Understanding Computation
Mathematics & Computation

Machines have helped with calculations for a long time.

Can we use machines to reason too?
Mathematics & Computation

Machines have helped with calculations for a long time

Can we use machines to reason too?
Mathematics & Computation

Machines have helped with calculations for a long time.

Can we use machines to reason too?

*Calculemus!*
Mathematics & Computation

Machines have helped with calculations for a long time

Can we use machines to reason too?

Calculemus!

Formal Logic: Reasoning made into a calculation
Mathematics & Computation
Mathematics & Computation

Formal systems based on axioms and logic: for machines & modern mathematicians
Mathematics & Computation

Formal systems based on axioms and logic: for machines & modern mathematicians

*Foundational problem*: How to choose one’s axioms?
Mathematics & Computation

Formal systems based on axioms and logic: for machines & modern mathematicians

*Foundational problem*: How to choose one’s axioms?
They should not give rise to contradictions!
Mathematics & Computation

Formal systems based on axioms and logic: for machines & modern mathematicians

**Foundational problem:** How to choose one’s axioms?  
*They should not give rise to contradictions!*

Early 1900s: Crisis in mathematical foundations
Mathematics & Computation

Formal systems based on axioms and logic: for machines & modern mathematicians

*Foundational problem:* How to choose one’s axioms?

They should not give rise to contradictions!

Early 1900s: Crisis in mathematical foundations

*Contradictions discovered while attempting to formalize notions involving infinite sets*
David Hilbert

• 1928, Hilbert’s Program:
  “Mechanize” mathematics
David Hilbert

• 1928, Hilbert’s Program:
  “Mechanize” mathematics

• Finite set of axioms and inference rules. An algorithm to determine the truth of any statement

  Need to find a consistent & complete set of axioms
• 1928, Hilbert’s Program:

  “Mechanize” mathematics

• Finite set of axioms and inference rules. An algorithm to determine the truth of any statement

  Need to find a consistent & complete set of axioms

• The system should also afford a proof of its own consistency
  • Based on “safe” axioms — i.e., axioms involving only finite objects — preferably
Mathematics & Computation

Mechanized math
Beyond just philosophical interest!
Can resolve stubborn open problems
Replace mathematicians with mathe-machines!
Goldbach’s Conjecture

Every even number > 2 is the sum of two primes

Letter from Goldbach to Euler dated 7 June 1742
Collatz Conjecture

**Program** Collatz (n:integer)

```
while n > 1 {
    if Even(n) then n := n/2
    else n := 3n+1
}
```
Collatz Conjecture

Program Collatz (n:integer)
    while n > 1 {
        if Even(n) then n := n/2
        else n := 3n+1
    }
Collatz Conjecture

**Program** Collatz (n:integer)

```plaintext
while n > 1 {
    if Even(n) then n := n/2
    else n := 3n+1
}
```

**Conjecture:** Collatz(n) halts for every n > 0
Kurt Gödel

• German logician, at age 25 (1931) proved:

“No matter what (consistent) set of axioms are used, a rich system will have true statements that can’t be proved”
Kurt Gödel

- German logician, at age 25 (1931) proved:

“This statement can’t be proved.”

“No matter what (consistent) set of axioms are used, a rich system will have true statements that can’t be proved.”
Kurt Gödel

- German logician, at age 25 (1931) proved:

  “No matter what (consistent) set of axioms are used, a rich system will have true statements that can’t be proved”
Kurt Gödel

• German logician, at age 25 (1931) proved:
  “No matter what (consistent) set of axioms are used, a rich system will have true statements that can’t be proved”

• Hilbert’s Program can’t work!

“This statement can’t be proved”
“The axioms are consistent”
Kurt Gödel

- German logician, at age 25 (1931) proved:
  
  “No matter what (consistent) set of axioms are used, a rich system will have true statements that can’t be proved”

- Hilbert’s Program can’t work!

