Algorithms — January 24

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

Today
- Some problems
- Introduction

Next time
- Stable matching

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

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
- Expect replies within a business day (but often same-day)
Chapter 1

Introduction:
Some Representative Problems
Matching Residents to Hospitals

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

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

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

Goal. Given n men and n women, find a "suitable" matching.
- Participants rate members of opposite sex.
- Each man lists women in order of preference from best to worst.
- Each woman lists men in order of preference from best to worst.

**Men's Preference Profile**

<table>
<thead>
<tr>
<th>favorite</th>
<th>least favorite</th>
</tr>
</thead>
<tbody>
<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>

**Women's Preference Profile**

<table>
<thead>
<tr>
<th>favorite</th>
<th>least favorite</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>2nd</td>
</tr>
<tr>
<td>Amy</td>
<td>Yancey</td>
</tr>
<tr>
<td>Bertha</td>
<td>Xavier</td>
</tr>
<tr>
<td>Clare</td>
<td>Xavier</td>
</tr>
</tbody>
</table>
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?

Men's Preference Profile

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

Women's Preference Profile

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

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

<table>
<thead>
<tr>
<th>Favorite</th>
<th>Least Favorite</th>
</tr>
</thead>
<tbody>
<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>

**Men’s Preference Profile**

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

**Women’s Preference Profile**
**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>
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>

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

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

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

- First an *algorithm*
  - Takes rankings tables
  - Returns a set of pairs
- *Proof* that the algorithm finds a (perfect) matching
- Proof that the matching is stable
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.
Bipartite Matching

**Input.** Bipartite graph.

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

**Input.** Graph.

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

subset of nodes such that no two joined by an edge
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.

```
10  1  5  15  5  1  5  1  15  10
```

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

Variations on a theme: independent set.

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

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

Today
Asymptotic measure
Sec. 2.1-2.4

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

Wednesday
Priority queues and heaps
Read Sec. 2.5

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

Contact information

- Web: cs.uwlax.edu/~jmarais
- Email: jmarais@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
How do we quantify efficiency for programs?

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

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

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

- By the clock time that passes during execution?
- CPU time?
- Statements?
  - Machine code statements?
- Key operations
  - Could be comparisons, assignments, etc.
Brute force. For many non-trivial problems, there is a natural brute force search algorithm that checks every possible solution.

- Typically takes $2^N$ time or worse for inputs of size $N$.
- Unacceptable in practice.

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

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

Def. An algorithm is poly-time if the above scaling property holds.
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.
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>$n = 10$</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>$n = 30$</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>$n = 50$</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>$n = 100$</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>$n = 1,000$</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>$n = 10,000$</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>$n = 100,000$</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>$n = 1,000,000$</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 \( \Theta(n^3) \).
Notation

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

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

We are discussing the runtime of *specific algorithms*

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

We are discussing the runtime of *specific algorithms*

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

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

Additivitiy.
- 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
}
```
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.

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.

```plaintext
min ← $(x_1 - x_2)^2 + (y_1 - y_2)^2$
for $i = 1$ to $n$
  for $j = i + 1$ to $n$
    $d ← (x_i - x_j)^2 + (y_i - y_j)^2$
    if ($d < min$)
      $min ← d$

don't need to take square roots
```

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

**Cubic time.** Enumerate all triples of elements.

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

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

```plaintext
foreach set $S_i$ {
    foreach other set $S_j$ {
        foreach element $p$ of $S_i$ {
            determine whether $p$ also belongs to $S_j$
        }
        if (no element of $S_i$ belongs to $S_j$)
            report that $S_i$ and $S_j$ are disjoint
    }
}
```
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.

```latex
\begin{align*}
\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} \\
\}
\end{align*}
```

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

poly-time for $k=17$, but not practical
Exponential Time

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

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

S* ← φ
foreach subset S of nodes {
    check whether S in an independent set
    if (S is largest independent set seen so far)
        update S* ← S
}

Algorithms — January 29

Today

Events

- CS120 tutoring, 231 Wing
  - Tonight, tomorrow, 6-9pm
  - Sunday, 3-6pm
  - Monday 4:45-7:45pm
- CODERS, community service via CS
  - Last regular meeting, Wednesday May 2nd, 3:15pm, 231 Wing
  - Last weekly study group, Thursday May 3rd, 4-6pm, 16 Wing
  - Exam Study Day, Monday May 7th, 12:15-2:15pm, Centennial 3214
- CS Club, student ACM chapter
  - Womyn’s group last meeting, Tuesday May 1st, 4pm, 3214 Centennial

Wednesday
Priority queues and heaps (Sec. 2.5)

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

Monday
Graphs

Contact information

- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
  - Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
  - Because advising or paperwork often require no meeting
How to do the problems

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

(In this or any advanced concepts course)

Do them all

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

Still: the best practice is to do them all

➢ This isn’t your only class
➢ And even if it is, that amount of homework may still not be feasible
➢ Not necessarily each of you, individually, alone, separately
➢ Group work is great here
➢ Especially in this size of class
➢ Come see me when you are blocked
How to do the problems

(In this or any advanced concepts course)

Do them all

► In an decent text, the authors develop skills across the problem set
► They’re walking you through a story
► In a perfect world, my grading staff would go over each problem as you do them

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

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

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

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

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

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

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

Exponentials. For every \(r > 1\) and every \(d > 0\), \(n^d = O(r^n)\).
So let’s turn back to stable matching
Propose-And-Reject Algorithm


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

\textbf{Pf.} Each time through the while loop a man proposes to a new woman. There are only \(n^2\) possible proposals.

\[n(n-1) + 1\] proposals required
Proof of Correctness: Perfection

Claim. All men and women get matched.
Pf. (by contradiction)
  • Suppose, for sake of contradiction, that Zeus is not matched upon termination of algorithm.
  • Then some woman, say Amy, is not matched upon termination.
  • By Observation 2, Amy was never proposed to.
  • But, Zeus proposes to everyone, since he ends up unmatched. □
Proof of Correctness: Stability

Claim. No unstable pairs.

Pf. (by contradiction)

- Suppose A-Z is an unstable pair: each prefers each other to partner in Gale-Shapley matching $S^*$.

  - Case 1: Z never proposed to A.
    - $\Rightarrow$ Z prefers his GS partner to A.
    - $\Rightarrow$ A-Z is stable.

  - Case 2: Z proposed to A.
    - $\Rightarrow$ A rejected Z (right away or later)
    - $\Rightarrow$ A prefers her GS partner to Z.
    - $\Rightarrow$ A-Z is stable.

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

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

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

Q. How to implement GS algorithm efficiently?

Q. If there are multiple stable matchings, which one does GS find?
Efficient Implementation

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 wife[m], and husband[w].
  - set entry to 0 if unmatched
  - if m matched to w then wife[m]=w and husband[w]=m

Men proposing.
- For each man, maintain a list of women, ordered by preference.
- Maintain an array 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.

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

<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>

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

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

**An instance with two stable matchings.**

- A-X, B-Y, C-Z.
- A-Y, B-X, C-Z.

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

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>Y</td>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>Bertha</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>Clare</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>
Understanding the Solution

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

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

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

\[ \text{no man and woman prefer to be with each other than assigned partner} \]

**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.

\[ \text{w is a valid partner of m if there exist some stable matching where m and w are paired} \]

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

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

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

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

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

Potentially deep social ramifications. [legal disclaimer]
Today
Priority queues and heaps (Sec. 2.5)

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

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting
Designing priority queues

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

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

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

A tree implemented as an array
- The parent of node $n$ is $n/2$ (rounding down)
- The children of node $n$ are nodes $2n$ and $2n+1$
- We are assuming a fixed maximum size
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.
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 $\text{key}[H[i]] < \text{key}[H[j]]$ then
         swap the array entries $H[i]$ and $H[j]$
         Heapify-up(H,j)
      Endif
   Endif
Endif
The Heapify-down process is moving element $w$ down, toward the leaves.

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

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

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

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

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

- Insertion of a new element
  - Add it to the end, then heapify-up, then down
  - $O(\log n)$
- Finding the minimum element
  - Always at the top — constant
- Deletion of an arbitrary (possibly the minimum) element
  - Move element at maximum position to point to be deleted, then heapify-up, then down
  - $O(\log n)$
Algorithms — February 5

Today
Graphs (Ch. 3)

Wednesday
More graphs

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

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
Undirected graph. $G = (V, E)$
- $V$ = nodes.
- $E$ = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|, m = |E|$.

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

Web graph.
- Node: web page.
- Edge: hyperlink from one page to another.
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 $v$. 

![Diagram of a graph with labeled nodes and edges.](image)
**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.

\[
\text{cycle } C = 1-2-4-5-3-1
\]
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.

```
a tree  the same tree, rooted at 1
```

```
root $r$
parent of $v$
child of $v$
```
Phylogeny Trees

Phylogeny trees. Describe evolutionary history of species.

- gut bacteria
- trees
- mushrooms
- fish
- mammals
- birds
- dragonflies
- beetles
GUI Containment Hierarchy

GUI containment hierarchy. Describe organization of GUI widgets.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  $\uparrow$

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

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

![Diagram showing a connected component with nodes 1, 2, 3, 4, 5, 6, 7, 8.]

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

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

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

![Diagram](image-url)
Flood Fill

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

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

recolor lime green blob to blue
Connected Component

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

---

R will consist of nodes to which s has a path
Initially R = {s}
While there is an edge (u, v) where u ∈ R and v ∉ 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.
More formal version of the definition:

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

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

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

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

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

![another drawing of $G$](image2)
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)**
**Bipartite Graphs**

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

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.

(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

**Pf.** (i)

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

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

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

Pf. (ii)

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

Def. A graph is \textit{strongly connected} if every pair of nodes is mutually reachable.

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

Pf. ⇒ Follows from definition.
Pf. ⇐ Path from u to v: concatenate u-s path with s-v path.
    Path from v to u: concatenate v-s path with s-u path. ▶
    ok if paths overlap
Strong Connectivity: Algorithm

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

**Pf.**
- Pick any node $s$.
- Run BFS from $s$ in $G$.
- Run BFS from $s$ in $G^{rev}$.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. □
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 \(v_j\).
Lemma. If $G$ has a topological order, then $G$ is a DAG.

Pf. (by contradiction)

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

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

Q. Does every DAG have a topological ordering?

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

Pf. (by contradiction)
- Suppose that $G$ is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$ we can walk backward to $u$.
- Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$.
- Repeat until we visit a node, say $w$, twice.
- Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle.
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:
  - $\text{count}[w] = \text{remaining number of incoming edges}$
  - $S = \text{set of remaining nodes with no incoming edges}$
- 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
Today
Graphs (Ch. 3)
▶ You should find the collected slides online now

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

Contact information
▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
    ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
    ▶ Wednesday 2: 3-5pm
    ▶ Friday 4: 2-4pm
    ▶ Monday 7: 10am-noon
▶ Or by appointment
    ▶ Email at least a day ahead
    ▶ Send several times when you are available
    ▶ Always describe what you need to discuss
        ▶ So I can be prepared
        ▶ Because advising or paperwork often require no meeting
Trees

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

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

- $G$ is connected.
- $G$ does not contain a cycle.
- $G$ has $n-1$ edges.
Proof (part 1)
Given: $G$ has $n$ nodes, is connected and acyclic
Prove: $G$ has $n - 1$ edges
Proof (part 1)
Given: $G$ has $n$ nodes, is connected and acyclic
Prove: $G$ has $n - 1$ edges

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

Assume the contrary

Assume the contrary

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

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

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

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

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

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

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

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

▶ Consider the graph $G' = (V \setminus \{v\}, E \setminus \{e\})$
▶ $G'$ is still connected, still acyclic, and has $n - 1$ nodes
▶ So by the induction hypothesis, $G'$ has $n - 2$ edges
▶ $G$ has one more edge than $G'$, so $n - 1$ edges — QED
Proof (part 1)
Given: $G$ has $n$ nodes, is connected and acyclic
Prove: $G$ has $n - 1$ edges

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

Main result: by induction on the number $n$ of nodes
   ▶ Case $n = 2$: construct directly
   ▶ For $n > 2$: by the lemma, there is some vertex $v$ attached only to edge $e$ in $G = (V, E)$
     ▶ Consider the graph $G' = (V \setminus \{v\}, E \setminus \{e\})$
       ▶ $G'$ 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

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

$G$ 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 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.
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
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

Therefore, $G$ 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
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
Proof (part 3)

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

By induction on the number $n$ of nodes

- Base case $n = 3$, observing the construction directly
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
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 same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
Breadth First Search

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

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

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

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

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

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

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

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

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

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

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

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 \( s \).

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

**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

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

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

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

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

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

![Diagram of bipartite and non-bipartite graphs](image-url)
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).
Corollary. A graph $G$ is bipartite iff it contain no odd length cycle.
Directed Graphs

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

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

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

Graph Search

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

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

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

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

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

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

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

**Pf.** ⇒ Follows from definition.

**Pf.** ⇐ Path from u to v: concatenate u-s path with s-v path.
Path from v to u: concatenate v-s path with s-u path. ∎

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

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

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

reverse orientation of every edge in $G$

![Diagram of strongly connected and not strongly connected graphs](image-url)
Directed Acyclic Graphs

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

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

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

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

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

Today
▶ Graphs, Greedy algorithms

Wednesday
▶ Greedy algorithms

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

Contact information
▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting
Exercise 3.2

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

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

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

▶ In a proof we can assert the existence of things, maybe with maximum/minimum or other bounds/properties
▶ In an algorithm, we must give the steps to produce such artifacts (or refer to previous algorithms and their cost)
▶ When modifying an algorithm we have already analyzed, we can just give the difference
  ▶ But we must be mindful of non-constant-time changes — they may change the overall cost of the algorithm
  ▶ For example, if we add an inner loop
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*. 
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?
Lemma. If $G$ is a DAG, then $G$ has a node with no incoming edges.

Pf. (by contradiction)

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

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

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

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

To compute a topological ordering of $G$:

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

Pf.

- Maintain the following information:
  - $\text{count}[w] =$ remaining number of incoming edges
  - $S =$ set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete $v$
  - remove $v$ from $S$
  - decrement $\text{count}[w]$ for all edges from $v$ to $w$, and add $w$ to $S$ if $\text{count}[w]$ 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.
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 \( f_1 \leq f_2 \leq \ldots \leq f_n \).
- **Set of jobs selected**
  
  \[
  A \leftarrow \emptyset \\
  \text{for } j = 1 \text{ to } n \{ \\
  \quad \text{if (job } j \text{ compatible with } A) \\
  \quad \quad A \leftarrow A \cup \{j\} \\
  \} \\
  \text{return } A
  \]

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

Pf. (by contradiction)
- Assume greedy is not optimal, and let's see what happens.
- Let $i_1, i_2, ... i_k$ 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 $r$.

Greedy: $i_1, i_2, ..., i_r, i_{r+1}$

OPT: $j_1, j_2, ..., j_r, \hat{j}_{r+1}$
Theorem. Greedy algorithm is optimal.

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

Greedy: $i_1 \quad i_2 \quad i_r \quad i_{r+1}$

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/~jmarais
▶ Email: jmarais@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting
Is greed good?

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

Interval partitioning.

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

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

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.
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.

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

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

```
Sort intervals by starting time so that $s_1 \leq s_2 \leq \ldots \leq s_n$.
\[ d \leftarrow 0 \quad \text{number of allocated classrooms} \]

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

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

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

**Theorem.** Greedy algorithm is optimal.

**Pf.**
- Let \( d \) = number of classrooms that the greedy algorithm allocates.
- Classroom \( d \) is opened because we needed to schedule a job, say \( j \), that is incompatible with all \( d-1 \) other classrooms.
- These \( d \) jobs each end after \( s_j \).
- Since we sorted by start time, all these incompatibilities are caused by lectures that start no later than \( s_j \).
- Thus, we have \( d \) lectures overlapping at time \( s_j + \varepsilon \).
- 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: \( l_j = \max \{ 0, f_j - d_j \} \).
- Goal: schedule all jobs to minimize maximum lateness \( L = \max l_j \).

Ex:

<table>
<thead>
<tr>
<th>( t_j )</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>4</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<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>

- \( d_3 = 9 \) \quad \text{lateness} = 2
- \( d_2 = 8 \) \quad \text{lateness} = 0
- \( d_6 = 15 \) \quad \text{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>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
</tr>
<tr>
<td>$d_j$</td>
<td>100</td>
</tr>
</tbody>
</table>

  counterexample

- [Smallest slack] Consider jobs in ascending order of slack $d_j - t_j$.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_j$</td>
<td>1</td>
</tr>
<tr>
<td>$d_j$</td>
<td>2</td>
</tr>
</tbody>
</table>

  counterexample
Minimizing Lateness: Greedy Algorithm

**Greedy algorithm.** Earliest deadline first.

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

\[
\begin{align*}
t &\leftarrow 0 \\
\text{for } j = 1 \text{ to } n \\
&\quad \text{Assign job } j \text{ to interval } [t, t + t_j] \\
&\quad s_j \leftarrow t, \ f_j \leftarrow t + t_j \\
&\quad t \leftarrow t + t_j \\
\text{output} \ intervals \ [s_j, f_j]
\end{align*}
\]

\[
\begin{array}{cccccccc}
  d_1 &=& 6 & d_2 &=& 8 & d_3 &=& 9 & d_4 &=& 9 \\
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\end{array}
\]

max lateness = 1
Minimizing Lateness: No Idle Time

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

![Diagram](d=4, d=6, d=12)

**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 two jobs $j$ and $i$ with an inversion between them.]

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.

[as before, we assume jobs are numbered so that $d_1 \leq d_2 \leq \ldots \leq d_n$]
**Def.** Given a schedule $S$, an inversion is a pair of jobs $i$ and $j$ such that: $i < j$ but $j$ scheduled before $i$.

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

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

- $\ell'_k = \ell_k$ for all $k \neq i, j$
- $\ell'_i \leq \ell_i$
- If job $j$ is late:

\[
\begin{align*}
\ell'_j &= f'_j - d_j \quad \text{(definition)} \\
&= f_i - d_j \quad \text{($j$ finishes at time $f_i$)} \\
&\leq f_i - d_i \quad \text{($i < j$)} \\
&\leq \ell_i \quad \text{(definition)}
\end{align*}
\]
Minimizing Lateness: Analysis of Greedy Algorithm

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

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

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


Greedy Analysis Strategies

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

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

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

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

Today

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

Monday

▶ Greedy algorithms

Wednesday

▶ Greedy algorithms
▶ Divide-and-conquer algorithms

Next Friday

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

Contact information

▶ Web: cs.uwlax.edu/~jmarais
▶ Email: jmarais@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
▶ Always describe what you need to discuss
  ▶ So I can be prepared
  ▶ Because advising or paperwork often require no meeting
Shortest path network.

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

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

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

\[
= 9 + 23 + 2 + 16 \\
= 50.
\]
Dijkstra's Algorithm

Dijkstra's algorithm.
- Maintain a set of explored nodes \( S \) for which we have determined the shortest path distance \( d(u) \) from \( s \) to \( u \).
- Initialize \( S = \{ s \} \), \( d(s) = 0 \).
- Repeatedly choose unexplored node \( v \) which minimizes 

\[
\pi(v) = \min_{e = (u,v): u \in S} d(u) + \ell_e ,
\]

add \( v \) to \( S \), and set \( d(v) = \pi(v) \).
Dijkstra's Algorithm

Dijkstra's algorithm.
- Maintain a set of explored nodes $S$ for which we have determined the shortest path distance $d(u)$ from $s$ to $u$.
- Initialize $S = \{s\}$, $d(s) = 0$.
- Repeatedly choose unexplored node $v$ which minimizes

$$
\pi(v) = \min_{e = (u,v): u \in S} d(u) + \ell_e,
$$

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

shortest path to some $u$ in explored part, followed by a single edge $(u, v)$
Today

▶ Greedy algorithms

Wednesday

▶ Greedy algorithms

Friday

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

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
   ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
   ▶ Wednesday 2: 3-5pm
   ▶ Friday 4: 2-4pm
   ▶ Monday 7: 10am-noon
▶ Or by appointment
   ▶ Email at least a day ahead
   ▶ Send several times when you are available
   ▶ Always describe what you need to discuss
      ▶ So I can be prepared
      ▶ Because advising or paperwork often require no meeting
Dijkstra's Algorithm: Proof of Correctness

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

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

**Base case:** \( |S| = 1 \) is trivial.

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

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

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

---

\( \ell \) (length) \( \pi \) (estimate)

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

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

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

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

<table>
<thead>
<tr>
<th>PQ Operation</th>
<th>Dijkstra</th>
<th>Array</th>
<th>Binary heap</th>
<th>d-way Heap</th>
<th>Fib heap†</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_{m/n} n )</td>
<td>( m + n \log n )</td>
<td></td>
</tr>
</tbody>
</table>

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

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

MST is fundamental problem with diverse applications.

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

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

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

- Cluster analysis.
Greedy Algorithms

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

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

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

**Remark.** All three algorithms produce an MST.
Greedy Algorithms

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

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

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

![Diagram showing e is in the MST](example)  
![Diagram showing f is not in the MST](example)
Cycles and Cuts

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

![Graph with labeled edges](image)

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

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

![Graph with labeled edges](image)

Cut $S = \{4, 5, 8\}$

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

**Pf.** (by picture)
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.  \[\square\]
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 $c_e < c_f$, cost($T'$) < cost($T^*$).
- This is a contradiction.\[\blacksquare\]
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 $S$. 
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.

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

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

Kruskal's algorithm. [Kruskal, 1956]

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

Case 1

Case 2
Implementation: Kruskal's Algorithm

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

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

essentially a constant

Kruskal($G$, $c$) {
  \textbf{Sort} edges weights so that $c_1 \leq c_2 \leq \ldots \leq c_m$.
  \[T \leftarrow \emptyset\]
  \textbf{foreach} ($u \in V$) make a set containing singleton $u$
  \textbf{for} $i = 1$ to $m$
    \textbf{if} ($u$ and $v$ are in different connected components) {
      \textbf{if} ($u$, $v$) = $e_i$
      \textbf{if} ($u$ and $v$ are in different sets) {
        $T \leftarrow T \cup \{e_i\}$
        merge the sets containing $u$ and $v$
      }
    }
  \textbf{return} $T$
}

merge two components
Lexicographic Tiebreaking

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

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

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

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

e.g., if all edge costs are integers, perturbing cost of edge e_i by i / n^2
Algorithms — February 21

Today, Friday

▶ Greedy algorithms

Monday

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

Next Friday

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

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)

▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon

▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting

▶ Open book, open notes, closed internet
▶ This one will be individual

End-of-semester office hours

▶ Wednesday 2: 3-5pm
▶ Friday 4: 2-4pm
▶ Monday 7: 10am-noon

▶ Or by appointment
▶ Email at least a day ahead
▶ Send several times when you are available
▶ Always describe what you need to discuss
  ▶ So I can be prepared
  ▶ Because advising or paperwork often require no meeting
Greedy Algorithms

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

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

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

![Diagram](image)

- $e$ 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.

[Diagram of a cycle with nodes labeled 1 to 8 and edges connecting them.]

