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 faster) during the semester
- Office hours through spring break
  - Tuesdays 1:30-3:30pm
  - Wednesdays 4-5pm
  - Fridays 2-3:30pm
- Or by appointment
  - 15-minutes slots, email at least a day ahead
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.

<table>
<thead>
<tr>
<th>Men’s Preference Profile</th>
<th>Women’s Preference Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>favorite</strong></td>
<td><strong>least favorite</strong></td>
</tr>
<tr>
<td>1st</td>
<td>2nd</td>
</tr>
<tr>
<td>Xavier</td>
<td>Amy</td>
</tr>
<tr>
<td>Yancey</td>
<td>Bertha</td>
</tr>
<tr>
<td>Zeus</td>
<td>Amy</td>
</tr>
</tbody>
</table>

1st, 2nd, 3rd
**Stable Matching Problem**

**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>Men’s Preference Profile</th>
<th>Women’s Preference Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Men</strong></td>
<td><strong>Women</strong></td>
</tr>
<tr>
<td>Xavier</td>
<td>Amy</td>
</tr>
<tr>
<td>Bertha</td>
<td>Bertha</td>
</tr>
<tr>
<td>Clare</td>
<td>Amy</td>
</tr>
<tr>
<td><strong>Amy</strong></td>
<td>Yancey</td>
</tr>
<tr>
<td><strong>Bertha</strong></td>
<td>Xavier</td>
</tr>
<tr>
<td><strong>Clare</strong></td>
<td>Yancey</td>
</tr>
</tbody>
</table>

**Men’s Preference Profiles:**
- **1st**: Xavier, Amy, Bertha
- **2nd**: Clare
- **3rd**: Z

**Women’s Preference Profiles:**
- **1st**: Amy, Yancey, Xavier
- **2nd**: Zeus
- **3rd**: B

**Stable Matching Problem**

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

Q. Is assignment X-C, Y-B, Z-A stable?
A. No. Bertha and Xavier will hook up.
Stable Matching Problem

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

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

<table>
<thead>
<tr>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>Yancey</td>
<td>Xavier</td>
</tr>
<tr>
<td>Bertha</td>
<td>Xavier</td>
<td>Yancey</td>
</tr>
<tr>
<td>Clare</td>
<td>Xavier</td>
<td>Yancey</td>
</tr>
<tr>
<td>Zeus</td>
<td>Xavier</td>
<td>Yancey</td>
</tr>
<tr>
<td>Zeus</td>
<td>Xavier</td>
<td>Yancey</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>Adam</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>Bob</td>
<td>C</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>Chris</td>
<td>A</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>Doofus</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

A-B, C-D \(\Rightarrow\) B-C unstable
A-C, B-D \(\Rightarrow\) A-B unstable
A-D, B-C \(\Rightarrow\) A-C unstable

Observation. Stable matchings do not always exist for stable roommate problem.
What is a solution?

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
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
Interval Scheduling

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

**Goal.** Find maximum cardinality subset of mutually compatible jobs.

jobs don't overlap
Weighted Interval Scheduling

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

**Goal.** Find *maximum weight* subset of mutually compatible jobs.
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
Competitive Facility Location

**Input.** Graph with weight on each 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/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a (business) day (but often faster) during the semester
- Office hours through spring break
  - Tuesdays 1:30-3:30pm
  - Wednesdays 4-5pm
  - Fridays 2-3:30pm
- Or by appointment
  - 15-minutes slots, email at least a day ahead
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 $c N^d$ 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 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.

<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>
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 $\Theta(n^3)$. 
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.** $a_0 + a_1 n + \ldots + a_d n^d$ 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 $d > 0$, $n^d = O(r^n)$.
Linear Time: $O(n)$

Linear time. Running time is proportional to input size.

