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?

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

<u>Foundational problem:</u> How to choose one's axioms?

Mathematics & Computation

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

<u>Foundational problem:</u> 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

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

- 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

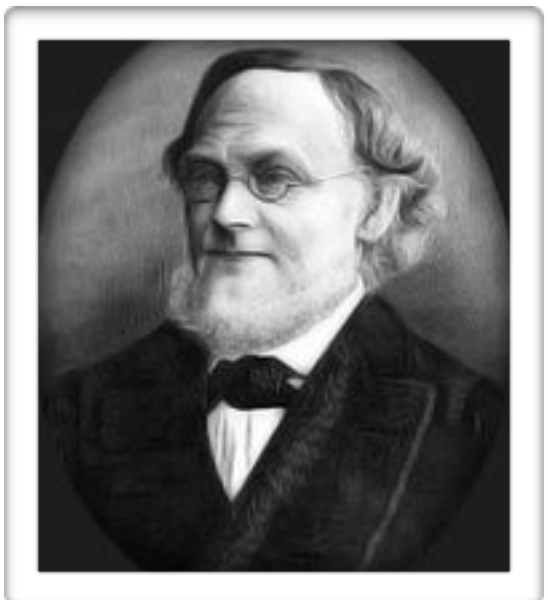
Beyond just philosophical interest!

Can resolve stubborn open problems

Replace mathematicians with mathe-machines!

Goldbach's Conjecture

Every even number > 2 is the sum of two primes



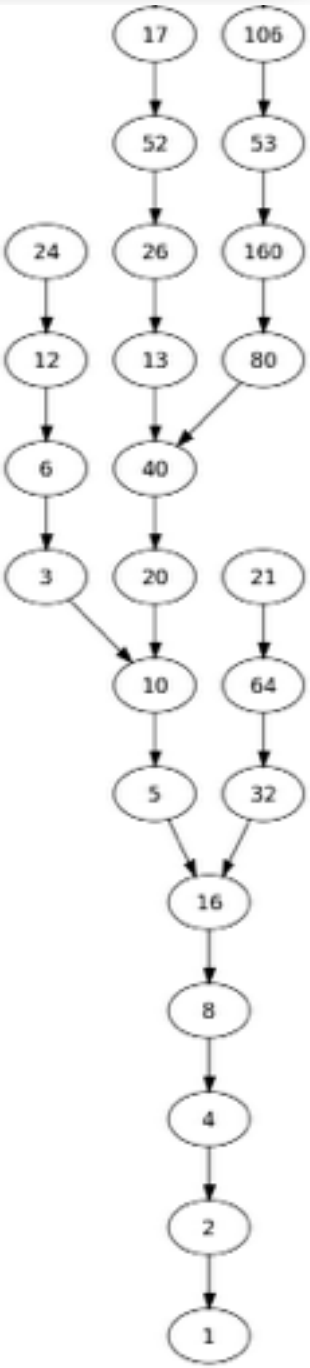
Letter from Goldbach to Euler dated 7 June 1742