**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 of a cutset with nodes labeled 1 to 8 and edges connecting them, showing a cutset D.]
Cycle-Cut Intersection

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

Pf. (by picture)
Greedy Algorithms

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

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

Pf. (exchange argument)
- Suppose $e$ does not belong to $T^*$, and let’s see what happens.
- Adding $e$ to $T^*$ creates a cycle $C$ in $T^*$.
- Edge $e$ is both in the cycle $C$ and in the cutset $D$ corresponding to $S$.
  $\Rightarrow$ there exists another edge, say $f$, that is in both $C$ and $D$.
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- 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 $S$ ⇒ there exists another edge, say $e$, that is in both $C$ and $D$.
- $T' = T^* \cup \{e\} - \{f\}$ is also a spanning tree.
- Since $c_e < c_f$, cost($T'$) < 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. [Jarník 1930, Dijkstra 1957, Prim 1959]

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

Implementation. Use a priority queue à la Dijkstra

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

- Greedy algorithms

Monday

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

Next Friday

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

Contact information

- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
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: 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](image1)

![Case 2](image2)
Union-Find

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

Three operations

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

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

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

Three operations

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

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

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

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

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

Simplest implementation is by an array from node index to name
- But $O(n)$ for Union
Union-Find

Three operations

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

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

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

Simplest implementation is by an array from node index to name

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

Three operations

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

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

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

Simplest implementation is by an array from node index to name

- But \( O(n) \) for Union

Can be improved by

- Using the name of the larger set in a union
- Tracking the size of each connected component

But we still need a better structure
The set \{s, u, w\} was merged into \{t, v, z\}.

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

Three operations

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

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

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

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

\[
\begin{align*}
m & \leq n^2 \quad \Rightarrow \quad \log m \text{ is } O(\log n) \\
\text{essentially a constant}
\end{align*}
\]

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 \( (u \text{ and } v \text{ are in different sets}) \) {
            \( T \leftarrow T \cup \{e_i\} \)
            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
}
```
Algorithms — February 26

Today

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

Wednesday

▶ Divide-and-conquer algorithms

Friday

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

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting
Divide-and-Conquer

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

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

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

Divide et impera.
Veni, vidi, vici.
- Julius Caesar
5.1 Mergesort
Mergesort

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

Jon von Neumann (1945)
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.**

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

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

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ \frac{2T(n/2)}{n} + \frac{n}{\log_2 n} & \text{otherwise} \end{cases}$$

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

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

Pf. (by induction on $n$)

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

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

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

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

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

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

\[
T(n) \leq T(n_1) + T(n_2) + n \\
\leq n_1 \lfloor \lg n_1 \rfloor + n_2 \lfloor \lg n_2 \rfloor + n \\
\leq n_1 \lfloor \lg n_2 \rfloor + n_2 \lfloor \lg n_2 \rfloor + n \\
= n \lfloor \lg n_2 \rfloor + n \\
\leq n( \lfloor \lg n \rfloor - 1 ) + n \\
= n \lfloor \lg n \rfloor
\]

\[
n_2 = \lceil n/2 \rceil \\
\leq 2^\lfloor \lg n \rfloor / 2 \\
= 2^\lfloor \lg n \rfloor / 2 \\
\Rightarrow \lg n_2 \leq \lfloor \lg n \rfloor - 1
\]
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.

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.

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

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

Divide-and-conquer.

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

<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

Conquer: $2T(n/2)$

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

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

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

5 blue-blue inversions
8 green-green inversions

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

Total = 5 + 8 + 9 = 22.

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

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

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

Count: $O(n)$

Merge: $O(n)$

$T(n) \leq T\left(\left\lceil n/2 \right\rceil \right) + T\left(\left\lfloor n/2 \right\rfloor \right) + O(n) \Rightarrow T(n) = O(n \log n)$
Counting Inversions: Implementation

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

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

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

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

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

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

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

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

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

---

\[ 
\text{fast closest pair inspired fast algorithms for these problems}
\]
Closest Pair of Points: First Attempt

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

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

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

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

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

**Algorithm.**

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

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

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

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

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

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

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

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

Find closest pair with one point in each side, assuming that distance < \( \delta \).
- Observation: only need to consider points within \( \delta \) of line \( L \).
- Sort points in \( 2\delta \)-strip by their y coordinate.
- Only check distances of those within 11 positions in sorted list!

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

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

Pf.

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

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

Closest-Pair($p_1, \ldots, p_n$) {

Compute separation line $L$ such that half the points are on one side and half on the other side.

$\delta_1 = \text{Closest-Pair(left half)}$
$\delta_2 = \text{Closest-Pair(right half)}$
$\delta = \min(\delta_1, \delta_2)$

Delete all points further than $\delta$ from separation line $L$

Sort remaining points by y-coordinate.

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

return $\delta$.

}
Closest Pair of Points: Analysis

Running time.

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

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.

\[ T(n) \leq 2T(n/2) + O(n) \implies T(n) = O(n \log n) \]
Algorithms — February 26

Today
- Divide-and-conquer algorithms

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

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

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
Recurrence relations roundup

Last time we saw:

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

Last time we saw:

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

Some other useful recurrence relations:

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

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

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

With reference translations from recurrence relations to asymptotic bounds, we can quickly analyze new applications for divide-and-conquer
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: a₁, a₂, ..., aₙ.
- Songs i and j inverted if i < j, but aᵢ > aⱼ.

Brute force: check all \( \Theta(n^2) \) pairs i and 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>

Inversions: 3-2, 4-2
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.

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

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

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

### Divide: \( O(1) \)

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

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

- **5 blue-blue inversions**
- **8 green-green inversions**

- 5-4, 5-2, 4-2, 8-2, 10-2
- 6-3, 9-3, 9-7, 12-3, 12-7, 12-11, 11-3, 11-7
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.

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

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

5 blue-blue inversions
8 green-green inversions

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

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

**Combine:** count blue-green inversions

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

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

Count: \( O(n) \)

Merge: \( O(n) \)

\[
T(n) \leq T\left(\left\lfloor n/2 \right\rfloor \right) + T\left(\left\lfloor n/2 \right\rfloor \right) + O(n) \quad \Rightarrow \quad T(n) = O(n \log n)
\]
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.
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. → seems like $\Theta(n^2)$
- Return best of 3 solutions.
Closest Pair of Points

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

\[ \delta = \min(12, 21) \]
Find closest pair with one point in each side, assuming that distance < $\delta$.
- Observation: only need to consider points within $\delta$ of line $L$. 

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

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

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

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

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

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

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

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

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

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

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

   \( \delta_1 = \text{Closest-Pair(left half)} \)
   \( \delta_2 = \text{Closest-Pair(right half)} \)
   \( \delta = \min(\delta_1, \delta_2) \)

   Delete all points further than \( \delta \) from separation line L

   Sort remaining points by y-coordinate.

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

   return \( \delta \).
}

\[ O(n \log n) \]
\[ 2T(n / 2) \]
\[ O(n) \]
\[ O(n \log n) \]
\[ O(n) \]
Closest Pair of Points: Analysis

Running time.

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.

\[
T(n) \leq 2T(n/2) + O(n \log n) \quad \Rightarrow \quad T(n) = O(n \log^2 n)
\]
Algorithms — March 2

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

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

Wednesday

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
How to Multiply
integers, matrices, and polynomials
Complex Multiplication

Complex multiplication. \((a + bi) (c + di) = x + yi.\)

Grade-school. \(x = ac - bd, \ y = bc + ad.\)

Q. Is it possible to do with fewer multiplications?
Complex Multiplication

Complex multiplication. \((a + bi) (c + di) = x + yi.\)

Grade-school. \(x = ac - bd, \; y = bc + ad.\)

4 multiplications, 2 additions

Q. Is it possible to do with fewer multiplications?
A. Yes. \([Gauss]\) \(x = ac - bd, \; y = (a + b) (c + d) - ac - bd.\)

3 multiplications, 5 additions

Remark. Improvement if no hardware multiply.
5.5 Integer Multiplication
### Integer Addition

**Addition.** Given two $n$-bit integers $a$ and $b$, compute $a + b$.

**Grade-school.** $\Theta(n)$ bit operations.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Remark.** Grade-school addition algorithm is optimal.
Integer Multiplication

**Multiplication.** Given two $n$-bit integers $a$ and $b$, compute $a \times b$.

**Grade-school.** $\Theta(n^2)$ bit operations.

```
  1 1 0 1 0 1 0 1
× 0 1 1 1 1 1 0 1
  1 1 0 1 0 1 0 1
+ 0 0 0 0 0 0 0 0
  1 1 0 1 0 1 0 1
+ 1 1 0 1 0 1 0 1
  1 1 0 1 0 1 0 1
+ 0 0 0 0 0 0 0 0
  1 1 0 1 0 1 0 1
+ 1 1 0 1 0 1 0 1
  1 1 0 1 0 1 0 1
+ 1 1 0 1 0 1 0 1
  1 1 0 1 0 1 0 1
+ 1 1 0 1 0 1 0 1
  1 1 0 1 0 1 0 1
+ 0 0 0 0 0 0 0 0
+-------------------
  0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1
```

**Q.** Is grade-school multiplication algorithm optimal?
Divide-and-Conquer Multiplication: Warmup

To multiply two \( n \)-bit integers \( a \) and \( b \):
- Multiply four \( \frac{1}{2}n \)-bit integers, recursively.
- Add and shift to obtain result.

\[
\begin{align*}
  a & = 2^{n/2} \cdot a_1 + a_0 \\
  b & = 2^{n/2} \cdot b_1 + b_0 \\
  a b & = \left(2^{n/2} \cdot a_1 + a_0\right) \left(2^{n/2} \cdot b_1 + b_0\right) = 2^n \cdot a_1 b_1 + 2^{n/2} \cdot (a_1 b_0 + a_0 b_1) + a_0 b_0
\end{align*}
\]

Ex. \( a = 10001101 \) \( \underbrace{1000}_{a_1} \underbrace{1101}_{a_0} \) \( \underbrace{1110}_{b_1} \underbrace{0001}_{b_0} \)

\[
T(n) = 4T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n^2)
\]
Karatsuba Multiplication

To multiply two $n$-bit integers $a$ and $b$:

- Add two $\frac{1}{2}n$ bit integers.
- Multiply three $\frac{1}{2}n$-bit integers, recursively.
- Add, subtract, and shift to obtain result.

\[
\begin{align*}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= 2^n \cdot a_1 b_1 + 2^{n/2} \cdot (a_1 b_0 + a_0 b_1) + a_0 b_0 \\
\end{align*}
\]

1. \[2^{n/2} \cdot a_1 b_1\]
2. \[2^{n/2} \cdot (a_1 b_0 + a_0 b_1)\]
3. \[a_0 b_0\]
Karatsuba Multiplication

To multiply two $n$-bit integers $a$ and $b$:
- Add two $\frac{1}{2}n$ bit integers.
- Multiply three $\frac{1}{2}n$-bit integers, recursively.
- Add, subtract, and shift to obtain result.

$$
\begin{align*}
a &= 2^{n/2} \cdot a_1 + a_0 \\
b &= 2^{n/2} \cdot b_1 + b_0 \\
ab &= 2^n \cdot a_1 b_1 + 2^{n/2} \cdot (a_1 b_0 + a_0 b_1) + a_0 b_0 \\
&= 2^n \cdot a_1 b_1 + 2^{n/2} \cdot ((a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0) + a_0 b_0
\end{align*}
$$

**Theorem.** [Karatsuba-Ofman 1962] Can multiply two $n$-bit integers in $O(n^{1.585})$ bit operations.

$$
T(n) \leq T\left(\lfloor n/2 \rfloor \right) + T\left(\lceil n/2 \rceil \right) + T\left(1 + \lceil n/2 \rceil \right) + \Theta(n) \quad \Rightarrow \quad T(n) = O(n^{\lg 3}) = O(n^{1.585})
$$

recursive calls

add, subtract, shift
Matrix Multiplication
Dot Product

**Dot product.** Given two length \( n \) vectors \( a \) and \( b \), compute \( c = a \cdot b \).

**Grade-school.** \( \Theta(n) \) arithmetic operations.

\[
a \cdot b = \sum_{i=1}^{n} a_i b_i
\]

\[
a = \begin{bmatrix} .70 & .20 & .10 \end{bmatrix} \\
b = \begin{bmatrix} .30 & .40 & .30 \end{bmatrix} \\
a \cdot b = (.70 \times .30) + (.20 \times .40) + (.10 \times .30) = .32
\]

**Remark.** Grade-school dot product algorithm is optimal.
Matrix Multiplication

**Matrix multiplication.** Given two $n$-by-$n$ matrices $A$ and $B$, compute $C = AB$.

**Grade-school.** $\Theta(n^3)$ arithmetic operations.

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

$$\begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{bmatrix}$$

$$\begin{bmatrix} .59 & .32 & .41 \\ .31 & .36 & .25 \\ .45 & .31 & .42 \end{bmatrix} = \begin{bmatrix} .70 & .20 & .10 \\ .30 & .60 & .10 \\ .50 & .10 & .40 \end{bmatrix} \times \begin{bmatrix} .80 & .30 & .50 \\ .10 & .40 & .10 \\ .10 & .30 & .40 \end{bmatrix}$$

**Q.** Is grade-school matrix multiplication algorithm optimal?
Block Matrix Multiplication

\[
C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} = \begin{bmatrix} 0 & 1 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 16 & 17 \\ 20 & 21 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix} \times \begin{bmatrix} 24 & 25 \\ 28 & 29 \end{bmatrix} = \begin{bmatrix} 152 & 158 \\ 504 & 526 \end{bmatrix}
\]
Matrix Multiplication: Warmup

To multiply two $n$-by-$n$ matrices $A$ and $B$:

- **Divide:** partition $A$ and $B$ into $\frac{1}{2}n$-by-$\frac{1}{2}n$ blocks.
- **Conquer:** multiply 8 pairs of $\frac{1}{2}n$-by-$\frac{1}{2}n$ matrices, recursively.
- **Combine:** add appropriate products using 4 matrix additions.

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix} = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix} \times \begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

\[
\begin{align*}
C_{11} & = (A_{11} \times B_{11}) + (A_{12} \times B_{21}) \\
C_{12} & = (A_{11} \times B_{12}) + (A_{12} \times B_{22}) \\
C_{21} & = (A_{21} \times B_{11}) + (A_{22} \times B_{21}) \\
C_{22} & = (A_{21} \times B_{12}) + (A_{22} \times B_{22})
\end{align*}
\]

\[
T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^3)
\]
Fast Matrix Multiplication

**Key idea.** multiply 2-by-2 blocks with only 7 multiplications.

\[
\begin{bmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{bmatrix}
= \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\times
\begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

- \[P_1 = A_{11} \times (B_{12} - B_{22})\]
- \[P_2 = (A_{11} + A_{12}) \times B_{22}\]
- \[P_3 = (A_{21} + A_{22}) \times B_{11}\]
- \[P_4 = A_{22} \times (B_{21} - B_{11})\]
- \[P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22})\]
- \[P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22})\]
- \[P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12})\]

- 7 multiplications.
- 18 = 8 + 10 additions and subtractions.
Fast Matrix Multiplication

To multiply two \( n \)-by-\( n \) matrices \( A \) and \( B \): [Strassen 1969]

- Divide: partition \( A \) and \( B \) into \( \frac{1}{2}n \)-by-\( \frac{1}{2}n \) blocks.
- Compute: 14 \( \frac{1}{2}n \)-by-\( \frac{1}{2}n \) matrices via 10 matrix additions.
- Conquer: multiply 7 pairs of \( \frac{1}{2}n \)-by-\( \frac{1}{2}n \) matrices, recursively.
- Combine: 7 products into 4 terms using 8 matrix additions.

Analysis.

- Assume \( n \) is a power of 2.
- \( T(n) = \# \) arithmetic operations.

\[
T(n) = 7T(n/2) + \Theta(n^2) \quad \Rightarrow \quad T(n) = \Theta(n^{\log_2 7}) = O(n^{2.81})
\]
Fast Matrix Multiplication: Practice

Implementation issues.
- Sparsity.
- Caching effects.
- Numerical stability.
- Odd matrix dimensions.
- Crossover to classical algorithm around $n = 128$.

Common misperception. "Strassen is only a theoretical curiosity."
- Apple reports 8x speedup on G4 Velocity Engine when $n \approx 2,500$.
- Range of instances where it's useful is a subject of controversy.

Remark. Can "Strassenize" $Ax = b$, determinant, eigenvalues, SVD, ....
Fast Matrix Multiplication: Theory

Q. Multiply two 2-by-2 matrices with 7 scalar multiplications?
A. Yes! [Strassen 1969]
\[ \Theta(n \log_2 7) = O(n^{2.807}) \]

Q. Multiply two 2-by-2 matrices with 6 scalar multiplications?
A. Impossible. [Hopcroft and Kerr 1971]
\[ \Theta(n \log_2 6) = O(n^{2.59}) \]

Q. Two 3-by-3 matrices with 21 scalar multiplications?
A. Also impossible.
\[ \Theta(n \log_3 21) = O(n^{2.77}) \]

Begun, the decimal wars have. [Pan, Bini et al, Schönhage, ...]
- Two 20-by-20 matrices with 4,460 scalar multiplications.
  \[ O(n^{2.805}) \]
- Two 48-by-48 matrices with 47,217 scalar multiplications.
  \[ O(n^{2.7801}) \]
- A year later.
Fast Matrix Multiplication: Theory

Best known. $O(n^{2.376})$ [Coppersmith-Winograd, 1987]

Conjecture. $O(n^{2+\varepsilon})$ for any $\varepsilon > 0$.

Caveat. Theoretical improvements to Strassen are progressively less practical.
Fast Fourier transforms

- Another divide-and-conquer application
- More mathematical than we’ll get into
- Applies to multiplication of polynomials, and then back to integers and matrices
Integer Multiplication, Redux

Integer multiplication. Given two $n$ bit integers $a = a_{n-1} \ldots a_1a_0$ and $b = b_{n-1} \ldots b_1b_0$, compute their product $a \cdot b$.

"the fastest bignum library on the planet"

Practice. [GNU Multiple Precision Arithmetic Library]
It uses brute force, Karatsuba, and FFT, depending on the size of $n$. 
Integer Arithmetic

Fundamental open question. What is complexity of arithmetic?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Upper Bound</th>
<th>Lower Bound</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>$O(n)$</td>
<td>$\Omega(n)$</td>
<td></td>
</tr>
<tr>
<td>multiplication</td>
<td>$O(n \log n \ 2^{O(\log^* n)})$</td>
<td>$\Omega(n)$</td>
<td></td>
</tr>
<tr>
<td>division</td>
<td>$O(n \log n \ 2^{O(\log^* n)})$</td>
<td>$\Omega(n)$</td>
<td></td>
</tr>
</tbody>
</table>
Factoring

Factoring. Given an $n$-bit integer, find its prime factorization.

$$2773 = 47 \times 59$$

$$2^{67} - 1 = 147573952589676412927 = 193707721 \times 761838257287$$

A disproof of Mersenne’s conjecture that $2^{67} - 1$ is prime

RSA-704
($30,000$ prize if you can factor)
Factoring and RSA

**Primality.** Given an \( n \)-bit integer, is it prime?

**Factoring.** Given an \( n \)-bit integer, find its prime factorization.

**Significance.** Efficient primality testing \( \Rightarrow \) can implement RSA.

**Significance.** Efficient factoring \( \Rightarrow \) can break RSA.

**Theorem.** [AKS 2002] Poly-time algorithm for primality testing.
Shor's Algorithm

Shor's algorithm. Can factor an $n$-bit integer in $O(n^3)$ time on a quantum computer.

Ramification. At least one of the following is wrong:
- RSA is secure.
- Textbook quantum mechanics.
- Extending Church-Turing thesis.
Algorithms — March 5

Today
- Problems due: 4.2, 4.8, 4.9, 5.1

This week
- Dynamic programming
- No quiz

Next week
- Spring break

Monday, March 19
- We’re back
- Problems due: 5.5, 5.6

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting

Expect replies within a business day (but often same-day)
Chapter 6
Dynamic Programming
Algorithmic Paradigms

**Greedy.** Build up a solution incrementally, myopically optimizing some local criterion.

**Divide-and-conquer.** Break up a problem into sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

**Dynamic programming.** Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
Dynamic Programming History

Bellman. [1950s] Pioneered the systematic study of dynamic programming.

Etymology.

- Dynamic programming = planning over time.
- Secretary of Defense was hostile to mathematical research.
- Bellman sought an impressive name to avoid confrontation.

"it's impossible to use dynamic in a pejorative sense"
"something not even a Congressman could object to"

Dynamic Programming Applications

Areas.

- Bioinformatics.
- Control theory.
- Information theory.
- Operations research.
- Computer science: theory, graphics, AI, compilers, systems, ....

Some famous dynamic programming algorithms.

- Unix diff for comparing two files.
- Viterbi for hidden Markov models.
- Smith-Waterman for genetic sequence alignment.
- Bellman-Ford for shortest path routing in networks.
- Cocke-Kasami-Younger for parsing context free grammars.
Illustrative overkill — Fibonacci

When we learn recursion, Fibonacci is usually one of the examples of how *not* to recur
Illustrative overkill — Fibonacci

When we learn recursion, Fibonacci is usually one of the examples of how not to recur

➤ Doing it wrong:

```java
public static long fib(int n) {
    switch (n) {
    case 0: return 0;
    case 1: return 1;
    default: return fib(n-1) + fib(n-2);
    }
}
```

requires $O(2^n)$ recursive calls
Illustrative overkill — Fibonacci

When we learn recursion, Fibonacci is usually one of the examples of how not to recur

- **Doing it wrong:**
  ```java
  public static long fib(int n) {
    switch (n) {
      case 0: return 0;
      case 1: return 1;
      default: return fib(n-1) + fib(n-2);
    }
  }
  ```
  requires $O(2^n)$ recursive calls

- This approach fails because the subproblems *overlap*
  - *fib*(n-1) is going to need *fib*(n-2)
  - We fail to take advantage of this structure
A dynamic solution to Fibonacci

Cache the results of the subproblems
  ▶ Then we can use each more than once

```java
public static long fib(int n) {
    final long[] fibs = new long[1+n];
    fibs[0] = 0;
    fibs[1] = 1;
    for(int i=2; i<=n; i++) {
        fibs[n] = fibs[n-1] + fibs[n-2];
    }
    return fibs[n];
}
```
A dynamic solution to Fibonacci

Cache the results of the subproblems

- Then we can use each more than once

