Asymptotic Order of Growth

**Upper bounds.** $T(n)$ is $O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$.

**Lower bounds.** $T(n)$ is $\Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.

**Tight bounds.** $T(n)$ is $\Theta(f(n))$ if $T(n)$ is both $O(f(n))$ and $\Omega(f(n))$.

**Ex:** $T(n) = 32n^2 + 17n + 32$.
- $T(n)$ is $O(n^2)$, $O(n^3)$, $\Omega(n^2)$, $\Omega(n)$, and $\Theta(n^2)$.
- $T(n)$ is not $O(n)$, $\Omega(n^3)$, $\Theta(n)$, or $\Theta(n^3)$.
**Notation**

**Slight abuse of notation.** $T(n) = O(f(n))$.
- Not transitive:
  - $f(n) = 5n^3; \ g(n) = 3n^2$
  - $f(n) = O(n^3) = g(n)$
  - but $f(n) \neq g(n)$.
- Better notation: $T(n) \in O(f(n))$.

**Meaningless statement.** Any comparison-based sorting algorithm requires at least $O(n \log n)$ comparisons.
- Statement doesn't "type-check."
- Use $\Omega$ for lower bounds.
Another notational issue

We are discussing the runtime of *specific algorithms*

- Not the runtime or needs of a *problem*
Another notational issue

We are discussing the runtime of specific algorithms.

- Not the runtime or needs of a problem.
- Later, we will consider more categorical statements about the possible algorithms for a problem.
Properties

Transitivity.
- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Additivity.
- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
- If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$.
Asymptotic Bounds for Some Common Functions

**Polynomials.** $a_0 + a_1 n + ... + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.

**Polynomial time.** Running time is $O(n^d)$ for some constant $d$ independent of the input size $n$.

**Logarithms.** $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$.

$\uparrow$

can avoid specifying the base

**Logarithms.** For every $x > 0$, $\log n = O(n^x)$.

$\uparrow$

log grows slower than every polynomial

**Exponentials.** For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.

$\uparrow$

every exponential grows faster than every polynomial
Linear Time: $O(n)$

Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of $n$ numbers $a_1, \ldots, a_n$.

```plaintext
max ← a_1
for i = 2 to n {
    if (a_i > max)
        max ← a_i
}
```
Linear Time: $O(n)$

**Merge.** Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into sorted whole.

```
i = 1, j = 1
while (both lists are nonempty) {
    if ($a_i \leq b_j$) append $a_i$ to output list and increment $i$
    else append $b_j$ to output list and increment $j$
}
append remainder of nonempty list to output list
```

**Claim.** Merging two lists of size $n$ takes $O(n)$ time.

**Pf.** After each comparison, the length of output list increases by 1.
O(n log n) Time

O(n log n) time. Arises in divide-and-conquer algorithms. Also referred to as linearithmic time.

Sorting. Mergesort and heapsort are sorting algorithms that perform O(n log n) comparisons.

Largest empty interval. Given n time-stamps $x_1, ..., x_n$ on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

O(n log n) solution. Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.
Quadratic Time: $O(n^2)$

**Quadratic time.** Enumerate all pairs of elements.

**Closest pair of points.** Given a list of $n$ points in the plane $(x_1, y_1), \ldots, (x_n, y_n)$, find the pair that is closest.

**$O(n^2)$ solution.** Try all pairs of points.

$$
\min \leftarrow (x_1 - x_2)^2 + (y_1 - y_2)^2 \\
\text{for } i = 1 \text{ to } n \{ \\
\quad \text{for } j = i+1 \text{ to } n \{ \\
\quad\quad d \leftarrow (x_i - x_j)^2 + (y_i - y_j)^2 \\
\quad\quad \text{if } (d < \min) \\
\quad\quad\quad \min \leftarrow d \\
\quad \} \\
\} 
$$

**Remark.** $\Omega(n^2)$ seems inevitable, but this is just an illusion. → see chapter 5
Cubic Time: \(O(n^3)\)

Cubic time. Enumerate all triples of elements.

Set disjointness. Given \(n\) sets \(S_1, \ldots, S_n\) each of which is a subset of \(1, 2, \ldots, n\), is there some pair of these which are disjoint?

\(O(n^3)\) solution. For each pairs of sets, determine if they are disjoint.

```plaintext
foreach set \(S_i\) {
    foreach other set \(S_j\) {
        foreach element \(p\) of \(S_i\) {
            determine whether \(p\) also belongs to \(S_j\)
        }
        if (no element of \(S_i\) belongs to \(S_j\))
            report that \(S_i\) and \(S_j\) are disjoint
    }
}
```
Polynomial Time: \( O(n^k) \) Time

**Independent set of size \( k \).** Given a graph, are there \( k \) nodes such that no two are joined by an edge?

\( O(n^k) \) solution. Enumerate all subsets of \( k \) nodes.

```
foreach subset S of k nodes {
    check whether S in an independent set
    if (S is an independent set)
        report S is an independent set
}
```

- Check whether \( S \) is an independent set = \( O(k^2) \).
- Number of \( k \) element subsets = \( \binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!} \)
- \( O(k^2 n^k / k!) = O(n^k) \).

Poly-time for \( k=17 \), but not practical
Exponential Time

Independent set. Given a graph, what is maximum size of an independent set?

\(O(n^2 2^n)\) solution. Enumerate all subsets.

```
S* ← \emptyset
foreach subset S of nodes {
    check whether S in an independent set
    if (S is largest independent set seen so far)
        update S* ← S
}
```
Algorithms — January 29

Today

Wednesday
Priority queues and heaps (Sec. 2.5)

Friday
Quiz on Sec. 2.1-2.4
Open-book (so bring the book)

Monday
Graphs
Ch. 3

Events
- CS120 tutoring, Mo./We./Th. 6-9pm, 231 Wing
- CODERS, community service via CS
  - Regular meeting, Wednesdays 3:15pm, 231 Wing
  - Weekly study group, Thursdays, 4-6pm, 16 Wing
- CS Club, student ACM chapter
  - Regular meeting, Mondays, 6pm, 3214 Centennial
  - Womyn’s group, Tuesday 4pm, 3214 Centennial
- Internship opportunities via Career Services, www.uwlax.edu/career-services/handshake

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- Office hours: Tues. 1:30-3:30pm, Wed.
How to do the problems

(In this or any advanced concepts course)
How to do the problems

(In this or any advanced concepts course)

Do them all

- In a decent text, the authors develop skills across the problem set
- They’re walking you through a story

Still: the best practice is to do them all

This isn’t your only class

And even if it is, that amount of homework may still not be feasible

Not necessarily each of you, individually, alone, separately

Group work is great here

Especially in this size of class

Come see me when you are blocked
How to do the problems

(In this or any advanced concepts course)

Do them all

▶ In an decent text, the authors develop skills across the problem set
▶ They’re walking you through a story
▶ In a perfect world, my grading staff would go over each problem as you do them
▶ We do not live in a perfect world
▶ I'm going to assign you all the homework (quizzes, etc.) that I'm confident I can grade and return quickly
▶ Still: the best practice is to do them all
▶ This isn't your only class
▶ And even if it is, that amount of homework may still not be feasible
▶ Not necessarily each of you, individually, alone, separately
▶ Group work is great here
▶ Especially in this size of class
▶ Come see me when you are blocked
How to do the problems

(In this or any advanced concepts course)

Do them all

- In an decent text, the authors develop skills across the problem set
- They’re walking you through a story
- In a perfect world, my grading staff would go over each problem as you do them
  - We do not live in a perfect world
  - I’m going to assign you all the homework (quizzes, etc.) that I’m confident I can grade and return quickly

Come see me when you are blocked
How to do the problems

(In this or any advanced concepts course)

Do them all

▶ In an decent text, the authors develop skills across the problem set
▶ They’re walking you through a story
▶ In a perfect world, my grading staff would go over each problem as you do them
  ▶ We do not live in a perfect world
  ▶ I’m going to assign you all the homework (quizzes, etc.) that I’m confident I can grade and return quickly
▶ Still: the best practice is to do them all
  ▶ This isn’t your only class
    ▶ And even if it is, that amount of homework may still not be feasible
  ▶ Not necessarily each of you, individually, alone, separately
  ▶ Group work is great here
  ▶ Especially in this size of class
  ▶ Come see me when you are blocked
Properties

Transitivity.
- If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
- If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
- If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Additivity.
- If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
- If $f = \Omega(h)$ and $g = \Omega(h)$ then $f + g = \Omega(h)$.
- If $f = \Theta(h)$ and $g = O(h)$ then $f + g = \Theta(h)$. 
Asymptotic Bounds for Some Common Functions

**Polynomials.** \( a_0 + a_1 n + \ldots + a_d n^d \) is \( \Theta(n^d) \) if \( a_d > 0 \).

**Polynomial time.** Running time is \( O(n^d) \) for some constant \( d \) independent of the input size \( n \).

**Logarithms.** \( O(\log_a n) = O(\log_b n) \) for any constants \( a, b > 0 \).

**Logarithms.** For every \( x > 0 \), \( \log n = O(n^x) \).

**Exponentials.** For every \( r > 1 \) and every \( d > 0 \), \( n^d = O(r^n) \).

---

can avoid specifying the base

can avoid specifying the base

log grows slower than every polynomial

---

every exponential grows faster than every polynomial
So let’s turn back to stable matching
Propose-And-Reject Algorithm


Initialize each person to be free.
\begin{verbatim}
while (some man is free and hasn't proposed to every woman) {
  Choose such a man m
  w = 1st woman on m's list to whom m has not yet proposed
  if (w is free)
    assign m and w to be engaged
  else if (w prefers m to her fiancé m')
    assign m and w to be engaged, and m' to be free
  else
    w rejects m
}
\end{verbatim}
Proof of Correctness: Termination

Observation 1. Men propose to women in decreasing order of preference.

Observation 2. Once a woman is matched, she never becomes unmatched; she only "trades up."

Claim. Algorithm terminates after at most $n^2$ iterations of while loop.

Pf. Each time through the while loop a man proposes to a new woman. There are only $n^2$ possible proposals. □
Proof of Correctness: Perfection

**Claim.** All men and women get matched.

**Pf.** (by contradiction)

- Suppose, for sake of contradiction, that Zeus is not matched upon termination of algorithm.
- Then some woman, say Amy, is not matched upon termination.
- By Observation 2, Amy was never proposed to.
- But, Zeus proposes to everyone, since he ends up unmatched. □
Proof of Correctness: Stability

Claim. No unstable pairs.

Pf. (by contradiction)

- Suppose A-Z is an unstable pair: each prefers each other to partner in Gale-Shapley matching \( S^* \).
  
  - Case 1: Z never proposed to A.
    - \( \Rightarrow \) Z prefers his GS partner to A.
    - \( \Rightarrow \) A-Z is stable.
  
  - Case 2: Z proposed to A.
    - \( \Rightarrow \) A rejected Z (right away or later)
    - \( \Rightarrow \) A prefers her GS partner to Z.
    - \( \Rightarrow \) A-Z is stable.

- In either case A-Z is stable, a contradiction. □
Summary

Stable matching problem. Given n men and n women, and their preferences, find a stable matching if one exists.

Gale-Shapley algorithm. Guarantees to find a stable matching for any problem instance.

Q. How to implement GS algorithm efficiently?

Q. If there are multiple stable matchings, which one does GS find?
Efficient implementation. We describe $O(n^2)$ time implementation.

Representing men and women.
- Assume men are named 1, ..., n.
- Assume women are named 1', ..., n'.

Engagements.
- Maintain a list of free men, e.g., in a queue.
- Maintain two arrays $\text{wife}[m]$, and $\text{husband}[w]$.
  - set entry to 0 if unmatched
  - if $m$ matched to $w$ then $\text{wife}[m]=w$ and $\text{husband}[w]=m$

Men proposing.
- For each man, maintain a list of women, ordered by preference.
- Maintain an array $\text{count}[m]$ that counts the number of proposals made by man $m$. 
Women rejecting/accepting.

- Does woman $w$ prefer man $m$ to man $m'$?
- For each woman, create inverse of preference list of men.
- Constant time access for each query after $O(n)$ preprocessing.

<table>
<thead>
<tr>
<th>Amy</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
<th>7th</th>
<th>8th</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pref</td>
<td>8</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Amy</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
<th>6th</th>
<th>7th</th>
<th>8th</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inverse</td>
<td>4th</td>
<td>8th</td>
<td>2nd</td>
<td>5th</td>
<td>6th</td>
<td>7th</td>
<td>3rd</td>
<td>1st</td>
</tr>
</tbody>
</table>

Amy prefers man 3 to 6 since $\text{inverse}[3] < \text{inverse}[6]$

```plaintext
for i = 1 to n
    inverse[pref[i]] = i
```

2 7
Understanding the Solution

Q. For a given problem instance, there may be several stable matchings. Do all executions of Gale-Shapley yield the same stable matching? If so, which one?

An instance with two stable matchings.
- A-X, B-Y, C-Z.
- A-Y, B-X, C-Z.

<table>
<thead>
<tr>
<th>Xavier</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Yancey</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Zeus</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>Y</td>
<td>X</td>
</tr>
<tr>
<td>Bertha</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>Clare</td>
<td>X</td>
<td>Y</td>
</tr>
</tbody>
</table>
Q. For a given problem instance, there may be several stable matchings. Do all executions of Gale-Shapley yield the same stable matching? If so, which one?

Def. Man $m$ is a valid partner of woman $w$ if there exists some stable matching in which they are matched.

Man-optimal assignment. Each man receives best valid partner.

Claim. All executions of GS yield man-optimal assignment, which is a stable matching!
- No reason a priori to believe that man-optimal assignment is perfect, let alone stable.
- Simultaneously best for each and every man.
Man Optimality

Claim. GS matching S* is man-optimal.
Pf. (by contradiction)

- Suppose some man is paired with someone other than best partner. Men propose in decreasing order of preference ⇒ some man is rejected by valid partner.
- Let Y be first such man, and let A be first valid woman that rejects him.
- Let S be a stable matching where A and Y are matched.
- When Y is rejected, A forms (or reaffirms) engagement with a man, say Z, whom she prefers to Y.
- Let B be Z's partner in S.
- Z not rejected by any valid partner at the point when Y is rejected by A. Thus, Z prefers A to B.
- But A prefers Z to Y.
- Thus A-Z is unstable in S. □
Stable Matching Summary

Stable matching problem. Given preference profiles of \( n \) men and \( n \) women, find a stable matching.

no man and woman prefer to be with each other than assigned partner

Gale-Shapley algorithm. Finds a stable matching in \( O(n^2) \) time.

Man-optimality. In version of GS where men propose, each man receives best valid partner.

w is a valid partner of m if there exist some stable matching where m and w are paired

Q. Does man-optimality come at the expense of the women?
Woman Pessimality

**Woman-pessimal assignment.** Each woman receives worst valid partner.

**Claim.** GS finds woman-pessimal stable matching $S^*$.  

**Pf.**  
- Suppose $A-Z$ matched in $S^*$, but $Z$ is not worst valid partner for $A$.  
- There exists stable matching $S$ in which $A$ is paired with a man, say $Y$, whom she likes less than $Z$.  
- Let $B$ be $Z$'s partner in $S$.  
- $Z$ prefers $A$ to $B$. $\leftarrow$ man-optimality  
- Thus, $A-Z$ is an unstable in $S$. $\blacksquare$
Lessons Learned

Powerful ideas learned in course.
- Isolate underlying structure of problem.
- Create useful and efficient algorithms.

Potentially deep social ramifications. [legal disclaimer]
Algorithms — January 31

Today
Priority queues and heaps (Sec. 2.5)

Friday
Quiz on Sec. 2.1-2.4
  ▶ Let’s also revisit Exercise 1.4 on the quiz
Open-book (so bring the book)

Contact information
  ▶ Web: cs.uwlax.edu/~jmaraist
  ▶ Email: jmaraist@uwlax.edu
    ▶ Expect replies within a business day (but often same-day)
  ▶ Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
  ▶ Or by appointment
    ▶ Email at least a day ahead
    ▶ Send several times when you are available
    ▶ Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Designing priority queues

Simple implementations of priority queues
- One queue, with the first of highest priority marked
- Multiple queues, one for each priority
- Single array or linked list

What is the time cost for
- Insertion of a new element
  - Remember that we must find the right queue
- Finding the minimum element
- Deletion of the minimum element
- Deletion of an arbitrary element

Can we do better than linear time for all of them?
Heap implementation

A tree implemented as an array

- The parent of node $n$ is $n/2$ (rounding down)
- The children of node $n$ are nodes $2n$ and $2n+1$
- We are assuming a fixed maximum size
Figure 2.3 Values in a heap shown as a binary tree on the left, and represented as an array on the right. The arrows show the children for the top three nodes in the tree.
The Heapify-up process is moving element \( v \) toward the root.

**Figure 2.4** The Heapify-up process. Key 3 (at position 16) is too small (on the left). After swapping keys 3 and 11, the heap violation moves one step closer to the root of the tree (on the right).
Algorithm 2.8, page 61

Heapify-up(H,i):
   If i > 1 then
      let j = parent(i) = \lfloor i/2 \rfloor
      If key[H[i]] < key[H[j]] then
         swap the array entries H[i] and H[j]
         Heapify-up(H,j)
      Endif
   Endif
Endif
The Heapify-down process is moving element $w$ down, toward the leaves.

Figure 2.5 The Heapify-down process:. Key 21 (at position 3) is too big (on the left). After swapping keys 21 and 7, the heap violation moves one step closer to the bottom of the tree (on the right).
Algorithm 2.9, page 63

Heapify-down(H,i):
    Let n = length(H)
    If 2i > n then
        Terminate with H unchanged
    Else if 2i < n then
        Let left = 2i, and right = 2i + 1
        Let j be the index that minimizes key[H[left]] and key[H[right]]
    Else if 2i = n then
        Let j = 2i
    Endif
    If key[H[j]] < key[H[i]] then
        swap the array entries H[i] and H[j]
        Heapify-down(H,j)
    Endif
So what are the costs?

- Insertion of a new element
- Finding the minimum element
- Deletion of an arbitrary (possibly the minimum) element
So what are the costs?

- Insertion of a new element
  - Add it to the end, then heapify-up, then down
  - $O(\log n)$
- Finding the minimum element
- Deletion of an arbitrary (possibly the minimum) element
So what are the costs?

- Insertion of a new element
  - Add it to the end, then heapify-up, then down
  - $O(\log n)$
- Finding the minimum element
  - Always at the top — constant
- Deletion of an arbitrary (possibly the minimum) element
So what are the costs?

- Insertion of a new element
  - Add it to the end, then heapify-up, then down
  - $O(\log n)$

- Finding the minimum element
  - Always at the top — constant

- Deletion of an arbitrary (possibly the minimum) element
  - Move element at maximum position to point to be deleted, then heapify-up, then down
  - $O(\log n)$
Algorithms — February 5

Today
Graphs (Ch. 3)

Wednesday
More graphs

Friday

- Problems due: from Chapter 2, exercise 7; from Chapter 3, exercises 1-2.

Contact information

- Web: cs.uwlax.edu/~jmarais
- Email: jmarais@uwlax.edu
  - Expect replies within a business day (but often same-day)
- Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Undirected Graphs

Undirected graph. $G = (V, E)$
- $V$ = nodes.
- $E$ = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|$, $m = |E|$.

V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}
E = \{ 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 \}
n = 8
m = 11
World Wide Web

Web graph.
- Node: web page.
- Edge: hyperlink from one page to another.
9-11 Terrorist Network

Social network graph.

- **Node:** people.
- **Edge:** relationship between two people.

Paths and Connectivity

**Def.** A path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair $v_i, v_{i+1}$ is joined by an edge in $E$.

**Def.** A path is **simple** if all nodes are distinct.

**Def.** An undirected graph is **connected** if for every pair of nodes $u$ and $v$, there is a path between $u$ and $v$.
Cycles

Def. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k-1$ nodes are all distinct.

cycle $C = 1-2-4-5-3-1$
Def. An undirected graph is a tree if it is connected and does not contain a cycle.

Theorem. Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements imply the third.

- $G$ is connected.
- $G$ does not contain a cycle.
- $G$ has $n-1$ edges.
Rooted Trees

**Rooted tree.** Given a tree $T$, choose a root node $r$ and orient each edge away from $r$.

**Importance.** Models hierarchical structure.

![Diagram of rooted trees](image)

- A tree
- The same tree, rooted at 1

**Definitions:**
- $v$: a vertex
- $r$: root
- $\text{parent of } v$: the parent of $v$
- $\text{child of } v$: the child of $v$
Phylogeny Trees

Phylogeny trees. Describe evolutionary history of species.
GUI Containment Hierarchy

GUI containment hierarchy. Describe organization of GUI widgets.