Collatz Conjecture

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

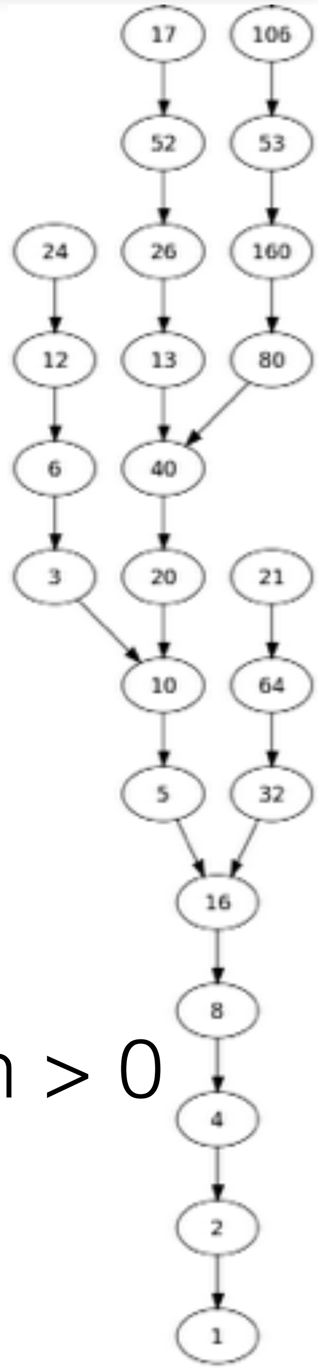
Collatz Conjecture

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



Collatz Conjecture

```
Program Collatz (n:integer)
   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

“This statement can’t be proved”

- 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

“<u>This</u> statement can’t be proved”

“The axioms are consistent”

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

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

Kurt Gödel

“<u>This</u> statement can’t be proved”

“The axioms are consistent”

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

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

- Hilbert’s Program can’t work!

Kurt Gödel

“<u>This</u> statement can’t be proved”

“The axioms are consistent”

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

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

- Hilbert’s Program can’t work!

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

Alan Turing

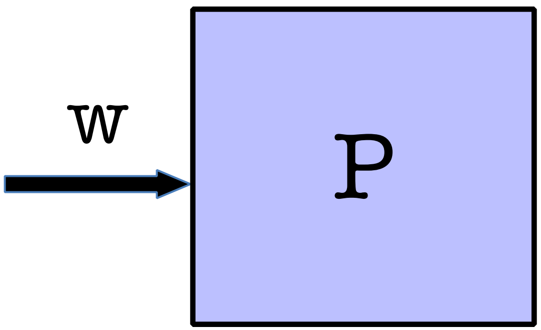
- British mathematician
  - cryptanalysis during WWII
  - arguably, father of AI, CS Theory
  - several books, movies

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

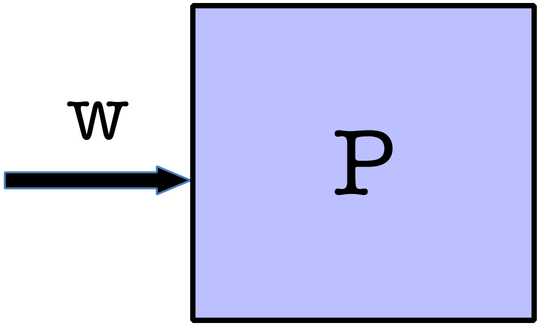
Halting Problem

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



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...

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

Does P halt ? ⟵ **Solves Goldbach conjecture!**

Why would we care about the Halting Problem?

Does <u>Find-proof</u> halt on w? ≡ Is w a provable theorem?

w

```
Program Find-proof(w)