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

```
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.

\[
i = 1, \quad j = 1
\]
\[
\text{while (both lists are nonempty)} \{
    \quad \text{if (} a_i \leq b_j \text{)} \quad \text{append } a_i \text{ to output list and increment } i \\
    \quad \text{else} \quad \text{append } b_j \text{ to output list and increment } j \\
\}
\]
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 \(x_1, \ldots, x_n\) 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.

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

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion.  

\[\text{see chapter 5}\]
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.

$$\text{foreach} \text{ subset } S \text{ of } k \text{ nodes} \{$$

$$\text{check whether } S \text{ in an independent set}$$

$$\text{if } (S \text{ is an independent set})$$

$$\text{report } S \text{ is an independent set}$$

$$\}$$

- 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

$k$ is a constant
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* ← φ
foreach subset S of nodes {
    check whether S is 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 faster) during the semester
- Office hours through spring break
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 an decent text, the authors develop skills across the problem set
▶ They’re walking you through a story
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
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. $a_0 + a_1 n + ... + a_d n^d$ 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 $d > 0$, $n^d = O(r^n)$.
Stable matching

So let’s turn back to stable matching

Initialize each person to be free.

while (some man is free and hasn't proposed to every woman) {
    Choose such a man m
    w = 1\textsuperscript{st} 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
}
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. □

<table>
<thead>
<tr>
<th>Victor</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
<th>5th</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>Wyatt</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>E</td>
</tr>
<tr>
<td>Xavier</td>
<td>C</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>Yancey</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>Zeus</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
<td>E</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>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Bertha</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>V</td>
<td>W</td>
</tr>
<tr>
<td>Clare</td>
<td>Y</td>
<td>Z</td>
<td>V</td>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>Diane</td>
<td>Z</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>Erika</td>
<td>V</td>
<td>W</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

$n(n-1) + 1$ proposals required
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

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 $m$. 
Efficient Implementation

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>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</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>


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

2 7
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.
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.
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 $\Rightarrow$ 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 $S$. $\blacksquare$
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. ← man-optimality
- Thus, A-Z is an unstable in S. □
Lessons Learned

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

**Potentially deep social ramifications.** [legal disclaimer]
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 faster) during the semester
▶ Office hours through spring break
  ▶ Tuesdays 1:30-3:30pm
  ▶ Wednesdays 4-5pm
  ▶ Fridays 2-3:30pm
▶ Or by appointment
  ▶ 15-minutes slots, email at least a day ahead
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
Each node’s key is at least as large as its parent’s.

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 $j = \text{parent}(i) = \lfloor i/2 \rfloor$
        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 \( n = \text{length}(H) \)
    If \( 2i > n \) then
        Terminate with \( H \) unchanged
    Else if \( 2i < n \) then
        Let \( \text{left} = 2i \), and \( \text{right} = 2i + 1 \)
        Let \( j \) be the index that minimizes \( \text{key}[H[\text{left}]] \) and \( \text{key}[H[\text{right}]] \)
    Else if \( 2i = n \) then
        Let \( j = 2i \)
    Endif
    If \( \text{key}[H[j]] < \text{key}[H[i]] \) 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/~jmarcaist
► Email: jmarcaist@uwlax.edu
  ► Expect replies within a (business) day (but often faster) during the semester
► Office hours through spring break
  ► Tuesdays 1:30-3:30pm
  ► Wednesdays 4-5pm
  ► Fridays 2-3:30pm
► Or by appointment
  ► 15-minutes slots, email at least a day ahead
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 $v_1, v_2, ..., v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k-1$ nodes are all distinct.

![Graph showing cycle C = 1-2-4-5-3-1]
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.
Rooted Trees

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

**Importance.** Models hierarchical structure.
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.** An 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 Diagram]  

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
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.**
- $L_0 = \{ s \}$.
- $L_1 = \text{all neighbors of } L_0$.
- $L_2 = \text{all nodes that do not belong to } L_0 \text{ or } L_1, \text{ and that have an edge to a node in } L_1$.
- $L_{i+1} = \text{all nodes that do not belong to an earlier layer, and that have an edge to a node in } L_i$.