Reference: http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html
Connectivity

**s-t connectivity problem.** Given two node s and t, is there a path between s and t?

**s-t shortest path problem.** Given two node s and t, what is the length of the shortest path between s and t?

**Applications.**
- Friendster.
- Maze traversal.
- Kevin Bacon number.
- Fewest number of hops in a communication network.
**Graph Representation: Adjacency Matrix**

**Adjacency matrix.** n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.
- Two representations of each edge.
- Space proportional to $n^2$.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.

```
1 2 3 4 5 6 7 8
1 0 1 1 0 0 0 0 0
2 1 0 1 1 1 0 0 0
3 1 1 0 0 1 0 1 1
4 0 1 0 1 1 0 0 0
5 0 1 1 1 0 1 0 0
6 0 0 0 0 1 0 0 0
7 0 0 1 0 0 0 0 1
8 0 0 1 0 0 0 1 0
```
**Graph Representation: Adjacency List**

**Adjacency list.** Node indexed array of lists.

- Two representations of each edge.
- Space proportional to $m + n$.
- Checking if $(u, v)$ is an edge takes $O(\text{deg}(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.
Breadth First Search

**BFS intuition.** Explore outward from \( s \) in all possible directions, adding nodes one "layer" at a time.

**BFS algorithm.**
- \( L_0 = \{ s \} \).
- \( L_1 = \) all neighbors of \( L_0 \).
- \( L_2 = \) all nodes that do not belong to \( L_0 \) or \( L_1 \), and that have an edge to a node in \( L_1 \).
- \( L_{i+1} = \) all nodes that do not belong to an earlier layer, and that have an edge to a node in \( L_i \).

**Theorem.** For each \( i \), \( L_i \) consists of all nodes at distance exactly \( i \) from \( s \). There is a path from \( s \) to \( t \) iff \( t \) appears in some layer.
Property. Let $T$ be a BFS tree of $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.
Theorem. The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Pf.

- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L[i]$
  - each node occurs on at most one list; for loop runs $\leq n$ times
  - when we consider node $u$, there are $\leq n$ incident edges $(u, v)$, and we spend $O(1)$ processing each edge

- Actually runs in $O(m + n)$ time:
  - when we consider node $u$, there are $\text{deg}(u)$ incident edges $(u, v)$
  - total time processing edges is $\Sigma_{u \in V} \text{deg}(u) = 2m$

  each edge $(u, v)$ is counted exactly twice in sum: once in $\text{deg}(u)$ and once in $\text{deg}(v)$
**Connected Component**

*Connected component.* Find all nodes reachable from $s$.

![Graph](image)

*Connected component containing node 1 = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}.*
**Flood Fill**

**Flood fill.** Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- **Node:** pixel.
- **Edge:** two neighboring lime pixels.
- **Blob:** connected component of lime pixels.

![Diagram of Flood Fill Process]
Flood Fill

Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- **Node**: pixel.
- **Edge**: two neighboring lime pixels.
- **Blob**: connected component of lime pixels.

recolor lime green blob to blue
Connected Component

**Connected component.** Find all nodes reachable from s.

---

\[ R \text{ will consist of nodes to which } s \text{ has a path} \]

Initially \( R = \{s\} \)

While there is an edge \((u, v)\) where \( u \in R \) and \( v \notin R \)
  
  Add \( v \) to \( R \)

Endwhile

---

**Theorem.** Upon termination, \( R \) is the connected component containing \( s \).

- **BFS** = explore in order of distance from \( s \).
- **DFS** = explore in a different way.
Bipartite Graphs

**Def.** An undirected graph \( G = (V, E) \) is **bipartite** if the nodes can be colored red or blue such that every edge has one red and one blue end.

**Applications.**
- **Stable marriage:** men = red, women = blue.
- **Scheduling:** machines = red, jobs = blue.

![A bipartite graph example](image)
More formal version of the definition:

An undirected graph $G = (V, E)$ is bipartite if there exist sets $V_1$ and $V_2$ such that

- $V_1 \cup V_2 = V$
- $V_1 \cap V_2 = \emptyset$
- For every edge $(v_1, v_2) \in E$, either
  - $v_1 \in V_1$ and $v_2 \in V_2$, or
  - $v_1 \in V_2$ and $v_2 \in V_1$
Testing bipartiteness. Given a graph $G$, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.

![A bipartite graph $G$](image1)

![Another drawing of $G$](image2)
Lemma. If a graph $G$ is bipartite, it cannot contain an odd length cycle.

Proof. Not possible to 2-color the odd cycle, let alone $G$. 
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf. (i)

- Suppose no edge joins two nodes in adjacent layers.
- By previous lemma, this implies all edges join nodes on same level.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.

Case (i)
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf. (ii)

- Suppose $(x, y)$ is an edge with $x, y$ in same level $L_j$.
- Let $z = \text{lca}(x, y) = \text{lowest common ancestor}$.
- Let $L_i$ be level containing $z$.
- Consider cycle that takes edge from $x$ to $y$, then path from $y$ to $z$, then path from $z$ to $x$.
- Its length is $1 + (j-i) + (j-i)$, which is odd.  

\[
\begin{align*}
(x, y) & \quad \text{path from } y \text{ to } z \\
\text{path from } z \text{ to } x
\end{align*}
\]

Diagram:

- $s$ is the root node.
- $x$ and $y$ are nodes in different layers $L_j$ and $L_i$.
- $z = \text{lca}(x, y)$ is the lowest common ancestor of $x$ and $y$.
- $z$ is in layer $L_i$.
- Path from $x$ to $y$ through $z$. 

Diagram:

- $s$ is the root node.
- $x$ and $y$ are nodes in different layers $L_j$ and $L_i$.
- $z = \text{lca}(x, y)$ is the lowest common ancestor of $x$ and $y$.
- $z$ is in layer $L_i$.
- Path from $x$ to $y$ through $z$. 

Diagram:

- $s$ is the root node.
- $x$ and $y$ are nodes in different layers $L_j$ and $L_i$.
- $z = \text{lca}(x, y)$ is the lowest common ancestor of $x$ and $y$.
- $z$ is in layer $L_i$.
- Path from $x$ to $y$ through $z$. 

Diagram:
Corollary. A graph $G$ is bipartite iff it contain no odd length cycle.
Directed Graphs

Directed graph. $G = (V, E)$
- Edge $(u, v)$ goes from node $u$ to node $v$.

Ex. Web graph - hyperlink points from one web page to another.
- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.
Ecological Food Web

Food web graph.

- Node = species.
- Edge = from prey to predator.

Graph Search

Directed reachability. Given a node s, find all nodes reachable from s.

Directed s-t shortest path problem. Given two node s and t, what is the length of the shortest path between s and t?

Graph search. BFS extends naturally to directed graphs.

Web crawler. Start from web page s. Find all web pages linked from s, either directly or indirectly.
**Strong Connectivity**

**Def.** Node $u$ and $v$ are **mutually reachable** if there is a path from $u$ to $v$ and also a path from $v$ to $u$.

**Def.** A graph is **strongly connected** if every pair of nodes is mutually reachable.

**Lemma.** Let $s$ be any node. $G$ is strongly connected iff every node is reachable from $s$, and $s$ is reachable from every node.

**Pf. $\Rightarrow$** Follows from definition.

**Pf. $\Leftarrow$**
- Path from $u$ to $v$: concatenate $u$-$s$ path with $s$-$v$ path.
- Path from $v$ to $u$: concatenate $v$-$s$ path with $s$-$u$ path.

\[\text{\textbullet}\]

ok if paths overlap
**Strong Connectivity: Algorithm**

**Theorem.** Can determine if $G$ is strongly connected in $O(m + n)$ time.

**Pf.**
- Pick any node $s$.
- Run BFS from $s$ in $G$.
- Run BFS from $s$ in $G_{rev}$.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. □

[Diagrams showing strongly connected and not strongly connected graphs]
Directed Acyclic Graphs

**Def.** An **DAG** is a directed graph that contains no directed cycles.

**Ex.** Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).

**Def.** A **topological order** of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, ..., v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).
Precedence Constraints

Precedence constraints. Edge \((v_i, v_j)\) means task \(v_i\) must occur before \(v_j\).

Applications.

- Course prerequisite graph: course \(v_i\) must be taken before \(v_j\).
- Compilation: module \(v_i\) must be compiled before \(v_j\). Pipeline of computing jobs: output of job \(v_i\) needed to determine input of job \(v_j\).
Lemma. If $G$ has a topological order, then $G$ is a DAG.

Pf. (by contradiction)

- Suppose that $G$ has a topological order $v_1, \ldots, v_n$ and that $G$ also has a directed cycle $C$. Let's see what happens.
- Let $v_i$ be the lowest-indexed node in $C$, and let $v_j$ be the node just before $v_i$; thus $(v_j, v_i)$ is an edge.
- By our choice of $i$, we have $i < j$.
- On the other hand, since $(v_j, v_i)$ is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction.
Directed Acyclic Graphs

**Lemma.** If $G$ has a topological order, then $G$ is a DAG.

**Q.** Does every DAG have a topological ordering?

**Q.** If so, how do we compute one?
Directed Acyclic Graphs

**Lemma.** If $G$ is a DAG, then $G$ has a node with no incoming edges.

**Pf.** (by contradiction)
- Suppose that $G$ is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$ we can walk backward to $u$.
- Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$.
- Repeat until we visit a node, say $w$, twice.
- Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle.
**Directed Acyclic Graphs**

**Lemma.** If $G$ is a DAG, then $G$ has a topological ordering.

**Pf.** (by induction on $n$)
- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
- $G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place $v$ first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges.

---

To compute a topological ordering of $G$:
- Find a node $v$ with no incoming edges and order it first
- Delete $v$ from $G$
- Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$
Theorem. Algorithm finds a topological order in $O(m + n)$ time.

Pf.

- Maintain the following information:
  - $\text{count}[w] =$ remaining number of incoming edges
  - $S =$ set of remaining nodes with no incoming edges

- Initialization: $O(m + n)$ via single scan through graph.

- Update: to delete $v$
  - remove $v$ from $S$
  - decrement $\text{count}[w]$ for all edges from $v$ to $w$, and add $w$ to $S$ if $c_{\text{count}[w]}$ hits 0
  - this is $O(1)$ per edge
Algorithms — February 7

Today
Graphs (Ch. 3)
▶ You should find the collected slides online now

Friday
▶ Graphs, greedy algorithms
▶ Problems due: from Chapter 2, exercise 7; from Chapter 3, exercises 1-2

Contact information
▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Trees

**Def.** An undirected graph is a **tree** if it is connected and does not contain a cycle.

**Theorem.** Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements imply the third.

- $G$ is connected.
- $G$ does not contain a cycle.
- $G$ has $n-1$ edges.
Proof (part 1)

Given: $G$ has $n$ nodes, is connected and acyclic
Prove: $G$ has $n - 1$ edges
Proof (part 1)
Given: G has n nodes, is connected and acyclic
Prove: G has n – 1 edges

Lemma: If G is acyclic and connected, then G has at least one vertex which connects to only one edge
  ▶ Assume the contrary

  ▶ Assume the contrary
Proof (part 1)
Given: $G$ has $n$ nodes, is connected and acyclic
Prove: $G$ has $n - 1$ edges

Lemma: If $G$ is acyclic and connected, then $G$ has at least one vertex which connects to only one edge

- Assume the contrary
- Let $v_a$ one endpoint of the longest simple path $p$ in $G$
  
  - (Or if there is more than one simple path with the same maximum length, pick any one of them)
Proof (part 1)

Given: \( G \) has \( n \) nodes, is connected and acyclic

Prove: \( G \) has \( n - 1 \) edges

Lemma: If \( G \) is acyclic and connected, then \( G \) has at least one vertex which connects to only one edge

- Assume the contrary
- Let \( v_a \) one endpoint of the longest simple path \( p \) in \( G \)
  - (Or if there is more than one simple path with the same maximum length, pick any one of them)
- But \( v \) must have another edge attached to it. Does that edge lead to a vertex which is in \( p \)?

Main result: by induction on the number \( n \) of nodes

- Case \( n = 2 \): construct directly
- For \( n > 2 \): by the lemma, there is some vertex \( v \) attached only to edge \( e \) in \( G = (V, E) \)

Consider the graph \( G' = (V \setminus \{v_a\}, E \setminus \{e\}) \)

- \( G' \) is still connected, still acyclic, and has \( n - 1 \) nodes
- So by the induction hypothesis, \( G' \) has \( n - 2 \) edges
- \( G \) has one more edge than \( G' \), so \( n - 1 \) edges — QED
Proof (part 1)

Given: $G$ has $n$ nodes, is connected and acyclic
Prove: $G$ has $n - 1$ edges

Lemma: If $G$ is acyclic and connected, then $G$ has at least one vertex which connects to only one edge

- Assume the contrary
- Let $v_a$ one endpoint of the longest simple path $p$ in $G$
  - (Or if there is more than one simple path with the same maximum length, pick any one of them)
- But $v$ must have another edge attached to it. Does that edge lead to a vertex which is in $p$?
  - If yes, then $G$ has a cycle

Main result: by induction on the number $n$ of nodes

- Case $n = 2$: construct directly
- For $n > 2$: by the lemma, there is some vertex $v$ attached only to edge $e$ in $G = (V, E)$
  - Consider the graph $G' = (V \{v\}, E \{e\})$
    - $G'$ is still connected, still acyclic, and has $n - 1$ nodes
    - So by the induction hypothesis, $G'$ has $n - 2$ edges
  - $G$ has one more edge than $G'$, so $n - 1$ edges — QED
Proof (part 1)

Given: $G$ has $n$ nodes, is connected and acyclic
Prove: $G$ has $n - 1$ edges

Lemma: If $G$ is acyclic and connected, then $G$ has at least one vertex which connects to only one edge

- Assume the contrary
- Let $v_a$ one endpoint of the longest simple path $p$ in $G$
  - (Or if there is more than one simple path with the same maximum length, pick any one of them)
- But $v$ must have another edge attached to it. Does that edge lead to a vertex which is in $p$?
  - If yes, then $G$ has a cycle
  - If no, then $p$ was not the longest simple path

Main result: by induction on the number $n$ of nodes

- Case $n = 2$: construct directly
- For $n > 2$: by the lemma, there is some vertex $v$ attached only to edge $e$ in $G$
  - Consider the graph $G' = (V \setminus \{v\}, E \setminus \{e\})$
  - $G'$ is still connected, still acyclic, and has $n - 1$ nodes
  - So by the induction hypothesis, $G'$ has $n - 2$ edges
- $G$ has one more edge than $G'$, so $n - 1$ edges — QED
Proof (part 1)

Given: $G$ has $n$ nodes, is connected and acyclic
Prove: $G$ has $n - 1$ edges

Lemma: If $G$ is acyclic and connected, then $G$ has at least one vertex which connects to only one edge

- Assume the contrary
- Let $v_a$ one endpoint of the longest simple path $p$ in $G$
  - (Or if there is more than one simple path with the same maximum length, pick any one of them)
- But $v$ must have another edge attached to it. Does that edge lead to a vertex which is in $p$?
  - If yes, then $G$ has a cycle
  - If no, then $p$ was not the longest simple path
- Either way, we reach a contradiction, so we reject the assumption, and conclude that $G$ does have at least one vertex which connects to only one edge

Main result: by induction on the number $n$ of nodes

- Case $n = 2$: construct directly
- For $n > 2$: by the lemma, there is some vertex $v$ attached only to edge $e$ in $G = (V, E)$
  - Consider the graph $G' = (V \setminus \{v\}, E \setminus \{e\})$
  - $G'$ is still connected, still acyclic, and has $n - 1$ nodes
  - So by the induction hypothesis, $G'$ has $n - 2$ edges
  - $G$ has one more edge than $G'$, so $n - 1$ edges — QED
Proof (part 1)

Given: $G$ has $n$ nodes, is connected and acyclic
Prove: $G$ has $n - 1$ edges

Lemma: If $G$ is acyclic and connected, then $G$ has at least one vertex which connects to only one edge

- Assume the contrary
  - Let $v_a$ one endpoint of the longest simple path $p$ in $G$
    - (Or if there is more than one simple path with the same maximum length, pick any one of them)
  - But $v$ must have another edge attached to it. Does that edge lead to a vertex which is in $p$?
    - If yes, then $G$ has a cycle
    - If no, then $p$ was not the longest simple path
  - Either way, we reach a contradiction, so we reject the assumption, and conclude that $G$ does have at least one vertex which connects to only one edge

Main result: by induction on the number $n$ of nodes
Proof (part 1)

Given: $G$ has $n$ nodes, is connected and acyclic
Prove: $G$ has $n - 1$ edges

Lemma: If $G$ is acyclic and connected, then $G$ has at least one vertex which connects to only one edge

- Assume the contrary
- Let $v_a$ one endpoint of the longest simple path $p$ in $G$
  - (Or if there is more than one simple path with the same maximum length, pick any one of them)
- But $v$ must have another edge attached to it. Does that edge lead to a vertex which is in $p$?
  - If yes, then $G$ has a cycle
  - If no, then $p$ was not the longest simple path
- Either way, we reach a contradiction, so we reject the assumption, and conclude that $G$ does have at least one vertex which connects to only one edge

Main result: by induction on the number $n$ of nodes

- Case $n = 2$: construct directly
Proof (part 1)

Given: $G$ has $n$ nodes, is connected and acyclic

Prove: $G$ has $n - 1$ edges

Lemma: If $G$ is acyclic and connected, then $G$ has at least one vertex which connects to only one edge

> Assume the contrary
> Let $v_a$ one endpoint of the longest simple path $p$ in $G$
>   (Or if there is more than one simple path with the same maximum length, pick any one of them)
> But $v$ must have another edge attached to it. Does that edge lead to a vertex which is in $p$?
>   If yes, then $G$ has a cycle
>   If no, then $p$ was not the longest simple path
> Either way, we reach a contradiction, so we reject the assumption, and conclude that $G$ does have at least one vertex which connects to only one edge

Main result: by induction on the number $n$ of nodes

> Case $n = 2$: construct directly
> For $n > 2$: by the lemma, there is some vertex $v$ attached only to edge $e$ in $G = (V, E)$
Proof (part 1)

Given: $G$ has $n$ nodes, is connected and acyclic
Prove: $G$ has $n - 1$ edges

Lemma: If $G$ is acyclic and connected, then $G$ has at least one vertex which connects to only one edge

- Assume the contrary
- Let $v_a$ one endpoint of the longest simple path $p$ in $G$
  - (Or if there is more than one simple path with the same maximum length, pick any one of them)
- But $v$ must have another edge attached to it. Does that edge lead to a vertex which is in $p$?
  - If yes, then $G$ has a cycle
  - If no, then $p$ was not the longest simple path
- Either way, we reach a contradiction, so we reject the assumption, and conclude that $G$ does have at least one vertex which connects to only one edge

Main result: by induction on the number $n$ of nodes

- Case $n = 2$: construct directly
- For $n > 2$: by the lemma, there is some vertex $v$ attached only to edge $e$ in $G = (V, E)$
  - Consider the graph $G' = (V \setminus \{v\}, E \setminus \{e\})$
Proof (part 1)

Given: \( G \) has \( n \) nodes, is connected and acyclic
Prove: \( G \) has \( n - 1 \) edges

Lemma: If \( G \) is acyclic and connected, then \( G \) has at least one vertex which connects to only one edge

- Assume the contrary
- Let \( v_a \) one endpoint of the longest simple path \( p \) in \( G \)
  - (Or if there is more than one simple path with the same maximum length, pick any one of them)
- But \( v \) must have another edge attached to it. Does that edge lead to a vertex which is in \( p \)?
  - If yes, then \( G \) has a cycle
  - If no, then \( p \) was not the longest simple path
- Either way, we reach a contradiction, so we reject the assumption, and conclude that \( G \) does have at least one vertex which connects to only one edge

Main result: by induction on the number \( n \) of nodes

- Case \( n = 2 \): construct directly
- For \( n > 2 \): by the lemma, there is some vertex \( v \) attached only to edge \( e \) in \( G = (V, E) \)
  - Consider the graph \( G' = (V \setminus \{v\}, E \setminus \{e\}) \)
  - \( G' \) is still connected, still acyclic, and has \( n - 1 \) nodes
    - So by the induction hypothesis, \( G' \) has \( n - 2 \) edges
Proof (part 1)

Given: \( G \) has \( n \) nodes, is connected and acyclic
Prove: \( G \) has \( n - 1 \) edges