p := empty-string
forever
    p := successor(p)
    if Verify-proof(w,p)
    then halt
```

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)

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)

Enough to model *interactive* computation too:
`P*(x,state)` outputs `(y,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

1	ε
2	0
3	1
4	00
5	01
6	10
7	11
8	000
9	001
10	010
11	011
12	100

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
- The set of all programs is countable.

1	ε
2	0
3	1
4	00
5	01
6	10
7	11
8	000
9	001
10	010
11	011
12	100

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

1	ε	***0***
2	0	***0***
3	1	***1***
4	00	***0***
5	01	***1***
6	10	***1***
7	11	***0***
8	000	***0***
9	001	***1***
10	010	***1***
11	011	***0***
12	100	***1***

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

1	ε	***0***
2	0	***0***
3	1	***1***
4	00	***0***
5	01	***1***
6	10	***1***
7	11	***0***
8	000	***0***
9	001	***1***
10	010	***1***
11	011	***0***
12	100	***1***

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]

1	ε	***0***
2	0	***0***
3	1	***1***
4	00	***0***
5	01	***1***
6	10	***1***
7	11	***0***
8	000	***0***
9	001	***1***
10	010	***1***
11	011	***0***
12	100	***1***

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

There are uncountably many functions!

But only countably many programs

Almost every function is uncomputable!

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

<u>Theorem</u> [Post'46]: Halting Problem (formulated for "Turing Machines") reduces to <u>PostCP</u> — a "combinatorial" problem

Post Correspondence Problem

<u>Theorem</u> [Post'46]: Halting Problem (formulated for "Turing Machines") reduces to <u>PostCP</u> — a "combinatorial" problem

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

b	**ba**	**abb**	**abb**	**a**
bbb	**bbb**	**a**	**baa**	**ab**

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

abb	**ba**	**abb**	**a**	**abb**	**b**
a	**bbb**	**a**	**ab**	**baa**	**bbb**

Post Correspondence Problem

<u>Theorem</u> [Post'46]: Halting Problem (formulated for "Turing Machines") reduces to <u>PostCP</u> — a "combinatorial" problem

<u>PostCP is uncomputable.</u>

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

b	**ba**	**abb**	**abb**	**a**
bbb	**bbb**	**a**	**baa**	**ab**

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

abb	**ba**	**abb**	**a**	**abb**	**b**
a	**bbb**	**a**	**ab**	**baa**	**bbb**

Post Correspondence Problem

<u>PostCP is uncomputable.</u>

Post Correspondence Problem

<u>PostCP is uncomputable.</u>

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:
http://en.wikipedia.org/wiki/List_of_undecidable_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 ✅

In ofs

- **Theorem** actly n-1 mov ble an n-piece ji espective of wh s)

Proof cti

Base -piece es 0 moves

Induc p: Conside

Assu n-1)-piece pu es

Consid iece puzzle:

n-2 moves

One more mov

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

Ind ofs

- **Theorem** ctly n-1 move ble an n-piece ji espective of wh s)

Proof cti

Base -piece es 0 moves

Induc p: Conside