- Shook the foundations of
  - mathematics
  - philosophy
  - science
  - everything
Alan Turing

- British mathematician
  - cryptanalysis during WWII
  - arguably, father of AI, CS Theory
  - several books, movies
Alan Turing

- British mathematician
  - cryptanalysis during WWII
  - arguably, father of AI, CS Theory
  - several books, movies

- Mathematically defined computation
  - and proved (1936) that **The Halting Problem** has no general algorithm
Halting Problem

• Given program $P$, input $w$: 

\[ w \rightarrow P \]
Halting Problem

- Given program $P$, input $w$:

Will $P(w)$ halt?
Why would we care about the Halting Problem?

- Suppose halting problem had an algorithm...
Why would we care about the Halting Problem?

- Suppose halting problem had an algorithm...

```plaintext
Program P()
    n := 4
    forever:
        if found-two-primes-that-sum-to(n)
            then n := n + 2
        else halt
```
Why would we care about the Halting Problem?

• Suppose halting problem had an algorithm...

Program P()
  n := 4
  forever:
    if found-two-primes-that-sum-to(n)
    then n := n + 2
    else halt

Does P halt?
Why would we care about the Halting Problem?

• Suppose halting problem had an algorithm...

Program $P()$

\begin{verbatim}
  n := 4
  forever:
    if found-two-primes-that-sum-to(n)
    then n := n + 2
    else halt
\end{verbatim}

Does $P$ halt? ← Solves Goldbach conjecture!
Why would we care about the Halting Problem?

Does \textbf{Find-proof} halt on \( w \)? = \textbf{Is} \( w \) a provable theorem?

\textbf{Program} \textbf{Find-proof}(w)

\begin{verbatim}
  p := empty-string
  forever
    p := successor(p)
    if Verify-proof(w,p)
    then halt
\end{verbatim}
Alas!
Alas!

There is no program that solves the Halting Problem!
No use trying to find one!
Alas!

There is no program that solves the Halting Problem!
   No use trying to find one!

How can there be problems that can’t be solved?
Alas!

There is no program that solves the Halting Problem!  
No use trying to find one!

How can there be problems that can’t be solved?  
What is a problem? What is a program?
Computation

**Problem:**
To compute a function $F$ that maps each input (a string) to an output bit

**Program:**
A finitely described process taking a string as input, and outputting a bit (or not halting)

$P$ solves $F$ if for every $x$, $P(x)$ outputs $F(x)$ and halts
Computation

**Problem:** To compute a function $F$ that maps each input (a string) to an output bit

**Program:** A finitely described process taking a string as input, and outputting a bit (or not halting)

$P$ solves $F$ if for every $x$, $P(x)$ outputs $F(x)$ and halts

Too restrictive?
Computation

**Problem:** To compute a function $F$ that maps each input (a string) to an output bit

**Program:** A finitely described process taking a string as input, and outputting a bit (or not halting)

$P$ solves $F$ if for every $x$, $P(x)$ outputs $F(x)$ and halts

Too restrictive?

Enough to compute functions with longer outputs too: $P(x,i)$ outputs the $i^{th}$ bit of $F(x)$
**Problem:** To compute a function $F$ that maps each input (a string) to an output bit

**Program:** A finitely described process taking a string as input, and outputting a bit (or not halting)

$P$ solves $F$ if for every $x$, $P(x)$ outputs $F(x)$ and halts

Too restrictive?

Enough to compute functions with longer outputs too:
$P(x,i)$ outputs the $i^{th}$ bit of $F(x)$

Enough to model *interactive* computation too:
$P^*(x,\text{state})$ outputs $(y,\text{new\_state})$
Computation

Problem: To compute a function $F$ that maps each input (a string) to an output bit

Program: A finitely described process taking a string as input, and outputting a bit (or not halting)