```java
public static long fib(int n) {
    final long[] fibs = new long[1+n];
    fibs[0] = 0;
    fibs[1] = 1;
    for(int i=2; i<=n; i++) {
        fibs[i] = fibs[i-1] + fibs[i-2];
    }
    return fibs[n];
}
```

- The smarter solution which we usually learn takes further advantage of the fact that the overlap is limited
  - We do not need to keep more than two members of the sequence at a time to make progress
  - So instead of an array, just track "this" element and "prev" (or "next") element
Weighted interval scheduling problem.

- Job $j$ starts at $s_j$, finishes at $f_j$, and has weight or value $v_j$.
- Two jobs compatible if they don't overlap.
- Goal: find maximum weight subset of mutually compatible jobs.
Recall. Greedy algorithm works if all weights are 1.
- Consider jobs in ascending order of finish time.
- Add job to subset if it is compatible with previously chosen jobs.

Observation. Greedy algorithm can fail spectacularly if arbitrary weights are allowed.
Weighted Interval Scheduling

**Notation.** Label jobs by finishing time: $f_1 \leq f_2 \leq \ldots \leq f_n$.

**Def.** $p(j) =$ largest index $i < j$ such that job $i$ is compatible with $j$.

**Ex:** $p(8) = 5$, $p(7) = 3$, $p(2) = 0$. 

![Graphical representation of the schedule with shaded bars indicating job intervals and time points labeled 0 to 11.]
Dynamic Programming: Binary Choice

**Notation.** \( OPT(j) = \) value of optimal solution to the problem consisting of job requests 1, 2, ..., j.

- **Case 1:** \( OPT \) selects job j.
  - collect profit \( v_j \)
  - can’t use incompatible jobs \{ p(j) + 1, p(j) + 2, ..., j - 1 \}
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., p(j)

- **Case 2:** \( OPT \) does not select job j.
  - must include optimal solution to problem consisting of remaining compatible jobs 1, 2, ..., j-1

\[
OPT(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ v_j + OPT(p(j)), \ OPT(j-1) \} & \text{otherwise}
\end{cases}
\]
**Weighted Interval Scheduling: Brute Force**

**Brute force algorithm.**

**Input:** \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

**Sort** jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

**Compute** \( p(1), p(2), \ldots, p(n) \)

**Compute-Opt(j)** {

\[
\text{if} \ (j = 0) \\
\text{return} \ 0 \\
\text{else} \\
\quad \text{return} \ \max (v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1))
\]

}
Weighted Interval Scheduling: Brute Force

**Observation.** Recursive algorithm fails spectacularly because of redundant sub-problems $\Rightarrow$ exponential algorithms.

**Ex.** Number of recursive calls for family of "layered" instances grows like Fibonacci sequence.

\[ p(1) = 0, \ p(j) = j-2 \]
Memoization. Store results of each sub-problem in a cache; lookup as needed.

**Input:** $n$, $s_1, \ldots, s_n$, $f_1, \ldots, f_n$, $v_1, \ldots, v_n$

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

**Compute** $p(1)$, $p(2)$, $\ldots$, $p(n)$

```
for j = 1 to n
    M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
    if (M[j] is empty)
        M[j] = max($v_j + M$-Compute-Opt($p(j)$), M-Compute-Opt($j-1$))
    return M[j]
}
```
Weighted Interval Scheduling: Running Time

**Claim.** Memoized version of algorithm takes $O(n \log n)$ time.
- Sort by finish time: $O(n \log n)$.
- Computing $p(\cdot)$: $O(n \log n)$ via sorting by start time.

- **M-Compute-Opt**$(j)$: each invocation takes $O(1)$ time and either
  - (i) returns an existing value $M[j]$
  - (ii) fills in one new entry $M[j]$ and makes two recursive calls

- Progress measure $\Phi = \#\text{ nonempty entries of } M[\cdot]$.
  - initially $\Phi = 0$, throughout $\Phi \leq n$.
  - (ii) increases $\Phi$ by 1 $\Rightarrow$ at most $2n$ recursive calls.

- Overall running time of $\text{M-Compute-Opt}(n)$ is $O(n)$. □

**Remark.** $O(n)$ if jobs are pre-sorted by start and finish times.
Weighted Interval Scheduling: Finding a Solution

Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?
A. Do some post-processing.

Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
    if (j = 0)
        output nothing
    else if (v_j + M[p(j)] > M[j-1])
        print j
        Find-Solution(p(j))
    else
        Find-Solution(j-1)
}

- # of recursive calls \leq n \Rightarrow O(n).
Weighted Interval Scheduling: Bottom-Up

**Bottom-up dynamic programming.** Unwind recursion.

**Input:** \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

**Sort** jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

**Compute** \( p(1), p(2), \ldots, p(n) \)

**Iterative-Compute-Opt** {  
  \[ M[0] = 0 \]
  \[ \text{for } j = 1 \text{ to } n \]
  \[ M[j] = \max(v_j + M[p(j)], M[j-1]) \]
}
6.4 Knapsack Problem
Knapsack Problem

Knapsack problem.
- Given \( n \) objects and a "knapsack."
- Item \( i \) weighs \( w_i > 0 \) kilograms and has value \( v_i > 0 \).
- Knapsack has capacity of \( W \) kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: \{ 3, 4 \} has value 40.

<table>
<thead>
<tr>
<th>#</th>
<th>value</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

\( W = 11 \)

Greedy: repeatedly add item with maximum ratio \( v_i / w_i \).
Ex: \{ 5, 2, 1 \} achieves only value = 35 \( \Rightarrow \) greedy not optimal.
Dynamic Programming: False Start

**Def.** $\text{OPT}(i) = \text{max profit subset of items } 1, \ldots, i.$

- **Case 1:** $\text{OPT}$ does not select item $i$.
  - $\text{OPT}$ selects best of $\{1, 2, \ldots, i-1\}$

- **Case 2:** $\text{OPT}$ selects item $i$.
  - Accepting item $i$ does not immediately imply that we will have to reject other items
  - Without knowing what other items were selected before $i$, we don't even know if we have enough room for $i$

**Conclusion.** Need more sub-problems!
Dynamic Programming: Adding a New Variable

**Def.** $OPT(i, w) = \text{max profit subset of items 1, \ldots, i with weight limit w.}$

- **Case 1:** $OPT$ does not select item i.
  - $OPT$ selects best of $\{1, 2, \ldots, i-1\}$ using weight limit $w$

- **Case 2:** $OPT$ selects item i.
  - new weight limit $= w - w_i$
  - $OPT$ selects best of $\{1, 2, \ldots, i-1\}$ using this new weight limit

$$OPT(i, w) = \begin{cases} 
  0 & \text{if } i = 0 \\
  OPT(i-1, w) & \text{if } w_i > w \\
  \max\{OPT(i-1, w), v_i + OPT(i-1, w-w_i)\} & \text{otherwise}
\end{cases}$$
Algorithms — March 7

Tonight
- 3-Minute Thesis competition, 1400 Centennial Hall, 6pm

This week
- Dynamic programming
- No quiz

Next week
- Spring break
- Lab closed Sat.-Sat., reopens Sunday regular hours

Monday, March 19
- We’re back
- Problems due: 5.5, 5.6

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
Dynamic Programming: Adding a New Variable

**Def.** $OPT(i, w) = \text{max profit subset of items } 1, ..., i \text{ with weight limit } w$.

- **Case 1:** $OPT$ does not select item $i$.
  - $OPT$ selects best of $\{1, 2, ..., i-1\}$ using weight limit $w$

- **Case 2:** $OPT$ selects item $i$.
  - new weight limit $= w - w_i$
  - $OPT$ selects best of $\{1, 2, ..., i-1\}$ using this new weight limit

\[
OPT(i, w) = \begin{cases} 
0 & \text{if } i = 0 \\
OPT(i-1, w) & \text{if } w_i > w \\
\max \{ OPT(i-1, w), \ w_i + OPT(i-1, w-w_i) \} & \text{otherwise}
\end{cases}
\]
Knapsack Problem: Bottom-Up

**Knapsack.** Fill up an n-by-W array.

**Input:** n, W, w₁,…,wₙ, v₁,…,vₙ

```plaintext
for w = 0 to W
    M[0, w] = 0

for i = 1 to n
    for w = 1 to W
        if (wᵢ > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w], vᵢ + M[i-1, w-wᵢ]}

return M[n, W]
```
Knapsack Algorithm

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>22</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>7</td>
</tr>
</tbody>
</table>

\[
\text{OPT: } \{ 4, 3 \} \\
\text{value } = 22 + 18 = 40
\]
Knapsack Problem: Running Time

Running time. $\Theta(n W)$.
- Not polynomial in input size!
- "Pseudo-polynomial."
- Decision version of Knapsack is NP-complete. [Chapter 8]

Knapsack approximation algorithm. There exists a poly-time algorithm that produces a feasible solution that has value within 0.01% of optimum. [Section 11.8]
This week
  ▶ Dynamic programming
  ▶ No quiz

Next week
  ▶ Spring break
  ▶ Lab closed Sat.-Sat., reopens Sunday regular hours

Monday, March 19
  ▶ We’re back
  ▶ Problems due: 5.5, 5.6

Friday, March 23
  ▶ Quiz on divide-and-conquer

Contact information
  ▶ Web: cs.uwlax.edu/~jmaraist
  ▶ Email: jmaraist@uwlax.edu
    ▶ Expect replies within a business day (but often same-day)
  ▶ End-of-semester office hours
    ▶ Wednesday 2: 3-5pm
    ▶ Friday 4: 2-4pm
    ▶ Monday 7: 10am-noon
  ▶ Or by appointment
    ▶ Email at least a day ahead
    ▶ Send several times when you are available
    ▶ Always describe what you need to discuss
      ▶ So I can be prepared
      ▶ Because advising or paperwork often require no meeting
6.3 Segmented Least Squares
Segmented Least Squares

Least squares.
- Foundational problem in statistic and numerical analysis.
- Given n points in the plane: \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\).
- Find a line \(y = ax + b\) that minimizes the sum of the squared error:

\[
SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2
\]

Solution. Calculus \(\Rightarrow\) min error is achieved when

\[
a = \frac{n \sum_i x_i y_i - (\sum_i x_i)(\sum_i y_i)}{n \sum_i x_i^2 - (\sum_i x_i)^2}, \quad b = \frac{\sum_i y_i - a \sum_i x_i}{n}
\]
Segmented Least Squares

Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given $n$ points in the plane $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ with
  - $x_1 < x_2 < \ldots < x_n$, find a sequence of lines that minimizes $f(x)$.

Q. What's a reasonable choice for $f(x)$ to balance accuracy and parsimony?
Segmented least squares.

- Points lie roughly on a sequence of several line segments.
- Given \( n \) points in the plane \((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\) with
  - \( x_1 < x_2 < \ldots < x_n \), find a sequence of lines that minimizes:
    - the sum of the sums of the squared errors \( E \) in each segment
    - the number of lines \( L \)
- Tradeoff function: \( E + cL \), for some constant \( c > 0 \).
Dynamic Programming: Multiway Choice

Notation.
- $\text{OPT}(j) =$ minimum cost for points $p_1, p_{i+1}, \ldots, p_j$.
- $e(i, j) =$ minimum sum of squares for points $p_i, p_{i+1}, \ldots, p_j$.

To compute $\text{OPT}(j)$:
- Last segment uses points $p_i, p_{i+1}, \ldots, p_j$ for some $i$.
- $\text{Cost} = e(i, j) + c + \text{OPT}(i-1)$.

\[
\begin{align*}
\text{OPT}(j) &= \begin{cases} 
0 & \text{if } j = 0 \\
\min_{1 \leq i \leq j} \{ e(i, j) + c + \text{OPT}(i-1) \} & \text{otherwise}
\end{cases}
\end{align*}
\]
Segmented Least Squares: Algorithm

INPUT: n, p₁,…,pₙ , c

Segmented-Least-Squares() {
    M[0] = 0
    for j = 1 to n
        for i = 1 to j
            compute the least square error eᵢⱼ for the segment pᵢ,…, pⱼ

    for j = 1 to n
        M[j] = min₁≤i≤j (eᵢⱼ + c + M[i-1])

    return M[n]
}

Running time. \( O(n^3) \). can be improved to \( O(n^2) \) by pre-computing various statistics

- Bottleneck = computing \( e(i, j) \) for \( O(n^2) \) pairs, \( O(n) \) per pair using previous formula.
6.5 RNA Secondary Structure
RNA Secondary Structure

**RNA.** String $B = b_1 b_2 \ldots b_n$ over alphabet $\{ A, C, G, U \}$.

**Secondary structure.** RNA is single-stranded so it tends to loop back and form base pairs with itself. This structure is essential for understanding behavior of molecule.

Ex: GUCGAUUGAGCGAAUGUAACAACGUGGCUCAGGCCGAGA

complementary base pairs: A-U, C-G
RNA Secondary Structure

Secondary structure. A set of pairs $S = \{(b_i, b_j)\}$ that satisfy:

- [Watson-Crick.] $S$ is a matching and each pair in $S$ is a Watson-Crick complement: $A-U$, $U-A$, $C-G$, or $G-C$.
- [No sharp turns.] The ends of each pair are separated by at least 4 intervening bases. If $(b_i, b_j) \in S$, then $i < j - 4$.
- [Non-crossing.] If $(b_i, b_j)$ and $(b_k, b_l)$ are two pairs in $S$, then we cannot have $i < k < j < l$.

Free energy. Usual hypothesis is that an RNA molecule will form the secondary structure with the optimum total free energy.

Goal. Given an RNA molecule $B = b_1b_2...b_n$, find a secondary structure $S$ that maximizes the number of base pairs.
RNA Secondary Structure: Examples

Examples.

- Base pair
- Sharp turn
- Crossing
First attempt. \( \text{OPT}(j) = \) maximum number of base pairs in a secondary structure of the substring \( b_1b_2...b_j \).

**Difficulty.** Results in two sub-problems.

- Finding secondary structure in: \( b_1b_2...b_{t-1} \). ← \( \text{OPT}(t-1) \)
- Finding secondary structure in: \( b_{t+1}b_{t+2}...b_{n-1} \). ← need more sub-problems

\[ \text{match } b_t \text{ and } b_n \]
Dynamic Programming Over Intervals

Notation. \( OPT(i, j) = \) maximum number of base pairs in a secondary structure of the substring \( b_i b_{i+1} \ldots b_j \).

- **Case 1.** If \( i \geq j - 4 \).
  - \( OPT(i, j) = 0 \) by no-sharp turns condition.

- **Case 2.** Base \( b_j \) is not involved in a pair.
  - \( OPT(i, j) = OPT(i, j-1) \)

- **Case 3.** Base \( b_j \) pairs with \( b_t \) for some \( i \leq t < j - 4 \).
  - non-crossing constraint decouples resulting sub-problems
  - \( OPT(i, j) = 1 + \max_t \{ OPT(i, t-1) + OPT(t+1, j-1) \} \)
  \( \text{take max over } t \text{ such that } i \leq t < j - 4 \) and \( b_t \) and \( b_j \) are Watson-Crick complements

Remark. Same core idea in CKY algorithm to parse context-free grammars.
**Bottom Up Dynamic Programming Over Intervals**

**Q.** What order to solve the sub-problems?
**A.** Do shortest intervals first.

```c
RNA(b_1, ..., b_n) {
    for k = 5, 6, ..., n-1
        for i = 1, 2, ..., n-k
            j = i + k
            Compute M[i, j]
        return M[i, j]
    return M[1, n]
}
```

**Running time.** $O(n^3)$.
Dynamic Programming Summary

Recipe.
- Characterize structure of problem.
- Recursively define value of optimal solution.
- Compute value of optimal solution.
- Construct optimal solution from computed information.

Dynamic programming techniques.
- Binary choice: weighted interval scheduling.
- Multi-way choice: segmented least squares.
- Adding a new variable: knapsack.
- Dynamic programming over intervals: RNA secondary structure.

Top-down vs. bottom-up: different people have different intuitions.
6.6 Sequence Alignment
String Similarity

How similar are two strings?

- occurrence
- occurrence

6 mismatches, 1 gap

1 mismatch, 1 gap

0 mismatches, 3 gaps
Applications.

- Basis for Unix diff.
- Speech recognition.
- Computational biology.


- Gap penalty $\delta$; mismatch penalty $\alpha_{pq}$.
- Cost = sum of gap and mismatch penalties.

\[ \alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA} \]

\[ 2\delta + \alpha_{CA} \]
Sequence Alignment

**Goal:** Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ find alignment of minimum cost.

**Def.** An **alignment** $M$ is a set of ordered pairs $x_i$-$y_j$ such that each item occurs in at most one pair and no crossings.

**Def.** The pair $x_i$-$y_j$ and $x_i'$-$y_j'$ **cross** if $i < i'$, but $j > j'$.

**Ex:** $CTACCG$ vs. $TACATG$.

**Sol:** $M = x_2$-$y_1$, $x_3$-$y_2$, $x_4$-$y_3$, $x_5$-$y_4$, $x_6$-$y_6$.

$$
\text{cost}(M) = \sum_{(x_i, y_j) \in M} \alpha_{x_i y_j} + \sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta
$$
Sequence Alignment: Problem Structure

**Def.** $OPT(i, j) = \text{min cost of aligning strings } x_1 x_2 \ldots x_i \text{ and } y_1 y_2 \ldots y_j$.

- **Case 1:** $OPT$ matches $x_i$-$y_j$.
  - pay mismatch for $x_i$-$y_j$ + min cost of aligning two strings $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_{j-1}$

- **Case 2a:** $OPT$ leaves $x_i$ unmatched.
  - pay gap for $x_i$ and min cost of aligning $x_1 x_2 \ldots x_{i-1}$ and $y_1 y_2 \ldots y_j$

- **Case 2b:** $OPT$ leaves $y_j$ unmatched.
  - pay gap for $y_j$ and min cost of aligning $x_1 x_2 \ldots x_i$ and $y_1 y_2 \ldots y_{j-1}$

$$OPT(i, j) = \begin{cases} 
  j\delta & \text{if } i = 0 \\
  \min \left\{ \begin{array}{ll}
  \alpha_{x_i, y_j} + OPT(i - 1, j - 1) & \\
  \delta + OPT(i - 1, j) & \delta + OPT(i, j - 1) \\
  i\delta & \text{if } j = 0
\end{array} \right. & \text{otherwise}
\end{cases}$$
Sequence Alignment: Algorithm

Sequence-Alignment(m, n, x₁x₂...xₘ, y₁y₂...yₙ, δ, α) {
    for i = 0 to m
        M[i, 0] = iδ
    for j = 0 to n
        M[0, j] = jδ

    for i = 1 to m
        for j = 1 to n
            M[i, j] = min(α[xᵢ, yⱼ] + M[i-1, j-1],
                        δ + M[i-1, j],
                        δ + M[i, j-1])

    return M[m, n]
}

Analysis. Θ(mn) time and space.

English words or sentences: m, n ≤ 10.

Computational biology: m = n = 100,000. 10 billions ops OK, but 10GB array?
6.7 Sequence Alignment in Linear Space
Sequence Alignment: Linear Space

Q. Can we avoid using quadratic space?

**Easy.** Optimal value in \( O(m + n) \) space and \( O(mn) \) time.
- Compute \( \text{OPT}(i, \cdot) \) from \( \text{OPT}(i-1, \cdot) \).
- No longer a simple way to recover alignment itself.

**Theorem.** [Hirschberg 1975] Optimal alignment in \( O(m + n) \) space and \( O(mn) \) time.
- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.
Edit distance graph.

- Let $f(i, j)$ be shortest path from $(0,0)$ to $(i, j)$.
- Observation: $f(i, j) = \text{OPT}(i, j)$. 

Sequence Alignment: Linear Space
Edit distance graph.
- Let $f(i, j)$ be shortest path from $(0,0)$ to $(i,j)$.
- Can compute $f(\cdot, j)$ for any $j$ in $O(mn)$ time and $O(m+n)$ space.
Edit distance graph.

- Let $g(i, j)$ be shortest path from $(i, j)$ to $(m, n)$.
- Can compute by reversing the edge orientations and inverting the roles of $(0, 0)$ and $(m, n)$.
Sequence Alignment: Linear Space

Edit distance graph.

- Let \( g(i, j) \) be shortest path from \((i, j)\) to \((m, n)\).
- Can compute \( g(\cdot, j) \) for any \( j \) in \( O(mn) \) time and \( O(m + n) \) space.
Observation 1. The cost of the shortest path that uses \((i, j)\) is \(f(i, j) + g(i, j)\).
Observation 2. Let $q$ be an index that minimizes $f(q, n/2) + g(q, n/2)$. Then, the shortest path from $(0, 0)$ to $(m, n)$ uses $(q, n/2)$. 
Divide: find index $q$ that minimizes $f(q, n/2) + g(q, n/2)$ using DP.

- Align $x_q$ and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.
Sequence Alignment: Running Time Analysis Warmup

**Theorem.** Let $T(m, n) = \max$ running time of algorithm on strings of length at most $m$ and $n$. $T(m, n) = O(mn \log n)$.

\[
T(m, n) \leq 2T(m, n/2) + O(mn) \quad \Rightarrow \quad T(m, n) = O(mn \log n)
\]

**Remark.** Analysis is not tight because two sub-problems are of size $(q, n/2)$ and $(m - q, n/2)$. In next slide, we save log n factor.
Theorem. Let $T(m, n) = \max$ running time of algorithm on strings of length $m$ and $n$. $T(m, n) = O(mn)$.

Pf. (by induction on $n$)
- $O(mn)$ time to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ and find index $q$.
- $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.
- Choose constant $c$ so that:
  
  $T(m, 2) \leq cm$
  $T(2, n) \leq cn$
  $T(m, n) \leq cmn + T(q, n/2) + T(m-q, n/2)$

- Base cases: $m = 2$ or $n = 2$.
- Inductive hypothesis: $T(m, n) \leq 2cmn$. 

\[
\begin{align*}
T(m,n) & \leq T(q,n/2) + T(m-q,n/2) + cmn \\
& \leq 2cq/2 + 2c(m-q)n/2 + cmn \\
& = cq + cmn - cq + cmn \\
& = 2cmn
\end{align*}
\]
Today
▶ We’re back
▶ Dynamic programming
▶ Problems due: 5.5, 5.6

Wednesday
▶ Dynamic programming

Friday
▶ Quiz on divide-and-conquer

Contact information
▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting
String Similarity

How similar are two strings?

- occurance
- occurrence

6 mismatches, 1 gap

1 mismatch, 1 gap

0 mismatches, 3 gaps
Applications.
- Basis for Unix diff.
- Speech recognition.
- Computational biology.

- Gap penalty $\delta$; mismatch penalty $\alpha_{pq}$.
- Cost = sum of gap and mismatch penalties.

\[
\alpha_{TC} + \alpha_{GT} + \alpha_{AG} + 2\alpha_{CA} \quad \text{and} \quad 2\delta + \alpha_{CA}
\]
Sequence Alignment

Goal: Given two strings $X = x_1 x_2 \ldots x_m$ and $Y = y_1 y_2 \ldots y_n$ find alignment of minimum cost.

Def. An alignment $M$ is a set of ordered pairs $x_i$-$y_j$ such that each item occurs in at most one pair and no crossings.

Def. The pair $x_i$-$y_j$ and $x_{i'}$-$y_{j'}$ cross if $i < i'$, but $j > j'$.

$$\text{cost}(M) = \sum_{(x_i, y_j) \in M} \alpha_{x_i, y_j} + \sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta$$

cost of diagonal $$\alpha$$

cost of mismatch $$\alpha$$

cost of gap $$\delta$$

cost of gap $$\delta$$

Ex: CTACCG vs. TACATG.

Sol: $M = x_2$-$y_1$, $x_3$-$y_2$, $x_4$-$y_3$, $x_5$-$y_4$, $x_6$-$y_6$. 
Sequence Alignment: Problem Structure

**Def.** \( \text{OPT}(i, j) = \min \text{ cost of aligning strings } x_1 x_2 \ldots x_i \text{ and } y_1 y_2 \ldots y_j. \)

- **Case 1:** \( \text{OPT} \) matches \( x_i \)-\( y_j \).
  - pay mismatch for \( x_i \)-\( y_j \) + min cost of aligning two strings \( x_1 x_2 \ldots x_{i-1} \text{ and } y_1 y_2 \ldots y_{j-1} \)

- **Case 2a:** \( \text{OPT} \) leaves \( x_i \) unmatched.
  - pay gap for \( x_i \) and min cost of aligning \( x_1 x_2 \ldots x_{i-1} \text{ and } y_1 y_2 \ldots y_j \)

- **Case 2b:** \( \text{OPT} \) leaves \( y_j \) unmatched.
  - pay gap for \( y_j \) and min cost of aligning \( x_1 x_2 \ldots x_i \text{ and } y_1 y_2 \ldots y_{j-1} \)

\[
\text{OPT}(i, j) = \begin{cases} 
  j\delta & \text{if } i = 0 \\
  \min \left\{ \begin{array}{l}
  \alpha_{x_i y_j} + \text{OPT}(i-1, j-1) \\
  \delta + \text{OPT}(i-1, j) \\
  \delta + \text{OPT}(i, j-1) \\
  i\delta \\
\end{array} \right\} & \text{otherwise} \\
\end{cases}
\]
Sequence Alignment: Algorithm

