CSC236 Week 3

Larry Zhang

Announcements

- Problem Set 1 due this Friday
- Make sure to read “Submission Instructions” on the course web page.
- “Search for Teammates” on Piazza
- Educational memes:
    - http://www.cs.toronto.edu/~ylzhang/csc236/memes.html
    - Contribute by emailing Larry.

Recap

We learned about **simple induction**

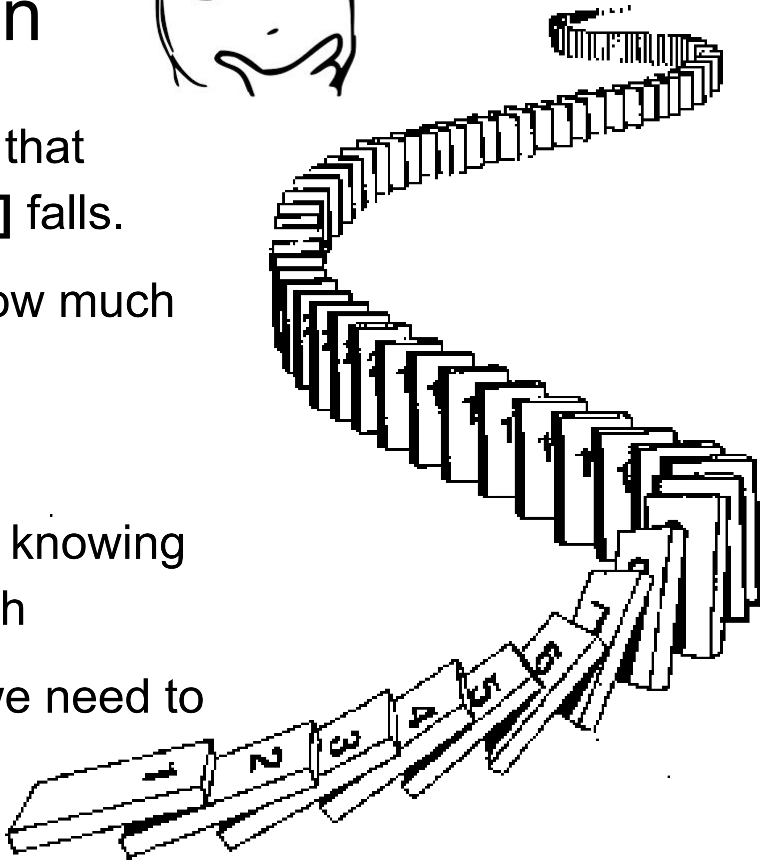
1. Define predicate P(n)
2. Base case: Show P(b)
3. Induction step
   - Assume P(k) # **Induction Hypothesis**
   - Show P(k+1)

Q.E.D.

Simple induction is great,
but sometimes it is not enough

Think about the dominoes again

- What simple induction says is that, to show that **d[236]** falls, all I need to know is that **d[235]** falls.
- But by knowing **d[235]** falls, we actually know much more…
- We also know **d[1]** to **d[234]** all fall
  - We didn't use this information because knowing that **d[235]** falls happened to be enough
  - But sometimes it is NOT enough and we need to use **all the information** we know.



In other words

What we did in **simple induction**

- Suppose **P(0)** is True
- Then we use **P(0)** to prove **P(1)** is True
- Then we use **P(1)** to prove **P(2)** is true.
- Then we use **P(2)** to prove **P(3)** is true
- ……

- Suppose **P(0)** is True
- Then we can use **P(0)** to prove **P(1)** is True
- Then we can use **both P(0) and P(1)** to prove **P(2)** is true.
- Then we can use **P(0), P(1) and P(2)** to prove **P(3)** is true
- ……
- This is called **complete (strong) induction**.

Complete (Strong) Induction

$\wedge$ means "and"

Principle of Complete Induction

(i) If $P(b)$ is True,

(ii) And $P(b) \wedge P(b+1) \wedge \ldots \wedge P(n-1) \Rightarrow P(n)$ is True for all $n > b$,

Then $P(n)$ is True for **all** integers $n \geq b$.