$P$ solves $F$ if for every $x$, $P(x)$ outputs $F(x)$ and halts

- A program is a finite bit string
Computation

**Problem:**
To compute a function $F$ that maps each input (a string) to an output bit

**Program:**
A finitely described process taking a string as input, and outputting a bit (or not halting)

$P$ solves $F$ if for every $x$, $P(x)$ outputs $F(x)$ and halts

- A program is a finite bit string
- Programs can be *enumerated* — listed sequentially — (say, lexicographically) so that every program appears somewhere in the list
**Problem: **
To compute a function F that maps each input (a string) to an output bit

**Program:**
A finitely described process taking a string as input, and outputting a bit (or not halting)

\[ P \] solves F if for every x, \( P(x) \) outputs \( F(x) \) and halts

- A program is a finite bit string
- Programs can be enumerated — listed sequentially — (say, lexicographically) so that every program appears somewhere in the list
- The set of all programs is countable.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ε</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>00</td>
</tr>
<tr>
<td>5</td>
<td>01</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>000</td>
</tr>
<tr>
<td>9</td>
<td>001</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
</tr>
</tbody>
</table>
Computation

**Problem:**
To compute a function $F$ that maps each input (a string) to an output bit

**Program:**
A finitely described process taking a string as input, and outputting a bit (or not halting)

$P$ solves $F$ if for every $x$, $P(x)$ outputs $F(x)$ and halts

- A function assigns a bit to each finite string

<table>
<thead>
<tr>
<th>Number</th>
<th>String</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\epsilon$</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>01</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>000</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>1</td>
</tr>
</tbody>
</table>
Computation

**Problem:**
To compute a function F that maps each input (a string) to an output bit

**Program:**
A finitely described process taking a string as input, and outputting a bit (or not halting)

P solves F if for every x, P(x) outputs F(x) and halts

- A function assigns a bit to each finite string
- Corresponds to an infinite bit string
Computation

**Problem:**
To compute a function $F$ that maps each input (a string) to an output bit

**Program:**
A finitely described process taking a string as input, and outputting a bit (or not halting)

$P$ solves $F$ if for every $x$, $P(x)$ outputs $F(x)$ and halts

- A function assigns a bit to each finite string
- Corresponds to an infinite bit string
- The set of all functions is **uncountable**!
  - As numerous as, say, real numbers in $[0,1]$
There are uncountably many functions!

Almost every function is uncomputable!

But only countably many programs
Uncomputable Problems
Uncomputable Problems

But that doesn’t tell us why some *interesting* problems are uncomputable
Uncomputable Problems

But that doesn’t tell us why some *interesting* problems are uncomputable

If *interesting = has a finite description in English*, then only countably many interesting problems!
Uncomputable Problems

But that doesn’t tell us why some *interesting* problems are uncomputable

If *interesting* = *has a finite description in English*, then only countably many interesting problems!

Proving that there are uncountably many real numbers:
“Diagonalization” argument by Cantor
Uncomputable Problems

But that doesn’t tell us why some interesting problems are uncomputable

If interesting = has a finite description in English, then only countably many interesting problems!

Proving that there are uncountably many real numbers: “Diagonalization” argument by Cantor

Showing Halting Problem to be uncomputable: a similar argument (later)
Uncomputable Problems
Uncomputable Problems

Once we know one interesting problem is uncomputable, show more using reductions:
Uncomputable Problems

Once we know one interesting problem is uncomputable, show more using reductions:

Reducing $F^*$ to $F$:
Use any program $P$ that solves $F$ to build a program $P^*$ that solves $F^*$
Uncomputable Problems

Once we know one interesting problem is uncomputable, show more using reductions:

Reducing $F^*$ to $F$:
Use any program $P$ that solves $F$
to build a program $P^*$ that solves $F^*$

If the Halting Problem can be reduced to $F$
then $F$ must be uncomputable!
Post Correspondence Problem

**Theorem** [Post’46]: Halting Problem (formulated for “Turing Machines”) reduces to PostCP — a “combinatorial” problem
Post Correspondence Problem

**Theorem** [Post’46]: Halting Problem (formulated for “Turing Machines”) reduces to PostCP — a “combinatorial” problem

Given: Dominoes, each with a top-word and a bottom-word

Can one arrange them (using any number of copies of each type) so that the top and bottom strings are identical?