Lemma: If \( G \) is acyclic and connected, then \( G \) has at least one vertex which connects to only one edge

- Assume the contrary
- Let \( v_a \) one endpoint of the longest simple path \( p \) in \( G \)
  - (Or if there is more than one simple path with the same maximum length, pick any one of them)
- But \( v \) must have another edge attached to it. Does that edge lead to a vertex which is in \( p \)?
  - If yes, then \( G \) has a cycle
  - If no, then \( p \) was not the longest simple path
- Either way, we reach a contradiction, so we reject the assumption, and conclude that \( G \) does have at least one vertex which connects to only one edge

Main result: by induction on the number \( n \) of nodes

- Case \( n = 2 \): construct directly
- For \( n > 2 \): by the lemma, there is some vertex \( v \) attached only to edge \( e \) in \( G = (V, E) \)
  - Consider the graph \( G' = (V \setminus \{v\}, E \setminus \{e\}) \)
  - \( G' \) is still connected, still acyclic, and has \( n - 1 \) nodes
    - So by the induction hypothesis, \( G' \) has \( n - 2 \) edges
  - \( G \) has one more edge than \( G' \), so \( n - 1 \) edges — QED
Proof (part 2)

Given: \( G = (\{v_1, \cdots, v_n\}, \{e_1, \cdots, e_{n-1}\}) \) is connected,
Prove: \( G \) does not contain a cycle
Proof (part 2)

Given: \( G = (\{v_1, \cdots, v_n\}, \{e_1, \cdots, e_{n-1}\}) \) is connected,
Prove: \( G \) does not contain a cycle

- Assume the contrary: that \( G \) does contain a cycle
  - Moreover we can assume without loss of generality that the largest cycle
    - Has \( h \) edges, and
    - Runs from \( v_1 \) through \( v_h \) and back to \( v_1 \)

Then there are \( n-h \) vertices not in the cycle.
Since \( G \) is connected, we can order them so that for every \( v_j \) with \( j > h \), there is at least one link from \( v_j \) to some \( v_i \) with \( i < j \).
This means that there are (at least) \( n-h \) edges attached to these vertices.
So there must be at least \( h + n - h = n \) edges in \( G \).
But this contradicts the assumption that \( G \) has \( n-1 \) edges.
So we reject the assumption, and conclude that \( G \) is acyclic — QED
Proof (part 2)

Given: \( G = (\{v_1, \cdots, v_n\}, \{e_1, \cdots, e_{n-1}\}) \) is connected,
Prove: \( G \) does not contain a cycle

- Assume the contrary: that \( G \) does contain a cycle
  - Moreover we can assume without loss of generality that the largest cycle
    - Has \( h \) edges, and
    - Runs from \( v_1 \) through \( v_h \) and back to \( v_1 \)
  - Thus there are \( n - h \) vertices not in the cycle

But this contradicts the assumption that \( G \) has \( n - 1 \) edges
So we reject the assumption, and conclude that \( G \) is acyclic — QED
Proof (part 2)

Given: $G = (\{v_1, \ldots, v_n\}, \{e_1, \ldots, e_{n-1}\})$ is connected,

Prove: $G$ does not contain a cycle

- Assume the contrary: that $G$ does contain a cycle
  - Moreover we can assume without loss of generality that the largest cycle
    - Has $h$ edges, and
    - Runs from $v_1$ through $v_h$ and back to $v_1$
  - Thus there are $n - h$ vertices not in the cycle
    - Since $G$ is connected, we can order them so that for every $v_j$ with $j > h$, there is at least one link from $v_j$ to some $v_i$ with $i < j$
    - This means that there are (at least) $n - h$ edges attached to these vertices

So there must be at least $h + n - h = n$ edges in $G$

But this contradicts the assumption that $G$ has $n - 1$ edges

So we reject the assumption, and conclude that $G$ is acyclic — QED
Given: \( G = (\{v_1, \cdots, v_n\}, \{e_1, \cdots, e_{n-1}\}) \) is connected,
Prove: \( G \) does not contain a cycle

- Assume the contrary: that \( G \) does contain a cycle
  - Moreover we can assume without loss of generality that the largest cycle
    - Has \( h \) edges, and
    - Runs from \( v_1 \) through \( v_h \) and back to \( v_1 \)
  - Thus there are \( n - h \) vertices not in the cycle
    - Since \( G \) is connected, we can order them so that for every \( v_j \) with \( j > h \), there is at least one link from \( v_j \) to some \( v_i \) with \( i < j \)
    - This means that there are (at least) \( n - h \) edges attached to these vertices
  - So there must be at least \( h + n - h = n \) edges in \( G \)
    - But this contradicts the assumption that \( G \) has \( n - 1 \) edges
    - So we reject the assumption, and conclude that \( G \) is acyclic — QED
Proof (part 3)

Given: $G = (\{v_1, \cdots, v_n\}, \{e_1, \cdots, e_{n-1}\})$ is acyclic

Prove: $G$ is connected
Given: $G = (\{v_1, \cdots, v_n\}, \{e_1, \cdots, e_{n-1}\})$ is acyclic
Prove: $G$ is connected

By induction on the number $n$ of nodes
Proof (part 3)

Given: $G = (\{v_1, \cdots, v_n\}, \{e_1, \cdots, e_{n-1}\})$ is acyclic

Prove: $G$ is connected

By induction on the number $n$ of nodes

- Base case $n = 3$, observing the construction directly
Proof (part 3)

Given: $G = (\{v_1, \cdots, v_n\}, \{e_1, \cdots, e_{n-1}\})$ is acyclic

Prove: $G$ is connected

By induction on the number $n$ of nodes

- Base case $n = 3$, observing the construction directly
- For $n > 3$, rely again on the lemma to remove one vertex $v$ and its sole edge $e$ for $G'$
  - Now $G'$ is acyclic, and connected
  - And $G$ is also connected, since $v$ has a path to any other node via $e$ — QED
Rooted Trees

**Rooted tree.** Given a tree $T$, choose a root node $r$ and orient each edge away from $r$.

**Importance.** Models hierarchical structure.

![Diagram of a tree and the same tree rooted at 1]
Phylogeny Trees

Phylogeny trees. Describe evolutionary history of species.
GUI Containment Hierarchy

GUI containment hierarchy. Describe organization of GUI widgets.

Reference: http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html
Connectivity

**s-t connectivity problem.** Given two node $s$ and $t$, is there a path between $s$ and $t$?

**s-t shortest path problem.** Given two node $s$ and $t$, what is the length of the shortest path between $s$ and $t$?

**Applications.**
- Friendster.
- Maze traversal.
- Kevin Bacon number.
- Fewest number of hops in a communication network.
Graph Representation: Adjacency Matrix

**Adjacency matrix.** n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.

- Two representations of each edge.
- Space proportional to $n^2$.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.
Graph Representation: Adjacency List

**Adjacency list.** Node indexed array of lists.
- Two representations of each edge.
- Space proportional to $m + n$.
- Checking if $(u, v)$ is an edge takes $O(\text{deg}(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.

\[
\text{degree} = \text{number of neighbors of } u
\]
Breadth First Search

**BFS intuition.** Explore outward from $s$ in all possible directions, adding nodes one "layer" at a time.

**BFS algorithm.**
- $L_0 = \{ s \}$.
- $L_1 =$ all neighbors of $L_0$.
- $L_2 =$ all nodes that do not belong to $L_0$ or $L_1$, and that have an edge to a node in $L_1$.
- $L_{i+1} =$ all nodes that do not belong to an earlier layer, and that have an edge to a node in $L_i$.

**Theorem.** For each $i$, $L_i$ consists of all nodes at distance exactly $i$ from $s$. There is a path from $s$ to $t$ iff $t$ appears in some layer.
Today

- Graphs
- Problems due: from Chapter 2, exercise 7; from Chapter 3, exercises 1-2

Monday

- Greedy algorithms

Next Friday

- Problems due: from Chapter 2, exercise 7; from Chapter 3, exercises 1-2

Contact information

- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Breadth First Search

**BFS intuition.** Explore outward from s in all possible directions, adding nodes one "layer" at a time.

**BFS algorithm.**
- $L_0 = \{s\}$.
- $L_1$ = all neighbors of $L_0$.
- $L_2$ = all nodes that do not belong to $L_0$ or $L_1$, and that have an edge to a node in $L_1$.
- $L_{i+1}$ = all nodes that do not belong to an earlier layer, and that have an edge to a node in $L_i$.

**Theorem.** For each $i$, $L_i$ consists of all nodes at distance exactly $i$ from $s$. There is a path from $s$ to $t$ iff $t$ appears in some layer.
**Breadth First Search**

**Property.** Let $T$ be a BFS tree of $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.
Graph Representation: Adjacency List

**Adjacency list.** Node indexed array of lists.
- Two representations of each edge.
- Space proportional to $m + n$.
- Checking if $(u, v)$ is an edge takes $O(deg(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.

degree = number of neighbors of $u$
Breadth First Search: Analysis

**Theorem.** The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

**Pf.**
- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L[i]$
  - each node occurs on at most one list; for loop runs $\leq n$ times
  - when we consider node $u$, there are $\leq n$ incident edges $(u, v)$, and we spend $O(1)$ processing each edge

- Actually runs in $O(m + n)$ time:
  - when we consider node $u$, there are $\deg(u)$ incident edges $(u, v)$
  - total time processing edges is $\sum_{u \in V} \deg(u) = 2m$

\[ \cdot \]

\[ \sum_{u \in V} \deg(u) \]

\[ = 2m \]

\[ \cdot \]

\[ \text{each edge } (u, v) \text{ is counted exactly twice} \]

\[ \text{in sum: once in } \deg(u) \text{ and once in } \deg(v) \]
Connected component. Find all nodes reachable from $s$.

Connected component containing node 1 = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}.
Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- Node: pixel.
- Edge: two neighboring lime pixels.
- Blob: connected component of lime pixels.
Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- **Node:** pixel.
- **Edge:** two neighboring lime pixels.
- **Blob:** connected component of lime pixels.

Click in the picture to fill that area with color.
**Connected Component**

**Connected component.** Find all nodes reachable from $s$.

---

$R$ will consist of nodes to which $s$ has a path.
Initially $R = \{s\}$
While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
   Add $v$ to $R$
Endwhile

---

**Theorem.** Upon termination, $R$ is the connected component containing $s$.
- **BFS** = explore in order of distance from $s$.
- **DFS** = explore in a different way.
Def. An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

Applications.

- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.
Bipartite graph

More formal version of the definition:

An undirected graph $G = (V, E)$ is bipartite if there exist sets $V_1$ and $V_2$ such that

- $V_1 \cup V_2 = V$
- $V_1 \cap V_2 = \emptyset$
- For every edge $(v_1, v_2) \in E$, either
  - $v_1 \in V_1$ and $v_2 \in V_2$, or
  - $v_1 \in V_2$ and $v_2 \in V_1$
Testing Bipartiteness

Testing bipartiteness. Given a graph $G$, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)

- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.

![Diagram of a bipartite graph $G$](image1)

![Another drawing of $G$](image2)
An Obstruction to Bipartiteness

**Lemma.** If a graph $G$ is bipartite, it cannot contain an odd length cycle.

**Pf.** Not possible to 2-color the odd cycle, let alone $G$. 

![Bipartite and non-bipartite graphs](image-url)
**Bipartite Graphs**

**Lemma.** Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

![Case (i)](image1)

![Case (ii)](image2)
Corollary. A graph $G$ is bipartite iff it contain no odd length cycle.
Directed Graphs

Directed graph. $G = (V, E)$
- Edge $(u, v)$ goes from node $u$ to node $v$.

Ex. Web graph - hyperlink points from one web page to another.
- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.
Ecological Food Web

Food web graph.
- Node = species.
- Edge = from prey to predator.

Graph Search

**Directed reachability.** Given a node $s$, find all nodes reachable from $s$.

**Directed $s$-$t$ shortest path problem.** Given two node $s$ and $t$, what is the length of the shortest path between $s$ and $t$?

**Graph search.** BFS extends naturally to directed graphs.

**Web crawler.** Start from web page $s$. Find all web pages linked from $s$, either directly or indirectly.
Strong Connectivity

Def. Node u and v are **mutually reachable** if there is a path from u to v and also a path from v to u.

Def. A graph is **strongly connected** if every pair of nodes is mutually reachable.

Lemma. Let s be any node. G is strongly connected iff every node is reachable from s, and s is reachable from every node.

Pf. ⇒ Follows from definition.
Pf. ⇐ Path from u to v: concatenate u-s path with s-v path.
Path from v to u: concatenate v-s path with s-u path. □

\[ \text{ok if paths overlap} \]
**Strong Connectivity: Algorithm**

**Theorem.** Can determine if $G$ is strongly connected in $O(m + n)$ time.

**Pf.**
- Pick any node $s$.
- Run BFS from $s$ in $G$.
- Run BFS from $s$ in $G^{\text{rev}}$.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. □

**Examples:**
- **Strongly connected**
- **Not strongly connected**
**Directed Acyclic Graphs**

**Def.** An **DAG** is a directed graph that contains no directed cycles.

**Ex.** Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).

**Def.** A **topological order** of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, ..., v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).
Precedence Constraints

Precedence constraints. Edge \((v_i, v_j)\) means task \(v_i\) must occur before \(v_j\).

Applications.

- Course prerequisite graph: course \(v_i\) must be taken before \(v_j\).
- Compilation: module \(v_i\) must be compiled before \(v_j\). Pipeline of computing jobs: output of job \(v_i\) needed to determine input of job \(v_j\).
Algorithms — February 12

Today
- Graphs, Greedy algorithms

Wednesday
- Greedy algorithms

Friday
- Greedy algorithms
- Problems due: from Chapter 3, exercises 5, 6, 10

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Exercise 3.2

Note the difference between what we can do in an *algorithm* and in a *proof*

- In a proof we can assert the existence of things, maybe with maximum/minimum or other bounds/properties
- In an algorithm, we must give the steps to produce such artifacts (or refer to previous algorithms and their cost)
Exercise 3.2

Note the difference between what we can do in an *algorithm* and in a *proof*

- In a proof we can assert the existence of things, maybe with maximum/minimum or other bounds/properties
- In an algorithm, we must give the steps to produce such artifacts (or refer to previous algorithms and their cost)
- When modifying an algorithm we have already analyzed, we can just give the difference
  - But we must be mindful of non-constant-time changes — they may change the overall cost of the algorithm
  - For example, if we add an *inner* loop

Here: Use a BFS

Two arrays, indexed by the node number

- *visited* contains booleans — usual for BFS
- Additional array *via* to contain node numbers, but initially all -1

When visiting any node

- Set its *visited* entry to true as usual

When enqueuing the neighbor $j$ of node $i$

- If *visited*[$j$] then we have found a cycle, can stop
- To retrieve cycle, start from *via*[$i$] and *via*[$j$]
- Else set *via*[$j$] to $i$
Exercise 3.2

Note the difference between what we can do in an *algorithm* and in a *proof*

- In a proof we can assert the existence of things, maybe with maximum/minimum or other bounds/properties
- In an algorithm, we must give the steps to produce such artifacts (or refer to previous algorithms and their cost)
- When modifying an algorithm we have already analyzed, we can just give the difference
  - But we must be mindful of non-constant-time changes — they may change the overall cost of the algorithm
  - For example, if we add an *inner* loop

Here: Use a BFS

- Two arrays, indexed by the node number
  - *visited* contains booleans — usual for BFS
  - Additional array *via* to contain node numbers, but initially all -1
- When visiting any node
  - Set its *visited* entry to true as usual
- When enqueueuing the neighbor *j* of node *i*
  - If *visited*[j] then we have found a cycle, can stop
    - To retrieve cycle, start from *via*[i] and *via*[j]
  - Else set *via*[j] to *i*
Lemma. If $G$ has a topological order, then $G$ is a DAG.

Pf. (by contradiction)
- Suppose that $G$ has a topological order $v_1, \ldots, v_n$ and that $G$ also has a directed cycle $C$. Let's see what happens.
- Let $v_i$ be the lowest-indexed node in $C$, and let $v_j$ be the node just before $v_i$; thus $(v_j, v_i)$ is an edge.
- By our choice of $i$, we have $i < j$.
- On the other hand, since $(v_j, v_i)$ is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. □
Directed Acyclic Graphs

**Lemma.** If $G$ has a topological order, then $G$ is a DAG.

**Q.** Does every DAG have a topological ordering?

**Q.** If so, how do we compute one?
Lemma. If $G$ is a DAG, then $G$ has a node with no incoming edges.

Pf. (by contradiction)

- Suppose that $G$ is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$ we can walk backward to $u$.
- Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$.
- Repeat until we visit a node, say $w$, twice.
- Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle.
Lemma. If $G$ is a DAG, then $G$ has a topological ordering.

Pf. (by induction on $n$)

- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
- $G - \{v\}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{v\}$ has a topological ordering.
- Place $v$ first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges.

---

To compute a topological ordering of $G$:

- Find a node $v$ with no incoming edges and order it first
- Delete $v$ from $G$
- Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$
Theorem. Algorithm finds a topological order in $O(m + n)$ time.

Pf.

- Maintain the following information:
  - $\text{count}[w] = \text{remaining number of incoming edges}$
  - $S = \text{set of remaining nodes with no incoming edges}$
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete $v$
  - remove $v$ from $S$
  - decrement $\text{count}[w]$ for all edges from $v$ to $w$, and add $w$ to $S$ if $\text{count}[w]$ hits 0
  - this is $O(1)$ per edge
Chapter 4
Greedy Algorithms
Interval Scheduling

Interval scheduling.

- Job $j$ starts at $s_j$ and finishes at $f_j$.
- Two jobs compatible if they don't overlap.
- Goal: find maximum subset of mutually compatible jobs.
Interval Scheduling: Greedy Algorithms

**Greedy template.** Consider jobs in some natural order. Take each job provided it’s compatible with the ones already taken.

- **[Earliest start time]** Consider jobs in ascending order of $s_j$.
- **[Earliest finish time]** Consider jobs in ascending order of $f_j$.
- **[Shortest interval]** Consider jobs in ascending order of $f_j - s_j$.
- **[Fewest conflicts]** For each job $j$, count the number of conflicting jobs $c_j$. Schedule in ascending order of $c_j$. 
**Greedy template.** Consider jobs in some natural order. Take each job provided it's compatible with the ones already taken.

- Counterexample for earliest start time
- Counterexample for shortest interval
- Counterexample for fewest conflicts
Greedy algorithm. Consider jobs in increasing order of finish time. Take each job provided it's compatible with the ones already taken.

Sort jobs by finish times so that $f_1 \leq f_2 \leq \ldots \leq f_n$.

```
set of jobs selected
A ← ∅
for j = 1 to n {
    if (job j compatible with A)
        A ← A ∪ {j}
}
return A
```

Implementation. $O(n \log n)$.
- Remember job $j^*$ that was added last to $A$.
- Job $j$ is compatible with $A$ if $s_j \geq f_{j^*}$.
**Theorem.** Greedy algorithm is optimal.

**Pf.** (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots, i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots, j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of $r$. 

![Diagram showing interval scheduling and greedy algorithm comparison]

**Greedy:**

- $i_1$
- $i_2$
- $i_r$
- $i_{r+1}$

**OPT:**

- $j_1$
- $j_2$
- $j_r$
- $j_{r+1}$
- $\cdots$

**Why not replace job $j_{r+1}$ with job $i_{r+1}$?**

**job $i_{r+1}$ finishes before $j_{r+1}$**
Theorem. Greedy algorithm is optimal.

Pf. (by contradiction)

- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, \ldots i_k$ denote set of jobs selected by greedy.
- Let $j_1, j_2, \ldots j_m$ denote set of jobs in the optimal solution with $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ for the largest possible value of $r$.

Greedy:  
\[  \begin{array}{|c|c|c|} \hline
  i_1 & i_2 & i_r & i_{r+1} \\ \hline
\end{array} \]

OPT:  
\[  \begin{array}{|c|c|c|} \hline
  j_1 & j_2 & j_r & i_{r+1} & \ldots \\ \hline
\end{array} \]

Job $i_{r+1}$ finishes before $j_{r+1}$

Solution still feasible and optimal, but contradicts maximality of $r$. 

Is greed good?

- Very often, it produces inferior solutions
- But sometimes it does give the best possible answer
  - And sometimes, it gives *good enough* results
- The skill of using greedy algorithms is in
  1. Picking the order in which we search
  2. Characterizing an "optimal solution"
  3. Demonstrating that an algorithm is optimal
Algorithms — February 14

Today

▶ Greedy algorithms

Friday

▶ Greedy algorithms
▶ Problems due: from Chapter 3, exercises 5, 6, 10

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Is greed good?