**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 $\deg(u)$ incident edges $(u, v)$
  - total time processing edges is $\sum_{u \in V} \deg(u) = 2m$

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

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.

recolor lime green blob to blue
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.

recolor lime green blob to blue
**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 \( v \not\in R \) 
    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.
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.
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 $G$. 
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)

Case (ii)
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.
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. □
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.

\[ \text{ok if paths overlap} \]
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.

![Diagram](image-url)

- 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 \(v_j\).
Directed Acyclic Graphs

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.

\[ \begin{array}{c}
\text{w} \\
\text{x} \\
\text{u} \\
\text{v} \\
\end{array} \]

\[ \begin{array}{c}
\text{w} \\
\text{v} \\
\end{array} \]

43
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 $\text{count}[w]$ hits 0
  - this is $O(1)$ per edge
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 faster) during the semester
▶ Office hours through spring break
  ▶ Tuesdays 1:30-3:30pm
  ▶ Wednesdays 4-5pm
  ▶ Fridays 2-3:30pm
▶ Or by appointment
  ▶ 15-minutes slots, email at least a day ahead
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

- 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 \( p \)?
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 $G = (V, E)$
  - 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

- 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
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 $v_1$
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: $G = (\{v_1, \cdots, v_n\}, \{e_1, \cdots, e_{n-1}\})$ 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
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
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
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
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.
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. An \( n \)-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.
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.
Algorithms — February 9

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 faster) during the semester
▶ Office hours through spring break
  ▶ Tuesdays 1:30-3:30pm
  ▶ Wednesdays 4-5pm
  ▶ Fridays 2-3:30pm
▶ Or by appointment
  ▶ 15-minutes slots, email at least a day ahead
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.
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.

![Graph Diagram]

degree = number of neighbors of $u$
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 $\sum_{u \in V} \deg(u) = 2m$  
    each edge $(u, v)$ is counted exactly twice in sum: once in $\deg(u)$ and once in $\deg(v)$
**Connected Component**

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

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.

recolor lime green blob to blue
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 $s$.

$R$ 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 $v \not\in R$
    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 \textit{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)
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 (2-colorable) 
not bipartite (not 2-colorable)
Lemma. Let $G$ be a connected graph, and let $L_0, ..., L_k$ 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).
Obstruction to Bipartiteness

**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.

\[\text{ok if paths overlap}\]
**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.**

---

**Diagram**: strongly connected vs. 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 \(v_1, v_2, \ldots, v_n\) 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 faster) during the semester
▶ Office hours through spring break
  ▶ Tuesdays 1:30-3:30pm
  ▶ Wednesdays 4-5pm
  ▶ Fridays 2-3:30pm
▶ Or by appointment
  ▶ 15-minutes slots, email at least a day ahead
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 enqueueing 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*
Directed Acyclic Graphs

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

Pf. (by contradiction)

- Suppose that G has a topological order $v_1, ..., v_n$ 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 $v_1, ..., v_n$ is a topological order, we must have $j < i$, a contradiction.

$v_1 \rightarrow v_i \rightarrow \ldots \rightarrow v_n$ the supposed topological order: $v_1, ..., v_n$

the directed cycle C
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.
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 $v$
Topological Sorting Algorithm: Running Time

**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 $c_j$. 
Interval Scheduling: Greedy Algorithms

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
**Interval Scheduling: Greedy Algorithm**

**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 ≤ f_2 ≤ ... ≤ f_n.

set of jobs selected
A ← φ
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 ≥ f_j*.
**Theorem.** Greedy algorithm is optimal.

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

![Diagram of Greedy vs OPT solutions with job $i_{r+1}$ finishing before job $j_{r+1}$, and question mark indicating why not replace job $j_{r+1}$ with job $i_{r+1}$?]}
Interval Scheduling: Analysis

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: \[ i_1 \quad i_2 \quad i_r \quad i_{r+1} \]

OPT: \[ j_1 \quad j_2 \quad j_r \quad i_{r+1} \]