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>ba</td>
<td>abb</td>
<td>abb</td>
<td></td>
<td>a</td>
</tr>
<tr>
<td>bbb</td>
<td>bbb</td>
<td>a</td>
<td>baa</td>
<td>ab</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>abb</td>
<td>ba</td>
<td>abb</td>
<td>a</td>
<td>abb</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>bbb</td>
<td>a</td>
<td>ab</td>
<td>baa</td>
<td>bbb</td>
</tr>
</tbody>
</table>
```
Post Correspondence Problem

**Theorem** [Post’46]: Halting Problem (formulated for “Turing Machines”) reduces to PostCP — a “combinatorial” problem

PostCP is uncomputable.

Given: Dominoes, each with a top-word and a bottom-word

Can one arrange them (using any number of copies of each type) so that the top and bottom strings are identical?
Post Correspondence Problem

PostCP is uncomputable.
Post Correspondence Problem

**PostCP is uncomputable.**

If PostCP can be reduced to F then F is uncomputable
Post Correspondence Problem

PostCP is uncomputable.

If PostCP can be reduced to F then F is uncomputable

Typically, easier than reducing Halting Problem directly to F
Post Correspondence Problem

PostCP is uncomputable.

If PostCP can be reduced to F then F is uncomputable.

Typically, easier than reducing Halting Problem directly to F.

Many more *interesting* problems:

Induction
Inductive Proofs

Example: How many “moves” to assemble a jigsaw puzzle?

- move = join two clumps
- clump = connected pieces
- only successful moves count

Theorem: It takes exactly n-1 moves to assemble an n-piece jigsaw puzzle (irrespective of which moves)
Inductive Proofs
Inductive Proofs

• **Theorem**: It takes exactly n-1 moves to assemble an n-piece jigsaw puzzle (irrespective of which moves)
Inductive Proofs

- **Theorem**: It takes exactly n-1 moves to assemble an n-piece jigsaw puzzle (irrespective of which moves)

**Proof by Induction:**
Inductive Proofs

- **Theorem**: It takes exactly \( n-1 \) moves to assemble an \( n \)-piece jigsaw puzzle (irrespective of which moves)

**Proof by Induction:**

**Base case**: 1-piece puzzle takes 0 moves. 🔄
Inductive Proofs

- **Theorem**: It takes exactly \( n-1 \) moves to assemble an \( n \)-piece jigsaw puzzle (irrespective of which moves)

**Proof by Induction:**

**Base case**: 1-piece puzzle takes 0 moves. ✅

**Inductive step**: Consider any \( n > 1 \)

Assume any \((n-1)\)-piece puzzle requires \( n-2 \) moves

Consider any \( n \)-piece puzzle:

- \( n-2 \) moves for all but last
- One more move for last

Total = \((n-2)+1 = n-1\)
Inductive Proofs

• **Theorem**: It takes exactly \( n-1 \) moves to assemble an \( n \)-piece jigsaw puzzle (irrespective of which moves)

**Proof by Induction:**

**Base case**: 1-piece puzzle takes 0 moves. ✓

**Inductive step**: Consider any \( n > 1 \)

Assume any \( (n-1) \)-piece puzzle requires \( n-2 \) moves

Consider any \( n \)-piece puzzle:

\( n-2 \) moves for all but last

One more move for last

Total = \( (n-2) + 1 = n-1 \)
Inductive Proofs

- **Theorem**: It takes exactly n-1 moves to assemble an n-piece jigsaw puzzle (irrespective of which moves)

**Proof by Induction:**

**Base case**: 1-piece puzzle takes 0 moves. ✅

**Inductive step**: Consider any n > 1

Assume any (n-1)-piece puzzle requires n-2 moves

Consider any n-piece puzzle:

n-2 moves for all but last

One more move for last

total = (n-2)+1 = n-1
Inductive Proofs

- **Theorem**: It takes exactly \( n-1 \) moves to assemble an \( n \)-piece jigsaw puzzle (irrespective of which moves)