- Very often, it produces inferior solutions
- But sometime it does give the best possible answer
  - And sometimes, it gives *good enough* results
- The skill of using greedy algorithms is in
  1. Picking the order in which we search
  2. Characterizing an "optimal solution"
  3. Demonstrating that an algorithm is optimal
Interval Partitioning

Interval partitioning.

- Lecture \( j \) starts at \( s_j \) and finishes at \( f_j \).
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses 4 classrooms to schedule 10 lectures.
Interval partitioning.

- Lecture $j$ starts at $s_j$ and finishes at $f_j$.
- Goal: find minimum number of classrooms to schedule all lectures so that no two occur at the same time in the same room.

Ex: This schedule uses only 3.
Def. The depth of a set of open intervals is the maximum number that contain any given time.

Key observation. Number of classrooms needed ≥ depth.

Ex: Depth of schedule below = 3 ⇒ schedule below is optimal.

Q. Does there always exist a schedule equal to depth of intervals?
Interval Partitioning: Greedy Algorithm

Greedy algorithm. Consider lectures in increasing order of start time: assign lecture to any compatible classroom.

```
Sort intervals by starting time so that \( s_1 \leq s_2 \leq \ldots \leq s_n \).
\[d \leftarrow 0\] \hspace{1cm} \text{number of allocated classrooms}

\text{for } j = 1 \text{ to } n \{ \\
\hspace{1cm} \text{if } (\text{lecture } j \text{ is compatible with some classroom } k) \\
\hspace{2cm} \text{schedule lecture } j \text{ in classroom } k \\
\hspace{1cm} \text{else} \\
\hspace{2cm} \text{allocate a new classroom } d + 1 \\
\hspace{2cm} \text{schedule lecture } j \text{ in classroom } d + 1 \\
\hspace{2cm} d \leftarrow d + 1 \\
\}
```

Implementation. \( O(n \log n) \).
- For each classroom \( k \), maintain the finish time of the last job added.
- Keep the classrooms in a priority queue.
Interval Partitioning: Greedy Analysis

**Observation.** Greedy algorithm never schedules two incompatible lectures in the same classroom.

**Theorem.** Greedy algorithm is optimal.

**Pf.**
- Let $d =$ number of classrooms that the greedy algorithm allocates.
- Classroom $d$ is opened because we needed to schedule a job, say $j$, that is incompatible with all $d-1$ other classrooms.
- These $d$ jobs each end after $s_j$.
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than $s_j$.
- Thus, we have $d$ lectures overlapping at time $s_j + \epsilon$.
- Key observation $\Rightarrow$ all schedules use $\geq d$ classrooms. ** □**
Scheduling to Minimizing Lateness

Minimizing lateness problem.
- Single resource processes one job at a time.
- Job \( j \) requires \( t_j \) units of processing time and is due at time \( d_j \).
- If \( j \) starts at time \( s_j \), it finishes at time \( f_j = s_j + t_j \).
- Lateness: \( \ell_j = \max \{ 0, f_j - d_j \} \).
- Goal: schedule all jobs to minimize maximum lateness \( L = \max \ell_j \).

Ex:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_j )</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>( d_j )</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

Lateness = 2
Lateness = 0
Max lateness = 6

\( d_3 = 9 \) \hspace{1cm} \( d_2 = 8 \) \hspace{1cm} \( d_6 = 15 \) \hspace{1cm} \( d_1 = 6 \) \hspace{1cm} \( d_5 = 14 \) \hspace{1cm} \( d_4 = 9 \)
Minimizing Lateness: Greedy Algorithms

**Greedy template.** Consider jobs in some order.

- **[Shortest processing time first]** Consider jobs in ascending order of processing time $t_j$.

- **[Earliest deadline first]** Consider jobs in ascending order of deadline $d_j$.

- **[Smallest slack]** Consider jobs in ascending order of slack $d_j - t_j$. 
Minimizing Lateness: Greedy Algorithms

Greedy template. Consider jobs in some order.

- [Shortest processing time first] Consider jobs in ascending order of processing time \( t_j \).

  
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
</tr>
</tbody>
</table>

  
  counterexample

- [Smallest slack] Consider jobs in ascending order of slack \( d_j - t_j \).

  
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

  
  counterexample
Minimizing Lateness: Greedy Algorithm

**Greedy algorithm.** Earliest deadline first.

Sort n jobs by deadline so that \(d_1 \leq d_2 \leq \ldots \leq d_n\)

\[
t \leftarrow 0
\]

**for** j = 1 to n

Assign job j to interval \([t, t + t_j]\)

\[
s_j \leftarrow t, f_j \leftarrow t + t_j
\]

\[
t \leftarrow t + t_j
\]

**output** intervals \([s_j, f_j]\)

max lateness = 1

<table>
<thead>
<tr>
<th>(d_1 = 6)</th>
<th>(d_2 = 8)</th>
<th>(d_3 = 9)</th>
<th>(d_4 = 9)</th>
<th>(d_5 = 14)</th>
<th>(d_6 = 15)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Minimizing Lateness: No Idle Time

**Observation.** There exists an optimal schedule with no idle time.

**Observation.** The greedy schedule has no idle time.
Minimizing Lateness: Inversions

**Def.** Given a schedule $S$, an inversion is a pair of jobs $i$ and $j$ such that: $i < j$ but $j$ scheduled before $i$.

**Observation.** Greedy schedule has no inversions.

**Observation.** If a schedule (with no idle time) has an inversion, it has one with a pair of inverted jobs scheduled consecutively.
**Def.** Given a schedule $S$, an inversion is a pair of jobs $i$ and $j$ such that: $i < j$ but $j$ scheduled before $i$.

**Claim.** Swapping two consecutive, inverted jobs reduces the number of inversions by one and does not increase the max lateness.

**Pf.** Let $\ell$ be the lateness before the swap, and let $\ell'$ be it afterwards.

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job $j$ is late:
  
  $$
  \begin{align*}
  \ell'_j &= f'_j - d_j \quad \text{(definition)} \\
  &= f_i - d_j \quad \text{($j$ finishes at time $f_i$)} \\
  &\leq f_i - d_i \quad \text{($i < j$)} \\
  &\leq \ell_i \quad \text{(definition)}
  \end{align*}
  $$
Minimizing Lateness: Analysis of Greedy Algorithm

**Theorem.** Greedy schedule $S$ is optimal.

**Pf.** Define $S^*$ to be an optimal schedule that has the fewest number of inversions, and let's see what happens.

- Can assume $S^*$ has no idle time.
- If $S^*$ has no inversions, then $S = S^*$.
- If $S^*$ has an inversion, let $i-j$ be an adjacent inversion.
  - swapping $i$ and $j$ does not increase the maximum lateness and strictly decreases the number of inversions
  - this contradicts definition of $S^*$
Greedy Analysis Strategies

**Greedy algorithm stays ahead.** Show that after each step of the greedy algorithm, its solution is at least as good as any other algorithm's.

**Structural.** Discover a simple "structural" bound asserting that every possible solution must have a certain value. Then show that your algorithm always achieves this bound.

**Exchange argument.** Gradually transform any solution to the one found by the greedy algorithm without hurting its quality.

**Other greedy algorithms.** Kruskal, Prim, Dijkstra, Huffman, ...
Algorithms — February 16

Today
- Greedy algorithms
- Problems due: from Chapter 3, exercises 5, 6, 10

Monday
- Greedy algorithms

Wednesday
- Greedy algorithms
- Divide-and-conquer algorithms

Next Friday
- Problems due: 3.12, 4.1, 4.3
- Quiz on Ch. 3
  - Open book, open notes, closed internet
  - This one will be individual
- Divide-and-conquer algorithms

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Shortest Path Problem

Shortest path network.
- Directed graph $G = (V, E)$.
- Source $s$, destination $t$.
- Length $l_e = \text{length of edge } e$.

Shortest path problem: find shortest directed path from $s$ to $t$.

Cost of path $s - 2 - 3 - 5 - t$

$= 9 + 23 + 2 + 16$

$= 50$. 
Dijkstra's Algorithm

Dijkstra's algorithm.

- Maintain a set of explored nodes $S$ for which we have determined the shortest path distance $d(u)$ from $s$ to $u$.
- Initialize $S = \{ s \}$, $d(s) = 0$.
- Repeatedly choose unexplored node $v$ which minimizes
  \[
  \pi(v) = \min_{e = (u,v) : u \in S} \{ d(u) + \ell_e \},
  \]
  add $v$ to $S$, and set $d(v) = \pi(v)$.

shortest path to some $u$ in explored part, followed by a single edge $(u,v)$
Dijkstra's algorithm.

- Maintain a set of explored nodes $S$ for which we have determined the shortest path distance $d(u)$ from $s$ to $u$.
- Initialize $S = \{s\}$, $d(s) = 0$.
- Repeatedly choose unexplored node $v$ which minimizes
  \[
  \pi(v) = \min_{e = (u, v) : u \in S} d(u) + \ell_e,
  \]

add $v$ to $S$, and set $d(v) = \pi(v)$. 

![Diagram of Dijkstra's Algorithm](image)
Algorithms — February 19

Today
- Greedy algorithms

Wednesday
- Greedy algorithms

Friday
- Problems due: 3.12, 4.1, 4.3
- Quiz on Ch. 3
  - Open book, open notes, closed internet
  - This one will be individual
- Divide-and-conquer algorithms

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Dijkstra's Algorithm: Proof of Correctness

**Invariant.** For each node $u \in S$, $d(u)$ is the length of the shortest $s$-$u$ path.

**Pf.** (by induction on $|S|$)

**Base case:** $|S| = 1$ is trivial.

**Inductive hypothesis:** Assume true for $|S| = k \geq 1$.

- Let $v$ be next node added to $S$, and let $u$-$v$ be the chosen edge.
- The shortest $s$-$u$ path plus $(u, v)$ is an $s$-$v$ path of length $\pi(v)$.
- Consider any $s$-$v$ path $P$. We'll see that it's no shorter than $\pi(v)$.
- Let $x$-$y$ be the first edge in $P$ that leaves $S$, and let $P'$ be the subpath to $x$.
- $P$ is already too long as soon as it leaves $S$.

$$
\ell(P) \geq \ell(P') + \ell(x, y) \geq d(x) + \ell(x, y) \geq \pi(y) \geq \pi(v)
$$

- nonnegative weights
- inductive hypothesis
- defn of $\pi(y)$
- Dijkstra chose $v$ instead of $y$
Dijkstra's Algorithm: Implementation

For each unexplored node, explicitly maintain \( \pi(v) = \min_{e = (u,v) : u \in S} d(u) + \ell_e \).

- Next node to explore = node with minimum \( \pi(v) \).
- When exploring \( v \), for each incident edge \( e = (v, w) \), update
  \[ \pi(w) = \min \{ \pi(w), \pi(v) + \ell_e \}. \]

**Efficient implementation.** Maintain a priority queue of unexplored nodes, prioritized by \( \pi(v) \).

<table>
<thead>
<tr>
<th>PQ Operation</th>
<th>Dijkstra</th>
<th>Array</th>
<th>Binary heap</th>
<th>d-way Heap</th>
<th>Fib heap $\dagger$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>( n )</td>
<td>( n )</td>
<td>( \log n )</td>
<td>( d \log_d n )</td>
<td>1</td>
</tr>
<tr>
<td>ExtractMin</td>
<td>( n )</td>
<td>( n )</td>
<td>( \log n )</td>
<td>( d \log_d n )</td>
<td>( \log n )</td>
</tr>
<tr>
<td>ChangeKey</td>
<td>( m )</td>
<td>1</td>
<td>( \log n )</td>
<td>( \log_d n )</td>
<td>1</td>
</tr>
<tr>
<td>IsEmpty</td>
<td>( n )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>( n^2 )</td>
<td>( m \log n )</td>
<td>( m \log_{m/n} n )</td>
<td>( m + n \log n )</td>
<td></td>
</tr>
</tbody>
</table>

$\dagger$ Individual ops are amortized bounds
4.5 Minimum Spanning Tree
Minimum spanning tree. Given a connected graph $G = (V, E)$ with real-valued edge weights $c_e$, an MST is a subset of the edges $T \subseteq E$ such that $T$ is a spanning tree whose sum of edge weights is minimized.

**Cayley's Theorem.** There are $n^{n-2}$ spanning trees of $K_n$. Can't solve by brute force.
Applications

MST is fundamental problem with diverse applications.

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road

- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree

- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network

- Cluster analysis.
Greedy Algorithms

Kruskal's algorithm. Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.

Reverse-Delete algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge $e$ from $T$ unless doing so would disconnect $T$.

Prim's algorithm. Start with some root node $s$ and greedily grow a tree $T$ from $s$ outward. At each step, add the cheapest edge $e$ to $T$ that has exactly one endpoint in $T$.

Remark. All three algorithms produce an MST.
Greedy Algorithms

Simplifying assumption. All edge costs $c_e$ are distinct.

Cut property. Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST contains $e$.

Cycle property. Let $C$ be any cycle, and let $f$ be the max cost edge belonging to $C$. Then the MST does not contain $f$. 

$e$ is in the MST

$f$ is not in the MST
Cycles and Cuts

**Cycle.** Set of edges the form a-b, b-c, c-d, ..., y-z, z-a.

![Graph showing a cycle](image)

- Cycle $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$

**Cutset.** A cut is a subset of nodes $S$. The corresponding cutset $D$ is the subset of edges with exactly one endpoint in $S$.

![Graph showing a cutset](image)

- Cut $S = \{4, 5, 8\}$
- Cutset $D = 5-6, 5-7, 3-4, 3-5, 7-8$
Claim. A cycle and a cutset intersect in an even number of edges.

Pf. (by picture)
Greedy Algorithms

Simplifying assumption. All edge costs \( c_e \) are distinct.

Cut property. Let \( S \) be any subset of nodes, and let \( e \) be the min cost edge with exactly one endpoint in \( S \). Then the MST \( T^* \) contains \( e \).

Pf. (exchange argument)
- Suppose \( e \) does not belong to \( T^* \), and let's see what happens.
- Adding \( e \) to \( T^* \) creates a cycle \( C \) in \( T^* \).
- Edge \( e \) is both in the cycle \( C \) and in the cutset \( D \) corresponding to \( S \).
  \( \Rightarrow \) there exists another edge, say \( f \), that is in both \( C \) and \( D \).
- \( T' = T^* \cup \{e\} - \{f\} \) is also a spanning tree.
- Since \( c_e < c_f \), cost(\( T' \)) < cost(\( T^* \)).
- This is a contradiction. □
Greedy Algorithms

Simplifying assumption. All edge costs $c_e$ are distinct.

Cycle property. Let $C$ be any cycle in $G$, and let $f$ be the max cost edge belonging to $C$. Then the MST $T^*$ does not contain $f$.

Pf. (exchange argument)

- Suppose $f$ belongs to $T^*$, and let's see what happens.
- Deleting $f$ from $T^*$ creates a cut $S$ in $T^*$.
- Edge $f$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$\[\Rightarrow\] there exists another edge, say $e$, that is in both $C$ and $D$.
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- This is a contradiction. 

![Diagram](image)
Prim's Algorithm: Proof of Correctness

**Prim's algorithm.** [Jarník 1930, Dijkstra 1957, Prim 1959]

- Initialize $S = \text{any node}$.
- Apply cut property to $S$.
- Add min cost edge in cutset corresponding to $S$ to $T$, and add one new explored node $u$ to $S$. 

![Diagram of Prim's algorithm with set $S$ highlighted and edges added to $T$.]
Implementation: Prim's Algorithm

Implementation. Use a priority queue ala Dijkstra.

- Maintain set of explored nodes $S$.
- For each unexplored node $v$, maintain attachment cost $a[v] = \text{cost of cheapest edge } v \text{ to a node in } S$.
- $O(n^2)$ with an array; $O(m \log n)$ with a binary heap.

```plaintext
Prim(G, c) {
    foreach (v ∈ V) a[v] ← ∞
    Initialize an empty priority queue Q
    foreach (v ∈ V) insert v onto Q
    Initialize set of explored nodes S ← φ

    while (Q is not empty) {
        u ← delete min element from Q
        S ← S U { u }
        foreach (edge e = (u, v) incident to u)
            if ((v ∉ S) and (c_e < a[v]))
                decrease priority a[v] to c_e
    }
```
Kruskal's Algorithm: Proof of Correctness

Kruskal's algorithm. [Kruskal, 1956]

- Consider edges in ascending order of weight.
- Case 1: If adding e to T creates a cycle, discard e according to cycle property.
- Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.

Case 1

Case 2
**Implementation:** Use the union-find data structure.

- Build set \( T \) of edges in the MST.
- Maintain set for each connected component.
- \( O(m \log n) \) for sorting and \( O(m \alpha(m, n)) \) for union-find.

\[
m \leq n^2 \Rightarrow \log m \text{ is } O(\log n)
\]

**Implementation:**

```
Kruskal(G, c) {
    Sort edges weights so that \( c_1 \leq c_2 \leq \ldots \leq c_m \).
    T ← φ

    foreach (u ∈ V) make a set containing singleton u

    for i = 1 to m
        (u,v) = e_i
        if (u and v are in different sets) {
            T ← T ∪ \{e_i\}
            merge the sets containing u and v
        }
    return T
}
```

\[m \leq n^2 \Rightarrow \log m \text{ is } O(\log n)\] essentially a constant
Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

**Impact.** Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

```
e.g., if all edge costs are integers, perturbing cost of edge e_i by i / n^2
```

**Implementation.** Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

```java
boolean less(i, j) {
    if (cost(e_i) < cost(e_j)) return true
    else if (cost(e_i) > cost(e_j)) return false
    else if (i < j) return true
    else return false
}
```
Today, Friday

▶ Greedy algorithms

Monday

▶ Problems due: 3.12, 4.1, 4.3
▶ Divide-and-conquer algorithms

Next Friday

▶ Quiz on Ch. 3
  ▶ Open book, open notes, closed internet
  ▶ This one will be individual

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Greedy Algorithms

Simplifying assumption. All edge costs $c_e$ are distinct.

Cut property. Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST contains $e$.

Cycle property. Let $C$ be any cycle, and let $f$ be the max cost edge belonging to $C$. Then the MST does not contain $f$. 

\[ S \]

\[ e \text{ is in the MST} \]

\[ C \]

\[ f \text{ is not in the MST} \]
Cycles and Cuts

Cycle. Set of edges the form \(a-b, b-c, c-d, \ldots, y-z, z-a\).

Cutset. A cut is a subset of nodes \(S\). The corresponding cutset \(D\) is the subset of edges with exactly one endpoint in \(S\).
**Cycle-Cut Intersection**

**Claim.** A cycle and a cutset intersect in an even number of edges.

**Pf.** (by picture)

Cycle $C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1$
Cutset $D = 3-4, 3-5, 5-6, 5-7, 7-8$
Intersection $= 3-4, 5-6$
Simplifying assumption. All edge costs $c_e$ are distinct.

Cut property. Let $S$ be any subset of nodes, and let $e$ be the min cost edge with exactly one endpoint in $S$. Then the MST $T^*$ contains $e$.

Pf. (exchange argument)
- Suppose $e$ does not belong to $T^*$, and let's see what happens.
- Adding $e$ to $T^*$ creates a cycle $C$ in $T^*$.
- Edge $e$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$.
  $\Rightarrow$ there exists another edge, say $f$, that is in both $C$ and $D$.
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, cost($T'$) < cost($T^*$).
- This is a contradiction.
Greedy Algorithms

Simplifying assumption. All edge costs $c_e$ are distinct.

Cycle property. Let $C$ be any cycle in $G$, and let $f$ be the max cost edge belonging to $C$. Then the MST $T^*$ does not contain $f$.

Pf. (exchange argument)
- Suppose $f$ belongs to $T^*$, and let's see what happens.
- Deleting $f$ from $T^*$ creates a cut $S$ in $T^*$.
- Edge $f$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$.
  $\Rightarrow$ there exists another edge, say $e$, that is in both $C$ and $D$.
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- This is a contradiction. □
Greedy Algorithms

**Kruskal's algorithm.** Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.

**Reverse-Delete algorithm.** Start with $T = E$. Consider edges in descending order of cost. Delete edge $e$ from $T$ unless doing so would disconnect $T$.

**Prim's algorithm.** Start with some root node $s$ and greedily grow a tree $T$ from $s$ outward. At each step, add the cheapest edge $e$ to $T$ that has exactly one endpoint in $T$.