```plaintext
Sequence-Alignment(m, n, x_1x_2...x_m, y_1y_2...y_n, \delta, \alpha) {
    for i = 0 to m
        M[i, 0] = i\delta
    for j = 0 to n
        M[0, j] = j\delta
    for i = 1 to m
        for j = 1 to n
            M[i, j] = min(\alpha[x_i, y_j] + M[i-1, j-1],
                            \delta + M[i-1, j],
                            \delta + M[i, j-1])
    return M[m, n]
}
```

**Analysis.** $\Theta(mn)$ time and space.

**English words or sentences:** $m, n \leq 10$.

**Computational biology:** $m = n = 100,000$. 10 billions ops OK, but 10GB array?
6.7 Sequence Alignment in Linear Space
Sequence Alignment: Linear Space

Q. Can we avoid using quadratic space?

Easy. Optimal value in $O(m + n)$ space and $O(mn)$ time.
- Compute $OPT(i, \cdot)$ from $OPT(i-1, \cdot)$.
- No longer a simple way to recover alignment itself.

Theorem. [Hirschberg 1975] Optimal alignment in $O(m + n)$ space and $O(mn)$ time.
- Clever combination of divide-and-conquer and dynamic programming.
- Inspired by idea of Savitch from complexity theory.
Sequence Alignment: Linear Space

Edit distance graph.
- Let $f(i, j)$ be shortest path from (0,0) to (i, j).
- Observation: $f(i, j) = \text{OPT}(i, j)$.
Sequence Alignment: Linear Space

Edit distance graph.
- Let $f(i, j)$ be shortest path from $(0,0)$ to $(i, j)$.
- Can compute $f(\cdot, j)$ for any $j$ in $O(mn)$ time and $O(m + n)$ space.
Sequence Alignment: Linear Space

Edit distance graph.
- Let $g(i, j)$ be shortest path from $(i, j)$ to $(m, n)$.
- Can compute by reversing the edge orientations and inverting the roles of $(0, 0)$ and $(m, n)$.
Edit distance graph.

- Let \( g(i, j) \) be shortest path from \((i, j)\) to \((m, n)\).
- Can compute \( g(\cdot, j) \) for any \( j \) in \( O(mn) \) time and \( O(m + n) \) space.
Observation 1. The cost of the shortest path that uses \((i, j)\) is 
\(f(i, j) + g(i, j)\).
Observation 2. Let $q$ be an index that minimizes $f(q, n/2) + g(q, n/2)$. Then, the shortest path from $(0, 0)$ to $(m, n)$ uses $(q, n/2)$.

Sequence Alignment: Linear Space
Divide: find index $q$ that minimizes $f(q, n/2) + g(q, n/2)$ using DP.
- Align $x_q$ and $y_{n/2}$.

Conquer: recursively compute optimal alignment in each piece.
Sequence Alignment: Running Time Analysis

**Theorem.** Let $T(m, n) = \max$ running time of algorithm on strings of length $m$ and $n$. $T(m, n) = O(mn)$.

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

- $O(mn)$ time to compute $f(\cdot, n/2)$ and $g(\cdot, n/2)$ and find index $q$.
- $T(q, n/2) + T(m - q, n/2)$ time for two recursive calls.
- Choose constant $c$ so that:

  $T(m, 2) \leq cm$
  $T(2, n) \leq cn$
  $T(m, n) \leq cmn + T(q, n/2) + T(m - q, n/2)$

- Base cases: $m = 2$ or $n = 2$.
- Inductive hypothesis: $T(m, n) \leq 2cmn$.

- $T(m,n) \leq T(q,n/2) + T(m - q,n/2) + cmn$
  $\leq 2cqn/2 + 2c(m - q)n/2 + cmn$
  $= cqn + cmn - cqn + cmn$
  $= 2cmn$
Algorithms — March 21

Today

▶ Dynamic programming

Friday

▶ Quiz on divide-and-conquer

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting
**Shortest Paths**

**Shortest path problem.** Given a directed graph $G = (V, E)$, with edge weights $c_{vw}$, find shortest path from node $s$ to node $t$.

Ex. Nodes represent agents in a financial setting and $c_{vw}$ is cost of transaction in which we buy from agent $v$ and sell immediately to $w$. 

![Graph with edge labels and node connections.](image-url)
Shortest Paths: Failed Attempts

**Dijkstra.** Can fail if negative edge costs.

![Graph](image)

**Re-weighting.** Adding a constant to every edge weight can fail.

![Re-weighted Graph](image)
Shortest Paths: Negative Cost Cycles

Negative cost cycle.

Observation. If some path from $s$ to $t$ contains a negative cost cycle, there does not exist a shortest $s$-$t$ path; otherwise, there exists one that is simple.
Shortest Paths: Dynamic Programming

**Def.** $\text{OPT}(i, v) =$ length of shortest $v$-$t$ path $P$ using at most $i$ edges.

- **Case 1:** $P$ uses at most $i-1$ edges.
  - $\text{OPT}(i, v) = \text{OPT}(i-1, v)$

- **Case 2:** $P$ uses exactly $i$ edges.
  - if $(v, w)$ is first edge, then $\text{OPT}$ uses $(v, w)$, and then selects best $w$-$t$ path using at most $i-1$ edges

\[
\text{OPT}(i, v) = \begin{cases}
0 & \text{if } i = 0 \\
\min \left\{ \text{OPT}(i-1, v), \min_{(v, w) \in E} \left\{ \text{OPT}(i-1, w) + c_{vw} \right\} \right\} & \text{otherwise}
\end{cases}
\]

**Remark.** By previous observation, if no negative cycles, then $\text{OPT}(n-1, v) =$ length of shortest $v$-$t$ path.
Shortest Paths: Implementation

Finding the shortest paths. Maintain a "successor" for each table entry.

Analysis. $\Theta(mn)$ time, $\Theta(n^2)$ space.
Shortest Paths: Practical Improvements

Practical improvements.

- Maintain only one array $M[v] =$ shortest v-t path that we have found so far.
- No need to check edges of the form (v, w) unless $M[w]$ changed in previous iteration.

Theorem. Throughout the algorithm, $M[v]$ is length of some v-t path, and after i rounds of updates, the value $M[v]$ is no larger than the length of shortest v-t path using $\leq i$ edges.

Overall impact.

- Memory: $O(m + n)$.
- Running time: $O(mn)$ worst case, but substantially faster in practice.
Bellman-Ford: Efficient Implementation

Push-Based-Shortest-Path(G, s, t) {
    foreach node v ∈ V {
        M[v] ← ∞
        successor[v] ← φ
    }
    M[t] = 0
    for i = 1 to n-1 {
        foreach node w ∈ V {
            if (M[w] has been updated in previous iteration) {
                foreach node v such that (v, w) ∈ E {
                    if (M[v] > M[w] + c_vw) {
                        M[v] ← M[w] + c_vw
                        successor[v] ← w
                    }
                }
            }
        }
        If no M[w] value changed in iteration i, stop.
    }
}
Detecting Negative Cycles

**Lemma.** If $\text{OPT}(n,v) = \text{OPT}(n-1,v)$ for all $v$, then no negative cycles.

**Pf.** Bellman-Ford algorithm.

**Lemma.** If $\text{OPT}(n,v) < \text{OPT}(n-1,v)$ for some node $v$, then (any) shortest path from $v$ to $t$ contains a cycle $W$. Moreover $W$ has negative cost.

**Pf.** (by contradiction)
- Since $\text{OPT}(n,v) < \text{OPT}(n-1,v)$, we know $P$ has exactly $n$ edges.
- By pigeonhole principle, $P$ must contain a directed cycle $W$.
- Deleting $W$ yields a $v$-$t$ path with $< n$ edges $\Rightarrow$ $W$ has negative cost.
Detecting Negative Cycles

**Theorem.** Can detect negative cost cycle in $O(mn)$ time.

- Add new node $t$ and connect all nodes to $t$ with 0-cost edge.
- Check if $OPT(n, v) = OPT(n-1, v)$ for all nodes $v$.
  - if yes, then no negative cycles
  - if no, then extract cycle from shortest path from $v$ to $t$
Detecting Negative Cycles: Application

**Currency conversion.** Given n currencies and exchange rates between pairs of currencies, is there an arbitrage opportunity?

**Remark.** Fastest algorithm very valuable!
Detecting Negative Cycles: Summary

\textbf{Bellman-Ford.} $O(mn)$ time, $O(m + n)$ space.
- Run Bellman-Ford for $n$ iterations (instead of $n-1$).
- Upon termination, Bellman-Ford successor variables trace a negative cycle if one exists.
- See p. 304 for improved version and early termination rule.
Algorithms — March 23

Today
- Quiz on divide-and-conquer
- Intro to network flow

Monday
- Network flow
- Problems due: 6.2, 6.3

Next Monday
- Problems due: 6.9, 6.11

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
Flow network.
- Abstraction for material **flowing** through the edges.
- $G = (V, E)$ = directed graph, no parallel edges.
- Two distinguished nodes: $s$ = source, $t$ = sink.
- $c(e)$ = capacity of edge $e$. 

Minimum Cut Problem
**Def.** An **s-t cut** is a partition \((A, B)\) of \(V\) with \(s \in A\) and \(t \in B\).

**Def.** The **capacity** of a cut \((A, B)\) is: 
\[
\text{cap}(A, B) = \sum_{e \text{ out of } A} c(e)
\]

![Graph Image]

Capacity = 10 + 5 + 15
= 30
Cuts

Def. An \textit{s-t cut} is a partition \((A, B)\) of \(V\) with \(s \in A\) and \(t \in B\).

Def. The \textit{capacity} of a cut \((A, B)\) is:

\[
\text{cap}(A, B) = \sum_{e \text{ out of } A} c(e)
\]

Capacity = 9 + 15 + 8 + 30 = 62
Min s-t cut problem. Find an s-t cut of minimum capacity.

Minimum Cut Problem

Min s-t cut problem. Find an s-t cut of minimum capacity.
**Def.** An s-t flow is a function that satisfies:

- For each $e \in E$: $0 \leq f(e) \leq c(e)$  
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$

**Def.** The value of a flow $f$ is: $v(f) = \sum_{e \text{ out of } s} f(e)$.

Flows