**Proof by Induction:**

**Base case**: 1-piece puzzle takes 0 moves. ✅

**Inductive step**: Consider any \( n > 1 \)

Assume any \( (n-1) \)-piece puzzle requires \( n-2 \) moves

Consider any \( n \)-piece puzzle:
- \( n-2 \) moves for all but last
- One more move for last

Total = \( (n-2)+1 = n-1 \) ✅
• **Theorem**: It takes exactly n-1 moves to assemble an n-piece jigsaw puzzle (irrespective of which pieces are used).

**Proof by Induction**:

- **Base case**: 1-piece puzzle takes 0 moves.
- **Inductive step**: Consider any n > 1.

  - Assume any (n-1)-piece puzzle requires n-2 moves.
  - Consider any n-piece puzzle:
    - n-2 moves for all but last piece
    - One more move for last piece
    - Total = (n-2) + 1 = n-1
Theorem: It takes exactly $n-1$ moves to assemble an $n$-piece jigsaw puzzle (irrespective of which moves).

Proof by Induction:

Base case: 1-piece puzzle takes 0 moves.

Inductive step: Consider any $n > 1$

Assume any $(n-1)$-piece puzzle requires $n-2$ moves.

Consider any $n$-piece puzzle: $n-2$ moves for all but last piece

One more move for last piece

total = $(n-2)+1 = n-1$
Inductive Proofs

Why must last move look like this?
Inductive Proofs

Why must last move look like this?

Last move could join two large clumps
Inductive Proofs

Why must last move look like this?

Last move could join two large clumps

The argument presented implicitly assumes puzzle is built piece-by-piece
Induction Template

• Base Case: Let \( n = \langle \text{some small values} \rangle \).
  Then \( \langle \text{show claim holds for n} \rangle \)

• Induction Step: Consider any arbitrary integer \( n \langle \text{greater than base-case values} \rangle \).

Induction hypothesis: Assume that for all integers \( k < n \) (and \( k \geq \langle \text{smallest value} \rangle \)), \( \langle \text{claim holds for k} \rangle \)

\( \langle \text{Prove that claim holds for n} \rangle \)
Induction Template

- **Base Case:** Let $n = \langle$ some small values $\rangle$. Then $\langle$ show claim holds for $n$ $\rangle$

- **Induction Step:** Consider any arbitrary integer $n \langle$ greater than base-case values $\rangle$.

Induction hypothesis: Assume that for all integers $k < n$ (and $k \geq \langle$ smallest value $\rangle$), $\langle$ claim holds for $k $ $\rangle$

$\langle$ Prove that claim holds for $n$ $\rangle$
Induction Template

- Base Case: Let \( n = \langle \text{some small values} \rangle \). Then \( \langle \text{show claim holds for } n \rangle \)

- Induction Step: Consider any *arbitrary* integer \( n \langle \text{greater than base-case values} \rangle \).

  Induction hypothesis: Assume that for all integers \( k < n \) (and \( k \geq \langle \text{smallest value} \rangle \)), \( \langle \text{claim holds for } k \rangle \)

\( \langle \text{Prove that claim holds for } n \rangle \)

Convention in this class: \( n \) here (not \( n+1 \))

May need a stronger claim than originally asked to prove
Induction Template

- **Base Case:** Let \( n = \langle \text{some small values} \rangle \). Then \( \langle \text{show claim holds for n} \rangle \)

- **Induction Step:** Consider any *arbitrary* integer \( n \langle \text{greater than base-case values} \rangle \).

**Induction hypothesis:** Assume that for all integers \( k < n \) (and \( k \geq \langle \text{smallest value} \rangle \)), \( \langle \text{claim holds for k} \rangle \)

\( \langle \text{Prove that claim holds for n} \rangle \)

*Always* use **strong induction**!

**Convention in this class:** you lose all points for using weak induction when strong needed
Induction Template