**Remark.** All three algorithms produce an MST.
Prim's Algorithm: Proof of Correctness

**Prim's algorithm.** [Jarník 1930, Dijkstra 1957, Prim 1959]

- Initialize $S =$ any node.
- Apply cut property to $S$.
- Add min cost edge in cutset corresponding to $S$ to $T$, and add one new explored node $u$ to $S$. 

![Diagram showing Prim's algorithm](image-url)
Implementation: Prim’s Algorithm

**Implementation.** Use a priority queue à la Dijkstra

- Maintain set of explored nodes $S$.
- For each unexplored node $v$, maintain attachment cost $a[v] = \text{cost of cheapest edge } v \text{ to a node in } S$.
- $O(n^2)$ with an array; $O(m \log n)$ with a binary heap.

**Algorithm $\text{Prim} (G, c)$**

- $\text{foreach } (v \in V) a[v] \leftarrow \infty$
- Initialize an empty priority queue $Q$
- $\text{foreach } (v \in V) \text{ insert } v \text{ onto } Q$
- Initialize set of explored nodes $S \leftarrow \emptyset$

- Pick starting node $s$
- Add $s$ to $S$
- $\text{foreach } (\text{edge } e = (s, u) \text{ incident to } s)$
  - Add $u$ to $Q$ with priority $c_u$

- while $(Q \text{ is not empty})$
  - $u \leftarrow \text{delete min element from } Q$
  - $S \leftarrow S \cup \{u\}$
  - $\text{foreach } (\text{edge } e = (u, v) \text{ incident to } u)$
    - if $((v \notin S) \text{ and } (c_e < a[v]))$
      - decrease priority $a[v]$ to $c_e$
Algorithms — February 23

Today
▶ Greedy algorithms

Monday
▶ Problems due: 3.12, 4.1, 4.3
▶ Divide-and-conquer algorithms

Next Friday
▶ Quiz on Ch. 3
    ▶ Open book, open notes, closed internet
    ▶ This one will be individual

Contact information
▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
    ▶ Expect replies within a business day (but often same-day)
▶ Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
▶ Or by appointment
    ▶ Email at least a day ahead
    ▶ Send several times when you are available
    ▶ Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Kruskal's algorithm. Start with $T = \emptyset$. Consider edges in ascending order of cost. Insert edge $e$ in $T$ unless doing so would create a cycle.

Reverse-Delete algorithm. Start with $T = E$. Consider edges in descending order of cost. Delete edge $e$ from $T$ unless doing so would disconnect $T$.

Prim's algorithm. Start with some root node $s$ and greedily grow a tree $T$ from $s$ outward. At each step, add the cheapest edge $e$ to $T$ that has exactly one endpoint in $T$.

Remark. All three algorithms produce an MST.
Kruskal's Algorithm: Proof of Correctness

*Kruskal's algorithm.* [Kruskal, 1956]

- Consider edges in ascending order of weight.
- Case 1: If adding $e$ to $T$ creates a cycle, discard $e$ according to cycle property.
- Case 2: Otherwise, insert $e = (u, v)$ into $T$ according to cut property where $S =$ set of nodes in $u$'s connected component.
Union-Find

Need a data structure to manage a graph and its connected components as we add edges.
Union-Find

Three operations

- **MakeUnionFind(S)**
  - The S are the initial nodes
  - Create a new structure with each node disconnected
  - Accept $O(n)$

- **Find(u)**
  - Return the name of the component containing node u
  - Want $O(\log n)$, some implementations give $O(1)$

- **Union(A, B)**
  - Updates the structure so that A and B are now connected
  - Want $O(\log n)$
Union-Find

Three operations

- **MakeUnionFind(S)**
  - The $S$ are the initial nodes
  - Create a new structure with each node disconnected
  - Accept $O(n)$

- **Find(u)**
  - Return the name of the component containing node $u$
  - Want $O(\log n)$, some implementations give $O(1)$

- **Union(A,B)**
  - Updates the structure so that $A$ and $B$ are now connected
  - Want $O(\log n)$

Not for removing edges, and creating new connected components
Union-Find

Three operations

- **MakeUnionFind(S)**
  - The S are the initial nodes
  - Create a new structure with each node disconnected
  - Accept $O(n)$

- **Find(u)**
  - Return the name of the component containing node u
  - Want $O(\log n)$, some implementations give $O(1)$

- **Union(A,B)**
  - Updates the structure so that A and B are now connected
  - Want $O(\log n)$

Simplest implementation is by an array from node index to name

- But $O(n)$ for Union
Union-Find

Three operations

- MakeUnionFind(S)
  - The S are the initial nodes
  - Create a new structure with each node disconnected
  - Accept $O(n)$

- Find(u)
  - Return the name of the component containing node u
  - Want $O(\log n)$, some implementations give $O(1)$

- Union(A,B)
  - Updates the structure so that A and B are now connected
  - Want $O(\log n)$

Simplest implementation is by an array from node index to name

- But $O(n)$ for Union
  - Can be improved by
    - Using the name of the larger set in a union
    - Tracking the size of each connected component
Union-Find

Three operations

- **MakeUnionFind(S)**
  - The $S$ are the initial nodes
  - Create a new structure with each node disconnected
  - Accept $O(n)$

- **Find(u)**
  - Return the name of the component containing node $u$
  - Want $O(\log n)$, some implementations give $O(1)$

- **Union(A,B)**
  - Updates the structure so that $A$ and $B$ are now connected
  - Want $O(\log n)$

Simplest implementation is by an array from node index to name

- **But $O(n)$ for Union**
  - Can be improved by
    - Using the name of the larger set in a union
    - Tracking the size of each connected component
  - But we still need a better structure
The set \{s, u, w\} was merged into \{t, v, z\}.

**Figure 4.12** A Union-Find data structure using pointers. The data structure has only two sets at the moment, named after nodes \(v\) and \(j\). The dashed arrow from \(u\) to \(v\) is the result of the last Union operation. To answer a Find query, we follow the arrows until we get to a node that has no outgoing arrow. For example, answering the query \text{Find}(i)\ would involve following the arrows \(i\) to \(x\), and then \(x\) to \(j\).
Union-Find

Three operations

- MakeUnionFind(S)
  - The S are the initial nodes
  - Create a new structure with each node disconnected
  - Accept $O(n)$
- Find(u)
  - Return the name of the component containing node u
  - Want $O(\log n)$, some implementations give $O(1)$
- Union(A,B)
  - Updates the structure so that A and B are now connected
  - Want $O(\log n)$

The pointer structure gives us $O(1)$ for Union

But now Find could be $O(n)$!

Again, we use the name of the larger set in a union

Gives us $O(\log n)$ again
Union-Find

Three operations
- MakeUnionFind(S)
  - The S are the initial nodes
  - Create a new structure with each node disconnected
  - Accept $O(n)$
- Find(u)
  - Return the name of the component containing node u
  - Want $O(\log n)$, some implementations give $O(1)$
- Union(A, B)
  - Updates the structure so that A and B are now connected
  - Want $O(\log n)$

The pointer structure gives us $O(1)$ for Union
- But now Find could be $O(n)$!
  - We could end up with a single long chain
Union-Find

Three operations

- MakeUnionFind(S)
  - The S are the initial nodes
  - Create a new structure with each node disconnected
  - Accept $O(n)$
- Find(u)
  - Return the name of the component containing node u
  - Want $O(\log n)$, some implementations give $O(1)$
- Union(A, B)
  - Updates the structure so that A and B are now connected
  - Want $O(\log n)$

The pointer structure gives us $O(1)$ for Union

- But now Find could be $O(n)$!
  - We could end up with a single long chain
- Again, we use the name of the larger set in a union
  - Gives us $O(\log n)$ again
Implementation: Kruskal's Algorithm

**Implementation.** Use the union-find data structure.

- Build set $T$ of edges in the MST.
- Maintain set for each connected component.
- $O(m \log n)$ for sorting and $O(m \alpha(m, n))$ for union-find.

$$m \leq n^2 \Rightarrow \log m \text{ is } O(\log n)$$

---

```plaintext
Kruskal(G, c) {
    Sort edges weights so that $c_1 \leq c_2 \leq \ldots \leq c_m$.
    $T \leftarrow \emptyset$

    foreach ($u \in V$) make a set containing singleton $u$

    for $i = 1$ to $m$ are $u$ and $v$ in different connected components?
        $(u,v) = e_i$
        if $(u$ and $v$ are in different sets) {
            $T \leftarrow T \cup \{e_i\}$
            merge the sets containing $u$ and $v$
        }
    return $T$
}
```
Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts to break any ties.

Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

Implementation. Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

```java
boolean less(i, j) {
    if (cost(e_i) < cost(e_j)) return true
    else if (cost(e_i) > cost(e_j)) return false
    else if (i < j) return true
    else return false
}
```
Algorithms — February 26

Today
▶ Problems due: 3.12, 4.1, 4.3
▶ Divide-and-conquer algorithms

Wednesday
▶ Divide-and-conquer algorithms

Friday
▶ Quiz on Ch. 3
  ▶ Open book, open notes, closed internet
  ▶ This one will be individual
▶ Divide-and-conquer algorithms

Contact information
▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Divide-and-Conquer

Divide-and-conquer.

- Break up problem into several parts.
- Solve each part recursively.
- Combine solutions to sub-problems into overall solution.

Most common usage.

- Break up problem of size $n$ into two equal parts of size $\frac{1}{2}n$.
- Solve two parts recursively.
- Combine two solutions into overall solution in linear time.

Consequence.

- Brute force: $n^2$.
- Divide-and-conquer: $n \log n$.

Divide et impera.
Veni, vidi, vici.

- Julius Caesar
5.1 Mergesort
Mergesort

Mergesort.
- Divide array into two halves.
- Recursively sort each half.
- Merge two halves to make sorted whole.

Jon von Neumann (1945)

\[
\begin{align*}
\text{divide} & \quad O(1) \\
\text{sort} & \quad 2T(n/2) \\
\text{merge} & \quad O(n)
\end{align*}
\]
**Merging**

**Merging.** Combine two pre-sorted lists into a sorted whole.

**How to merge efficiently?**

- Linear number of comparisons.
- Use temporary array.

---

**Challenge for the bored.** In-place merge. [Kronrud, 1969]

using only a constant amount of extra storage
A Useful Recurrence Relation

**Def.** $T(n) = \text{number of comparisons to mergesort an input of size } n.$

**Mergesort recurrence.**

\[
T(n) \leq \begin{cases} 
0 & \text{if } n = 1 \\
T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & \text{otherwise}
\end{cases}
\]

**Solution.** $T(n) = O(n \log_2 n).$

**Assorted proofs.** We describe several ways to prove this recurrence. Initially we assume $n$ is a power of 2 and replace $\leq$ with $=.$
**Proof by Telescoping**

**Claim.** If \( T(n) \) satisfies this recurrence, then \( T(n) = n \log_2 n \).

\[
\begin{align*}
T(n) &= \begin{cases} 
0 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases} \\
\text{sorting both halves} & \text{merging}
\end{align*}
\]

\[\uparrow\] assumes \( n \) is a power of 2

**Pf.** For \( n > 1 \):

\[
\begin{align*}
\frac{T(n)}{n} &= \frac{2T(n/2)}{n} + 1 \\
&= \frac{T(n/2)}{n/2} + 1 \\
&= \frac{T(n/4)}{n/4} + 1 + 1 \\
&\vdots \\
&= \frac{T(n/n)}{n/n} + 1 + \cdots + 1 \\
&= \log_2 n
\end{align*}
\]
Proof by Induction

Claim. If $T(n)$ satisfies this recurrence, then $T(n) = n \log_2 n$.

\[
T(n) = \begin{cases} 
0 & \text{if } n = 1 \\
2T(n/2) + n & \text{otherwise}
\end{cases}
\]

Pf. (by induction on $n$)

- Base case: $n = 1$.
- Inductive hypothesis: $T(n) = n \log_2 n$.
- Goal: show that $T(2n) = 2n \log_2 (2n)$.

\[
T(2n) = 2T(n) + 2n = 2n \log_2 n + 2n = 2n(\log_2(2n) - 1) + 2n = 2n \log_2(2n)
\]
Claim. If \( T(n) \) satisfies the following recurrence, then \( T(n) \leq n \lfloor \log n \rfloor \).

\[
T(n) \leq \begin{cases} 
0 & \text{if } n = 1 \\
\left\lfloor \frac{n}{2} \right\rfloor \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + \left\lfloor \frac{n}{2} \right\rfloor \cdot T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + n & \text{otherwise}
\end{cases}
\]

Pf. (by induction on \( n \))

- Base case: \( n = 1 \).
- Define \( n_1 = \lfloor n / 2 \rfloor \), \( n_2 = \lceil n / 2 \rceil \).
- Induction step: assume true for \( 1, 2, \ldots, n-1 \).

\[
T(n) \leq T(n_1) + T(n_2) + n
\]

\[
\leq n_1 \lfloor \log n_1 \rfloor + n_2 \lfloor \log n_2 \rfloor + n
\]

\[
\leq n_1 \lfloor \log n_2 \rfloor + n_2 \lfloor \log n_2 \rfloor + n
\]

\[
= n \lfloor \log n_2 \rfloor + n
\]

\[
\leq n(\lfloor \log n \rfloor - 1) + n
\]

\[
= n \lfloor \log n \rfloor
\]
5.3 Counting Inversions
Counting Inversions

Music site tries to match your song preferences with others.
- You rank \( n \) songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.
- My rank: \( 1, 2, \ldots, n \).
- Your rank: \( a_1, a_2, \ldots, a_n \).
- Songs \( i \) and \( j \) inverted if \( i < j \), but \( a_i > a_j \).

<table>
<thead>
<tr>
<th>Songs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td><strong>Me</strong></td>
</tr>
<tr>
<td><strong>You</strong></td>
</tr>
</tbody>
</table>

**Inversions**
3-2, 4-2

Brute force: check all \( \Theta(n^2) \) pairs \( i \) and \( j \).
Applications

- Voting theory.
- Collaborative filtering.
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.
- Rank aggregation for meta-searching on the Web.
- Nonparametric statistics (e.g., Kendall's Tau distance).
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- **Divide**: separate list into two pieces.

\[
\begin{array}{cccccccccc}
1 & 5 & 4 & 8 & 10 & 2 & 6 & 9 & 12 & 11 & 3 & 7 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
1 & 5 & 4 & 8 & 10 & 2 & 6 & 9 & 12 & 11 & 3 & 7 \\
\end{array}
\]

Divide: $O(1)$. 
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.
- **Divide**: separate list into two pieces.
- **Conquer**: recursively count inversions in each half.

```
1 5 4 8 10 2 6 9 12 11 3 7
```

**Divide**: $O(1)$.

```
1 5 4 8 10 2 6 9 12 11 3 7
```

5 blue-blue inversions
5-4, 5-2, 4-2, 8-2, 10-2

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

8 green-green inversions

**Conquer**: $2T(n/2)$
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- **Divide**: separate list into two pieces.
- **Conquer**: recursively count inversions in each half.
- **Combine**: count inversions where \( a_i \) and \( a_j \) are in different halves, and return sum of three quantities.

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>2</th>
<th>6</th>
<th>9</th>
<th>12</th>
<th>11</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
</table>

- Divide: \( O(1) \).
- Conquer: \( 2T(n / 2) \)
- Combine: ???

5 blue-blue inversions

8 green-green inversions

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

**Total** = 5 + 8 + 9 = 22.
Counting Inversions: Combine

**Combine:** count blue-green inversions
- Assume each half is sorted.
- Count inversions where \( a_i \) and \( a_j \) are in different halves.
- **Merge** two sorted halves into sorted whole.

To maintain sorted invariant

13 blue-green inversions: 6 + 3 + 2 + 2 + 0 + 0

Count: \( O(n) \)

Merge: \( O(n) \)

\[
T(n) \leq T\left(\left\lfloor n/2 \right\rfloor \right) + T\left(\left\lfloor n/2 \right\rfloor \right) + O(n) \quad \Rightarrow \quad T(n) = O(n \log n)
\]
Counting Inversions: Implementation

Pre-condition. [Merge-and-Count] A and B are sorted.
Post-condition. [Sort-and-Count] L is sorted.

Sort-and-Count(L) {
    if list L has one element
        return 0 and the list L

    Divide the list into two halves A and B
    (r_A, A) ← Sort-and-Count(A)
    (r_B, B) ← Sort-and-Count(B)
    (r, L) ← Merge-and-Count(A, B)

    return r = r_A + r_B + r and the sorted list L
}
5.4 Closest Pair of Points
Closest Pair of Points

**Closest pair.** Given \( n \) points in the plane, find a pair with smallest Euclidean distance between them.

**Fundamental geometric primitive.**
- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of nearest neighbor, Euclidean MST, Voronoi.

**Brute force.** Check all pairs of points \( p \) and \( q \) with \( \Theta(n^2) \) comparisons.

**1-D version.** \( O(n \log n) \) easy if points are on a line.

**Assumption.** No two points have same \( x \) coordinate.

\( \text{fast closest pair inspired fast algorithms for these problems} \)

to make presentation cleaner
Closest Pair of Points: First Attempt

**Divide.** Sub-divide region into 4 quadrants.
Closest Pair of Points: First Attempt

**Divide.** Sub-divide region into 4 quadrants.

**Obstacle.** Impossible to ensure $n/4$ points in each piece.
Closest Pair of Points

Algorithm.
- **Divide:** draw vertical line $L$ so that roughly $\frac{1}{2}n$ points on each side.
Closest Pair of Points

**Algorithm.**

- **Divide:** draw vertical line $L$ so that roughly $\frac{1}{2}n$ points on each side.
- **Conquer:** find closest pair in each side recursively.
Closest Pair of Points

**Algorithm.**
- **Divide:** draw vertical line $L$ so that roughly $\frac{1}{2}n$ points on each side.
- **Conquer:** find closest pair in each side recursively.
- **Combine:** find closest pair with one point in each side. ← seems like $\Theta(n^2)$
- **Return** best of 3 solutions.
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < $\delta$. 

$\delta = \min(12, 21)$
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance $< \delta$.
- Observation: only need to consider points within $\delta$ of line $L$. 

\[ \delta = \min(12, 21) \]
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance $< \delta$.

- **Observation:** only need to consider points within $\delta$ of line $L$.
- Sort points in $2\delta$-strip by their $y$ coordinate.

$\delta = \min(12, 21)$
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance $< \delta$.

- Observation: only need to consider points within $\delta$ of line $L$.
- Sort points in $2\delta$-strip by their $y$ coordinate.
- Only check distances of those within 11 positions in sorted list!

$$\delta = \min(12, 21)$$
**Def.** Let \( s_i \) be the point in the \( 2\delta \)-strip, with the \( i^{th} \) smallest \( y \)-coordinate.

**Claim.** If \( |i-j| \geq 12 \), then the distance between \( s_i \) and \( s_j \) is at least \( \delta \).

**Pf.**
- No two points lie in same \( \frac{1}{2}\delta \)-by-\( \frac{1}{2}\delta \) box.
- Two points at least 2 rows apart have distance \( \geq 2(\frac{1}{2}\delta) \).

**Fact.** Still true if we replace 12 with 7.
**Closest Pair Algorithm**

Closest-Pair(p₁, ..., pₙ) {
    Compute separation line L such that half the points are on one side and half on the other side.

    δ₁ = Closest-Pair(left half)
    δ₂ = Closest-Pair(right half)
    δ = min(δ₁, δ₂)

    **Delete** all points further than δ from separation line L

    **Sort** remaining points by y-coordinate.

    **Scan** points in y-order and compare distance between each point and next 11 neighbors. If any of these distances is less than δ, update δ.

    **return** δ.
}
Closest Pair of Points: Analysis

Running time.

T(n) ≤ 2T(n/2) + O(n log n)  ⇒  T(n) = O(n log^2 n)

Q. Can we achieve O(n log n)?

A. Yes. Don’t sort points in strip from scratch each time.
   - Each recursive returns two lists: all points sorted by y coordinate, and all points sorted by x coordinate.
   - Sort by merging two pre-sorted lists.