Assu n-1)-piece pu es

Consid iece puzzle:

n-2 moves

One more mov

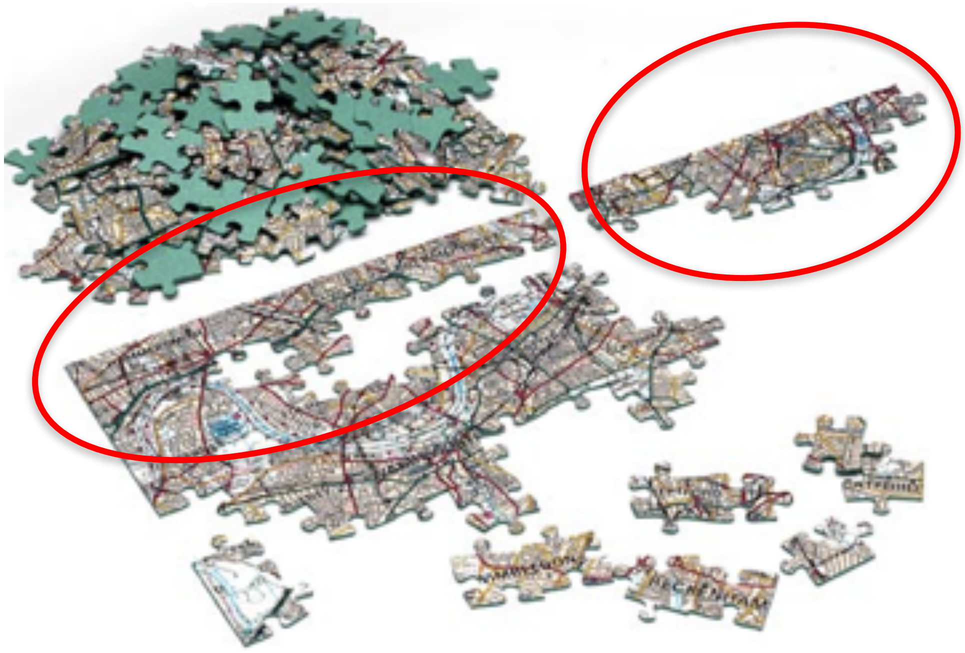
total = (n-2)+1 = n-1

Inductive Proofs

Why must last move look like this?

Inductive Proofs

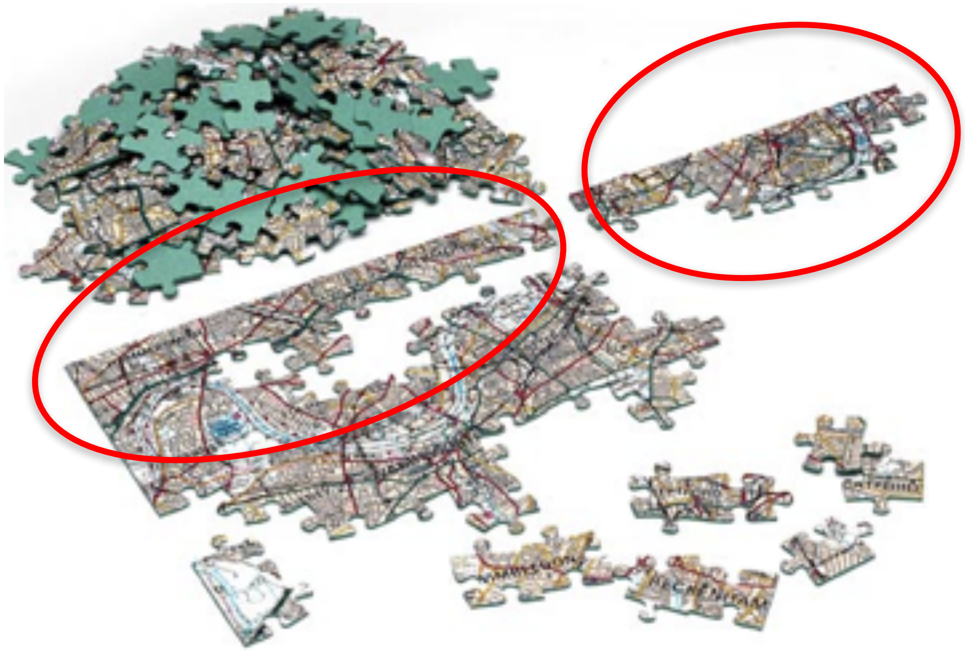
Why must last move look like this?



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 = ⟨**some small values**⟩.
  Then ⟨**show claim holds for n**⟩

- Induction Step: Consider any *arbitrary* integer n ⟨**greater than base-case values**⟩.

  Induction hypothesis: Assume that for all integers k < n (and k ≥ ⟨**smallest value**⟩), ⟨**claim holds for k**⟩

  ⟨**Prove that claim holds for n**⟩

May need a stronger claim than originally asked to prove

Induction Template

- Base Case: Let n = ⟨**some small values**⟩.
  Then ⟨**show claim holds for n**⟩
- Induction Step: Consider any *arbitrary* integer n ⟨**greater than base-case values**⟩.

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

  ⟨**Prove that claim holds for n**⟩

May need a stronger claim than originally asked to prove

Induction Template

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

- Base Case: Let n = ⟨**some small values**⟩. Then ⟨**show claim holds for n**⟩
- Induction Step: Consider any *arbitrary* integer n ⟨**greater than base-case values**⟩.

  Induction hypothesis: Assume that for all integers k < n (and k ≥ ⟨**smallest value**⟩), ⟨**claim holds for k**⟩

  ⟨**Prove that claim holds for n**⟩

May need a stronger claim than originally asked to prove

Induction Template

- Base Case: Let n = ⟨**some small values**⟩. Then ⟨**show claim holds for n**⟩

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

- Induction Step: Consider any *arbitrary* integer n ⟨**greater than base-case values**⟩.

  Induction hypothesis: Assume that for all integers k < n (and k ≥ ⟨**smallest value**⟩), ⟨**claim holds for k**⟩

  ⟨**Prove that claim holds for n**⟩

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

Induction Template

- Base Case: Let n = ⟨**some small values**⟩. Then ⟨**show claim holds for n**⟩

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

- Induction Step: Consider any *arbitrary* integer n ⟨**greater than base-case values**⟩.

  Induction hypothesis: Assume that for all integers k < n (and k ≥ ⟨**smallest value**⟩), ⟨**claim holds for k**⟩

  ⟨**Prove that claim holds for n**⟩

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

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

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

Example

- Base Case: Let n = **1**.
  Then, **any <u>clump</u> 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 ≥ **1**), **any clump with k pieces needs k-1 moves to assemble**

  ⟨**Prove that claim holds for n**⟩

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

Example

- 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 ≥ **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 ≤ 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/
- Immediately join Piazza
- Immediately check access to Moodle

Links to Piazza and Moodle are on course home page