- Base Case: Let $n = \langle\text{some small values}\rangle$. Then $\langle\text{show claim holds for } n\rangle$

- Induction Step: Consider any arbitrary integer $n \langle\text{greater than base-case values}\rangle$.

Induction hypothesis: Assume that for all integers $k < n$ (and $k \geq \langle\text{smallest value}\rangle$), $\langle\text{claim holds for } k\rangle$.

$\langle\text{Prove that claim holds for } n\rangle$

The clever stuff. Be careful to consider arbitrary instance of size $n$. Relate it to one or more instances for which IH is assumed.

Convention in this class: $n$ here (not $n+1$)

Always use strong induction!

Convention in this class: you lose all points for using weak induction when strong needed

May need a stronger claim than originally asked to prove
Example

**Stronger Claim:** Any *clump* with $n$ pieces takes exactly $n-1$ moves to assemble.

- **Base Case:** Let $n = 1$. Then, any clump with $n$ pieces is just a single piece, and it needs $0 = n-1$ moves to assemble.

- **Induction Step:** Consider any arbitrary integer $n > 1$.

  Induction hypothesis: Assume that for all integers $k < n$ (and $k \geq 1$), any clump with $k$ pieces needs $k-1$ moves to assemble.

  \[\text{Prove that claim holds for } n\]
**Example**

**Stronger Claim:** Any *clump* with \( n \) pieces takes exactly \( n-1 \) moves to assemble

- **Base Case:** Let \( n = 1 \).
  Then, *any clump* with \( n \) pieces is just a single piece, and it needs \( 0 = n-1 \) moves to assemble.

- **Induction Step:** Consider any *arbitrary* integer \( n > 1 \).
  
  **Induction hypothesis:** Assume that for all integers \( k < n \) (and \( k \geq 1 \)), *any clump* with \( k \) pieces needs \( k-1 \) moves to assemble.

  Consider an *arbitrary* clump with \( n \) pieces, and an *arbitrary* sequence of moves to assemble it.
  - Last move joins 2 clumps of size \( k \) and \( n-k \), where \( 1 \leq k < n \).
  - By IH, the two clumps took \( k-1 \) and \( n-k-1 \) moves each.
  - Overall \( (k-1) + (n-k-1) + 1 = n-1 \) moves. ✅
Simple non-inductive proof
Simple non-inductive proof

- Sometimes non-inductive proofs work, like in this example!
Simple non-inductive proof

• Sometimes non-inductive proofs work, like in this example!
  ▸ A single move reduces number of clumps by exactly 1.
Simple non-inductive proof

• Sometimes non-inductive proofs work, like in this example!
  ▸ A single move reduces number of clumps by exactly 1.
    ▸ $m$ moves reduce it by $m$
Simple non-inductive proof

• Sometimes non-inductive proofs work, like in this example!
  ▸ A single move reduces number of clumps by exactly 1.
    ▸ m moves reduce it by m
  ▸ Initially, n clumps (each of one piece)
  ▸ At the end, 1 clump (of all pieces)
Simple non-inductive proof

- Sometimes non-inductive proofs work, like in this example!
  - A single move reduces number of clumps by exactly 1.
    - m moves reduce it by m
  - Initially, n clumps (each of one piece)
  - At the end, 1 clump (of all pieces)
  - Therefore, if m moves overall, 1 = n - m.
Simple non-inductive proof

- Sometimes non-inductive proofs work, like in this example!
  - A single move reduces number of clumps by exactly 1.
    - m moves reduce it by m
  - Initially, n clumps (each of one piece)
  - At the end, 1 clump (of all pieces)
  - Therefore, if m moves overall, \(1 = n - m\).
  - Hence m = n - 1
If you came in late:

- [https://courses.engr.illinois.edu/cs374/](https://courses.engr.illinois.edu/cs374/)
- Immediately join Piazza
- Immediately check access to Moodle

Links to Piazza and Moodle are on course home page