T(n) ≤ 2T(n/2) + O(n)  ⇒  T(n) = O(n log n)
Algorithms — February 26

Today

► Divide-and-conquer algorithms

Friday

► Quiz on Ch. 3
  ► Open book, open notes, closed internet
  ► This one will be individual
► Divide-and-conquer algorithms

Monday

► Problems due: 4.2, 4.8, 4.9, 5.1
► Divide-and-conquer algorithms

Contact information

► Web: cs.uwlax.edu/~jmaraist
► Email: jmaraist@uwlax.edu
  ► Expect replies within a business day (but often same-day)
► Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
► Or by appointment
  ► Email at least a day ahead
  ► Send several times when you are available
  ► Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Recurrence relations roundup

Last time we saw:

- If \( T(n) \leq 2T(n/2) + cn \) for \( n > 2 \) and \( T(2) \leq c \)
  then \( T = O(n \log n) \)
Recurrence relations roundup

Last time we saw:

- If \( T(n) \leq 2T(n/2) + cn \) for \( n > 2 \) and \( T(2) \leq c \)
  then \( T = O(n \log n) \)

Some other useful recurrence relations:

- If \( T(n) \leq qT(n/2) + cn \) for \( q > 2, n > 2 \) and \( T(2) \leq c \)
  then \( T = O(n^{\log_2 q}) \)

- If \( T(n) \leq T(n/2) + cn \) for \( n > 2 \) and \( T(2) \leq c \)
  then \( T = O(n) \)

- If \( T(n) \leq 2T(n/2) + cn^2 \) for \( n > 2 \) and \( T(2) \leq c \)
  then \( T = O(n^2) \)

With reference translations from recurrence relations to asymptotic bounds, we can quickly analyze new applications for divide-and-conquer
Music site tries to match your song preferences with others.
- You rank n songs.
- Music site consults database to find people with similar tastes.

Similarity metric: number of inversions between two rankings.
- My rank: 1, 2, ..., n.
- Your rank: \( a_1, a_2, ..., a_n \).
- Songs \( i \) and \( j \) inverted if \( i < j \), but \( a_i > a_j \).

Brute force: check all \( \Theta(n^2) \) pairs \( i \) and \( j \).
Applications

- Voting theory.
- Collaborative filtering.
- Measuring the "sortedness" of an array.
- Sensitivity analysis of Google's ranking function.
- Rank aggregation for meta-searching on the Web.
- Nonparametric statistics (e.g., Kendall's Tau distance).
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

1 5 4 8 10 2 6 9 12 11 3 7
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- **Divide**: separate list into two pieces.

```
1 5 4 8 10 2 6 9 12 11 3 7
```

Divide: $O(1)$.
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- **Divide**: separate list into two pieces.
- **Conquer**: recursively count inversions in each half.

4 8 10 2 1 5 12 11 3 7

5 blue-blue inversions
5-4, 5-2, 4-2, 8-2, 10-2

6-9, 9-12, 11-3, 7-3

6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7

Divide: $O(1)$.

Conquer: $2T(n/2)$
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

- **Divide:** separate list into two pieces.
- **Conquer:** recursively count inversions in each half.
- **Combine:** count inversions where $a_i$ and $a_j$ are in different halves, and return sum of three quantities.

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>2</th>
<th>6</th>
<th>9</th>
<th>12</th>
<th>11</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
</table>

5 blue-blue inversions
8 green-green inversions

9 blue-green inversions
5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Total $= 5 + 8 + 9 = 22$.  

Divide: $O(1)$.  
Conquer: $2T(n / 2)$  
Combine: ???
Counting Inversions: Combine

**Combine:** count blue-green inversions
- Assume each half is sorted.
- Count inversions where \( a_i \) and \( a_j \) are in different halves.
- **Merge** two sorted halves into sorted whole.

![Image showing the process of counting and merging inversions]

13 blue-green inversions: \( 6 + 3 + 2 + 2 + 0 + 0 \)

**Count:** \( O(n) \)

![Image showing the sorted halves]

**Merge:** \( O(n) \)

\[
T(n) \leq T\left(\left\lfloor n/2 \right\rfloor \right) + T\left(\left\lfloor n/2 \right\rfloor \right) + O(n) \quad \Rightarrow \quad T(n) = O(n \log n)
\]
**Closest Pair of Points**

**Closest pair.** Given n points in the plane, find a pair with smallest Euclidean distance between them.

**Fundamental geometric primitive.**
- Graphics, computer vision, geographic information systems, molecular modeling, air traffic control.
- Special case of nearest neighbor, Euclidean MST, Voronoi.

**Brute force.** Check all pairs of points p and q with $\Theta(n^2)$ comparisons.

**1-D version.** $O(n \log n)$ easy if points are on a line.

**Assumption.** No two points have same x coordinate.

\[\text{to make presentation cleaner}\]
Closest Pair of Points: First Attempt

**Divide.** Sub-divide region into 4 quadrants.
Closest Pair of Points: First Attempt

**Divide.** Sub-divide region into 4 quadrants.

**Obstacle.** Impossible to ensure $n/4$ points in each piece.

![Diagram showing points divided into 4 quadrants](image)
Closest Pair of Points

Algorithm.
- **Divide**: draw vertical line $L$ so that roughly $\frac{1}{2}n$ points on each side.
Closest Pair of Points

Algorithm.
- **Divide:** draw vertical line \( L \) so that roughly \( \frac{1}{2}n \) points on each side.
- **Conquer:** find closest pair in each side recursively.
**Closest Pair of Points**

**Algorithm.**
- **Divide:** draw vertical line $L$ so that roughly $\frac{1}{2}n$ points on each side.
- **Conquer:** find closest pair in each side recursively.
- **Combine:** find closest pair with one point in each side. $\leftarrow$ seems like $\Theta(n^2)$
- Return best of 3 solutions.
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < \( \delta \).

\[ \delta = \min(12, 21) \]
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < $\delta$.

- Observation: only need to consider points within $\delta$ of line $L$. 

$$\delta = \min(12, 21)$$
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < \( \delta \).
- Observation: only need to consider points within \( \delta \) of line \( L \).
- Sort points in \( 2\delta \)-strip by their \( y \) coordinate.

\[ \delta = \min(12, 21) \]
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < \( \delta \).
- Observation: only need to consider points within \( \delta \) of line \( L \).
- Sort points in \( 2\delta \)-strip by their \( y \) coordinate.
- Only check distances of those within 11 positions in sorted list!

\[ \delta = \min(12, 21) \]
Closest Pair of Points

**Def.** Let \( s_i \) be the point in the \( 2\delta \)-strip, with the \( i^{th} \) smallest \( y \)-coordinate.

**Claim.** If \( |i - j| \geq 12 \), then the distance between \( s_i \) and \( s_j \) is at least \( \delta \).

**Pf.**
- No two points lie in same \( \frac{1}{2}\delta \)-by-\( \frac{1}{2}\delta \) box.
- Two points at least 2 rows apart have distance \( \geq 2(\frac{1}{2}\delta) \).

**Fact.** Still true if we replace 12 with 7.
Closest Pair Algorithm

Closest-Pair(p₁, ..., pₙ) {

    Compute separation line L such that half the points are on one side and half on the other side.

    δ₁ = Closest-Pair(left half)
    δ₂ = Closest-Pair(right half)
    δ = min(δ₁, δ₂)

    Delete all points further than δ from separation line L

    Sort remaining points by y-coordinate.

    Scan points in y-order and compare distance between each point and next 11 neighbors. If any of these distances is less than δ, update δ.

    return δ.

}
Closest Pair of Points: Analysis

Running time.

\[ T(n) \leq 2T(n/2) + O(n \log n) \Rightarrow T(n) = O(n \log^2 n) \]

Q. Can we achieve \(O(n \log n)\)?

A. Yes. Don’t sort points in strip from scratch each time.
   - Each recursive returns two lists: all points sorted by \(y\) coordinate, and all points sorted by \(x\) coordinate.
   - Sort by merging two pre-sorted lists.

\[ T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n) \]
Algorithms — March 2

Today
▶ Quiz on Ch. 3
  ▶ Open book, open notes, closed internet
  ▶ This one will be individual
▶ Divide-and-conquer algorithms

Monday
▶ Problems due: 4.2, 4.8, 4.9, 5.1
▶ Divide-and-conquer algorithms

Wednesday

Contact information
▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
How to Multiply
integers, matrices, and polynomials
Complex Multiplication

**Complex multiplication.** \((a + bi) (c + di) = x + yi.\)

**Grade-school.** \(x = ac - bd, \ y = bc + ad.\)

\[\text{4 multiplications, 2 additions}\]

**Q.** Is it possible to do with fewer multiplications?
Complex Multiplication

Complex multiplication. \((a + bi) \, (c + di) = x + yi.\)

Grade-school. \(x = ac - bd, \ y = bc + ad.\)

Q. Is it possible to do with fewer multiplications?
A. Yes. [Gauss] \(x = ac - bd, \ y = (a + b) \, (c + d) - ac - bd.\)

Remark. Improvement if no hardware multiply.
5.5 Integer Multiplication
**Integer Addition**

**Addition.** Given two $n$-bit integers $a$ and $b$, compute $a + b$.

**Grade-school.** $\Theta(n)$ bit operations.

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
\hline
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
+ & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\
\end{array}
\]

**Remark.** Grade-school addition algorithm is optimal.
Integer Multiplication

Multiplication. Given two $n$-bit integers $a$ and $b$, compute $a \times b$.

Grade-school. $\Theta(n^2)$ bit operations.

Q. Is grade-school multiplication algorithm optimal?
Divide-and-Conquer Multiplication: Warmup

To multiply two $n$-bit integers $a$ and $b$:
- Multiply four $\frac{1}{2}n$-bit integers, recursively.
- Add and shift to obtain result.

\[
\begin{align*}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= (2^{n/2} \cdot a_1 + a_0) (2^{n/2} \cdot b_1 + b_0) = 2^n \cdot a_1b_1 + 2^{n/2} \cdot (a_1b_0 + a_0b_1) + a_0b_0
\end{align*}
\]

Ex. \[a = \overbrace{10001101}^{a_1 \ a_0} \quad b = \overbrace{11100001}^{b_1 \ b_0}\]

\[T(n) = 4T(n/2) + \Theta(n) \implies T(n) = \Theta(n^2)\]
Karatsuba Multiplication

To multiply two $n$-bit integers $a$ and $b$:

- Add two $\frac{1}{2}n$ bit integers.
- Multiply three $\frac{1}{2}n$-bit integers, recursively.
- Add, subtract, and shift to obtain result.

\[
\begin{align*}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= 2^n \cdot a_1 b_1 + 2^{n/2} \cdot (a_1 b_0 + a_0 b_1) + a_0 b_0 \\
    &= 2^n \cdot a_1 b_1 + 2^{n/2} \cdot ((a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0) + a_0 b_0
\end{align*}
\]
Karatsuba Multiplication

To multiply two \( n \)-bit integers \( a \) and \( b \):

- Add two \( \frac{1}{2}n \) bit integers.
- Multiply three \( \frac{1}{2}n \)-bit integers, recursively.
- Add, subtract, and shift to obtain result.

\[
\begin{align*}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= 2^n \cdot a_1 b_1 + 2^{n/2} \cdot (a_1 b_0 + a_0 b_1) + a_0 b_0 \\
&= 2^n \cdot a_1 b_1 + 2^{n/2} \cdot ((a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0) + a_0 b_0
\end{align*}
\]

1 2 1 3 3

Theorem. [Karatsuba-Ofman 1962] Can multiply two \( n \)-bit integers in \( O(n^{1.585}) \) bit operations.

\[
T(n) \leq T\left(\left\lceil \frac{n}{2} \right\rceil \right) + T\left(\left\lceil \frac{n}{2} \right\rceil \right) + T\left(1 + \left\lceil \frac{n}{2} \right\rceil \right) + \Theta(n) \\
\implies T(n) = O(n^{\lg 3}) = O(n^{1.585})
\]
Matrix Multiplication
Dot Product

**Dot product.** Given two length $n$ vectors $a$ and $b$, compute $c = a \cdot b$.

**Grade-school.** $\Theta(n)$ arithmetic operations.

\[
a \cdot b = \sum_{i=1}^{n} a_i b_i
\]

<table>
<thead>
<tr>
<th>$a$</th>
<th>.70</th>
<th>.20</th>
<th>.10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b$</td>
<td>.30</td>
<td>.40</td>
<td>.30</td>
</tr>
</tbody>
</table>

$a \cdot b = (.70 \times .30) + (.20 \times .40) + (.10 \times .30) = .32$

**Remark.** Grade-school dot product algorithm is optimal.
Matrix Multiplication

Matrix multiplication. Given two \(n\)-by-\(n\) matrices \(A\) and \(B\), compute \(C = AB\).

Grade-school. \(\Theta(n^3)\) arithmetic operations.

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}
\]

\[
\begin{bmatrix}
c_{11} & c_{12} & \cdots & c_{1n} \\
c_{21} & c_{22} & \cdots & c_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
c_{n1} & c_{n2} & \cdots & c_{nn}
\end{bmatrix} = \begin{bmatrix}
 a_{11} & a_{12} & \cdots & a_{1n} \\
 a_{21} & a_{22} & \cdots & a_{2n} \\
 \vdots & \vdots & \ddots & \vdots \\
 a_{n1} & a_{n2} & \cdots & a_{nn}
\end{bmatrix} \times \begin{bmatrix}
 b_{11} & b_{12} & \cdots & b_{1n} \\
 b_{21} & b_{22} & \cdots & b_{2n} \\
 \vdots & \vdots & \ddots & \vdots \\
 b_{n1} & b_{n2} & \cdots & b_{nn}
\end{bmatrix}
\]

\[
\begin{bmatrix}
.59 & .32 & .41 \\
.31 & .36 & .25 \\
.45 & .31 & .42
\end{bmatrix} = \begin{bmatrix}
.70 & .20 & .10 \\
.30 & .60 & .10 \\
.50 & .10 & .40
\end{bmatrix} \times \begin{bmatrix}
.80 & .30 & .50 \\
.10 & .40 & .10 \\
.10 & .30 & .40
\end{bmatrix}
\]

Q. Is grade-school matrix multiplication algorithm optimal?
Block Matrix Multiplication

\[
C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 16 & 17 \\ 20 & 21 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 24 & 25 \\ 28 & 29 \end{bmatrix} = \begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}
\]
Matrix Multiplication: Warmup

To multiply two $n$-by-$n$ matrices $A$ and $B$:

- **Divide:** partition $A$ and $B$ into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
- **Conquer:** multiply 8 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
- **Combine:** add appropriate products using 4 matrix additions.

\[
\begin{bmatrix}
  C_{11} & C_{12} \\
  C_{21} & C_{22}
\end{bmatrix} = \begin{bmatrix}
  A_{11} & A_{12} \\
  A_{21} & A_{22}
\end{bmatrix} \times \begin{bmatrix}
  B_{11} & B_{12} \\
  B_{21} & B_{22}
\end{bmatrix}
\]

\[
\begin{align*}
  C_{11} &= (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\
  C_{12} &= (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\
  C_{21} &= (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\
  C_{22} &= (A_{21} \times B_{12}) + (A_{22} \times B_{22})
\end{align*}
\]

\[
T(n) = 8T(\frac{n}{2}) + \Theta(n^2) \implies T(n) = \Theta(n^3)
\]
Fast Matrix Multiplication

**Key idea.** multiply 2-by-2 blocks with only 7 multiplications.

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix} = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix} \times \begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

**Key steps:**

- \( C_{11} = P_5 + P_4 - P_2 + P_6 \)
- \( C_{12} = P_1 + P_2 \)
- \( C_{21} = P_3 + P_4 \)
- \( C_{22} = P_5 + P_1 - P_3 - P_7 \)

- 7 multiplications.
- 18 = 8 + 10 additions and subtractions.

- \( P_1 = A_{11} \times (B_{12} - B_{22}) \)
- \( P_2 = (A_{11} + A_{12}) \times B_{22} \)
- \( P_3 = (A_{21} + A_{22}) \times B_{11} \)
- \( P_4 = A_{22} \times (B_{21} - B_{11}) \)
- \( P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) \)
- \( P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \)
- \( P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12}) \)
Fast Matrix Multiplication

To multiply two $n$-by-$n$ matrices $A$ and $B$: [Strassen 1969]

- **Divide:** partition $A$ and $B$ into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
- **Compute:** $14 \frac{1}{2}n$-by-$\frac{1}{2}n$ matrices via $10$ matrix additions.
- **Conquer:** multiply $7$ pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
- **Combine:** $7$ products into $4$ terms using $8$ matrix additions.

**Analysis.**

- Assume $n$ is a power of $2$.
- $T(n) = \#$ arithmetic operations.

\[
T(n) = 7T(n/2) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})
\]
Fast Matrix Multiplication: Practice

Implementation issues.
- Sparsity.
- Caching effects.
- Numerical stability.
- Odd matrix dimensions.
- Crossover to classical algorithm around $n = 128$.

Common misperception. "Strassen is only a theoretical curiosity."
- Apple reports 8x speedup on G4 Velocity Engine when $n \approx 2,500$.
- Range of instances where it's useful is a subject of controversy.

Remark. Can "Strassenize" $Ax = b$, determinant, eigenvalues, SVD, ....
Fast Matrix Multiplication: Theory

Q. Multiply two 2-by-2 matrices with 7 scalar multiplications?
A. Yes! [Strassen 1969] \[ \Theta(n^{\log_2 7}) = O(n^{2.807}) \]

Q. Multiply two 2-by-2 matrices with 6 scalar multiplications?
A. Impossible. [Hopcroft and Kerr 1971] \[ \Theta(n^{\log_2 6}) = O(n^{2.59}) \]

Q. Two 3-by-3 matrices with 21 scalar multiplications?
A. Also impossible. \[ \Theta(n^{\log_3 21}) = O(n^{2.77}) \]

Begun, the decimal wars have. [Pan, Bini et al, Schönhage, …]
- Two 20-by-20 matrices with 4,460 scalar multiplications. \[ O(n^{2.805}) \]
- Two 48-by-48 matrices with 47,217 scalar multiplications. \[ O(n^{2.7801}) \]
- A year later. \[ O(n^{2.7799}) \]
- December, 1979. \[ O(n^{2.521813}) \]
- January, 1980. \[ O(n^{2.521801}) \]
Fast Matrix Multiplication: Theory

Best known. $O(n^{2.376})$ [Coppersmith-Winograd, 1987]

Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.

Fig. 1. $\omega(t)$ is the best exponent announced by time $t$. 

---

22
Fast Fourier transforms

- Another divide-and-conquer application
- More mathematical than we’ll get into
- Applies to multiplication of polynomials, and then back to integers and matrices
Integer Multiplication, Redux

**Integer multiplication.** Given two $n$ bit integers $a = a_{n-1} \ldots a_1a_0$ and $b = b_{n-1} \ldots b_1b_0$, compute their product $a \cdot b$.

"the fastest bignum library on the planet"

**Practice.** [GNU Multiple Precision Arithmetic Library] It uses brute force, Karatsuba, and FFT, depending on the size of $n$. 
Integer Arithmetic

Fundamental open question. What is complexity of arithmetic?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Upper Bound</th>
<th>Lower Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>$O(n)$</td>
<td>$\Omega(n)$</td>
</tr>
<tr>
<td>multiplication</td>
<td>$O(n \log n \cdot 2^{\omega(n)})$</td>
<td>$\Omega(n)$</td>
</tr>
<tr>
<td>division</td>
<td>$O(n \log n \cdot 2^{\omega(n)})$</td>
<td>$\Omega(n)$</td>
</tr>
</tbody>
</table>
Factoring

**Factoring.** Given an $n$-bit integer, find its prime factorization.

\[ 2773 = 47 \times 59 \]

\[ 2^{67}-1 = 147573952589676412927 = 193707721 \times 761838257287 \]

A disproof of Mersenne’s conjecture that $2^{67}-1$ is prime

\[
\begin{align*}
740375634795617128280467960974295731425931888892312890849 \\
362326389727650340282662768919964196251178439958943305021 \\
275853701189680982867331732731089309005525051168770632990 \\
72396380786710086096962537934650563796359
\end{align*}
\]

**RSA-704**

($30,000$ prize if you can factor)
Factoring and RSA

**Primality.** Given an \( n \)-bit integer, is it prime?

**Factoring.** Given an \( n \)-bit integer, find its prime factorization.

**Significance.** Efficient primality testing \( \Rightarrow \) can implement RSA.

**Significance.** Efficient factoring \( \Rightarrow \) can break RSA.

**Theorem.** [AKS 2002] Poly-time algorithm for primality testing.
Shor's Algorithm

**Shor's algorithm.** Can factor an \( n \)-bit integer in \( O(n^3) \) time on a quantum computer.

**Algorithm uses quantum QFT!**

**Ramification.** At least one of the following is wrong:
- RSA is secure.
- Textbook quantum mechanics.
- Extending Church-Turing thesis.
Algorithms — March 5

Today
▶ Problems due: 4.2, 4.8, 4.9, 5.1

This week
▶ Dynamic programming
▶ No quiz

Next week
▶ Spring break

Monday, March 19
▶ We’re back
▶ Problems due: 5.5, 5.6

Contact information
▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Chapter 6
Dynamic Programming
Algorithmic Paradigms

**Greedy.** Build up a solution incrementally, myopically optimizing some local criterion.

**Divide-and-conquer.** Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

**Dynamic programming.** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
Dynamic Programming History

Bellman. [1950s] Pioneered the systematic study of dynamic programming.