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 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
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 faster) during the semester
▶ Office hours through spring break
  ▶ Tuesdays 1:30-3:30pm
  ▶ Wednesdays 4-5pm
  ▶ Fridays 2-3:30pm
▶ Or by appointment
  ▶ 15-minutes slots, email at least a day ahead
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

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.

![Diagram showing the schedule with lectures a through j at different times using 3 classrooms.](image-url)
Interval Partitioning: Lower Bound on Optimal Solution

**Def.** The depth of a set of open intervals is the maximum number that contain any given time.

**Key observation.** Number of classrooms needed $\geq$ depth.

**Ex:** Depth of schedule below = 3 $\Rightarrow$ schedule below is optimal.

\begin{align*}
9:30 &\quad 9:30 \\
10 &\quad 10:30 \\
11 &\quad 11:30 \\
12 &\quad 12:30 \\
1:30 &\quad 1:30 \\
2:30 &\quad 2:30 \\
3:30 &\quad 3:30 \\
4:30 &\quad 4:30 \\
\end{align*}

$a, b, c$ all contain 9:30

**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 \( s_1 \leq s_2 \leq \ldots \leq s_n \).
\[
\begin{align*}
\text{d} &\leftarrow 0 & \text{number of allocated classrooms} \\
\text{for } j = 1 \text{ to } n \{ \\
\quad \text{if (lecture } j \text{ is compatible with some classroom } k) \\
\quad \quad \text{schedule lecture } j \text{ in classroom } k \\
\quad \text{else} \\
\quad \quad \text{allocate a new classroom } d + 1 \\
\quad \quad \text{schedule lecture } j \text{ in classroom } d + 1 \\
\quad \text{d} &\leftarrow d + 1 \\
\} 
\end{align*}
\]

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 $s_j + \varepsilon$.
- 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>

- $d_3 = 9$, $d_2 = 8$, $d_6 = 15$, $d_1 = 6$, $d_5 = 14$, $d_4 = 9$
- Lateness = 2
- Lateness = 0
- Max lateness = 6
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></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<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></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>$d_j$</td>
<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
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 $i$.

![Diagram showing an inversion between jobs $j$ and $i$]

[as before, we assume jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$]

**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.
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 $i$.

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:

$$
\begin{align*}
\ell'_j &= f'_j - d_j & \text{(definition)} \\
&= f_i - d_j & \text{($j$ finishes at time $f_i$)} \\
&\leq f_i - d_i & \text{($i < j$)} \\
&\leq \ell_i & \text{(definition)}
\end{align*}
$$
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 faster) during the semester
- Office hours through spring break
  - Tuesdays 1:30-3:30pm
  - Wednesdays 4-5pm
  - Fridays 2-3:30pm
- Or by appointment
  - 15-minutes slots, email at least a day ahead
  - Office hours through spring break
Shortest Path Problem

**Shortest path network.**
- Directed graph \( G = (V, E) \).
- Source \( s \), destination \( t \).
- Length \( \ell_e \) = 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.

- 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 $d(v) = \pi(v)$.

\[ S \]

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 \( d(v) = \pi(v) \).
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 faster) during the semester
▶ Office hours through spring break
  ▶ Tuesdays 1:30-3:30pm
  ▶ Wednesdays 4-5pm
  ▶ Fridays 2-3:30pm
▶ Or by appointment
  ▶ 15-minutes slots, email at least a day ahead
▶ Expect replies within a (business) day (but often faster) during the semester
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 $S$.

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

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

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

- Next node to explore = node with minimum \( \pi(v) \).
- When exploring \( v \), for each incident edge \( e = (v, w) \), update

\[
\pi(w) = \min \{ \pi(w), \pi(v) + l_e \}.
\]

**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>Total</td>
<td>( n^2 )</td>
<td>( m \log n )</td>
<td>( m \log \frac{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.

$G = (V, E)$

$T$, $\sum_{e \in T} c_e = 50$

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 $f$. 

---

![Diagram](image.png)

- $e$ 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.

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

**Diagram:**

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

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.

\[\square\]
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. $\blacksquare$
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 $S$. 

Prim's Algorithm: Proof of Correctness
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 \in V) a[v] \leftarrow \infty
    Initialize an empty priority queue $Q$
    foreach (v \in V) insert $v$ onto $Q$
    Initialize set of explored nodes $S \leftarrow \phi$

    while ($Q$ is not empty) {
        $u \leftarrow$ delete min element from $Q$
        $S \leftarrow S \cup \{ u \}$
        foreach (edge $e = (u, v)$ incident to $u$)
            if ((v \notin 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: 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.

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

$m \leq n^2 \Rightarrow \log m$ 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 faster) during the semester
- Office hours through spring break
  - Tuesdays 1:30-3:30pm
  - Wednesdays 4-5pm
  - Fridays 2-3:30pm
- Or by appointment
  - 15-minutes slots, email at least a day ahead
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 $f$. 

![Diagram showing Cut property and Cycle property](image-url)
Cycles and Cuts

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