```
+----+    +----+    +----+    +----+    +----+
|    | 9   | 0   | 15  | 0   | 10  |
|    |      |      |      |      |      |
| 2   | 4     | 4    | 0    | 0    | 0    |
| 3   | 10    | 4    | 15   | 15   | 0    |
| 4   | 0     | 4    | 0    | 0    | 10   |
| 5   | 9     | 0    | 15   | 0    | 0    |
| 6   | 0     | 8    | 8    | 15   | 0    |
| 7   | 30    | 0    | 6    | 15   | 0    |
| t   | 0     | 0    | 0    | 0    | 10   |
```

Value = 4
Def. An s-t flow is a function that satisfies:
- For each $e \in E$: $0 \leq f(e) \leq c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [conservation]

Def. The value of a flow $f$ is: $v(f) = \sum_{e \text{ out of } s} f(e)$.

![Flow network diagram]

Value = 24
Max flow problem. Find s-t flow of maximum value.
Maximum Flow and Minimum Cut

Max flow and min cut.
- Two very rich algorithmic problems.
- Cornerstone problems in combinatorial optimization.
- Beautiful mathematical duality.

Nontrivial applications / reductions.
- Data mining.
- Open-pit mining.
- Project selection.
- Airline scheduling.
- Bipartite matching.
- Baseball elimination.
- Image segmentation.
- Network connectivity.

- Network reliability.
- Distributed computing.
- Egalitarian stable matching.
- Security of statistical data.
- Network intrusion detection.
- Multi-camera scene reconstruction.
- Many many more ...
Flows and Cuts

Flow value lemma. Let $f$ be any flow, and let $(A, B)$ be any $s$-$t$ cut. Then, the net flow sent across the cut is equal to the amount leaving $s$.

$$\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f)$$

![Flow network diagram]
Flow value lemma. Let $f$ be any flow, and let $(A, B)$ be any $s$-$t$ cut. Then, the net flow sent across the cut is equal to the amount leaving $s$.

$$
\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f)
$$

![Graph diagram]

Value = $6 + 0 + 8 - 1 + 11 = 24$
**Flows and Cuts**

**Flow value lemma.** Let \( f \) be any flow, and let \((A, B)\) be any s-t cut. Then, the net flow sent across the cut is equal to the amount leaving \( s \).

\[
\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f)
\]
**Flows and Cuts**

**Weak duality.** Let \( f \) be any flow, and let \((A, B)\) be any \( s-t \) cut. Then the value of the flow is at most the capacity of the cut.

\[
\text{Cut capacity} = 30 \quad \Rightarrow \quad \text{Flow value} \leq 30
\]
**Certificate of Optimality**

**Corollary.** Let $f$ be any flow, and let $(A, B)$ be any cut. If $v(f) = \text{cap}(A, B)$, then $f$ is a max flow and $(A, B)$ is a min cut.

Value of flow = 28
Cut capacity = 28 $\Rightarrow$ Flow value $\leq$ 28
Algorithms — March 26

Today
- Network flow
- Problems due: 6.2, 6.3

Next Monday
- Problems due: 6.9, 6.11

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
Towards a Max Flow Algorithm

**Greedy algorithm.**
- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an $s$-$t$ path $P$ where each edge has $f(e) < c(e)$.
- Augment flow along path $P$.
- Repeat until you get stuck.

Flow value = 0
Towards a Max Flow Algorithm

**Greedy algorithm.**
- Start with \( f(e) = 0 \) for all edge \( e \in E \).
- Find an \( s-t \) path \( P \) where each edge has \( f(e) < c(e) \).
- Augment flow along path \( P \).
- Repeat until you get stuck.

![Diagram of a network flow problem with a flow value of 20]

**Flow value = 20**
Towards a Max Flow Algorithm

**Greedy algorithm.**
- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an $s$-$t$ path $P$ where each edge has $f(e) < c(e)$.
- Augment flow along path $P$.
- Repeat until you get stuck.

![Diagram](image)

- **greedy = 20**
- **opt = 30**

 locally optimality $\not\Rightarrow$ global optimality
**Residual Graph**

**Original edge:**  $e = (u, v) \in E$.
- Flow $f(e)$, capacity $c(e)$.

**Residual edge.**
- "Undo" flow sent.
- $e = (u, v)$ and $e^R = (v, u)$.
- Residual capacity:
  $$c_f(e) = \begin{cases} 
  c(e) - f(e) & \text{if } e \in E \\
  f(e) & \text{if } e^R \in E
  \end{cases}$$

**Residual graph:**  $G_f = (V, E_f)$.
- Residual edges with positive residual capacity.
- $E_f = \{e : f(e) < c(e)\} \cup \{e^R : f(e) > 0\}$. 
Ford-Fulkerson Algorithm

$G$:  

- Source (s) to Node 2: 10 units
- Node 2 to Node 4: 4 units
- Node 3 to Node 5: 8 units
- Node 4 to Node 5: 6 units
- Node 5 to Sink (+): 10 units

Capacity arrows indicate the maximum flow that can be sent through each edge.
Augmenting Path Algorithm

Augment \( f, c, P \) {
    \( b \leftarrow \text{bottleneck}(P) \)
    \( \text{foreach } e \in P \) {
        \text{if } (e \in E) f(e) \leftarrow f(e) + b
        \text{else } f(e^R) \leftarrow f(e^R) - b
    }
    \text{return } f
}

Ford-Fulkerson \( G, s, t, c \) {
    \( \text{foreach } e \in E \ f(e) \leftarrow 0 \)
    \( G_f \leftarrow \text{residual graph} \)
    \text{while } (\text{there exists augmenting path } P) \{
        f \leftarrow \text{Augment}(f, c, P)
        \text{update } G_f
    }
    \text{return } f
}
Algorithms — March 30

Today

▶ Network flow

Monday, April 2

▶ Problems due: 6.9, 6.11

Friday, April 6

▶ Quiz on Chapter 6

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
   ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
   ▶ Wednesday 2: 3-5pm
   ▶ Friday 4: 2-4pm
   ▶ Monday 7: 10am-noon
▶ Or by appointment
   ▶ Email at least a day ahead
   ▶ Send several times when you are available
   ▶ Always describe what you need to discuss
      ▶ So I can be prepared
      ▶ Because advising or paperwork often require no meeting
Augmenting Path Algorithm

Augment(f, c, P) {
    b ← bottleneck(P)
    foreach e ∈ P {
        if (e ∈ E) f(e) ← f(e) + b
        else f(e^R) ← f(e^R) − b
    }
    return f
}

Ford-Fulkerson(G, s, t, c) {
    foreach e ∈ E  f(e) ← 0
    G_f ← residual graph

    while (there exists augmenting path P) {
        f ← Augment(f, c, P)
        update G_f
    }
    return f
}
Questions about Ford-Fulkerson

1. Does it always produce a flow?
   - Yes it does (discussed on Monday)
2. Does its while loop always terminate?
3. How much time does it require?
4. Does it produce a maximum flow?
5. Does it matter what augmenting path we choose?
   - For correctness?
   - For time cost?
Reminder of some notation

- Each directed graph edge has a capacity
- A flow describes how we use part (or all, or none) of each edge's capacity
  - \(0 \leq f(e) \leq c_e\)
  - The sum of the flow into a vertex must equal the flow out of that vertex (except for the source and sink)
    - No flow into the source, or out from the sink
  - The value of the flow \(v(f)\) is the sum of the flows of the edges leading from the source
- The residual graph for a flow has
  - Forward edges labeled with remaining capacity
  - Backwards edges labeled with the amount of flow which we could decrease or divert
- An augmenting path is a simple path from source to sink in the residual graph
  - Its bottleneck capacity is the lowest capacity of its edges
- A cut of a network graph is a partition \((A, B)\) of the nodes with the source in \(A\) and the sink in \(B\)
  - The capacity of a cut, \(c(A, B)\), is the sum of the capacities of the edges outgoing from \(A\)
Three steps towards termination

We will need three observations to argue that the `while` loop in Ford-Fulkerson always terminates.
Three steps towards termination

We will need three observations to argue that the while loop in Ford-Fulkerson always terminates.

1. The flow values and residual capacities are always all integers.

This observation may seem banal, but it will help us to prove termination.

- Clearly true before any iteration of the while loop — the flow values are all zero, and the residual capacities are the original integer weights.
- Suppose it is true after $j$ iterations: so the capacities in the residual graph are all integers.
- So on the $(j + 1)^{st}$ iteration, the bottleneck capacity of any augmenting path must also be an integer.
- This means that the changes to the flow graph will be by (and will produce) integer values as well.
Three steps towards termination

We will need three observations to argue that the while loop in Ford-Fulkerson always terminates

1. The flow values and residual capacities are always all integers

This observation may seem banal, but it will help us to prove termination

- Clearly true before any iteration of the while loop — the flow values are all zero, and the residual capacities are the original integer weights
Three steps towards termination

We will need three observations to argue that the while loop in Ford-Fulkerson always terminates:

1. The flow values and residual capacities are always all integers.

This observation may seem banal, but it will help us to prove termination:

- Clearly true before any iteration of the while loop — the flow values are all zero, and the residual capacities are the original integer weights.
- Suppose it is true after \( j \) iterations: so the capacities in the residual graph are all integers.

  - So on the \((j + 1)^{st}\) iteration, the bottleneck capacity of any augmenting path must also be an integer.
  - This means that the changes to the flow graph will be by (and will produce) integer values as well.
Three steps towards termination

Consider the augmenting path through the residual graph

We will need three observations to argue that the while loop in Ford-Fulkerson always terminates

1. The flow values and residual capacities are always all integers
2. Each addition to the flow graph increases its value
Three steps towards termination

We will need three observations to argue that the while loop in Ford-Fulkerson always terminates

1. The flow values and residual capacities are always all integers
2. Each addition to the flow graph increases its value

Consider the augmenting path through the residual graph

- Since the path runs from source to sink, the first edge $e$ must be from $s$
- Since the original graph has no edges to $s$, $e$ must be a forward edge
Three steps towards termination

We will need three observations to argue that the while loop in Ford-Fulkerson always terminates

1. The flow values and residual capacities are always all integers
2. Each addition to the flow graph increases its value

Consider the augmenting path through the residual graph

- Since the path runs from source to sink, the first edge $e$ must be from $s$
- Since the original graph has no edges to $s$, $e$ must be a forward edge
- So we increase the flow on $e$ by the bottleneck value
Three steps towards termination

We will need three observations to argue that the `while` loop in Ford-Fulkerson always terminates

1. The flow values and residual capacities are always all integers
2. Each addition to the flow graph increases its value

Consider the augmenting path through the residual graph

- Since the path runs from source to sink, the first edge \( e \) must be from \( s \)
- Since the original graph has no edges to \( s \), \( e \) must be a forward edge
- So we increase the flow on \( e \) by the bottleneck value
- Since the path is simple, no other edge has \( s \) as an endpoint — so we do not change the flow on any other edge from \( s \)
Three steps towards termination

We will need three observations to argue that the while loop in Ford-Fulkerson always terminates

1. The flow values and residual capacities are always all integers

2. Each addition to the flow graph increases its value

Consider the augmenting path through the residual graph

- Since the path runs from source to sink, the first edge $e$ must be from $s$
- Since the original graph has no edges to $s$, $e$ must be a forward edge
- So we increase the flow on $e$ by the bottleneck value
- Since the path is simple, no other edge has $s$ as an endpoint — so we do not change the flow on any other edge from $s$
- So the overall change to the flow on the edges from $s$ is to increase it
Three steps towards termination

We will need three observations to argue that the `while` loop in Ford-Fulkerson always terminates:

1. The flow values and residual capacities are always all integers.
2. Each addition to the flow graph increases its value.
3. The flow value has an upper bound.

- The flow on each edge is bounded by the capacity of that edge.
- So the overall flow value cannot exceed the sum of the capacities of the outgoing edges from `s`.
We will need three observations to argue that the while loop in Ford-Fulkerson always terminates:

1. The flow values and residual capacities are always all integers.
2. Each addition to the flow graph increases its value.
3. The flow value has an upper bound.

Putting the three steps together:

- Let $C$ be the sum of the edges out of $s$. The flow starts at 0, increases at each iteration to another integer, and cannot exceed $C$. So the while loop can run at most $C$ times — Ford-Fulkerson does always terminate.
Three steps towards termination

We will need three observations to argue that the `while` loop in Ford-Fulkerson always terminates.

1. The flow values and residual capacities are always all integers.
2. Each addition to the flow graph increases its value.
3. The flow value has an upper bound.

Putting the three steps together:

- Let $C$ be the sum of the edges out of $s$.
- The flow starts at 0, increases at each iteration to another integer, and cannot exceed $C$.
Three steps towards termination

We will need three observations to argue that the `while` loop in Ford-Fulkerson always terminates

1. The flow values and residual capacities are always all integers
2. Each addition to the flow graph increases its value
3. The flow value has an upper bound

Putting the three steps together:

- Let $C$ be the sum of the edges out of $s$
- The flow starts at 0, increases at each iteration to another integer, and cannot exceed $C$
- So the `while` loop can run at most $C$ times — Ford-Fulkerson does always terminate
What happen on each iteration?

Let’s assume that the network graph is connected

- So \( m \geq n/2 \)
- And so \( O(m + n) = O(m) \).
What happen on each iteration?

Let's assume that the network graph is connected

- So $m \geq n/2$
- And so $O(m + n) = O(m)$.

The residual graph has at most $2m$ edges

- On each iteration we find a path from $s$ to $t$
  - Can use BFS, $O(2m + n) = O(m)$
- Then augment the flow graph with the path
  - Path has at most $n - 1$ edges, so $O(n) = O(m)$
- Then update the residual graph
  - Requires visiting every edge, so $O(m)$

So each iteration requires $O(m)$ time, and we will execute the loop at most $C$ times

Worst-case runtime is $O(mC)$

Another pseudo-polynomial — the bound includes factors other than the size of the input graph

We can do better — but not today!
What happen on each iteration?

Let's assume that the network graph is connected

▶ So \( m \geq n/2 \)
▶ And so \( O(m + n) = O(m) \).

The residual graph has at most \( 2m \) edges

▶ On each iteration we find a path from \( s \) to \( t \)
  ▶ Can use BFS, \( O(2m + n) = O(m) \)
▶ Then augment the flow graph with the path
  ▶ Path has at most \( n - 1 \) edges, so \( O(n) = O(m) \)
▶ Then update the residual graph
  ▶ Requires visiting every edge, so \( O(m) \)

So each iteration requires \( O(m) \) time, and we will execute the loop at most \( C \) times

▶ Worst-case runtime is \( O(mC) \)
▶ Another *pseudo-polynomial* — the bound includes factors other than the size of the input graph
▶ We can do better — but not today!
The final residual graph

In the final residual graph, there is no $s - t$ path

- Divide the nodes into $A^*$, $B^*$ where $A^*$ includes $s$ plus the nodes reachable from $s$ via the final residual graph, and $B^*$ are the other nodes
- This $(A^*, B^*)$ is a cut

How does this help us?

Remember the relationship between flows and cuts. . .

By weak duality, this means Ford-Fulkerson does in fact find the maximum flow
In the final residual graph, there is no $s - t$ path

- Divide the nodes into $A^*$, $B^*$ where $A^*$ includes $s$ plus the nodes reachable from $s$ via the final residual graph, and $B^*$ are the other nodes
- This $(A^*, B^*)$ is a cut
- What is the flow on edges from $A^*$ to $B^*$?
  - If it were less than the capacity, then the residual graph would have a forward edge
  - But by construction of $A^*$, this cannot be so

How does this help us?

Remember the relationship between flows and cuts.

By weak duality, this means Ford-Fulkerson does in fact find the maximum flow
The final residual graph

In the final residual graph, there is no $s - t$ path

- Divide the nodes into $A^*$, $B^*$ where $A^*$ includes $s$ plus the nodes reachable from $s$ via the final residual graph, and $B^*$ are the other nodes

- This $(A^*, B^*)$ is a cut

- What is the flow on edges from $A^*$ to $B^*$?
  - If it were less than the capacity, then the residual graph would have a forward edge
  - But by construction of $A^*$, this cannot be so

- What is the flow on edges from $B^*$ to $A^*$?
  - If it were above zero, then the residual graph would have a backward edge
  - But by construction of $A^*$, this cannot be so

So $v(f) = c(A^*, B^*)$

How does this help us?

Remember the relationship between flows and cuts. . .

By weak duality, this means Ford-Fulkerson does in fact find the maximum flow
The final residual graph

In the final residual graph, there is no $s - t$ path

- Divide the nodes into $A^*$, $B^*$ where $A^*$ includes $s$ plus the nodes reachable from $s$ via the final residual graph, and $B^*$ are the other nodes
- This $(A^*, B^*)$ is a cut
- What is the flow on edges from $A^*$ to $B^*$?
  - If it were less than the capacity, then the residual graph would have a forward edge
  - But by construction of $A^*$, this cannot be so
- What is the flow on edges from $B^*$ to $A^*$?
  - If it were above zero, then the residual graph would have a backward edge
  - But by construction of $A^*$, this cannot be so
- So $\nu(f) = c(A^*, B^*)$
  - How does this help us?
  - Remember the relationship between flows and cuts...
Flow value lemma. Let $f$ be any flow, and let $(A, B)$ be any $s$-$t$ cut. Then, the net flow sent across the cut is equal to the amount leaving $s$.

\[
\sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) = v(f)
\]

Value = $6 + 0 + 8 - 1 + 11 = 24$
**Weak duality.** Let $f$ be any flow, and let $(A, B)$ be any $s$-$t$ cut. Then the value of the flow is at most the capacity of the cut.
The final residual graph

In the final residual graph, there is no $s - t$ path

- Divide the nodes into $A^*$, $B^*$ where $A^*$ includes $s$ plus the nodes reachable from $s$ via the final residual graph, and $B^*$ are the other nodes
- This $(A^*, B^*)$ is a cut
- What is the flow on edges from $A^*$ to $B^*$?
  - If it were less than the capacity, then the residual graph would have a forward edge
  - But by construction of $A^*$, this cannot be so
- What is the flow on edges from $B^*$ to $A^*$?
  - If it were above zero, then the residual graph would have a backward edge
  - But by construction of $A^*$, this cannot be so
- So $v(f) = c(A^*, B^*)$

  - How does this help us?
  - Remember the relationship between flows and cuts...
  - By weak duality, this means Ford-Fulkerson does in fact find the maximum flow
Algorithms — March 30

Contact information

- Web: cs.uwlax.edu/~jmarais
- Email: jmarais@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting

Today

- Network flow

Monday, April 2

- Problems due: 6.9, 6.11

Friday, April 6

- Quiz on Chapter 6
Questions about Ford-Fulkerson

1. Does it always produce a flow?
   ▶ Yes it does (discussed on Monday)

2. Does its while loop always terminate?
   ▶ Yes

3. How much time does it require?
   ▶ $O(mC)$

4. Does it produce a *maximum* flow?
   ▶ Yes

5. Does it matter what augmenting path we choose?
   ▶ For correctness?
   ▶ For time cost?
Ford-Fulkerson: Exponential Number of Augmentations

**Q.** Is generic Ford-Fulkerson algorithm polynomial in input size?

**A.** No. If max capacity is $C$, then algorithm can take $C$ iterations.

$m, n, \text{ and } \log C$
Choosing Good Augmenting Paths

Use care when selecting augmenting paths.
- Some choices lead to exponential algorithms.
- Clever choices lead to polynomial algorithms.
- If capacities are irrational, algorithm not guaranteed to terminate!

Goal: choose augmenting paths so that:
- Can find augmenting paths efficiently.
- Few iterations.

Choose augmenting paths with: [Edmonds-Karp 1972, Dinitz 1970]
- Max bottleneck capacity.
- Sufficiently large bottleneck capacity.
- Fewest number of edges.
**Capacity Scaling**

**Intuition.** Choosing path with highest bottleneck capacity increases flow by max possible amount.

- Don't worry about finding exact highest bottleneck path.
- Maintain scaling parameter $\Delta$.
- Let $G_f(\Delta)$ be the subgraph of the residual graph consisting of only arcs with capacity at least $\Delta$. 

![Diagram](attachment:image.png)
From Ford-Fulkerson to Capacity Scaling Max-Flow

Ford-Fulkerson

Ford-Fulkerson(G, s, t, c)

- foreach e∈ E
  - f(e) ← 0
- Gf ← residual graph
- while ∃ augmenting path P in Gf
  - f ← augment(f, c, P)
  - update Gf

Capacity scaling

Scaling-Max-Flow(G, s, t, c)

- foreach e∈ E
  - f(e) ← 0
- Δ ← smallest power of 2 ≥ C
- Gf ← residual graph
- while Δ ≥ 1
  - Gf ← $\Delta$-residual graph
  - while ∃ augmenting path P in Gf(Δ)
    - f ← augment(f, c, P)
    - update Gf(Δ)
  - Δ ← Δ / 2
- return f
Capacity Scaling: Correctness

**Assumption.** All edge capacities are integers between 1 and $C$.

**Integrality invariant.** All flow and residual capacity values are integral.

**Correctness.** If the algorithm terminates, then $f$ is a max flow.

**Pf.**
- By integrality invariant, when $\Delta = 1 \Rightarrow G_f(\Delta) = G_f$.
- Upon termination of $\Delta = 1$ phase, there are no augmenting paths. □
Lemma 1. The outer while loop repeats $1 + \lceil \log_2 C \rceil$ times.
Pf. Initially $C \leq \Delta < 2C$. $\Delta$ decreases by a factor of 2 each iteration. □

Lemma 2. Let $f$ be the flow at the end of a $\Delta$-scaling phase. Then the value of the maximum flow is at most $v(f) + m \Delta$. ◄ proof on next slide

Lemma 3. There are at most $2m$ augmentations per scaling phase.
- Let $f$ be the flow at the end of the previous scaling phase.
- $L2 \Rightarrow v(f^*) \leq v(f) + m (2\Delta)$.
- Each augmentation in a $\Delta$-phase increases $v(f)$ by at least $\Delta$. □

Theorem. The scaling max-flow algorithm finds a max flow in $O(m \log C)$ augmentations. It can be implemented to run in $O(m^2 \log C)$ time. □
More work on tuning Ford-Fulkerson

In the last 50 years there has been much more work on improving Ford-Fulkerson

- If each iteration chooses the augmenting path with the fewest edges, the algorithm terminates in \(O(mn)\) time
- Different current algorithms terminate in
  - \(O(mn \log n)\)
  - \(O(n^3)\)
  - \(O(\min(n^{\frac{2}{3}}, m^{\frac{1}{2}}) m \log n \log U)\), all capacities integral and at most \(U\)
7.5 Bipartite Matching
Matching

Matching.
- Input: undirected graph $G = (V, E)$.
- $M \subseteq E$ is a matching if each node appears in at most edge in $M$.
- Max matching: find a max cardinality matching.
Bipartite matching.

- Input: undirected, bipartite graph $G = (L \cup R, E)$.
- $M \subseteq E$ is a matching if each node appears in at most edge in $M$.
- Max matching: find a max cardinality matching.
Bipartite matching.

- Input: undirected, bipartite graph $G = (L \cup R, E)$.
- $M \subseteq E$ is a matching if each node appears in at most one edge in $M$.
- Max matching: find a max cardinality matching.

1-1', 2-2', 3-3', 4-4'
Max flow formulation.
- Create digraph $G' = (L \cup R \cup \{s, t\}, E')$.
- Direct all edges from $L$ to $R$, and assign infinite (or unit) capacity.
- Add source $s$, and unit capacity edges from $s$ to each node in $L$.
- Add sink $t$, and unit capacity edges from each node in $R$ to $t$.

Bipartite Matching
Def. A matching $M \subseteq E$ is **perfect** if each node appears in exactly one edge in $M$.

**Q.** When does a bipartite graph have a perfect matching?

**Structure of bipartite graphs with perfect matchings.**
- Clearly we must have $|L| = |R|$.
- What other conditions are necessary?
- What conditions are sufficient?
**Notation.** Let $S$ be a subset of nodes, and let $N(S)$ be the set of nodes adjacent to nodes in $S$.

**Observation.** If a bipartite graph $G = (L \cup R, E)$, has a perfect matching, then $|N(S)| \geq |S|$ for all subsets $S \subseteq L$.

**Pf.** Each node in $S$ has to be matched to a different node in $N(S)$. 

**Example:**

No perfect matching:

- $S = \{2, 4, 5\}$
- $N(S) = \{2', 5'\}$. 

![Diagram of a bipartite graph with nodes and edges illustrating the perfect matching concept.]

- Nodes 1 and 1' are matched.
- Nodes 2 and 2' are matched.
- Nodes 3 and 3' are matched.
- Nodes 4 and 4' are matched.
- Nodes 5 and 5' are matched.
Bipartite Matching: Running Time

Which max flow algorithm to use for bipartite matching?
- Generic augmenting path: $O(m \text{ val}(f^*)) = O(mn)$.
- Capacity scaling: $O(m^2 \log C) = O(m^2)$.
- Shortest augmenting path: $O(m n^{1/2})$.

Non-bipartite matching.
- Structure of non-bipartite graphs is more complicated, but well-understood. [Tutte-Berge, Edmonds-Galai]
- Blossom algorithm: $O(n^4)$. [Edmonds 1965]
- Best known: $O(m n^{1/2})$. [Micali-Vazirani 1980]
7.6 Disjoint Paths
Edge Disjoint Paths

**Disjoint path problem.** Given a digraph $G = (V, E)$ and two nodes $s$ and $t$, find the max number of edge-disjoint $s$-$t$ paths.

**Def.** Two paths are **edge-disjoint** if they have no edge in common.

**Ex:** communication networks.
Disjoint path problem. Given a digraph $G = (V, E)$ and two nodes $s$ and $t$, find the max number of edge-disjoint $s$-$t$ paths.

Def. Two paths are edge-disjoint if they have no edge in common.

Ex: communication networks.
**Max flow formulation:** assign unit capacity to every edge.

**Theorem.** Max number edge-disjoint s-t paths equals max flow value.

**Pf.** ≤

- Suppose there are k edge-disjoint paths $P_1, \ldots, P_k$.
- Set $f(e) = 1$ if $e$ participates in some path $P_i$; else set $f(e) = 0$.
- Since paths are edge-disjoint, $f$ is a flow of value $k$. ▪
Today
- Disjoint paths
- Circulations
- Problems due: 6.9, 6.11

Wednesday, April 4
- Some more involved applications

Friday, April 6
- Baseball!
- Quiz on Chapter 6

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
7.6 Disjoint Paths
Disjoint path problem. Given a digraph $G = (V, E)$ and two nodes $s$ and $t$, find the max number of edge-disjoint $s$-$t$ paths.

Def. Two paths are edge-disjoint if they have no edge in common.

Ex: communication networks.
Disjoint path problem. Given a digraph $G = (V, E)$ and two nodes $s$ and $t$, find the max number of edge-disjoint $s$-$t$ paths.

Def. Two paths are edge-disjoint if they have no edge in common.

Ex: communication networks.
**Edge Disjoint Paths**

**Max flow formulation:** assign unit capacity to every edge.

**Theorem.** Max number edge-disjoint s-t paths equals max flow value.

**Pf. ≤**
- Suppose there are \( k \) edge-disjoint paths \( P_1, \ldots, P_k \).
- Set \( f(e) = 1 \) if \( e \) participates in some path \( P_i \); else set \( f(e) = 0 \).
- Since paths are edge-disjoint, \( f \) is a flow of value \( k \).
**Max flow formulation:** assign unit capacity to every edge.

**Theorem.** Max number edge-disjoint s-t paths equals max flow value.

**Pf.**
- Suppose max flow value is k.
- Integrality theorem $\Rightarrow$ there exists 0-1 flow $f$ of value k.
- Consider edge $(s, u)$ with $f(s, u) = 1$.
  - by conservation, there exists an edge $(u, v)$ with $f(u, v) = 1$
  - continue until reach $t$, always choosing a new edge
- Produces $k$ (not necessarily simple) edge-disjoint paths.

can eliminate cycles to get simple paths if desired
Network Connectivity

Network connectivity. Given a digraph $G = (V, E)$ and two nodes $s$ and $t$, find min number of edges whose removal disconnects $t$ from $s$.

Def. A set of edges $F \subseteq E$ disconnects $t$ from $s$ if every $s$-$t$ path uses at least one edge in $F$. 
Theorem. [Menger 1927] The max number of edge-disjoint s-t paths is equal to the min number of edges whose removal disconnects t from s.

Pf. ≤
  - Suppose the removal of $F \subseteq E$ disconnects t from s, and $|F| = k$.
  - Every s-t path uses at least one edge in F.
    Hence, the number of edge-disjoint paths is at most k.

Diagram: Two networks with edge-disjoint paths highlighted.
Theorem. [Menger 1927] The max number of edge-disjoint s-t paths is equal to the min number of edges whose removal disconnects t from s.

Pf. ≥

- Suppose max number of edge-disjoint paths is $k$.
- Then max flow value is $k$.
- Max-flow min-cut $\Rightarrow$ cut $(A, B)$ of capacity $k$.
- Let $F$ be set of edges going from $A$ to $B$.
- $|F| = k$ and disconnects $t$ from $s$. ▪
7.7 Extensions to Max Flow
Circulation with Demands

Circulation with demands.
- Directed graph $G = (V, E)$.
- Edge capacities $c(e), e \in E$.
- Node supply and demands $d(v), v \in V$.

\[
\text{demand if } d(v) > 0; \text{ supply if } d(v) < 0; \text{ transshipment if } d(v) = 0
\]

Def. A circulation is a function that satisfies:
- For each $e \in E$: $0 \leq f(e) \leq c(e)$ (capacity)
- For each $v \in V$: $\sum_{e \text{ in to } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$ (conservation)

Circulation problem: given $(V, E, c, d)$, does there exist a circulation?
Circulation with Demands

**Necessary condition:** sum of supplies = sum of demands.

\[
\sum_{v : d(v) > 0} d(v) = \sum_{v : d(v) < 0} -d(v) =: D
\]

**Pf.** Sum conservation constraints for every demand node \( v \).
Circulation with Demands

Max flow formulation.

G:

-7 10 6 -8 7 -6

3 10 7 4 4

9 0

11

demand

supply
Circulation with Demands

Max flow formulation.
- Add new source $s$ and sink $t$.
- For each $v$ with $d(v) < 0$, add edge $(s, v)$ with capacity $-d(v)$.
- For each $v$ with $d(v) > 0$, add edge $(v, t)$ with capacity $d(v)$.
- Claim: $G$ has circulation iff $G'$ has max flow of value $D$.

$G'$:

- saturates all edges leaving $s$ and entering $t$
- supply
- demand
Circulation with Demands

**Integrality theorem.** If all capacities and demands are integers, and there exists a circulation, then there exists one that is integer-valued.

**Pf.** Follows from max flow formulation and integrality theorem for max flow.

**Characterization.** Given \((V, E, c, d)\), there does not exist a circulation iff there exists a node partition \((A, B)\) such that \(\sum_{v \in B} d_v > \text{cap}(A, B)\)

**Pf idea.** Look at min cut in \(G'\).

\[ 	ext{demand by nodes in } B \text{ exceeds supply of nodes in } B \text{ plus max capacity of edges going from } A \text{ to } B \]
Circulation with Demands and Lower Bounds

Feasible circulation.
- Directed graph $G = (V, E)$.
- Edge capacities $c(e)$ and lower bounds $\ell(e)$, $e \in E$.
- Node supply and demands $d(v)$, $v \in V$.

Def. A circulation is a function that satisfies:
- For each $e \in E$: $\ell(e) \leq f(e) \leq c(e)$ (capacity)
- For each $v \in V$: $\sum_{e \text{ in to } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$ (conservation)

Circulation problem with lower bounds. Given $(V, E, \ell, c, d)$, does there exists a a circulation?
Idea. Model lower bounds with demands.
- Send \( \ell(e) \) units of flow along edge \( e \).
- Update demands of both endpoints.

\[ \begin{align*}
\text{lower bound} & \quad \text{upper bound} \\
 v & \quad [2, 9] \\
d(v) & \quad d(w)
\end{align*} \]

\[ \begin{align*}
\text{capacity} \\
 v & \quad 7 \\
d(v) + 2 & \quad d(w) - 2
\end{align*} \]

Theorem. There exists a circulation in \( G \) iff there exists a circulation in \( G' \). If all demands, capacities, and lower bounds in \( G \) are integers, then there is a circulation in \( G \) that is integer-valued.

Pf sketch. \( f(e) \) is a circulation in \( G \) iff \( f'(e) = f(e) - \ell(e) \) is a circulation in \( G' \).
7.8 Survey Design
Survey Design

Survey design.

- Design survey asking $n_1$ consumers about $n_2$ products.
- Can only survey consumer $i$ about product $j$ if they own it.
- Ask consumer $i$ between $c_i$ and $c_i'$ questions.
- Ask between $p_j$ and $p_j'$ consumers about product $j$.

Goal. Design a survey that meets these specs, if possible.

Bipartite perfect matching. Special case when $c_i = c_i' = p_i = p_i' = 1$. 

one survey question per product
**Algorithm.** Formulate as a circulation problem with lower bounds.

- Include an edge \((i, j)\) if consumer \(j\) owns product \(i\).
- Integer circulation \(\iff\) feasible survey design.
Algorithms — April 4

Today
  ▶ Applications for min-cut

Friday, April 6
  ▶ Baseball!

Monday, April 9
  ▶ Problems due

Friday, April 13
  ▶ Quiz on Chapter 6

Contact information
  ▶ Web: cs.uwlax.edu/~jmaraist
  ▶ Email: jmaraist@uwlax.edu
    ▶ Expect replies within a business day (but often same-day)
  ▶ End-of-semester office hours
    ▶ Wednesday 2: 3-5pm
    ▶ Friday 4: 2-4pm
    ▶ Monday 7: 10am-noon
  ▶ Or by appointment
    ▶ Email at least a day ahead
    ▶ Send several times when you are available
    ▶ Always describe what you need to discuss
      ▶ So I can be prepared
      ▶ Because advising or paperwork often require no meeting
7.10 Image Segmentation
Image Segmentation

Image segmentation.
- Central problem in image processing.
- Divide image into coherent regions.

Ex: Three people standing in front of complex background scene. Identify each person as a coherent object.
Image Segmentation

Foreground / background segmentation.
- Label each pixel in picture as belonging to foreground or background.
- \( V = \text{set of pixels}, \ E = \text{pairs of neighboring pixels} \).
- \( a_i \geq 0 \) is likelihood pixel \( i \) in foreground.
- \( b_i \geq 0 \) is likelihood pixel \( i \) in background.
- \( p_{ij} \geq 0 \) is separation penalty for labeling one of \( i \) and \( j \) as foreground, and the other as background.

Goals.
- Accuracy: if \( a_i > b_i \) in isolation, prefer to label \( i \) in foreground.
- Smoothness: if many neighbors of \( i \) are labeled foreground, we should be inclined to label \( i \) as foreground.
- Find partition \((A, B)\) that maximizes: \( \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{(i,j) \in E} p_{ij} \) where \( |A \cap \{i,j\}| = 1 \).
Image Segmentation

Formulate as min cut problem.
- Maximization.
- No source or sink.
- Undirected graph.

Turn into minimization problem.

- Maximizing

\[
\sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{(i, j) \in E} p_{ij} = 1
\]

is equivalent to minimizing

\[
\left( \sum_{i \in V} a_i + \sum_{j \in V} b_j \right) - \sum_{i \in A} a_i - \sum_{j \in B} b_j + \sum_{(i, j) \in E} p_{ij} = 1
\]

or alternatively

\[
\sum_{j \in B} a_j + \sum_{i \in A} b_i + \sum_{(i, j) \in E} p_{ij} = 1
\]
Image Segmentation

Formulate as min cut problem.
- $G' = (V', E')$.
- Add source to correspond to foreground; add sink to correspond to background.
- Use two anti-parallel edges instead of undirected edge.
Image Segmentation

Consider min cut \((A, B)\) in \(G'\).

- \(A = \text{foreground.}\)

\[
\text{cap}(A, B) = \sum_{j \in B} a_j + \sum_{i \in A} b_i + \sum_{(i, j) \in E} p_{ij} \quad \text{if \(i\) and \(j\) on different sides,}
\]

\[
p_{ij} \text{ counted exactly once}
\]

- Precisely the quantity we want to minimize.
7.11 Project Selection
Projects with prerequisites.

- Set $P$ of possible projects. Project $v$ has associated revenue $p_v$.
  - some projects generate money: create interactive e-commerce interface, redesign web page
  - others cost money: upgrade computers, get site license

- Set of prerequisites $E$. If $(v, w) \in E$, can't do project $v$ and unless also do project $w$.

- A subset of projects $A \subseteq P$ is feasible if the prerequisite of every project in $A$ also belongs to $A$.

Project selection. Choose a feasible subset of projects to maximize revenue.
Project Selection: Prerequisite Graph

Prerequisite graph.
- Include an edge from v to w if can't do v without also doing w.
- \( \{v, w, x\} \) is feasible subset of projects.
- \( \{v, x\} \) is infeasible subset of projects.
**Min cut formulation.**

- Assign capacity $\infty$ to all prerequisite edge.
- Add edge $(s, v)$ with capacity $p_v$ if $p_v > 0$.
- Add edge $(v, t)$ with capacity $-p_v$ if $p_v < 0$.
- For notational convenience, define $p_s = p_t = 0$. 

![Diagram of network flow problem](image)
Claim. \((A, B)\) is min cut iff \(A - \{s\}\) is optimal set of projects.

- Infinite capacity edges ensure \(A - \{s\}\) is feasible.
- Max revenue because:

\[
\text{cap}(A, B) = \sum_{v \in B: p_v > 0} p_v + \sum_{v \in A: p_v < 0} (-p_v) \\
= \sum_{v: p_v > 0} p_v - \sum_{v \in A} p_v
\]

constant
Open-pit mining. (studied since early 1960s)

- Blocks of earth are extracted from surface to retrieve ore.
- Each block \( v \) has net value \( p_v = \text{value of ore} - \text{processing cost} \).
- Can't remove block \( v \) before \( w \) or \( x \).
Algorithms — April 6

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting

Today
▶ Baseball!

Friday, April 13
▶ Quiz on Chapter 6

Monday, April 16
▶ Problems due
Baseball Elimination

<table>
<thead>
<tr>
<th>Team</th>
<th>Wins $w_i$</th>
<th>Losses $l_i$</th>
<th>To play $r_i$</th>
<th>Against $= r_{ij}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Atl</td>
</tr>
<tr>
<td>Atlanta</td>
<td>83</td>
<td>71</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>Philly</td>
<td>80</td>
<td>79</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>New York</td>
<td>78</td>
<td>78</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Montreal</td>
<td>77</td>
<td>82</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Which teams have a chance of finishing the season with most wins?

- Montreal eliminated since it can finish with at most 80 wins, but Atlanta already has 83.
- $w_i + r_i < w_j \implies$ team i eliminated.
- Only reason sports writers appear to be aware of.
- Sufficient, but not necessary!
### Baseball Elimination

<table>
<thead>
<tr>
<th>Team</th>
<th>Wins</th>
<th>Losses</th>
<th>To play</th>
<th>Against = $r_{ij}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$w_i$</td>
<td>$l_i$</td>
<td>$r_i$</td>
<td>Atl</td>
</tr>
<tr>
<td>Atlanta</td>
<td>83</td>
<td>71</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>Philly</td>
<td>80</td>
<td>79</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>New York</td>
<td>78</td>
<td>78</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Montreal</td>
<td>77</td>
<td>82</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Which teams have a chance of finishing the season with most wins?
- Philly can win 83, but still eliminated . . .
- If Atlanta loses a game, then some other team wins one.

**Remark.** Answer depends not just on how many games already won and left to play, but also on whom they're against.
Baseball Elimination

SPORTING GI

49ers, Young Get Big Break

Quarterback may

By Gary Swan
Chronicle Staff Writer

The bye week has come at a perfect time for the 49ers and quarterback Steve Young. If they had a game next Sunday, there's a good chance Young would not play.

GIANTS OFFICIALLY LEAVE THE NL WEST RACE

By Nancy Gay
Chronicle Staff Writer

With the smack of another National League West bat 500 miles away, the Giants' run at the division title ended last night. Just as they were handing the visiting St. Louis Cardinals an even bigger lead in the NL Central.

CARDINALS 6
GIANTS 2

In San Diego, Greg Vaughn's three-run homer in the eighth pushed the Padres over the Pirates and officially shoved the rest of the Giants' season into the background. On the heels of their tedious 6-2 loss before an announced crowd of 10,307 at Candlestick Park, the Giants fell 1 1/2 games off the lead.

As it is, the worst the Padres (80-65) can finish is 60-82. The Giants have fallen to 59-83 with 20 games left; they cannot win 80 games. Coming off a miserable 2-9 mark on a three-city road trip that saw their road record drop to 27-47, the Giants were hoping to get off on the right foot in their longest homestand of the year (15 games, 14 days).

Financing in Place
For Giants' New Stadium

SEE PAGE B1, MAIN NEWS

“Where we are, you’re going to be eliminated sooner or later,” Baker said quietly. “But it doesn’t alter the fact that we’ve still got to play ball. You’ve still got to play hard, the fans come out to watch you play. You’ve got to play for the fact of loving to play, no matter where you are in the standings.”

“Your got to play the role of spoiler, to not make it easier on GIANTS. Page D5 Col. 3
Baseball Elimination

Baseball elimination problem.
- Set of teams $S$.
- Distinguished team $s \in S$.
- Team $x$ has won $w_x$ games already.
- Teams $x$ and $y$ play each other $r_{xy}$ additional times.
- Is there any outcome of the remaining games in which team $s$ finishes with the most (or tied for the most) wins?
Can team 3 finish with most wins?

- Assume team 3 wins all remaining games \( \Rightarrow w_3 + r_3 \) wins.
- Divvy remaining games so that all teams have \( \leq w_3 + r_3 \) wins.

Baseball Elimination: Max Flow Formulation

\[
\begin{align*}
\text{team nodes} & \quad \text{game nodes} \\
2-5 & \quad 1-5 \\
1-4 & \quad 2-4 \\
1-2 & \\
\text{s} & \quad r_{24} = 7 \\
& \quad \text{team 4 can still win this many more games}
\end{align*}
\]
**Theorem.** Team 3 is not eliminated iff max flow saturates all edges leaving source.

- Integrality theorem $\Rightarrow$ each remaining game between $x$ and $y$ added to number of wins for team $x$ or team $y$.
- Capacity on $(x, t)$ edges ensure no team wins too many games.

Baseball Elimination: Max Flow Formulation

![Diagram showing a network flow problem with nodes and edges labeled with game nodes and team nodes. The diagram includes a flow from source $s$ to sink $t$ with capacities and labels indicating games left and games won.](Diagram.png)
**Baseball Elimination: Explanation for Sports Writers**

<table>
<thead>
<tr>
<th>Team</th>
<th>Wins $w_i$</th>
<th>Losses $l_i$</th>
<th>To play $r_i$</th>
<th>Against = $r_{ij}$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NY</td>
</tr>
<tr>
<td>NY</td>
<td>75</td>
<td>59</td>
<td>28</td>
<td>-</td>
</tr>
<tr>
<td>Baltimore</td>
<td>71</td>
<td>63</td>
<td>28</td>
<td>3</td>
</tr>
<tr>
<td>Boston</td>
<td>69</td>
<td>66</td>
<td>27</td>
<td>8</td>
</tr>
<tr>
<td>Toronto</td>
<td>63</td>
<td>72</td>
<td>27</td>
<td>7</td>
</tr>
<tr>
<td>Detroit</td>
<td>49</td>
<td>86</td>
<td>27</td>
<td>3</td>
</tr>
</tbody>
</table>

*AL East: August 30, 1996*

**Which teams have a chance of finishing the season with most wins?**
- Detroit could finish season with $49 + 27 = 76$ wins.
Baseball Elimination: Explanation for Sports Writers

Which teams have a chance of finishing the season with most wins?
- Detroit could finish season with \(49 + 27 = 76\) wins.

Certificate of elimination. \(R = \{ \text{NY, Bal, Bos, Tor}\}\)
- Have already won \(w(R) = 278\) games.
- Must win at least \(r(R) = 27\) more.
- Average team in \(R\) wins at least \(305/4 > 76\) games.
Certificate of elimination.

\[ T \subseteq S, \quad w(T) := \sum_{i \in T} w_i, \quad g(T) := \sum_{\{x,y\} \subseteq T} g_{xy}, \]

If \( \frac{w(T) + g(T)}{|T|} > w_z + g_z \) then \( z \) is eliminated (by subset \( T \)).

Theorem. [Hoffman-Rivlin 1967] Team \( z \) is eliminated iff there exists a subset \( T^* \) that eliminates \( z \).

Proof idea. Let \( T^* \) = team nodes on source side of min cut.
Baseball Elimination: Explanation for Sports Writers

Pf of theorem.

- Use max flow formulation, and consider min cut \((A, B)\).
- Define \(T^*\) = team nodes on source side of min cut.
- Observe \(x-y \in A\) iff both \(x \in T^*\) and \(y \in T^*\).
  - infinite capacity edges ensure if \(x-y \in A\) then \(x \in A\) and \(y \in A\)
  - if \(x \in A\) and \(y \in A\) but \(x-y \in T\), then adding \(x-y\) to \(A\) decreases capacity of cut
Contact information

- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
Classify Problems According to Computational Requirements

Q. Which problems will we be able to solve in practice?


<table>
<thead>
<tr>
<th>Yes</th>
<th>Probably no</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest path</td>
<td>Longest path</td>
</tr>
<tr>
<td>Matching</td>
<td>3D-matching</td>
</tr>
<tr>
<td>Min cut</td>
<td>Max cut</td>
</tr>
<tr>
<td>2-SAT</td>
<td>3-SAT</td>
</tr>
<tr>
<td>Planar 4-color</td>
<td>Planar 3-color</td>
</tr>
<tr>
<td>Bipartite vertex cover</td>
<td>Vertex cover</td>
</tr>
<tr>
<td>Primality testing</td>
<td>Factoring</td>
</tr>
</tbody>
</table>
Classify Problems

Desiderata. Classify problems according to those that can be solved in polynomial-time and those that cannot.

Provably requires exponential-time.

- Given a Turing machine, does it halt in at most k steps?
- Given a board position in an n-by-n generalization of chess, can black guarantee a win?

Frustrating news. Huge number of fundamental problems have defied classification for decades.

This chapter. Show that these fundamental problems are "computationally equivalent" and appear to be different manifestations of one really hard problem.
Polynomial-Time Reduction

Desiderata'. Suppose we could solve $X$ in polynomial-time. What else could we solve in polynomial time?

Reduction. Problem $X$ polynomial reduces to problem $Y$ if arbitrary instances of problem $X$ can be solved using:

- Polynomial number of standard computational steps, plus
- Polynomial number of calls to oracle that solves problem $Y$.

Notation. $X \leq_p Y$.

Remarks.
- We pay for time to write down instances sent to black box $\Rightarrow$ instances of $Y$ must be of polynomial size.
- Note: Cook reducibility.

computational model supplemented by special piece of hardware that solves instances of $Y$ in a single step

in contrast to Karp reductions
Polynomial-Time Reduction

**Purpose.** Classify problems according to relative difficulty.

**Design algorithms.** If $X \leq_p Y$ and $Y$ can be solved in polynomial-time, then $X$ can also be solved in polynomial time.

**Establish intractability.** If $X \leq_p Y$ and $X$ cannot be solved in polynomial-time, then $Y$ cannot be solved in polynomial time.

**Establish equivalence.** If $X \leq_p Y$ and $Y \leq_p X$, we use notation $X \equiv_p Y$. 

up to cost of reduction
Reduction By Simple Equivalence

Basic reduction strategies.

- Reduction by simple equivalence.
- Reduction from special case to general case.
- Reduction by encoding with gadgets.
INDEPENDENT SET: Given a graph $G = (V, E)$ and an integer $k$, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in $S$?

Ex. Is there an independent set of size $\geq 6$? Yes.
Ex. Is there an independent set of size $\geq 7$? No.
**Vertex Cover**

**VERTEX COVER:** Given a graph $G = (V, E)$ and an integer $k$, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge, at least one of its endpoints is in $S$?

*Ex.* Is there a vertex cover of size $\leq 4$? Yes.

*Ex.* Is there a vertex cover of size $\leq 3$? No.
Claim. \textsc{vertex-cover} \equiv_p \textsc{independent-set}.

Pf. We show $S$ is an independent set iff $V - S$ is a vertex cover.
Claim. VERTEX-COVER \equiv_p INDEPENDENT-SET.

Pf. We show \( S \) is an independent set iff \( V - S \) is a vertex cover.

\[ \Rightarrow \]

- Let \( S \) be any independent set.
- Consider an arbitrary edge \((u, v)\).
- \( S \) independent \( \Rightarrow u \not\in S \) or \( v \not\in S \) \( \Rightarrow u \in V - S \) or \( v \in V - S \).
- Thus, \( V - S \) covers \((u, v)\).

\[ \Leftarrow \]

- Let \( V - S \) be any vertex cover.
- Consider two nodes \( u \in S \) and \( v \in S \).
- Observe that \((u, v) \not\in E\) since \( V - S \) is a vertex cover.
- Thus, no two nodes in \( S \) are joined by an edge \( \Rightarrow S \) independent set. \( \blacksquare \)
Reduction from Special Case to General Case

Basic reduction strategies.

- Reduction by simple equivalence.
- Reduction from special case to general case.
- Reduction by encoding with gadgets.
SET COVER: Given a set $U$ of elements, a collection $S_1, S_2, \ldots, S_m$ of subsets of $U$, and an integer $k$, does there exist a collection of $\leq k$ of these sets whose union is equal to $U$?

Sample application.

- $m$ available pieces of software.
- Set $U$ of $n$ capabilities that we would like our system to have.
- The $i$th piece of software provides the set $S_i \subseteq U$ of capabilities.
- Goal: achieve all $n$ capabilities using fewest pieces of software.

Ex:

\[
\begin{align*}
    U &= \{1, 2, 3, 4, 5, 6, 7\} \\
    k &= 2 \\
    S_1 &= \{3, 7\} \\
    S_2 &= \{3, 4, 5, 6\} \\
    S_3 &= \{1\} \\
    S_4 &= \{2, 4\} \\
    S_5 &= \{5\} \\
    S_6 &= \{1, 2, 6, 7\}
\end{align*}
\]
**Claim.** VERTEX-COVER $\leq_p$ SET-COVER.

**Pf.** Given a VERTEX-COVER instance $G = (V, E)$, $k$, we construct a set cover instance whose size equals the size of the vertex cover instance.

**Construction.**

- Create SET-COVER instance:
  - $k = k$, $U = E$, $S_v = \{e \in E : e \text{ incident to } v\}$

- Set-cover of size $\leq k$ iff vertex cover of size $\leq k$. □
Algorithms — April 11

Today

▶ Reduction with gadgets

Friday, April 13

▶ Quiz on Chapter 6

Monday, April 16

▶ Problems due

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting
First a quick review
  ▶ Independent set
  ▶ Polynomial reduction
Then a new problem — boolean formula satisfiability
INDEPENDENT SET: Given a graph $G = (V, E)$ and an integer $k$, is there a subset of vertices $S \subseteq V$ such that $|S| \geq k$, and for each edge at most one of its endpoints is in $S$?

Ex. Is there an independent set of size $\geq 6$? Yes.
Ex. Is there an independent set of size $\geq 7$? No.
Desiderata'. Suppose we could solve X in polynomial-time. What else could we solve in polynomial time?

Reduction. Problem X polynomial reduces to problem Y if arbitrary instances of problem X can be solved using:

- Polynomial number of standard computational steps, plus
- Polynomial number of calls to oracle that solves problem Y.

Notation. $X \leq_P Y$.

Remarks.
- We pay for time to write down instances sent to black box $\Rightarrow$ instances of Y must be of polynomial size.
- Note: Cook reducibility.

in contrast to Karp reductions

don’t confuse with reduces from computational model supplemented by special piece of hardware that solves instances of Y in a single step

Satisfiability

Literal: A Boolean variable or its negation. \( x_i \) or \( \overline{x_i} \)

Clause: A disjunction of literals. \( C_j = x_1 \lor \overline{x_2} \lor x_3 \)

Conjunctive normal form: A propositional formula \( \Phi \) that is the conjunction of clauses. \(\Phi = C_1 \land C_2 \land C_3 \land C_4 \)

SAT: Given CNF formula \( \Phi \), does it have a satisfying truth assignment?

3-SAT: SAT where each clause contains exactly 3 literals.

Ex: \( (\overline{x_1} \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor x_3) \land (x_2 \lor x_3) \land (\overline{x_1} \lor \overline{x_2} \lor \overline{x_3}) \)

Yes: \( x_1 = \text{true}, x_2 = \text{true}, x_3 = \text{false} \).
Claim. \(3\text{-SAT} \leq_{P} \text{INDEPENDENT-SET}\).

Pf. Given an instance \(\Phi\) of 3-SAT, we construct an instance \((G, k)\) of \(\text{INDEPENDENT-SET}\) that has an independent set of size \(k\) iff \(\Phi\) is satisfiable.

Construction.
- \(G\) contains 3 vertices for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.

\[
\Phi = (\overline{x_1} \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_1} \lor x_2 \lor x_4)
\]
Claim. $G$ contains independent set of size $k = |\Phi|$ iff $\Phi$ is satisfiable.

Pf. $\Rightarrow$ Let $S$ be independent set of size $k$.
- $S$ must contain exactly one vertex in each triangle.
- Set these literals to true. and any other variables in a consistent way
- Truth assignment is consistent and all clauses are satisfied.

Pf $\Leftarrow$ Given satisfying assignment, select one true literal from each triangle. This is an independent set of size $k$.

$G$

$k = 3$

$\Phi = (\overline{x_1} \lor x_2 \lor x_3) \land (x_1 \lor \overline{x_2} \lor x_3) \land (\overline{x_1} \lor x_2 \lor x_4)$
Basic reduction strategies.

- Simple equivalence: \textsc{Independent-set} \equiv_p \textsc{Vertex-cover}.
- Special case to general case: \textsc{Vertex-cover} \leq_p \textsc{Set-cover}.
- Encoding with gadgets: \textsc{3-Sat} \leq_p \textsc{Independent-set}.

Transitivity. If \(X \leq_p Y\) and \(Y \leq_p Z\), then \(X \leq_p Z\).

Pf idea. Compose the two algorithms.

Ex: \textsc{3-Sat} \leq_p \textsc{Independent-set} \leq_p \textsc{Vertex-cover} \leq_p \textsc{Set-cover}.
8.3 Definition of NP
Decision Problems

Decision problem.
- $X$ is a set of strings.
- Instance: string $s$.
- Algorithm $A$ solves problem $X$: $A(s) = \text{yes}$ iff $s \in X$.

Polynomial time. Algorithm $A$ runs in poly-time if for every string $s$, $A(s)$ terminates in at most $p(|s|)$ "steps", where $p(\cdot)$ is some polynomial.

PRIMES: $X = \{ 2, 3, 5, 7, 11, 13, 17, 23, 29, 31, 37, \ldots \}$

# Definition of P

**P.** Decision problems for which there is a poly-time algorithm.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Description</th>
<th>Algorithm</th>
<th>Yes</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTIPLE</td>
<td>Is (x) a multiple of (y)?</td>
<td>Grade school division</td>
<td>51, 17</td>
<td>51, 16</td>
</tr>
<tr>
<td>RELPRIME</td>
<td>Are (x) and (y) relatively prime?</td>
<td>Euclid (300 BCE)</td>
<td>34, 39</td>
<td>34, 51</td>
</tr>
<tr>
<td>PRIMES</td>
<td>Is (x) prime?</td>
<td>AKS (2002)</td>
<td>53</td>
<td>51</td>
</tr>
<tr>
<td>EDIT-DISTANCE</td>
<td>Is the edit distance between (x) and (y) less than 5?</td>
<td>Dynamic programming</td>
<td>neither</td>
<td>acgggt ttttta</td>
</tr>
<tr>
<td>LSOLVE</td>
<td>Is there a vector (x) that satisfies (Ax = b)?</td>
<td>Gauss-Edmonds elimination</td>
<td>![Matrix]</td>
<td>![Matrix]</td>
</tr>
</tbody>
</table>
Certification algorithm intuition.

- Certifier views things from "managerial" viewpoint.
- Certifier doesn't determine whether \( s \in X \) on its own; rather, it checks a proposed proof \( t \) that \( s \in X \).

**Def.** Algorithm \( C(s, t) \) is a **certifier** for problem \( X \) if for every string \( s \), \( s \in X \) iff there exists a string \( t \) such that \( C(s, t) = \text{yes} \).

**NP.** Decision problems for which there exists a **poly-time** certifier.

\( C(s, t) \) is a poly-time algorithm and \( |t| \leq p(|s|) \) for some polynomial \( p(\cdot) \).

**Remark.** NP stands for **nondeterministic** polynomial-time.
Certifiers and Certificates: Composite

**COMPOSITES.** Given an integer s, is s composite?

**Certificate.** A nontrivial factor t of s. Note that such a certificate exists iff s is composite. Moreover |t| ≤ |s|.

**Certifier.**