Induction Hypothesis

Notice the detail with n-1 and n, n > b and n >= b.
Exercise: rewrite it into an equivalent form using P(n+1)

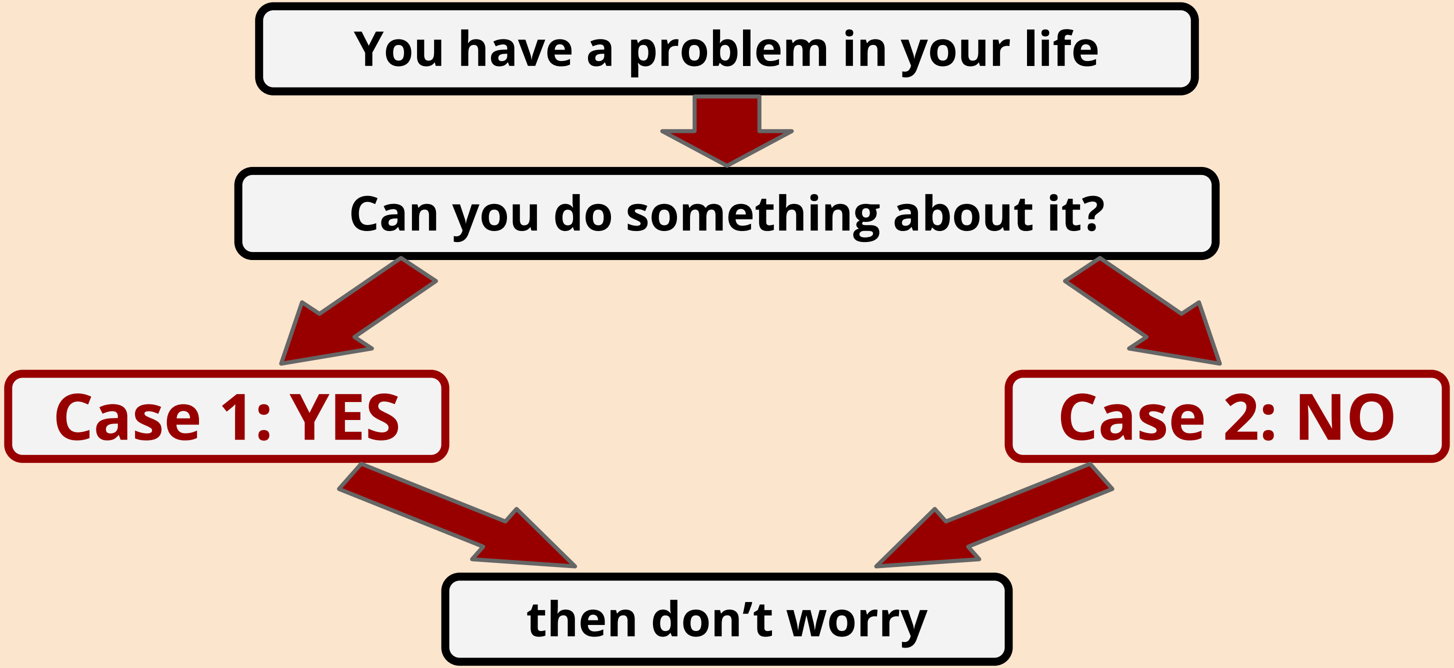
Example 1

Interlude: proof by cases

→ **split** your argument into differences cases

→ prove the conclusion **for each** case

Prove: If you have a problem in your life, then don't worry.



What makes it a valid proof?

The union of the cases are covering ALL possibilities.

Prime or Product of Primes

Prove that every natural number greater than 1 can be written as a product of primes.

For example:

$2 = 2$
$3 = 3$
$4 = 2 \times 2$
$5 = 5$
$6 = 2 \times 3$
$28 = 2 \times 2 \times 7$
$236 = 2 \times 2 \times 59$

Let’s try simple induction ...

Define predicate P(n): n can be decomposed into a product of primes

Base case: n=2

2 is already a product of primes (2 is prime), so we're done.

Induction Step:

Assume **n >= 2** and that **n** can be written as a product of primes.
Need to prove that **n+1** can be written as a product of primes...

Imagine that we know that 8 can be written as a product of primes. (2x2x2)
How does this help us decompose 9 into a product of primes? (3x3)
Not obvious!