```
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
```

**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 $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- This is a contradiction. $\blacksquare$
**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 $S$.
  \[ \Rightarrow \text{there exists another edge, say } e, \text{ that is in both } C \text{ and } D. \]
- $T' = T^* \cup \{ e \} - \{ f \}$ 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 $S = \text{any node}$.
- 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](image-url)
Implementation: Prim’s Algorithm

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

- foreach $(v \in V)$ $a[v] \leftarrow \infty$
- Initialize an empty priority queue $Q$
- foreach $(v \in V)$ insert $v$ onto $Q$
- Initialize set of explored nodes $S \leftarrow \emptyset$
- Pick starting node $s$
- Add $s$ to $S$
- foreach (edge $e = (s, u)$ incident to $s$)
  - Add $u$ to $Q$ with priority $c_u$
- while ($Q$ is not empty) {
  - $u \leftarrow$ delete min element from $Q$
  - $S \leftarrow S \cup \{ u \}$
  - foreach (edge $e = (u, v)$ incident to $u$)
    - if ($v \notin S$ and ($c_e < a[v]$))
      - decrease priority $a[v]$ to $c_e$
- $O(n^2)$ with an array;
- $O(m \log n)$ with a binary heap.

Implementation. Use a priority queue à la Dijkstra
- Maintain set of explored nodes $S$.
- For each unexplored node $v$, maintain attachment cost $a[v] =$ cost of cheapest edge $v$ to a node in $S$.
- $O(n^2)$ with an array;
- $O(m \log n)$ with a binary heap.
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 faster) during the semester
▶ Office hours through spring break
  ▶ Tuesdays 1:30-3:30pm
  ▶ Wednesdays 4-5pm
  ▶ Fridays 2-3:30pm
▶ Or by appointment
  ▶ 15-minutes slots, email at least a day ahead
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.
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
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
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 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.
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.

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.

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

- 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 faster) during the semester
- Office hours through spring break
  - Tuesdays 1:30-3:30pm
  - Wednesdays 4-5pm
  - Fridays 2-3:30pm
- Or by appointment
  - 15-minutes slots, email at least a day ahead
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: $n \log n$.
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)

\[
\begin{array}{cccccccc}
A & L & G & O & R & I & T & H & M & S \\
A & L & G & O & R & & I & T & H & M & S \\
A & G & L & O & R & & H & I & M & S & T \\
A & G & H & I & L & M & O & R & S & T \\
\end{array}
\]

\[
\begin{align*}
\text{divide} & \quad O(1) \\
\text{sort} & \quad 2T(n/2) \\
\text{merge} & \quad O(n) 
\end{align*}
\]
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.** $T(n) =$ number of comparisons to mergesort an input of size $n$.

**Mergesort recurrence.**

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

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

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

\(\text{sorting both halves}\)
\(\text{merging}\)

\(\uparrow\) assumes $n$ is a power of 2

**Pf.** For $n > 1$:

\[
\frac{T(n)}{n} = \frac{2T(n/2)}{n} + 1
\]

\[
= \frac{T(n/2)}{n/2} + 1
\]

\[
= \frac{T(n/4)}{n/4} + 1 + 1
\]

\[\vdots\]

\[
= \frac{T(n/n)}{n/n} + 1 + \cdots + 1
\]

\[
= \log_2 n
\]
Proof by Induction

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

**Pf.** (by induction on $n$)

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

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

assumes $n$ is a power of 2

\[
T(2n) = 2T(n) + 2n \\
= 2n \log_2 n + 2n \\
= 2n(\log_2 (2n) - 1) + 2n \\
= 2n \log_2 (2n)
\]
Analysis of Mergesort Recurrence

**Claim.** If $T(n)$ satisfies the following recurrence, then $T(n) \leq n \lceil \lg n \rceil$.

$$T(n) \leq \begin{cases} 0 & \text{if } n = 1 \\ \frac{T(\lceil n/2 \rceil)}{\text{solve left half}} + \frac{T(\lfloor n/2 \rfloor)}{\text{solve right half}} + \frac{n}{\text{merging}} & \text{otherwise} \end{cases}$$

**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 $1, 2, \ldots, n-1$.

$$T(n) \leq T(n_1) + T(n_2) + n$$
$$\leq n_1 \lceil \lg n_1 \rceil + n_2 \lfloor \lg n_2 \rfloor + n$$
$$\leq n_1 \lceil \lg n_2 \rceil + n_2 \lfloor \lg n_2 \rfloor + n$$
$$= n \lfloor \lg n_2 \rfloor + n$$
$$\leq n(\lceil \lg n \rceil - 1) + n$$
$$= n \lceil \lg n \rceil$$

$$n_2 = \lfloor n/2 \rfloor$$
$$\leq 2^\lceil \lg n \rceil / 2$$
$$= 2^\lfloor \lg n \rfloor / 2$$
$$\Rightarrow \lg n_2 \leq \lfloor \lg n \rfloor - 1$$
5.3 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, ..., n.
- Your rank: \( a_1, a_2, ..., a_n \).
- Songs \( i \) and \( j \) inverted if \( i < j \), but \( a_i > a_j \).

<table>
<thead>
<tr>
<th>Songs</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Me</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>You</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

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.