```java
boolean C(s, t) {
    if (t ≤ 1 or t ≥ s)
        return false
    else if (s is a multiple of t)
        return true
    else
        return false
}
```

**Instance.** s = 437,669.

**Certificate.** t = 541 or 809.  437,669 = 541 × 809

**Conclusion.** COMPOSITES is in NP.
Certifiers and Certificates: 3-Satisfiability

**SAT.** Given a CNF formula $\Phi$, is there a satisfying assignment?

**Certificate.** An assignment of truth values to the $n$ boolean variables.

**Certifier.** Check that each clause in $\Phi$ has at least one true literal.

Ex.

\[
\left( \overline{x_1} \lor x_2 \lor x_3 \right) \land \left( x_1 \lor \overline{x_2} \lor x_3 \right) \land \left( x_1 \lor x_2 \lor x_4 \right) \land \left( \overline{x_1} \lor \overline{x_3} \lor \overline{x_4} \right)
\]

instance $s$

$\begin{align*}
x_1 &= 1, \ x_2 &= 1, \ x_3 &= 0, \ x_4 &= 1
\end{align*}$

certificate $\top$

**Conclusion.** SAT is in NP.
Certifiers and Certificates: Hamiltonian Cycle

**HAM-CYCLE.** Given an undirected graph $G = (V, E)$, does there exist a simple cycle $C$ that visits every node?

**Certificate.** A permutation of the $n$ nodes.

**Certifier.** Check that the permutation contains each node in $V$ exactly once, and that there is an edge between each pair of adjacent nodes in the permutation.

**Conclusion.** HAM-CYCLE is in NP.
P, NP, EXP

P. Decision problems for which there is a poly-time algorithm.
EXP. Decision problems for which there is an exponential-time algorithm.
NP. Decision problems for which there is a poly-time certifier.

Claim. P ⊆ NP.
Pf. Consider any problem X in P.
   ■ By definition, there exists a poly-time algorithm A(s) that solves X.
   ■ Certificate: t = ε, certifier C(s, t) = A(s).

Claim. NP ⊆ EXP.
Pf. Consider any problem X in NP.
   ■ By definition, there exists a poly-time certifier C(s, t) for X.
   ■ To solve input s, run C(s, t) on all strings t with |t| ≤ p(|s|).
   ■ Return yes, if C(s, t) returns yes for any of these.
Algorithms — April 13

Today

▶ Intractability
▶ Quiz on Chapter 6

Monday, April 16

▶ Problems due: Ch. 7, #3-#6

Friday, April 20

▶ Quiz on Chapter 8
  ▶ Definitions and short-answer questions
  ▶ Closed-book

Monday, April 23

▶ Problems due: Ch. 7, #8, #9

Contact information

▶ Web: cs.uwlax.edu/~jmarais
▶ Email: jmarais@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting
Theory at UWL

The ideas behind computing, rather than the engineering or applications

Theory and theory-inclusive classes at UWL include:

- **Complexity and computability:**
  - CS353 Analysis of Algorithms
  - CS453 Theory of Computation
    - Automata show up in CS442 Compiler Construction
  - MTH317 Graph Theory (fall, odd years)

- **Numerical methods:**
  - CS351 Simulation
  - CS419 Optimization
  - CS419 Machine Learning (this fall)
  - MTH311 Number Theory

- **Languages, logic and semantics**
  - CS421 Programming Language Concepts
  - MTH411/412 Abstract algebra, as a basis for the formal semantics of languages
  - PHL302 Symbolic Logic (spring, odd years)

There is also **CS395/499** — independent study/research

- Can be tricky with theory
- But always ask!
- Faculty particularly interested/experienced in more theoretical topics include Allen, Maraist, Matthias (arriving in the fall), J Sauppe, Senger
The Main Question: P Versus NP

Does $P = NP$? [Cook 1971, Edmonds, Levin, Yablonski, Gödel]
- Is the decision problem as easy as the certification problem?
- Clay $1$ million prize.

If $P = NP$:
- Efficient algorithms for 3-COLOR, TSP, FACTOR, SAT, ...

If $P \neq NP$:
- RSA cryptography would break (and potentially collapse economy)

Consensus opinion on $P = NP$? Probably no.
Polynomial Transformation

Def. Problem X **polynomial reduces** (Cook) to problem Y if arbitrary instances of problem X can be solved using:

- Polynomial number of standard computational steps, plus
- Polynomial number of calls to oracle that solves problem Y.

Def. Problem X **polynomial transforms** (Karp) to problem Y if given any input x to X, we can construct an input y such that x is a *yes* instance of X iff y is a *yes* instance of Y.

Note. Polynomial transformation is polynomial reduction with just one call to oracle for Y, exactly at the end of the algorithm for X. Almost all previous reductions were of this form.

Open question. Are these two concepts the same?

we abuse notation \(\leq_p\) and blur distinction
NP-Complete

**NP-complete.** A problem $Y$ in NP with the property that for every problem $X$ in NP, $X \leq_P Y$.

**Theorem.** Suppose $Y$ is an NP-complete problem. Then $Y$ is solvable in poly-time iff $P = NP$.

**Pf.** $\Leftarrow$ If $P = NP$ then $Y$ can be solved in poly-time since $Y$ is in NP.

**Pf.** $\Rightarrow$ Suppose $Y$ can be solved in poly-time.

- Let $X$ be any problem in NP. Since $X \leq_P Y$, we can solve $X$ in poly-time. This implies $NP \subseteq P$.
- We already know $P \subseteq NP$. Thus $P = NP$. □

**Fundamental question.** Do there exist "natural" NP-complete problems?
**Circuit Satisfiability**

**CIRCUIT-SAT.** Given a combinational circuit built out of AND, OR, and NOT gates, is there a way to set the circuit inputs so that the output is 1?

![Circuit Diagram](image)

**yes: 1 0 1**
The "First" NP-Complete Problem

**Theorem.** CIRCUIT-SAT is NP-complete. [Cook 1971, Levin 1973]

**Pf.** (sketch)

- Any algorithm that takes a fixed number of bits $n$ as input and produces a yes/no answer can be represented by such a circuit. Moreover, if algorithm takes poly-time, then circuit is of poly-size.

  sketchy part of proof: fixing the number of bits is important, and reflects basic distinction between algorithms and circuits

- Consider some problem $X$ in NP. It has a poly-time certifier $C(s, t)$. To determine whether $s$ is in $X$, need to know if there exists a certificate $t$ of length $p(|s|)$ such that $C(s, t) = \text{yes}$.

- View $C(s, t)$ as an algorithm on $|s| + p(|s|)$ bits (input $s$, certificate $t$) and convert it into a poly-size circuit $K$.
  - first $|s|$ bits are hard-coded with $s$
  - remaining $p(|s|)$ bits represent bits of $t$

- Circuit $K$ is satisfiable iff $C(s, t) = \text{yes}$. 
Example

Ex. Construction below creates a circuit $K$ whose inputs can be set so that $K$ outputs true iff graph $G$ has an independent set of size 2.

$G = (V, E), n = 3$

\[
\begin{align*}
\binom{n}{2} & \text{ hard-coded inputs (graph description)} \\
& \text{ } \\
n & \text{ inputs (nodes in independent set)}
\end{align*}
\]
Establishing NP-Completeness

Remark. Once we establish first "natural" NP-complete problem, others fall like dominoes.

Recipe to establish NP-completeness of problem Y.
- Step 1. Show that Y is in NP.
- Step 2. Choose an NP-complete problem X.
- Step 3. Prove that $X \leq_p Y$.

Justification. If X is an NP-complete problem, and Y is a problem in NP with the property that $X \leq_p Y$ then Y is NP-complete.

Pf. Let W be any problem in NP. Then $W \leq_p X \leq_p Y$.
- By transitivity, $W \leq_p Y$.
- Hence Y is NP-complete. •
3-SAT is NP-Complete

**Theorem.** 3-SAT is NP-complete.

**Pf.** Suffices to show that CIRCUIT-SAT $\leq_p$ 3-SAT since 3-SAT is in NP.

- Let $K$ be any circuit.
- Create a 3-SAT variable $x_i$ for each circuit element $i$.
- Make circuit compute correct values at each node:
  - $x_2 = \neg x_3 \Rightarrow$ add 2 clauses: $x_2 \lor x_3$, $\neg x_2 \lor \neg x_3$
  - $x_1 = x_4 \lor x_5 \Rightarrow$ add 3 clauses: $x_1 \lor \neg x_4$, $x_1 \lor \neg x_5$, $\neg x_1 \lor x_4 \lor x_5$
  - $x_0 = x_1 \land x_2 \Rightarrow$ add 3 clauses: $\neg x_0 \lor x_1$, $\neg x_0 \lor x_2$, $\neg x_0 \lor \neg x_1 \lor \neg x_2$

- Hard-coded input values and output value.
  - $x_5 = 0 \Rightarrow$ add 1 clause: $\neg x_5$
  - $x_0 = 1 \Rightarrow$ add 1 clause: $x_0$

- Final step: turn clauses of length < 3 into clauses of length exactly 3.
Observation. All problems below are NP-complete and polynomial reduce to one another!
Some NP-Complete Problems

Six basic genres of NP-complete problems and paradigmatic examples.

- Packing problems: SET-PACKING, INDEPENDENT SET.
- Covering problems: SET-COVER, VERTEX-COVER.
- Constraint satisfaction problems: SAT, 3-SAT.
- Sequencing problems: HAMILTONIAN-CYCLE, TSP.
- Partitioning problems: 3D-MATCHING 3-COLOR.
- Numerical problems: SUBSET-SUM, KNAPSACK.

Practice. Most NP problems are either known to be in P or NP-complete.

Notable exceptions. Factoring, graph isomorphism, Nash equilibrium.
Extent and Impact of NP-Completeness

Extent of NP-completeness. [Papadimitriou 1995]

- Prime intellectual export of CS to other disciplines.
- 6,000 citations per year (title, abstract, keywords).
  - more than "compiler", "operating system", "database"
- Broad applicability and classification power.
- "Captures vast domains of computational, scientific, mathematical endeavors, and seems to roughly delimit what mathematicians and scientists had been aspiring to compute feasibly."

NP-completeness can guide scientific inquiry.

- 1926: Ising introduces simple model for phase transitions.
- 1944: Onsager solves 2D case in tour de force.
- 19xx: Feynman and other top minds seek 3D solution.
More Hard Computational Problems

**Aerospace engineering:** optimal mesh partitioning for finite elements.

**Biology:** protein folding.

**Chemical engineering:** heat exchanger network synthesis.

**Civil engineering:** equilibrium of urban traffic flow.

**Economics:** computation of arbitrage in financial markets with friction.

**Electrical engineering:** VLSI layout.

**Environmental engineering:** optimal placement of contaminant sensors.

**Financial engineering:** find minimum risk portfolio of given return.

**Game theory:** find Nash equilibrium that maximizes social welfare.

**Genomics:** phylogeny reconstruction.

**Mechanical engineering:** structure of turbulence in sheared flows.

**Medicine:** reconstructing 3-D shape from biplane angiocardiogram.

**Operations research:** optimal resource allocation.

**Physics:** partition function of 3-D Ising model in statistical mechanics.

**Politics:** Shapley-Shubik voting power.

**Pop culture:** Minesweeper consistency.

**Statistics:** optimal experimental design.
The Complexity Zoo

A moderated wiki of the state of complexity theory

- The zoo keepers currently list over 500 notable complexity classes
- https://complexityzoo.uwaterloo.ca
- The "petting zoo" is a simpler, introductory subwiki

Some highlights

- **Conjecture**: NP $\neq$ P
- BPP, MA, AM are about randomized algorithms
  - **Conjecture**: BPP=P
- BQP describes problems efficiently solvable by quantum computers
**Observation.** All problems below are NP-complete and polynomial reduce to one another!
Establishing NP-Completeness

**Remark.** Once we establish first "natural" NP-complete problem, others fall like dominoes.

**Recipe to establish NP-completeness of problem Y.**

- Step 1. Show that \( Y \) is in NP.
- Step 2. Choose an NP-complete problem \( X \).
- Step 3. Prove that \( X \leq_p Y \).

**Justification.** If \( X \) is an NP-complete problem, and \( Y \) is a problem in NP with the property that \( X \leq_p Y \) then \( Y \) is NP-complete.

**Pf.** Let \( W \) be any problem in NP. Then \( W \leq_p X \leq_p Y \).

- By transitivity, \( W \leq_p Y \).
- Hence \( Y \) is NP-complete. □
Theorem. 3-SAT is NP-complete.

Pf. Suffices to show that CIRCUIT-SAT ≤ₚ 3-SAT since 3-SAT is in NP.

- Let K be any circuit.
- Create a 3-SAT variable \( x_i \) for each circuit element \( i \).
- Make circuit compute correct values at each node:
  - \( x_2 = \neg x_3 \Rightarrow \) add 2 clauses: \( x_2 \lor x_3, \ x_2 \lor \neg x_3 \)
  - \( x_1 = x_4 \lor x_5 \Rightarrow \) add 3 clauses: \( x_1 \lor \neg x_4, \ x_1 \lor \neg x_5, \ \neg x_1 \lor x_4 \lor x_5 \)
  - \( x_0 = x_1 \land x_2 \Rightarrow \) add 3 clauses: \( \neg x_0 \lor x_1, \ \neg x_0 \lor x_2, \ x_0 \lor \neg x_1 \lor \neg x_2 \)

- Hard-coded input values and output value.
  - \( x_5 = 0 \Rightarrow \) add 1 clause: \( \neg x_5 \)
  - \( x_0 = 1 \Rightarrow \) add 1 clause: \( x_0 \)

- Final step: turn clauses of length < 3 into clauses of length exactly 3.
Asymmetry of NP

Asymmetry of NP. We only need to have short proofs of yes instances.

Ex 1. SAT vs. TAUTOLOGY.
- Can prove a CNF formula is satisfiable by giving such an assignment.
- How could we prove that a formula is not satisfiable?

Ex 2. HAM-CYCLE vs. NO-HAM-CYCLE.
- Can prove a graph is Hamiltonian by giving such a Hamiltonian cycle.
- How could we prove that a graph is not Hamiltonian?

Remark. SAT is NP-complete and $SAT \equiv_p TAUTOLOGY$, but how do we classify TAUTOLOGY?

\[\text{not even known to be in NP}\]
NP and co-NP

**NP.** Decision problems for which there is a poly-time certifier.
*Ex.* SAT, HAM-CYCLE, COMPOSITES.

**Def.** Given a decision problem \( X \), its complement \( \overline{X} \) is the same problem with the yes and no answers reverse.

*Ex.* \( \overline{X} = \{ 0, 1, 4, 6, 8, 9, 10, 12, 14, 15, ... \} \)
\( X = \{ 2, 3, 5, 7, 11, 13, 17, 23, 29, ... \} \)

**co-NP.** Complements of decision problems in NP.
*Ex.* TAUTOLOGY, NO-HAM-CYCLE, PRIMES.
Fundamental question. Does NP = co-NP?
  - Do yes instances have succinct certificates iff no instances do?
  - Consensus opinion: no.

Theorem. If NP ≠ co-NP, then P ≠ NP.

Pf idea.
  - P is closed under complementation.
  - If P = NP, then NP is closed under complementation.
  - In other words, NP = co-NP.
  - This is the contrapositive of the theorem.
Good Characterizations

**Good characterization.** [Edmonds 1965] \( \text{NP} \cap \text{co-NP} \).

- If problem X is in both \( \text{NP} \) and \( \text{co-NP} \), then:
  - for *yes* instance, there is a succinct certificate
  - for *no* instance, there is a succinct disqualifier
- Provides conceptual leverage for reasoning about a problem.

**Ex.** Given a bipartite graph, is there a perfect matching.

- If *yes*, can exhibit a perfect matching.
- If *no*, can exhibit a set of nodes \( S \) such that \( |N(S)| < |S| \).
Good Characterizations

Observation. $P \subseteq NP \cap co-NP$.
- Proof of max-flow min-cut theorem led to stronger result that max-flow and min-cut are in $P$.
- Sometimes finding a good characterization seems easier than finding an efficient algorithm.

Fundamental open question. Does $P = NP \cap co-NP$?
- Mixed opinions.
- Many examples where problem found to have a non-trivial good characterization, but only years later discovered to be in $P$.
  - linear programming [Khachiyan, 1979]
  - primality testing [Agrawal-Kayal-Saxena, 2002]

Fact. Factoring is in $NP \cap co-NP$, but not known to be in $P$.

if poly-time algorithm for factoring, can break RSA cryptosystem
Theorem. PRIMES is in NP ∩ co-NP.

Pf. We already know that PRIMES is in co-NP, so it suffices to prove that PRIMES is in NP.

Pratt’s Theorem. An odd integer $s$ is prime iff there exists an integer $1 < t < s$ s.t.

\[
\begin{align*}
    t^{s-1} &\equiv 1 \pmod{s} \\
    t^{(s-1)/p} &\not\equiv 1 \pmod{s}
\end{align*}
\]

for all prime divisors $p$ of $s-1$

Input. $s = 437,677$
Certificate. $t = 17, 2^2 \times 3 \times 36,473$

Certifier.
- Check $s-1 = 2 \times 2 \times 3 \times 36,473$.
- Check $17^{s-1} = 1 \pmod{s}$.
- Check $17^{(s-1)/2} \equiv 437,676 \pmod{s}$.
- Check $17^{(s-1)/3} \equiv 329,415 \pmod{s}$.
- Check $17^{(s-1)/36,473} \equiv 305,452 \pmod{s}$.

prime factorization of $s-1$
also need a recursive certificate
to assert that 3 and 36,473 are prime

use repeated squaring
FACTOR is in $\text{NP} \cap \text{co-NP}$

**FACTORIZE.** Given an integer $x$, find its prime factorization.

**FACTOR.** Given two integers $x$ and $y$, does $x$ have a nontrivial factor less than $y$?

**Theorem.** $\text{FACTOR} \equiv_p \text{FACTORIZE}$.

**Theorem.** $\text{FACTOR}$ is in $\text{NP} \cap \text{co-NP}$.

**Pf.**
- **Certificate:** a factor $p$ of $x$ that is less than $y$.
- **Disqualifier:** the prime factorization of $x$ (where each prime factor is less than $y$), along with a certificate that each factor is prime.
We established: \( \text{PRIMES} \leq_p \text{COMPOSITES} \leq_p \text{FACTOR} \).

Natural question: Does \( \text{FACTOR} \leq_p \text{PRIMES} \)?

Consensus opinion. No.

State-of-the-art.
- PRIMES is in \( P \). \( \leftarrow \) proved in 2001
- FACTOR not believed to be in \( P \).

RSA cryptosystem.
- Based on dichotomy between complexity of two problems.
- To use RSA, must generate large primes efficiently.
- To break RSA, suffixes to find efficient factoring algorithm.
Algorithms — April 18

Today
► PSPACE

Friday
► Quiz on Chapter 8
  ► Definitions and short-answer questions
  ► Closed-book

Monday, April 23
► Problems due: Ch. 7, #8, #9

Contact information
► Web: cs.uwlax.edu/~jmaraist
► Email: jmaraist@uwlax.edu
  ► Expect replies within a business day (but often same-day)
► End-of-semester office hours
  ► Wednesday 2: 3-5pm
  ► Friday 4: 2-4pm
  ► Monday 7: 10am-noon
► Or by appointment
  ► Email at least a day ahead
  ► Send several times when you are available
  ► Always describe what you need to discuss
    ► So I can be prepared
    ► Because advising or paperwork often require no meeting
9.1 PSPACE
PSPACE

Binary counter. Count from 0 to \(2^n - 1\) in binary.

Algorithm. Use \(n\) bit odometer.

Claim. 3-SAT is in PSPACE.

Pf.
- Enumerate all \(2^n\) possible truth assignments using counter.
- Check each assignment to see if it satisfies all clauses. □

Theorem. NP ⊆ PSPACE.

Pf. Consider arbitrary problem \(Y\) in NP.
- Since \(Y \leq_p 3\text{-SAT}\), there exists algorithm that solves \(Y\) in poly-time plus polynomial number of calls to 3-SAT black box.
- Can implement black box in poly-space. □
9.3 Quantified Satisfiability
Quantified Satisfiability

**QSAT.** Let $\Phi(x_1, \ldots, x_n)$ be a Boolean CNF formula. Is the following propositional formula true?

$$\exists x_1 \ \forall x_2 \ \exists x_3 \ \forall x_4 \ldots \ \forall x_{n-1} \ \exists x_n \ \Phi(x_1, \ldots, x_n)$$

↑

assume $n$ is odd

**Intuition.** Amy picks truth value for $x_1$, then Bob for $x_2$, then Amy for $x_3$, and so on. Can Amy satisfy $\Phi$ no matter what Bob does?

**Ex.** $(x_1 \lor x_2) \land (x_2 \lor \overline{x}_3) \land (x_1 \lor x_2 \lor x_3)$

**Yes.** Amy sets $x_1$ true; Bob sets $x_2$; Amy sets $x_3$ to be same as $x_2$.

**Ex.** $(x_1 \lor x_2) \land (\overline{x}_2 \lor \overline{x}_3) \land (x_1 \lor x_2 \lor x_3)$

**No.** If Amy sets $x_1$ false; Bob sets $x_2$ false; Amy loses;

if Amy sets $x_1$ true; Bob sets $x_2$ true; Amy loses.
**Theorem.** QSAT ∈ PSPACE.

**Pf.** Recursively try all possibilities.
- Only need one bit of information from each subproblem.
- Amount of space is proportional to depth of function call stack.

\[
\begin{align*}
\exists & \; \forall \; x_1 = 0 \\
\exists & \; \exists \; x_2 = 0 \\
\exists & \; x_3 = 0 \\
\Phi(0, 0, 0) & \\
\Phi(0, 0, 1) & \\
\Phi(0, 1, 0) & \\
\Phi(0, 1, 1) & \\
\Phi(1, 0, 0) & \\
\Phi(1, 0, 1) & \\
\Phi(1, 1, 0) & \\
\Phi(1, 1, 1) & \\
\end{align*}
\]
9.4 Planning Problem
8-puzzle, 15-puzzle. [Sam Loyd 1870s]

- Board: 3-by-3 grid of tiles labeled 1-8.
- Legal move: slide neighboring tile into blank (white) square.
- Find sequence of legal moves to transform initial configuration into goal configuration.