Etymology.
- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

"it's impossible to use dynamic in a pejorative sense"
"something not even a Congressman could object to"

Dynamic Programming Applications

Areas.
- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ....

Some famous dynamic programming algorithms.
- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.
Illustrative overkill — Fibonacci

When we learn recursion, Fibonacci is usually one of the examples of how not to recur
When we learn recursion, Fibonacci is usually one of the examples of how \textit{not} to recur

\begin{itemize}
  \item Doing it wrong:
  \end{itemize}

\begin{verbatim}
public static long fib(int n) {
    switch (n) {
        case 0: return 0;
        case 1: return 1;
        default: return fib(n-1) + fib(n-2);
    }
}
\end{verbatim}

requires $O(2^n)$ recursive calls
Illustrative overkill — Fibonacci

When we learn recursion, Fibonacci is usually one of the examples of how not to recur

➤ Doing it wrong:

```java
public static long fib(int n) {
    switch (n) {
        case 0: return 0;
        case 1: return 1;
        default: return fib(n-1) + fib(n-2);
    }
}
```

requires $O(2^n)$ recursive calls

➤ This approach fails because the subproblems overlap
  ➤ fib(n-1) is going to need fib(n-2)
  ➤ We fail to take advantage of this structure
A dynamic solution to Fibonacci

Cache the results of the subproblems
- Then we can use each more than once

```java
public static long fib(int n) {
    final long[] fibs = new long[1+n];
    fibs[0] = 0;
    fibs[1] = 1;
    for(int i=2; i<=n; i++) {
        fibs[n] = fibs[n-1] + fibs[n-2];
    }
    return fibs[n];
}
```
A dynamic solution to Fibonacci

Cache the results of the subproblems

- Then we can use each more than once

```java
public static long fib(int n) {
    final long[] fibs = new long[1+n];
    fibs[0] = 0;
    fibs[1] = 1;
    for(int i=2; i<=n; i++) {
        fibs[n] = fibs[n-1] + fibs[n-2];
    }
    return fibs[n];
}
```

- The smarter solution which we usually learn takes further advantage of the fact that the overlap is limited
  - We do not need to keep more than two members of the sequence at a time to make progress
  - So instead of an array, just track "this" element and "prev" (or "next") element
Weighted interval scheduling problem.

- Job $j$ starts at $s_j$, finishes at $f_j$, and has weight or value $v_j$.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.
Recall. Greedy algorithm works if all weights are 1.
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.
Weighted Interval Scheduling

**Notation.** Label jobs by finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_n \).

**Def.** \( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).

**Ex:** \( p(8) = 5, p(7) = 3, p(2) = 0. \)
Dynamic Programming: Binary Choice

Notation. \( \text{OPT}(j) = \) value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- **Case 1:** OPT selects job j.
  - collect profit \( v_j \)
  - can’t use incompatible jobs \{ p(j) + 1, p(j) + 2, ..., j - 1 \}
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

- **Case 2:** OPT does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ v_j + \text{OPT}(p(j)), \text{OPT}(j-1) \} & \text{otherwise}
\end{cases}
\]
Weighted Interval Scheduling: Brute Force

**Brute force algorithm.**

**Input:** \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

Sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

Compute \( p(1), p(2), \ldots, p(n) \)

Compute-Opt\( (j) \) {
  if \( (j = 0) \)
    return 0
  else
    return max\( (v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)) \)
}
**Observation.** Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

**Ex.** Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.

\[
p(1) = 0, \quad p(j) = j-2
\]
Weighted Interval Scheduling: Memoization

**Memoization.** Store results of each sub-problem in a cache; lookup as needed.

---

**Input:** n, s₁,...,sₙ, f₁,...,fₙ, v₁,...,vₙ

Sort jobs by finish times so that \( f_1 \leq f_2 \leq ... \leq f_n \).

Compute \( p(1), p(2), ..., p(n) \)

for \( j = 1 \) to \( n \)

\[
\begin{align*}
&M[j] = \text{empty} \\
&M[0] = 0
\end{align*}
\]

M-Compute-Opt(j) {
  if (M[j] is empty)
    \[
    M[j] = \max(v_j + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
    \]
  return M[j]
}

---
Claim. Memoized version of algorithm takes $O(n \log n)$ time.

- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time.

- $M$-Compute-Opt($j$): each invocation takes $O(1)$ time and either
  - (i) returns an existing value $M[j]$ 
  - (ii) fills in one new entry $M[j]$ and makes two recursive calls

- Progress measure $\Phi = \#$ nonempty entries of $M[\cdot]$.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases $\Phi$ by 1 $\Rightarrow$ at most $2n$ recursive calls.

- Overall running time of $M$-Compute-Opt($n$) is $O(n)$. □

Remark. $O(n)$ if jobs are pre-sorted by start and finish times.
Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?
A. Do some post-processing.

```plaintext
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (v_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}
```

- # of recursive calls \leq n \Rightarrow O(n).
Weighted Interval Scheduling: Bottom-Up

**Bottom-up dynamic programming.** Unwind recursion.

**Input:** n, s₁, s₂, sₙ, f₁, f₂, fₙ, v₁, v₂, vₙ

Sort jobs by finish times so that f₁ ≤ f₂ ≤ ... ≤ fₙ.

Compute p(1), p(2), ..., p(n)

Iterative-Compute-Opt {
    M[0] = 0
    for j = 1 to n
        M[j] = max(vⱼ + M[p(j)], M[j-1])
}
6.4 Knapsack Problem
Knapsack problem.

- Given \( n \) objects and a "knapsack."
- Item \( i \) weighs \( w_i > 0 \) kilograms and has value \( v_i > 0 \).
- Knapsack has capacity of \( W \) kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: \( \{ 3, 4 \} \) has value 40.

<table>
<thead>
<tr>
<th>#</th>
<th>value</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

\( W = 11 \)

**Greedy:** repeatedly add item with maximum ratio \( v_i / w_i \).

Ex: \( \{ 5, 2, 1 \} \) achieves only value = 35 \( \Rightarrow \) greedy not optimal.
**Dynamic Programming: False Start**

**Def.** \( \text{OPT}(i) = \text{max profit subset of items } 1, ..., i. \)

- **Case 1:** \( \text{OPT} \) does not select item \( i \).
  - \( \text{OPT} \) selects best of \( \{ 1, 2, ..., i-1 \} \)

- **Case 2:** \( \text{OPT} \) selects item \( i \).
  - accepting item \( i \) does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before \( i \), we don't even know if we have enough room for \( i \)

**Conclusion.** Need more sub-problems!
Dynamic Programming: Adding a New Variable

Def. \( OPT(i, w) = \text{max profit subset of items 1, \ldots, i with weight limit } w \).

- **Case 1:** \( OPT \) does not select item \( i \).
  - \( OPT \) selects best of \( \{1, 2, \ldots, i-1\} \) using weight limit \( w \)

- **Case 2:** \( OPT \) selects item \( i \).
  - new weight limit = \( w - w_i \)
  - \( OPT \) selects best of \( \{1, 2, \ldots, i-1\} \) using this new weight limit

\[
OPT(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
OPT(i - 1, w) & \text{if } w_i > w \\
\max\{OPT(i - 1, w), \ v_i + OPT(i - 1, w - w_i)\} & \text{otherwise}
\end{cases}
\]
Algorithms — March 7

Tonight
- 3-Minute Thesis competition, 1400 Centennial Hall, 6pm

This week
- Dynamic programming
- No quiz

Next week
- Spring break
- Lab closed Sat.-Sat., reopens Sunday regular hours

Monday, March 19
- We’re back
- Problems due: 5.5, 5.6

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Dynamic Programming: Adding a New Variable

**Def.** $\text{OPT}(i, w) = \max \text{ profit subset of items } 1, \ldots, i \text{ with weight limit } w$.

- **Case 1:** $\text{OPT}$ does not select item $i$.
  - $\text{OPT}$ selects best of $\{1, 2, \ldots, i-1\}$ using weight limit $w$

- **Case 2:** $\text{OPT}$ selects item $i$.
  - new weight limit $= w - w_i$
  - $\text{OPT}$ selects best of $\{1, 2, \ldots, i-1\}$ using this new weight limit

\[
\text{OPT}(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
\text{OPT}(i-1, w) & \text{if } w_i > w \\
\max \{ \text{OPT}(i-1, w), \ v_i + \text{OPT}(i-1, w - w_i) \} & \text{otherwise}
\end{cases}
\]
Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

Input: n, W, \( w_1, \ldots, w_N, v_1, \ldots, v_N \)

\begin{verbatim}
for w = 0 to W
    M[0, w] = 0

for i = 1 to n
    for w = 1 to W
        if (w_i > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = \max \{ M[i-1, w], v_i + M[i-1, w-w_i] \}

return M[n, W]
\end{verbatim}
### Knapsack Algorithm

**OPT: \{ 4, 3 \}**

value = 22 + 18 = 40
Knapsack Problem: Running Time

Running time. $\Theta(n W)$.

- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm. There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]
This week
- Dynamic programming
- No quiz

Next week
- Spring break
- Lab closed Sat.-Sat., reopens Sunday regular hours

Monday, March 19
- We’re back
- Problems due: 5.5, 5.6

Friday, March 23
- Quiz on divide-and-conquer

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
6.3 Segmented Least Squares
Segmented Least Squares

**Least squares.**

- Foundational problem in statistic and numerical analysis.
- Given $n$ points in the plane: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$.
- Find a line $y = ax + b$ that minimizes the sum of the squared error:

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$$

**Solution.** Calculus $\Rightarrow$ min error is achieved when

$$a = \frac{n \sum x_i y_i - (\sum x_i)(\sum y_i)}{n \sum x_i^2 - (\sum x_i)^2}, \quad b = \frac{\sum y_i - a \sum x_i}{n}$$
Segmented least squares.
- Points lie roughly on a sequence of several line segments.
- Given $n$ points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

**Q.** What's a reasonable choice for $f(x)$ to balance accuracy and parsimony?
Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given $n$ points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes:
  - the sum of the sums of the squared errors $E$ in each segment
  - the number of lines $L$
- Tradeoff function: $E + cL$, for some constant $c > 0$. 
Dynamic Programming: Multiway Choice

Notation.
- $OPT(j) =$ minimum cost for points $p_1, p_{i+1}, \ldots, p_j$.
- $e(i, j) =$ minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.

To compute $OPT(j)$:
- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some $i$.
- Cost = $e(i, j) + c + OPT(i-1)$.

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\min_{1 \leq i \leq j} \{ e(i, j) + c + OPT(i-1) \} & \text{otherwise}
\end{cases}
\]
Segmented Least Squares: Algorithm

**INPUT:** \( n, p_1, \ldots, p_N, c \)

Segmented-Least-Squares() {
    \( M[0] = 0 \)
    for \( j = 1 \) to \( n \)
        for \( i = 1 \) to \( j \)
            compute the least square error \( e_{ij} \) for the segment \( p_i, \ldots, p_j \)

    for \( j = 1 \) to \( n \)
        \( M[j] = \min_{1 \leq i \leq j} (e_{ij} + c + M[i-1]) \)

    return \( M[n] \)
}

**Running time.** \( O(n^3) \).  
- Bottleneck = computing \( e(i, j) \) for \( O(n^2) \) pairs, \( O(n) \) per pair using previous formula.

\[ \text{can be improved to } O(n^2) \text{ by pre-computing various statistics} \]
6.5 RNA Secondary Structure
RNA. String $B = b_1b_2\ldots b_n$ over alphabet $\{A, C, G, U\}$.

Secondary structure. RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

Ex: GUCGAUUGAGCGAUGUAACAGUGCUGCUCGGCGAGA

complementary base pairs: $A-U, C-G$
RNA Secondary Structure

Secondary structure. A set of pairs \( S = \{ (b_i, b_j) \} \) that satisfy:

- [Watson-Crick.] \( S \) is a matching and each pair in \( S \) is a Watson-Crick complement: A-U, U-A, C-G, or G-C.
- [No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If \((b_i, b_j) \in S\), then \( i < j - 4 \).
- [Non-crossing.] If \((b_i, b_j)\) and \((b_k, b_l)\) are two pairs in \( S \), then we cannot have \( i < k < j < l \).

Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

Goal. Given an RNA molecule \( B = b_1b_2\ldots b_n \), find a secondary structure \( S \) that maximizes the number of base pairs.
RNA Secondary Structure: Examples

Examples.

![RNA Secondary Structure Diagrams](image)

- **Base Pair:**
- **Sharp Turn:**
- **Crossing:**

- **OK:**
- **≤4:**

Note: The diagrams illustrate various RNA secondary structures, including base pairs, sharp turns, and crossings, along with corresponding nucleotide sequences.
RNA Secondary Structure: Subproblems

First attempt. $\text{OPT}(j) =$ maximum number of base pairs in a secondary structure of the substring $b_1b_2...b_j$.

Difficulty. Results in two sub-problems.
- Finding secondary structure in: $b_1b_2...b_{t-1}$.
- Finding secondary structure in: $b_{t+1}b_{t+2}...b_{n-1}$.

match $b_t$ and $b_n$
Dynamic Programming Over Intervals

**Notation.** \( \text{OPT}(i, j) = \) maximum number of base pairs in a secondary structure of the substring \( b_i b_{i+1} \ldots b_j \).

- **Case 1.** If \( i \geq j - 4 \).
  - \( \text{OPT}(i, j) = 0 \) by no-sharp turns condition.

- **Case 2.** Base \( b_j \) is not involved in a pair.
  - \( \text{OPT}(i, j) = \text{OPT}(i, j-1) \)

- **Case 3.** Base \( b_j \) pairs with \( b_t \) for some \( i \leq t < j - 4 \).
  - non-crossing constraint decouples resulting sub-problems
  - \( \text{OPT}(i, j) = 1 + \max_t \{ \text{OPT}(i, t-1) + \text{OPT}(t+1, j-1) \} \)

  take max over \( t \) such that \( i \leq t < j-4 \) and \( b_t \) and \( b_j \) are Watson-Crick complements

**Remark.** Same core idea in CKY algorithm to parse context-free grammars.
Bottom Up Dynamic Programming Over Intervals

Q. What order to solve the sub-problems?
A. Do shortest intervals first.

\[
\text{RNA}(b_1, \ldots, b_n) \begin{cases} 
\text{for } k = 5, 6, \ldots, n-1 \\
\quad \text{for } i = 1, 2, \ldots, n-k \\
\quad \quad j = i + k \\
\quad \quad \text{Compute } M[i, j] \\
\text{return } M[1, n] \end{cases}
\]

\text{using recurrence}

Running time. \( O(n^3) \).
Dynamic Programming Summary

Recipe.
- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Dynamic programming techniques.
- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares.
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

Top-down vs. bottom-up: different people have different intuitions.
6.6 Sequence Alignment
String Similarity

How similar are two strings?

- occurrence
- occurrence

6 mismatches, 1 gap

1 mismatch, 1 gap

0 mismatches, 3 gaps
Applications.
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

- Gap penalty $\delta$; mismatch penalty $\alpha_{pq}$.
- Cost = sum of gap and mismatch penalties.

$$
\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA} = 2\delta + \alpha_{CA}
$$
Sequence Alignment

**Goal:** Given two strings $X = x_1x_2\ldots x_m$ and $Y = y_1y_2\ldots y_n$ find alignment of minimum cost.

**Def.** An alignment $M$ is a set of ordered pairs $x_i$-$y_j$ such that each item occurs in at most one pair and no crossings.

**Def.** The pair $x_i$-$y_j$ and $x_{i'}$-$y_{j'}$ cross if $i < i'$, but $j > j'$.

$$
\text{cost}(M) = \sum_{(x_i,y_j) \in M} \alpha_{x_i y_j} \quad + \quad \sum_{i : x_i \text{ unmatched}} \delta \quad + \quad \sum_{j : y_j \text{ unmatched}} \delta
$$

**Ex:** CTACCG vs. TACATG.

**Sol:** $M = x_2$-$y_1$, $x_3$-$y_2$, $x_4$-$y_3$, $x_5$-$y_4$, $x_6$-$y_6$. 
Sequence Alignment: Problem Structure

Def. \( \text{OPT}(i, j) = \min \text{ cost of aligning strings } x_1 x_2 \ldots x_i \text{ and } y_1 y_2 \ldots y_j. \)

- **Case 1:** \( \text{OPT} \) matches \( x_i-y_j \).
  - pay mismatch for \( x_i-y_j \) + min cost of aligning two strings
    \( x_1 x_2 \ldots x_{i-1} \text{ and } y_1 y_2 \ldots y_{j-1} \)

- **Case 2a:** \( \text{OPT} \) leaves \( x_i \) unmatched.
  - pay gap for \( x_i \) and min cost of aligning \( x_1 x_2 \ldots x_{i-1} \text{ and } y_1 y_2 \ldots y_j \)

- **Case 2b:** \( \text{OPT} \) leaves \( y_j \) unmatched.
  - pay gap for \( y_j \) and min cost of aligning \( x_1 x_2 \ldots x_i \text{ and } y_1 y_2 \ldots y_{j-1} \)

\[
\text{OPT}(i,j) = \begin{cases} 
  j\delta & \text{if } i = 0 \\
  \min \left\{ \begin{array}{ll}
    \alpha_{x_i y_j} + \text{OPT}(i-1, j-1) & \delta + \text{OPT}(i-1, j) \\
    \delta + \text{OPT}(i, j-1) & \end{array} \right. & \text{otherwise} \\
  i\delta & \text{if } j = 0
\end{cases}
\]
Sequence Alignment: Algorithm

```
Sequence-Alignment(m, n, x_1x_2...x_m, y_1y_2...y_n, δ, α) {
    for i = 0 to m
        M[i, 0] = iδ
    for j = 0 to n
        M[0, j] = jδ

    for i = 1 to m
        for j = 1 to n
            M[i, j] = min(α[x_i, y_j] + M[i-1, j-1],
                            δ + M[i-1, j],
                            δ + M[i, j-1])
    return M[m, n]
}
```

**Analysis.** Θ(mn) time and space.

**English words or sentences:** m, n ≤ 10.

**Computational biology:** m = n = 100,000. 10 billions ops OK, but 10GB array?
6.7 Sequence Alignment in Linear Space
Sequence Alignment: Linear Space

Q. Can we avoid using quadratic space?

Easy. Optimal value in $O(m + n)$ space and $O(mn)$ time.
   - Compute $\text{OPT}(i, \cdot)$ from $\text{OPT}(i-1, \cdot)$.
   - No longer a simple way to recover alignment itself.

Theorem. [Hirschberg 1975] Optimal alignment in $O(m + n)$ space and $O(mn)$ time.
   - Clever combination of divide-and-conquer and dynamic programming.
   - Inspired by idea of Savitch from complexity theory.
**Sequence Alignment: Linear Space**

**Edit distance graph.**
- Let $f(i, j)$ be shortest path from $(0,0)$ to $(i, j)$.
- Observation: $f(i, j) = \text{OPT}(i, j)$. 
Edit distance graph.

- Let $f(i, j)$ be shortest path from $(0, 0)$ to $(i, j)$.
- Can compute $f(\cdot, j)$ for any $j$ in $O(mn)$ time and $O(m + n)$ space.
Sequence Alignment: Linear Space

Edit distance graph.

- Let $g(i, j)$ be shortest path from $(i, j)$ to $(m, n)$.
- Can compute by reversing the edge orientations and inverting the roles of $(0, 0)$ and $(m, n)$.
Edit distance graph.

- Let $g(i, j)$ be shortest path from $(i, j)$ to $(m, n)$.
- Can compute $g(\cdot, j)$ for any $j$ in $O(mn)$ time and $O(m + n)$ space.

Sequence Alignment: Linear Space

```
  j
 /  /
/    /
0-0 Y1  Y2  Y3  Y4  Y5  Y6
|
|  x1                   x2
|  |
|  x3
```

```
i-j
m-n
```

48
Observation 1. The cost of the shortest path that uses \((i, j)\) is \(f(i, j) + g(i, j)\).
Observation 2. Let \( q \) be an index that minimizes \( f(q, n/2) + g(q, n/2) \). Then, the shortest path from \((0, 0)\) to \((m, n)\) uses \((q, n/2)\).
Sequence Alignment: Linear Space

**Divide:** find index $q$ that minimizes $f(q, n/2) + g(q, n/2)$ using DP.

- Align $x_q$ and $y_{n/2}$.