```
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
```
Counting Inversions: Divide-and-Conquer

Divide-and-conquer.

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

Divide: \( O(1) \).

Conquer: \( 2T(n / 2) \)

<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

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

8 green-green inversions

5-4, 5-2, 4-2, 8-2, 10-2
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

Divide: \( O(1) \).

Conquer: \( 2T(n/2) \)

Combine: ???

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.

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

Count: $O(n)$

2 3 7 10 11 14 16 17 18 19 23 25

Merge: $O(n)$

$T(n) \leq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n) \implies T(n) = O(n \log n)$
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.

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.
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. \(\text{\textbullet\ rule}\) 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 < \( \delta \).
Closest Pair of Points

Find closest pair with one point in each side, assuming that distance < δ.

- Observation: only need to consider points within δ of line L.

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

Find closest pair with one point in each side, assuming that distance < δ.
- Observation: only need to consider points within δ of line L.
- Sort points in 2δ-strip by their y coordinate.

δ = 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

\[
\text{Closest-Pair}(p_1, \ldots, p_n) \{ \\
\text{Compute separation line } L \text{ such that half the points are on one side and half on the other side.} \\
\delta_1 = \text{Closest-Pair(left half)} \\
\delta_2 = \text{Closest-Pair(right half)} \\
\delta = \min(\delta_1, \delta_2) \\
\text{Delete all points further than } \delta \text{ from separation line } L \\
\text{Sort remaining points by } y\text{-coordinate.} \\
\text{Scan points in } y\text{-order and compare distance between each point and next 11 neighbors. If any of these distances is less than } \delta, \text{ update } \delta. \\
\text{return } \delta.
\}
\]
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.

\[ T(n) \leq 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n) \]