```
 initial configuration

1 2 3
4 5 6
8 7

move 12

1 2 3
4 5
8 7 6

?  

goal configuration

1 2 3
4 5 6
7 8
```
Planning Problem

**Conditions.** Set $C = \{ C_1, \ldots, C_n \}$.

**Initial configuration.** Subset $c_0 \subseteq C$ of conditions initially satisfied.

**Goal configuration.** Subset $c^* \subseteq C$ of conditions we seek to satisfy.

**Operators.** Set $O = \{ O_1, \ldots, O_k \}$.

- To invoke operator $O_i$, must satisfy certain prereq conditions.
- After invoking $O_i$ certain conditions become true, and certain conditions become false.

**PLANNING.** Is it possible to apply sequence of operators to get from initial configuration to goal configuration?

**Examples.**
- 15-puzzle.
- Rubik’s cube.
- Logistical operations to move people, equipment, and materials.
Planning Problem: 8-Puzzle

Planning example. Can we solve the 8-puzzle?

Conditions. $C_{ij}, 1 \leq i, j \leq 9$. $C_{ij}$ means tile $i$ is in square $j$

Initial state. $c_0 = \{C_{11}, C_{22}, ..., C_{66}, C_{78}, C_{87}, C_{99}\}$.

Goal state. $c^* = \{C_{11}, C_{22}, ..., C_{66}, C_{77}, C_{88}, C_{99}\}$.

Operators.
- Precondition to apply $O_i = \{C_{11}, C_{22}, ..., C_{66}, C_{78}, C_{87}, C_{99}\}$.
- After invoking $O_i$, conditions $C_{79}$ and $C_{97}$ become true.
- After invoking $O_i$, conditions $C_{78}$ and $C_{99}$ become false.

Solution. No solution to 8-puzzle or 15-puzzle!
8-puzzle invariant. Any legal move preserves the parity of the number of pairs of pieces in reverse order (inversions).

- Original state:
  
  \[
  \begin{array}{ccc}
  3 & 1 & 2 \\
  4 & 5 & 6 \\
  8 & 7 & \\
  \end{array}
  \]

  3 inversions:
  
  1-3, 2-3, 7-8

- After one move:
  
  \[
  \begin{array}{ccc}
  3 & 1 & 2 \\
  4 & 5 & 6 \\
  8 & 7 & \\
  \end{array}
  \]

  3 inversions:
  
  1-3, 2-3, 7-8

  \[
  \begin{array}{ccc}
  3 & 1 & 2 \\
  4 & 6 & \\
  8 & 5 & 7 \\
  \end{array}
  \]

  5 inversions:
  
  1-3, 2-3, 7-8, 5-8, 5-6

- Final state:
  
  \[
  \begin{array}{ccc}
  1 & 2 & 3 \\
  4 & 5 & 6 \\
  8 & 7 & \\
  \end{array}
  \]

  1 inversion:
  
  7-8

  \[
  \begin{array}{ccc}
  1 & 2 & 3 \\
  4 & 5 & 6 \\
  8 & 7 & \\
  \end{array}
  \]

  0 inversions
Planning Problem: Binary Counter

Planning example. Can we increment an n-bit counter from the all-zeroes state to the all-ones state?

Conditions. $C_1, \ldots, C_n$. $\leftarrow C_i$ corresponds to bit $i = 1$
Initial state. $c_0 = \emptyset$. $\leftarrow$ all 0s
Goal state. $c^* = \{C_1, \ldots, C_n\}$. $\leftarrow$ all 1s
Operators. $O_1, \ldots, O_n$.

- To invoke operator $O_i$, must satisfy $C_1, \ldots, C_{i-1}$. $\leftarrow$ i-1 least significant bits are 1
- After invoking $O_i$, condition $C_i$ becomes true. $\leftarrow$ set bit $i$ to 1
- After invoking $O_i$, conditions $C_1, \ldots, C_{i-1}$ become false. $\leftarrow$ set i-1 least significant bits to 0

Solution. $\emptyset \Rightarrow \{C_1\} \Rightarrow \{C_2\} \Rightarrow \{C_1, C_2\} \Rightarrow \{C_3\} \Rightarrow \{C_3, C_1\} \Rightarrow \ldots$

Observation. Any solution requires at least $2^n - 1$ steps.
Planning Problem: In Exponential Space

**Configuration graph** $G$.
- Include node for each of $2^n$ possible configurations.
- Include an edge from configuration $c'$ to configuration $c''$ if one of the operators can convert from $c'$ to $c''$.

**PLANNING.** Is there a path from $c_0$ to $c^*$ in configuration graph?

**Claim.** PLANNING is in EXPTIME.
**Pf.** Run BFS to find path from $c_0$ to $c^*$ in configuration graph.  

**Note.** Configuration graph can have $2^n$ nodes, and shortest path can be of length $= 2^n - 1$. 

↑

binary counter
Theorem. PLANNING is in PSPACE.

Pf.

- Suppose there is a path from $c_1$ to $c_2$ of length $L$.
- Path from $c_1$ to midpoint and from $c_2$ to midpoint are each $\leq L/2$.
- Enumerate all possible midpoints.
- Apply recursively. Depth of recursion = $\log_2 L$. 