**Conquer:** recursively compute optimal alignment in each piece.
Sequence Alignment: Running Time Analysis Warmup

**Theorem.** Let $T(m, n) = \text{max running time of algorithm on strings of length at most } m \text{ and } n$. $T(m, n) = O(mn \log n)$.

\[
T(m, n) \leq 2T(m, n/2) + O(mn) \quad \Rightarrow \quad T(m, n) = O(mn \log n)
\]

**Remark.** Analysis is not tight because two sub-problems are of size $(q, n/2)$ and $(m - q, n/2)$. In next slide, we save log $n$ factor.
Sequence Alignment: Running Time Analysis

**Theorem.** Let \( T(m, n) = \max \) running time of algorithm on strings of length \( m \) and \( n \). \( T(m, n) = O(mn) \).

**Pf.** (by induction on \( n \))
- \( O(mn) \) time to compute \( f(\cdot, n/2) \) and \( g(\cdot, n/2) \) and find index \( q \).
- \( T(q, n/2) + T(m - q, n/2) \) time for two recursive calls.
- Choose constant \( c \) so that:

\[
T(m, 2) \leq cm \\
T(2, n) \leq cn \\
T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)
\]

- Base cases: \( m = 2 \) or \( n = 2 \).
- Inductive hypothesis: \( T(m, n) \leq 2cmn \).

\[
T(m,n) \leq T(q,n/2) + T(m - q,n/2) + cmn \\
\leq 2cqn/2 + 2c(m-q)n/2 + cmn \\
= cqn + cmn - cqn + cmn \\
= 2cmn
\]
Algorithms — March 19

Today

▶ We’re back
▶ Dynamic programming
▶ Problems due: 5.5, 5.6

Wednesday

▶ Dynamic programming

Friday

▶ Quiz on divide-and-conquer

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
String Similarity

How similar are two strings?

- occurrance
- occurrence

6 mismatches, 1 gap

1 mismatch, 1 gap

0 mismatches, 3 gaps
Applications.
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

- Gap penalty $\delta$; mismatch penalty $\alpha_{pq}$.
- Cost = sum of gap and mismatch penalties.

$$\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA} \quad \text{and} \quad 2\delta + \alpha_{CA}$$
Goal: Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ find alignment of minimum cost.

Def. An alignment $M$ is a set of ordered pairs $x_i$-$y_j$ such that each item occurs in at most one pair and no crossings.

Def. The pair $x_i$-$y_j$ and $x_i'$-$y_j'$ cross if $i < i'$, but $j > j'$.

\[
\text{cost}(M) = \sum_{(x_i, y_j) \in M} \alpha_{x_i y_j} + \sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta
\]

Ex: CTACCG vs. TACATG.

Sol: $M = x_2$-$y_1$, $x_3$-$y_2$, $x_4$-$y_3$, $x_5$-$y_4$, $x_6$-$y_6$. 
**Sequence Alignment: Problem Structure**

**Def.** $OPT(i, j) = \text{min cost of aligning strings } x_1 x_2 \ldots x_i \text{ and } y_1 y_2 \ldots y_j$.

- **Case 1:** $OPT$ matches $x_i$-$y_j$.
  - pay mismatch for $x_i$-$y_j$ + min cost of aligning two strings $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_{j-1}$
- **Case 2a:** $OPT$ leaves $x_i$ unmatched.
  - pay gap for $x_i$ and min cost of aligning $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_j$
- **Case 2b:** $OPT$ leaves $y_j$ unmatched.
  - pay gap for $y_j$ and min cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$

$$OPT(i, j) = \begin{cases} 
  j \delta & \text{if } i = 0 \\
  \min \left\{ \begin{array}{ll}
  \alpha_{x_i y_j} + OPT(i-1, j-1) \\
  \delta + OPT(i-1, j) \\
  \delta + OPT(i, j-1) \\
  i \delta 
\end{array} \right. & \text{otherwise} \\
\end{cases}$$
Sequence Alignment: Algorithm

```plaintext
Sequence-Alignment(m, n, x_1x_2...x_m, y_1y_2...y_n, δ, α) {
    for i = 0 to m
        M[i, 0] = iδ
    for j = 0 to n
        M[0, j] = jδ
    for i = 1 to m
        for j = 1 to n
            M[i, j] = min(α[x_i, y_j] + M[i-1, j-1],
                           δ + M[i-1, j],
                           δ + M[i, j-1])
    return M[m, n]
}
```

**Analysis.** Θ(mn) time and space.

**English words or sentences:** m, n ≤ 10.

**Computational biology:** m = n = 100,000. 10 billions ops OK, but 10GB array?
6.7 Sequence Alignment in Linear Space
**Sequence Alignment: Linear Space**

**Q.** Can we avoid using quadratic space?

**Easy.** Optimal value in $O(m + n)$ space and $O(mn)$ time.
- Compute $OPT(i, \cdot)$ from $OPT(i-1, \cdot)$.
- No longer a simple way to recover alignment itself.

**Theorem.** [Hirschberg 1975] Optimal alignment in $O(m + n)$ space and $O(mn)$ time.
- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.
Edit distance graph.

- Let $f(i, j)$ be shortest path from $(0,0)$ to $(i, j)$.
- Observation: $f(i, j) = \text{OPT}(i, j)$.

Sequence Alignment: Linear Space
Edit distance graph.

- Let $f(i, j)$ be shortest path from $(0,0)$ to $(i, j)$.
- Can compute $f(\cdot, j)$ for any $j$ in $O(mn)$ time and $O(m + n)$ space.
Sequence Alignment: Linear Space

Edit distance graph.
- Let $g(i, j)$ be shortest path from $(i, j)$ to $(m, n)$.
- Can compute by reversing the edge orientations and inverting the roles of $(0, 0)$ and $(m, n)$
Sequence Alignment: Linear Space

**Edit distance graph.**
- Let $g(i, j)$ be shortest path from $(i, j)$ to $(m, n)$.
- Can compute $g(\cdot, j)$ for any $j$ in $O(mn)$ time and $O(m + n)$ space.
Observation 1. The cost of the shortest path that uses \((i, j)\) is \(f(i, j) + g(i, j)\).
Observation 2. Let $q$ be an index that minimizes $f(q, n/2) + g(q, n/2)$. Then, the shortest path from $(0, 0)$ to $(m, n)$ uses $(q, n/2)$.
**Sequence Alignment: Linear Space**

**Divide:** find index $q$ that minimizes $f(q, n/2) + g(q, n/2)$ using DP.
- Align $x_q$ and $y_{n/2}$.

**Conquer:** recursively compute optimal alignment in each piece.
**Theorem.** Let $T(m, n) = \max$ running time of algorithm on strings of length $m$ and $n$. $T(m, n) = O(mn)$.

**Pf.** (by induction on $n$)
- $O(mn)$ time to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ and find index $q$.
- $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.
- Choose constant $c$ so that:

  $T(m, 2) \leq cm$
  $T(2, n) \leq cn$
  $T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$

- Base cases: $m = 2$ or $n = 2$.
- Inductive hypothesis: $T(m, n) \leq 2cmn$.  

  $T(m, n) \leq T(q, n/2) + T(m - q, n/2) + cmn$
  $\leq 2cqn/2 + 2c(m - q)n/2 + cmn$
  $= cqn + cmn - cqn + cmn$
  $= 2cmn$
Today

▶ Dynamic programming

Friday

▶ Quiz on divide-and-conquer

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
   ▶ Expect replies within a business day (but often same-day)
▶ Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
▶ Or by appointment
   ▶ Email at least a day ahead
   ▶ Send several times when you are available
   ▶ Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Shortest path problem. Given a directed graph $G = (V, E)$, with edge weights $c_{vw}$, find shortest path from node $s$ to node $t$.

Ex. Nodes represent agents in a financial setting and $c_{vw}$ is cost of transaction in which we buy from agent $v$ and sell immediately to $w$. 
Shortest Paths: Failed Attempts

**Dijkstra.** Can fail if negative edge costs.

![Graph](image1)

**Re-weighting.** Adding a constant to every edge weight can fail.

![Graph](image2)
Shortest Paths: Negative Cost Cycles

Negative cost cycle.

Observation. If some path from $s$ to $t$ contains a negative cost cycle, there does not exist a shortest $s$-$t$ path; otherwise, there exists one that is simple.
Shortest Paths: Dynamic Programming

Def. $OPT(i, v) =$ length of shortest $v$-$t$ path $P$ using at most $i$ edges.

- **Case 1:** $P$ uses at most $i-1$ edges.
  - $OPT(i, v) = OPT(i-1, v)$

- **Case 2:** $P$ uses exactly $i$ edges.
  - if $(v, w)$ is first edge, then $OPT$ uses $(v, w)$, and then selects best $w$-$t$ path using at most $i-1$ edges

$OPT(i, v) = \begin{cases} 
0 & \text{if } i = 0 \\
\min \left\{ OPT(i - 1, v), \min_{(v, w) \in E} \left\{ OPT(i - 1, w) + c_{vw} \right\} \right\} & \text{otherwise}
\end{cases}$

Remark. By previous observation, if no negative cycles, then $OPT(n-1, v) =$ length of shortest $v$-$t$ path.
Shortest Paths: Implementation

```
Shortest-Path(G, t) {
    foreach node v ∈ V
        M[0, v] ← ∞
    M[0, t] ← 0

    for i = 1 to n-1
        foreach node v ∈ V
            M[i, v] ← M[i-1, v]
        foreach edge (v, w) ∈ E
            M[i, v] ← min { M[i, v], M[i-1, w] + c_{vw} }
}
```

Analysis. \( \Theta(mn) \) time, \( \Theta(n^2) \) space.

Finding the shortest paths. Maintain a "successor" for each table entry.
Practical improvements.

- Maintain only one array $M[v] =$ shortest v-t path that we have found so far.
- No need to check edges of the form $(v, w)$ unless $M[w]$ changed in previous iteration.

**Theorem.** Throughout the algorithm, $M[v]$ is length of some v-t path, and after $i$ rounds of updates, the value $M[v]$ is no larger than the length of shortest v-t path using $\leq i$ edges.

**Overall impact.**

- Memory: $O(m + n)$.
- Running time: $O(mn)$ worst case, but substantially faster in practice.
Bellman-Ford: Efficient Implementation

Push-Based-Shortest-Path(G, s, t) {
    foreach node v ∈ V {
        M[v] ← ∞
        successor[v] ← φ
    }

    M[t] = 0
    for i = 1 to n-1 {
        foreach node w ∈ V {
            if (M[w] has been updated in previous iteration) {
                foreach node v such that (v, w) ∈ E {
                    if (M[v] > M[w] + c_{vw}) {
                        M[v] ← M[w] + c_{vw}
                        successor[v] ← w
                    }
                }
            }
        }
        If no M[w] value changed in iteration i, stop.
    }
}
Detecting Negative Cycles

Lemma. If $\text{OPT}(n,v) = \text{OPT}(n-1,v)$ for all $v$, then no negative cycles.


Lemma. If $\text{OPT}(n,v) < \text{OPT}(n-1,v)$ for some node $v$, then (any) shortest path from $v$ to $t$ contains a cycle $W$. Moreover $W$ has negative cost.

Pf. (by contradiction)

- Since $\text{OPT}(n,v) < \text{OPT}(n-1,v)$, we know $P$ has exactly $n$ edges.
- By pigeonhole principle, $P$ must contain a directed cycle $W$.
- Deleting $W$ yields a $v$-$t$ path with $< n$ edges $\Rightarrow W$ has negative cost.
Detecting Negative Cycles

**Theorem.** Can detect negative cost cycle in $O(mn)$ time.
- Add new node $t$ and connect all nodes to $t$ with 0-cost edge.
- Check if $OPT(n, v) = OPT(n-1, v)$ for all nodes $v$.
  - if yes, then no negative cycles
  - if no, then extract cycle from shortest path from $v$ to $t$
Detecting Negative Cycles: Application

**Currency conversion.** Given n currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

**Remark.** Fastest algorithm very valuable!
Detecting Negative Cycles: Summary

**Bellman-Ford.** $O(mn)$ time, $O(m + n)$ space.
- Run Bellman-Ford for $n$ iterations (instead of $n-1$).
- Upon termination, Bellman-Ford successor variables trace a negative cycle if one exists.
- See p. 304 for improved version and early termination rule.
Algorithms — March 23

Today
- Quiz on divide-and-conquer
- Intro to network flow

Monday
- Network flow
- Problems due: 6.2, 6.3

Next Monday
- Problems due: 6.9, 6.11

Contact information
- Web: cs.uwlax.edu/~jmarais
- Email: jmarais@uwlax.edu
  - Expect replies within a business day (but often same-day)
- Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Flow network.

- Abstraction for material **flowing** through the edges.
- \( G = (V, E) \) = directed graph, no parallel edges.
- Two distinguished nodes: \( s = \) source, \( t = \) sink.
- \( c(e) \) = capacity of edge \( e \).

Minimum Cut Problem

![Flow Network Diagram](image-url)
**Def.** An \( s-t \) cut is a partition \((A, B)\) of \(V\) with \(s \in A\) and \(t \in B\).

**Def.** The capacity of a cut \((A, B)\) is:

\[
\text{Cap}(A, B) = \sum_{e \text{ out of } A} c(e)
\]
Cuts

Def. An \( s \)-\( t \) cut is a partition \((A, B)\) of \( V \) with \( s \in A \) and \( t \in B \).

Def. The \textit{capacity} of a cut \((A, B)\) is:

\[
\text{cap}(A, B) = \sum_{\text{e out of } A} c(e)
\]

\[
\text{Capacity} = 9 + 15 + 8 + 30 = 62
\]
Min s-t cut problem. Find an s-t cut of minimum capacity.

Minimum Cut Problem

Capacity = 10 + 8 + 10 = 28


**Flows**

**Def.** An \( s \rightarrow t \) flow is a function that satisfies:

- For each \( e \in E \):
  \[ 0 \leq f(e) \leq c(e) \]  
  [capacity]

- For each \( v \in V - \{s, t\} \):
  \[ \sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e) \]  
  [conservation]

**Def.** The value of a flow \( f \) is:

\[ v(f) = \sum_{e \text{ out of } s} f(e) \].
**Flows**

**Def.** An $s$-$t$ flow is a function that satisfies:

- For each $e \in E$: \[ 0 \leq f(e) \leq c(e) \] [capacity]
- For each $v \in V - \{s, t\}$: \[ \sum_{e \in \text{in to } v} f(e) = \sum_{e \in \text{out of } v} f(e) \] [conservation]

**Def.** The value of a flow $f$ is: \[ v(f) = \sum_{e \text{ out of } s} f(e) \].

![Flow Network Diagram]

Value = 24
Max flow problem. Find s-t flow of maximum value.
Maximum Flow and Minimum Cut

Max flow and min cut.
- Two very rich algorithmic problems.
- Cornerstone problems in combinatorial optimization.
- Beautiful mathematical duality.

Nontrivial applications / reductions.
- Data mining.
- Open-pit mining.
- Project selection.
- Airline scheduling.
- Bipartite matching.
- Baseball elimination.
- Image segmentation.
- Network connectivity.
- Network reliability.
- Distributed computing.
- Egalitarian stable matching.
- Security of statistical data.
- Network intrusion detection.
- Multi-camera scene reconstruction.
- Many many more ...
**Flow value lemma.** Let $f$ be any flow, and let $(A, B)$ be any $s$-$t$ cut. Then, the net flow sent across the cut is equal to the amount leaving $s$.

$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f)$$
Flow value lemma. Let $f$ be any flow, and let $(A, B)$ be any $s$-$t$ cut. Then, the net flow sent across the cut is equal to the amount leaving $s$.

$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f)$$

Value $= 6 + 0 + 8 - 1 + 11 = 24$
**Flows and Cuts**

**Flow value lemma.** Let $f$ be any flow, and let $(A, B)$ be any $s$-$t$ cut. Then, the net flow sent across the cut is equal to the amount leaving $s$.

\[
\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f)
\]

Value $= 10 - 4 + 8 - 0 + 10 = 24$
Weak duality. Let $f$ be any flow, and let $(A, B)$ be any s-t cut. Then the value of the flow is at most the capacity of the cut.

Cut capacity = 30 $\Rightarrow$ Flow value $\leq$ 30
Corollary. Let $f$ be any flow, and let $(A, B)$ be any cut. If $v(f) = \text{cap}(A, B)$, then $f$ is a max flow and $(A, B)$ is a min cut.

Value of flow = 28  
Cut capacity = 28  ⇒  Flow value ≤ 28
Algorithms — March 26

Today

- Network flow
- Problems due: 6.2, 6.3

Next Monday

- Problems due: 6.9, 6.11

Contact information

- Web: cs.uwlax.edu/~jmarais
- Email: jmarais@uwlax.edu
  - Expect replies within a business day (but often same-day)
- Office hours: Tues. 1:30-3:30pm, Wed. 4-5pm, Fri. 2-3:30pm
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss — for me to prepare, and often advising or paperwork will not require meeting
Greedy algorithm.

- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an s-t path $P$ where each edge has $f(e) < c(e)$.
- Augment flow along path $P$.
- Repeat until you get stuck.

Flow value = 0
Towards a Max Flow Algorithm

**Greedy algorithm.**
- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an $s$-$t$ path $P$ where each edge has $f(e) < c(e)$.
- Augment flow along path $P$.
- Repeat until you get stuck.

Flow value = 20
Towards a Max Flow Algorithm

**Greedy algorithm.**
- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an s-t path $P$ where each edge has $f(e) < c(e)$.
- Augment flow along path $P$.
- Repeat until you get stuck.

$\text{Greedy} = 20$

$\text{Opt} = 30$

locally optimality $\Rightarrow$ global optimality
Residual Graph

Original edge: \( e = (u, v) \in E \).
- Flow \( f(e) \), capacity \( c(e) \).

Residual edge.
- "Undo" flow sent.
- \( e = (u, v) \) and \( e^R = (v, u) \).
- Residual capacity:
  \[
  c_f(e) = \begin{cases} 
  c(e) - f(e) & \text{if } e \in E \\
  f(e) & \text{if } e^R \in E 
  \end{cases}
  \]

Residual graph: \( G_f = (V, E_f) \).
- Residual edges with positive residual capacity.
- \( E_f = \{ e : f(e) < c(e) \} \cup \{ e^R : f(e) > 0 \} \).
Ford-Fulkerson Algorithm

$G$:  

![Graph with capacities](image-url)
Augmenting Path Algorithm

Augment(f, c, P) {
    b ← bottleneck(P)
    foreach e ∈ P {
        if (e ∈ E) f(e) ← f(e) + b
        else f(e^R) ← f(e^R) - b
    }
    return f
}

Ford-Fulkerson(G, s, t, c) {
    foreach e ∈ E  f(e) ← 0
    G_f ← residual graph

    while (there exists augmenting path P) {
        f ← Augment(f, c, P)
        update G_f
    }
    return f
}
Max-Flow Min-Cut Theorem

Augmenting path theorem. Flow $f$ is a max flow iff there are no augmenting paths.


Pf. We prove both simultaneously by showing TFAE:

(i) There exists a cut $(A, B)$ such that $v(f) = \text{cap}(A, B)$.
(ii) Flow $f$ is a max flow.
(iii) There is no augmenting path relative to $f$.

(i) $\Rightarrow$ (ii) This was the corollary to weak duality lemma.

(ii) $\Rightarrow$ (iii) We show contrapositive.
   - Let $f$ be a flow. If there exists an augmenting path, then we can improve $f$ by sending flow along path.
Proof of Max-Flow Min-Cut Theorem

(iii) $\Rightarrow$ (i)

- Let $f$ be a flow with no augmenting paths.
- Let $A$ be set of vertices reachable from $s$ in residual graph.
- By definition of $A$, $s \in A$.
- By definition of $f$, $t \notin A$.

$$v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e)$$

$$= \sum_{e \text{ out of } A} c(e)$$

$$= \text{cap}(A, B)$$
**Running Time**

**Assumption.** All capacities are integers between 1 and $C$.

**Invariant.** Every flow value $f(e)$ and every residual capacity $c_f(e)$ remains an integer throughout the algorithm.

**Theorem.** The algorithm terminates in at most $v(f^*) \leq nC$ iterations.

**Pf.** Each augmentation increase value by at least 1. □

**Corollary.** If $C = 1$, Ford-Fulkerson runs in $O(mn)$ time.

**Integrality theorem.** If all capacities are integers, then there exists a max flow $f$ for which every flow value $f(e)$ is an integer.

**Pf.** Since algorithm terminates, theorem follows from invariant. □