```java
boolean hasPath(c1, c2, L) {
    if (L <= 1) return correct answer
    enumerate using binary counter
    foreach configuration c' {
        boolean x = hasPath(c1, c', L/2)
        boolean y = hasPath(c2, c', L/2)
        if (x and y) return true
    }
    return false
}
```
9.5 PSPACE-Complete
PSPACE. Decision problems solvable in polynomial space.

PSPACE-Complete. Problem \( Y \) is PSPACE-complete if (i) \( Y \) is in PSPACE and (ii) for every problem \( X \) in PSPACE, \( X \leq_p Y \).


Theorem. PSPACE \( \subseteq \) EXPTIME.

Pf. Previous algorithm solves QSAT in exponential time, and QSAT is PSPACE-complete. ▪

Summary. \( P \subseteq NP \subseteq PSPACE \subseteq EXPTIME \).

\[ \uparrow \quad \uparrow \quad \uparrow \]

It is known that \( P \neq EXPTIME \), but unknown which inclusion is strict; conjectured that all are
PSPACE-Complete Problems

More PSPACE-complete problems.

- Competitive facility location.
- Natural generalizations of games.
  - Othello, Hex, Geography, Rush-Hour, Instant Insanity
  - Shanghai, go-moku, Sokoban
- Given a memory restricted Turing machine, does it terminate in at most k steps?
- Do two regular expressions describe different languages?
- Is it possible to move and rotate complicated object with attachments through an irregularly shaped corridor?
- Is a deadlock state possible within a system of communicating processors?
Competitive Facility Location

**Input.** Graph with positive edge weights, and target B.

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

**Competitive facility location.** Can second player guarantee at least B units of profit?

Yes if B = 20; no if B = 25.
Claim. COMPETITIVE-FACILITY is PSPACE-complete.

Pf.

- To solve in poly-space, use recursion like QSAT, but at each step there are up to n choices instead of 2.

- To show that it's complete, we show that QSAT polynomial reduces to it. Given an instance of QSAT, we construct an instance of COMPETITIVE-FACILITY such that player 2 can force a win iff QSAT formula is true.
**Construction.** Given instance $\Phi(x_1, ..., x_n) = C_1 \land C_1 \land ... \land C_k$ of QSAT.

- Include a node for each literal and its negation and connect them.
  - at most one of $x_i$ and its negation can be chosen
- Choose $c \geq k+2$, and put weight $c^i$ on literal $x^i$ and its negation;
  set $B = c^{n-1} + c^{n-3} + ... + c^4 + c^2 + 1$.
  - ensures variables are selected in order $x_n, x_{n-1}, ..., x_1$.
- As is, player 2 will lose by 1 unit: $c^{n-1} + c^{n-3} + ... + c^4 + c^2$. 

<diagram>
\begin{itemize}
  \item \hspace{1cm} 10^n \\
  \hspace{1cm} x_n \quad x_n^n \\
  \quad \quad \quad \vdots \\
  \quad 100 \quad 100 \\
  \quad x_2 \quad x_2 \\
  \quad 10 \quad 10 \\
  \quad x_1 \quad x_1
\end{itemize>
Competitive Facility Location

Construction. Given instance $\Phi(x_1, \ldots, x_n) = C_1 \land C_1 \land \ldots \land C_k$ of QSAT.
- Give player 2 one last move on which she can try to win.
- For each clause $C_j$, add node with value 1 and an edge to each of its literals.
- Player 2 can make last move iff truth assignment defined alternately by the players failed to satisfy some clause.

\[
\begin{align*}
x_1 \lor x_2 \lor \overline{x_n} & \\
1 & \\
\overline{x_n} & \\
10 & \vdots \quad 100 & \\
100 & 10 & \\
\overline{x_2} & \\
\overline{x_1} & \\
x_2 & \\
x_1 & \\
\end{align*}
\]
Algorithms — April 18

Today

▶ Graph library code walkthrough
  ▶ Java functional interfaces briefing
▶ Quiz on Chapter 8
  ▶ Definitions and short-answer questions
  ▶ Closed-book

Monday, April 23

▶ Problems due: Ch. 7, #8, #9

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting
Algorithms — April 23

Today

▶ Problems due: Ch. 7, #8, #9
▶ PSPACE-completeness
▶ Approximation algorithms

Friday, April 27

▶ Quiz on Chapter 9
  ▶ Definitions and short-answer questions
  ▶ Closed-book

Monday, April 30

▶ Presentations Round 1

Wednesday, May 2

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting
**PSPACE-Complete**

**PSPACE.** Decision problems solvable in polynomial space.

**PSPACE-Complete.** Problem $Y$ is PSPACE-complete if (i) $Y$ is in PSPACE and (ii) for every problem $X$ in PSPACE, $X \leq_p Y$.

**Theorem.** [Stockmeyer-Meyer 1973] QSAT is PSPACE-complete.

**Theorem.** $\text{PSPACE} \subseteq \text{EXPTIME}$.

**Pf.** Previous algorithm solves QSAT in exponential time, and QSAT is PSPACE-complete. □

**Summary.** $P \subseteq \text{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME}$.

It is known that $P \neq \text{EXPTIME}$, but unknown which inclusion is strict; conjectured that all are
PSPACE-Complete Problems

More PSPACE-complete problems.

- Competitive facility location.
- Natural generalizations of games.
  - Othello, Hex, Geography, Rush-Hour, Instant Insanity
  - Shanghai, go-moku, Sokoban
- Given a memory restricted Turing machine, does it terminate in at most k steps?
- Do two regular expressions describe different languages?
- Is it possible to move and rotate complicated object with attachments through an irregularly shaped corridor?
- Is a deadlock state possible within a system of communicating processors?
Competitive Facility Location

**Input.** Graph with positive edge weights, and target B.

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

**Competitive facility location.** Can second player guarantee at least B units of profit?

Yes if B = 20; no if B = 25.
Claim. COMPETITIVE-FACILITY is PSPACE-complete.

Pf.

- To solve in poly-space, use recursion like QSAT, but at each step there are up to $n$ choices instead of 2.

- To show that it's complete, we show that QSAT polynomial reduces to it. Given an instance of QSAT, we construct an instance of COMPETITIVE-FACILITY such that player 2 can force a win iff QSAT formula is true.
Construction. Given instance $\Phi(x_1, \ldots, x_n) = C_1 \land C_1 \land \ldots C_k$ of QSAT.

- Include a node for each literal and its negation and connect them. at most one of $x_i$ and its negation can be chosen
- Choose $c \geq k+2$, and put weight $c_i$ on literal $x^i$ and its negation; set $B = c_{n-1} + c_{n-3} + \ldots + c^4 + c^2 + 1.$
  - ensures variables are selected in order $x_n, x_{n-1}, \ldots, x_1$.
- As is, player 2 will lose by 1 unit: $c_{n-1} + c_{n-3} + \ldots + c^4 + c^2$. 

\[ 10^n \quad 10^n \]
\[ x_n \quad \overline{x_n} \]
\[ \vdots \]
\[ 100 \quad 100 \]
\[ x_2 \quad \overline{x_2} \]
\[ 10 \quad 10 \]
\[ x_1 \quad \overline{x_1} \]

Assume $n$ is odd.
Competitive Facility Location

Construction. Given instance $\Phi(x_1, \ldots, x_n) = C_1 \land C_1 \land \ldots C_k$ of QSAT.

- Give player 2 one last move on which she can try to win.
- For each clause $C_j$, add node with value 1 and an edge to each of its literals.
- Player 2 can make last move iff truth assignment defined alternately by the players failed to satisfy some clause. ▪
Algorithms — April 25

Today

▶ Problems due: Ch. 7, #8, #9
▶ Approximation algorithms

Friday, April 27

▶ Sample presentation
▶ Quiz on Chapter 9
  ▶ Definitions and short-answer questions
  ▶ Closed-book

Monday, April 30

▶ Presentations Round 1

Wednesday, May 2

▶ Presentations Round 2

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no
Algorithms next time

This is the time in the semester when faculty make notes about what we’ll do differently next time
  ▶ Especially after the first time(s) teaching a class
Algorithms next time

This is the time in the semester when faculty make notes about what we’ll do differently next time

- Especially after the first time(s) teaching a class
- I’d welcome your feedback — in person, or via the SEIs
  - With or without suggestions please fill out your SEIs
  - In a class this size, it is important to get everyone’s response
Algorithms next time

This is the time in the semester when faculty make notes about what we’ll do differently next time

▶ Especially after the first time(s) teaching a class
▶ I’d welcome your feedback — in person, or via the SEIs
  ▶ With or without suggestions please fill out your SEIs
  ▶ In a class this size, it is important to get everyone’s response

Three things I’d change for the next CS353
Algorithms next time

This is the time in the semester when faculty make notes about what we’ll do differently next time

- Especially after the first time(s) teaching a class
- I’d welcome your feedback — in person, or via the SEIs
  - With or without suggestions *please fill out your SEIs*
  - In a class this size, it is important to get everyone’s response

Three things I’d change for the next CS353

- Programming assignments earlier
  - Better to tie the theory to real practice
  - Make each homework set a better combination of theory and implementation
Algorithms next time
This is the time in the semester when faculty make notes about what we’ll do differently next time

- Especially after the first time(s) teaching a class
- I’d welcome your feedback — in person, or via the SEIs
  - With or without suggestions please fill out your SEIs
  - In a class this size, it is important to get everyone’s response

Three things I’d change for the next CS353

- Programming assignments earlier
  - Better to tie the theory to real practice
  - Make each homework set a better combination of theory and implementation
- More time on Chapters 4-7 (Greedy through Network Flow)
  - In particular, more worked examples
  - Possibly including another problem-type chapter from another book
Algorithms next time

This is the time in the semester when faculty make notes about what we’ll do differently next time

▶ Especially after the first time(s) teaching a class
▶ I’d welcome your feedback — in person, or via the SEIs
  ▶ With or without suggestions please fill out your SEIs
  ▶ In a class this size, it is important to get everyone’s response

Three things I’d change for the next CS353

▶ Programming assignments earlier
  ▶ Better to tie the theory to real practice
  ▶ Make each homework set a better combination of theory and implementation
▶ More time on Chapters 4-7 (Greedy through Network Flow)
  ▶ In particular, more worked examples
  ▶ Possibly including another problem-type chapter from another book
  ▶ So a more compressed version of Chapter 9 (PSPACE) in particular
    ▶ Something like a 1-day overview NP-completeness, PSPACE and the complexity beastiary
Algorithms next time

This is the time in the semester when faculty make notes about what we’ll do differently next time

▶ Especially after the first time(s) teaching a class
▶ I’d welcome your feedback — in person, or via the SEIs
  ▶ With or without suggestions *please fill out your SEIs*
  ▶ In a class this size, it is important to get everyone’s response

Three things I’d change for the next CS353

▶ Programming assignments earlier
  ▶ Better to tie the theory to real practice
  ▶ Make each homework set a better combination of theory and implementation
▶ More time on Chapters 4-7 (Greedy through Network Flow)
  ▶ In particular, more worked examples
  ▶ Possibly including another problem-type chapter from another book
  ▶ So a more compressed version of Chapter 9 (PSPACE) in particular
    ▶ Something like a 1-day overview NP-completeness, PSPACE and the complexity beastiary
▶ Continue with the \LaTeXed exercise solutions, but prep them prior to the semester
  ▶ Homework grading took too long
  ▶ Having the solutions ready would speed things up, so that I could focus just on person-specific feedback
Algorithms next time

This is the time in the semester when faculty make notes about what we’ll do differently next time

▶ Especially after the first time(s) teaching a class
▶ I’d welcome your feedback — in person, or via the SEIs
  ▶ With or without suggestions please fill out your SEIs
  ▶ In a class this size, it is important to get everyone’s response

Three things I’d change for the next CS353

▶ Programming assignments earlier
  ▶ Better to tie the theory to real practice
  ▶ Make each homework set a better combination of theory and implementation
▶ More time on Chapters 4-7 (Greedy through Network Flow)
  ▶ In particular, more worked examples
  ▶ Possibly including another problem-type chapter from another book
  ▶ So a more compressed version of Chapter 9 (PSPACE) in particular
    ▶ Something like a 1-day overview NP-completeness, PSPACE and the complexity beastiary
▶ Continue with the \LaTeX ed exercise solutions, but prep them prior to the semester
  ▶ Homework grading took too long
  ▶ Having the solutions ready would speed things up, so that I could focus just on person-specific feedback
▶ Keep the general structure of assessment
  ▶ Quizzes and homeworks through the semester, just one exam
  ▶ When outscoring some week’s work on the exam, plug that improvement back to the earlier work
Q. Suppose I need to solve an NP-complete problem. What should I do?
A. Theory says you're unlikely to find poly-time algorithm.

Must sacrifice one of three desired features.
- Solve problem to optimality.
- Solve problem in polynomial time.
- Solve arbitrary instances of the problem.

This lecture. Solve some special cases of NP-complete problems that arise in practice.
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>$n = 10$</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>$n = 30$</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>$n = 50$</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>$n = 100$</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>$n = 1,000$</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>$n = 10,000$</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>$n = 100,000$</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>$n = 1,000,000$</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>
10.1 Finding Small Vertex Covers
**Vertex Cover**

**VERTEX COVER:** Given a graph $G = (V, E)$ and an integer $k$, is there a subset of vertices $S \subseteq V$ such that $|S| \leq k$, and for each edge $(u, v)$ either $u \in S$, or $v \in S$, or both.

\[ k = 4 \]
\[ S = \{3, 6, 7, 10\} \]
Finding Small Vertex Covers

**Q.** What if $k$ is small?

**Brute force.** $O(k \cdot n^{k+1})$.
- Try all $C(n, k) = O(n^k)$ subsets of size $k$.
- Takes $O(k \cdot n)$ time to check whether a subset is a vertex cover.

**Goal.** Limit exponential dependency on $k$, e.g., to $O(2^k \cdot k \cdot n)$.

**Ex.** $n = 1,000$, $k = 10$.
- **Brute.** $k \cdot n^{k+1} = 10^{34} \Rightarrow$ infeasible.
- **Better.** $2^k \cdot k \cdot n = 10^7 \Rightarrow$ feasible.

**Remark.** If $k$ is a constant, algorithm is poly-time; if $k$ is a small constant, then it's also practical.
Finding Small Vertex Covers

Claim. Let $u-v$ be an edge of $G$. $G$ has a vertex cover of size $\leq k$ iff at least one of $G - \{u\}$ and $G - \{v\}$ has a vertex cover of size $\leq k-1$.

Pf. $\Rightarrow$

- Suppose $G$ has a vertex cover $S$ of size $\leq k$.
- $S$ contains either $u$ or $v$ (or both). Assume it contains $u$.
- $S - \{u\}$ is a vertex cover of $G - \{u\}$.

Pf. $\Leftarrow$

- Suppose $S$ is a vertex cover of $G - \{u\}$ of size $\leq k-1$.
- Then $S \cup \{u\}$ is a vertex cover of $G$.

Claim. If $G$ has a vertex cover of size $k$, it has $\leq k(n-1)$ edges.

Pf. Each vertex covers at most $n-1$ edges.
Claim. The following algorithm determines if \( G \) has a vertex cover of size \( \leq k \) in \( O(2^k \cdot kn) \) time.

\[
\text{boolean } \text{Vertex-Cover}(G, k) \{
    \text{if (G contains no edges) return true}
    \text{if (G contains } \geq \text{ kn edges) return false}
    \text{let (u, v) be any edge of G}
    a = \text{Vertex-Cover}(G - \{u\}, k-1)
    b = \text{Vertex-Cover}(G - \{v\}, k-1)
    \text{return a or b}
\}
\]

Pf.
- Correctness follows from previous two claims.
- There are \( \leq 2^{k+1} \) nodes in the recursion tree; each invocation takes \( O(kn) \) time.
Finding Small Vertex Covers: Recursion Tree

\[
T(n, k) \leq \begin{cases} 
  c & \text{if } k = 0 \\
  cn & \text{if } k = 1 \\
  2T(n, k - 1) + ckn & \text{if } k > 1 
\end{cases} \Rightarrow T(n, k) \leq 2^k c kn
\]
10.2 Solving NP-Hard Problems on Trees
Independent set on trees. Given a tree, find a maximum cardinality subset of nodes such that no two share an edge.

Fact. A tree on at least two nodes has at least two leaf nodes.

Key observation. If \( v \) is a leaf, there exists a maximum size independent set containing \( v \).

Pf. (exchange argument)
- Consider a max cardinality independent set \( S \).
- If \( v \in S \), we're done.
- If \( u \notin S \) and \( v \notin S \), then \( S \cup \{ v \} \) is independent \( \implies S \) not maximum.
- If \( u \in S \) and \( v \notin S \), then \( S \cup \{ v \} - \{ u \} \) is independent. □
Theorem. The following greedy algorithm finds a maximum cardinality independent set in forests (and hence trees).

Pf. Correctness follows from the previous key observation. □

Remark. Can implement in $O(n)$ time by considering nodes in postorder.
Weighted Independent Set on Trees

**Weighted independent set on trees.** Given a tree and node weights $w_v > 0$, find an independent set $S$ that maximizes $\sum_{v \in S} w_v$.

**Observation.** If $(u, v)$ is an edge such that $v$ is a leaf node, then either $OPT$ includes $u$, or it includes all leaf nodes incident to $u$.

**Dynamic programming solution.** Root tree at some node, say $r$.
- $OPT_{in}(u) = \max$ weight independent set of subtree rooted at $u$, containing $u$.
- $OPT_{out}(u) = \max$ weight independent set of subtree rooted at $u$, not containing $u$.

\[
OPT_{in}(u) = w_u + \sum_{v \in \text{children}(u)} OPT_{out}(v)
\]

\[
OPT_{out}(u) = \sum_{v \in \text{children}(u)} \max \{ OPT_{in}(v), OPT_{out}(v) \}
\]
Theorem. The dynamic programming algorithm finds a maximum weighted independent set in a tree in $O(n)$ time.

Weighted-Independent-Set-In-A-Tree(T) {
    Root the tree at a node $r$
    foreach (node $u$ of $T$ in postorder) {
        if ($u$ is a leaf) {
            $M_{in}[u] = w_u$
            $M_{out}[u] = 0$
        } else {
            $M_{in}[u] = w_u + \sum_{v \in \text{children}(u)} M_{out}[v]$
            $M_{out}[u] = \sum_{v \in \text{children}(u)} \max(M_{in}[v], M_{out}[v])$
        }
    }
    return $\max(M_{in}[r], M_{out}[r])$
}

Pf. Takes $O(n)$ time since we visit nodes in postorder and examine each edge exactly once. □ can also find independent set itself (not just value)
Context

**Independent set on trees.** This structured special case is tractable because we can find a node that breaks the communication among the subproblems in different subtrees.

**Graphs of bounded tree width.** Elegant generalization of trees that:
- Captures a rich class of graphs that arise in practice.
- Enables decomposition into independent pieces.

*see Chapter 10.4, but proceed with caution*
Algorithms — April 27

Today

▶ Sample presentation
▶ Quiz on Chapter 9
  ▶ Definitions and short-answer questions
  ▶ Closed-book

Monday

Presentations Round 1
1. Adam, 6.12
2. Gus, 6.15b
3. Aaron, 6.19
4. Will, 6.20
5. Zach, 6.24

Wednesday

Presentations Round 2
1. Zack, 7.16
2. Adam, 7.7
3. Aaron, 7.31
4. Gus, 7.13
5. Will, 7.21b

Wednesday, May 9

▶ Final exam, in here
▶ First half closed-book
▶ Then take a break if you like
▶ Second half open-book

Contact information

▶ Web: cs.uwlax.edu/~jmaraist
▶ Email: jmaraist@uwlax.edu
  ▶ Expect replies within a business day (but often same-day)
▶ End-of-semester office hours
  ▶ Wednesday 2: 3-5pm
  ▶ Friday 4: 2-4pm
  ▶ Monday 7: 10am-noon
▶ Or by appointment
  ▶ Email at least a day ahead
  ▶ Send several times when you are available
  ▶ Always describe what you need to discuss
    ▶ So I can be prepared
    ▶ Because advising or paperwork often require no meeting

Friday — Last lecture
Algorithms — April 30

Today
Presentations Round 1
1. Adam, 6.12
2. Gus, 6.15b
3. Aaron, 6.19
4. Will, 6.20
5. Zach, 6.24

Wednesday
Presentations Round 2
1. Zack, 7.16
2. Adam, 7.7
3. Aaron, 7.31
4. Gus, 7.13
5. Will, 7.21b

Friday — Last lecture

Wednesday, May 9

- Final exam, in here
- First half, closed-book
  - So bring the book
- Then you can take a break if you like
- Second half, open-book

Contact information

- Web: cs.uwlax.edu/~jmarais
- Email: jmarais@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting
Algorithms — May 2

Today
Presentations Round 2
1. Zach, 6.24
2. Zach, 7.16
3. Adam, 7.7
4. Aaron, 7.31
5. Gus, 7.13
6. Will, 7.21b

Friday — Last lecture

Wednesday, May 9
- Final exam, in here
- First half, closed-book
  - So bring the book
- Then you can take a break if you like
- Second half, open-book

Contact information
- Web: cs.uwlax.edu/~jmaraist
- Email: jmaraist@uwlax.edu
  - Expect replies within a business day (but often same-day)
- End-of-semester office hours
  - Wednesday 2: 3-5pm
  - Friday 4: 2-4pm
  - Monday 7: 10am-noon
- Or by appointment
  - Email at least a day ahead
  - Send several times when you are available
  - Always describe what you need to discuss
    - So I can be prepared
    - Because advising or paperwork often require no meeting