Problem: There is no obvious relation between the decomposition of k and the decomposition of k+1. Simple induction not working!

Use Complete Induction

Define predicate P(n): n can be decomposed into a product of primes. (same as before)

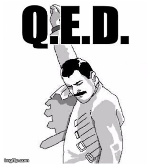
Base case: n=2, 2 is already a product of primes (2 is prime), so we're done. (same as before)

Induction Step:

Assume ***P(2) ∧ P(3) ∧ P(4) ∧ … ∧ P(n-1)***, i..e, all numbers from 2 to n-1 can be written as a product of primes. (**Induction Hypothesis of Complete Induction**)

Now need to show ***P(n)***, i.e., ***n*** can be written as a product of primes

- Case 1: n is prime …
  - then n is already a product of primes, done
- Case 2: n is composite (not prime) …
  - then n can be written as **n = a x b**, where a & b satisfies **2 <= a,b <= n-1**
  - According to **I.H.**, each of ***a*** and ***b*** can be written as a product of primes.
  - So n = **a x b** can be written as a product of primes.



Takeaways

- If jumping “**one number back**” is sufficient to prove the claim for the next number, then use **simple** induction
- If jumping **further back** is necessary, then use **complete** induction
- The structure/steps of complete induction is **very similar** to that of simple induction; the only difference is how the **induction hypothesis** is made.

Example 2

The Unstacking Game (A Simple Version)

The Unstacking Game

The rule:

- You begin with a stack of **n** cups
- Each move of the game involves dividing a stack of cups into **two** stacks
- The game ends when you have **n** stacks, with **one** cup per stack
- For each move, you get **points**. (This is a single player game)
- If you divide a stack of **a + b** cups into a stack of **a** cups and a stack of **b** cups, you get **a x b** points
- Your **final score** is the **sum** of the points for each move

The Unstacking Game

Example:

- Start with 5 cups in one stack, **xxxxx**
- Move #1, divide into two stacks of 2 and 3, **xx xxx**
  - Get 6 points
- Move #2, divide the 2 into 1 and 1, **x x xxx**
  - Get 1 point
- Move #3, divide 3 into 1 and 2, **x x x xx**
  - Get 2 points
- Move #4, divide 2 into 1 and 1, **x x x x x**
  - Get 1 point
- **Final score: 6 + 1 + 2 + 1 = 10**

There's more than one way. Can you do better than this?

The Unstacking Game

Prove that, given **n** cups, no matter how we unstack them, the final score we get is always:

$$\frac{n(n-1)}{2}$$

The Unstacking Game

$$\frac{n(n-1)}{2}$$

Step 1:

Define the predicate

P(n): with **n** cups, the final score is **n(n-1)/2**

The Unstacking Game

$$\frac{n(n-1)}{2}$$

Step 2:

Base Case:

n = 1

No unstacking move can be made, so final score is **0**

which is equal to **1(1-1)/2 = 0**.

So base case done.

The Unstacking Game

$$\frac{n(n-1)}{2}$$

Step 3:

Induction Hypothesis:

Assume ***P(1) ∧ P(2) ∧ … ∧ P(n-1)***, i.e., the final scores for games with **1** to **n-1** cups conform to the formula.

Now, make first move which divide **n** into a **k** stack and a **n-k** stack,

for some **0 < k < n**. (the choice of **k** is arbitrary)

- Points we get from this first move: **k(n-k)** points
- According to I.H., unstacking the k stack give us **k(k-1)/2** points
- Again according to I.H., unstacking the **n-k** stack gives **(n-k)(n-k-1)/2** pints

Now we just need to show the sum of these three parts is actually n(n-1)/2

$$\frac{n(n-1)}{2}$$

The Unstacking Game

Just need to show:

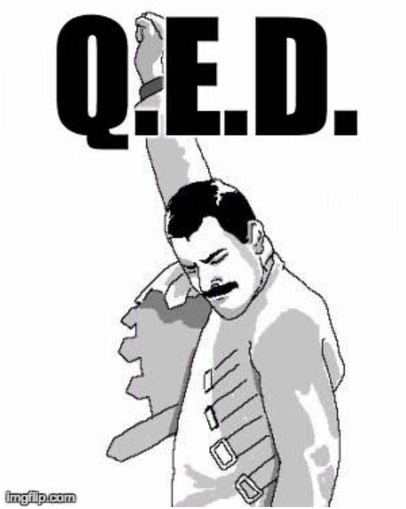
$$k(n-k) + (k(k-1))/2 + ((n-k)(n-k-1))/2$$

$$= kn - k^2 + (k^2 - k)/2 + (n^2 - kn - n - kn + k^2 + k)/2$$

$$= (2kn - 2k^2 + k^2 - k + n^2 - kn - n - kn + k^2 + k)/2$$

$$= (n^2 - n)/2$$

$$= n(n-1)/2$$



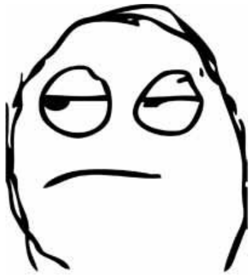
Summary

Your friend

I played that unstacking game with 200 cups and got 19000 points. You think you can beat me?

You

Of course …, don't even need to use my brain to play; BTW you got your score wrong.



Serious Summary

The proof structure for both simple and complete induction:

- Define the **predicate** P(n)
- Prove for **base case**
- Make **induction hypothesis**, and use the induction hypothesis to prove the induction step
- Use **principle of (simple/complete) induction** to conclude that P(n) is true for the base and all larger numbers.

The **difference** between simple and complete: the **induction hypothesis**.

- Simple induction: Assume P(n-1) show P(n), or assume P(n), show P(n+1)
- Complete induction: Assume P(b) ∧ … ∧ P(n-1) show P(n)

For Home Thinking

For the Unstacking Game, could we choose n=0 as the base case, instead of choosing n=1?

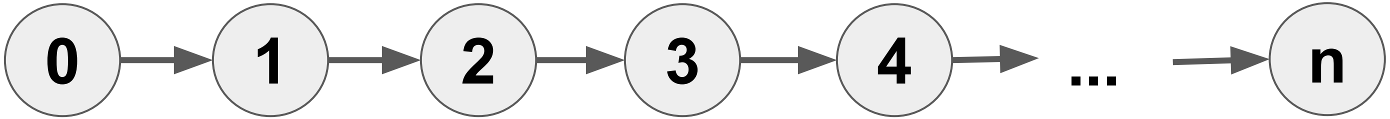
The answer is NO, but why?

Structural Induction

a more general type of induction

What simple and complete inductions share in common

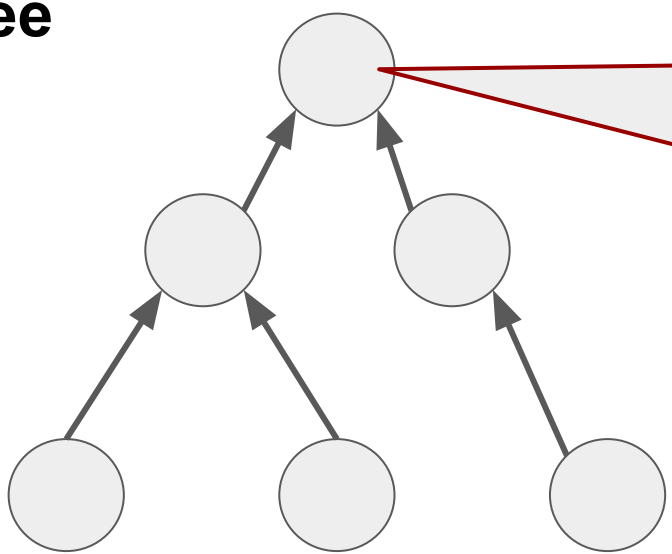
- They both prove things on **natural numbers**.
- The **structure** of natural numbers is very simple
- It is just **a single line**



What if we have something whose structure is more complex than a single line?

An example of something with more complex structure

A **tree**



For example, if I want to prove some property of the root node (or of the whole tree), **simple** and **complete** induction would NOT work, because it's not clear what 1, 2, 3, …, k and k+1 are.

Structural induction would work for this.

Structural Induction can be used to prove statements on **recursively-defined sets**

What's this?

Recursively-Defined Sets

They are the sets that are defined like this:

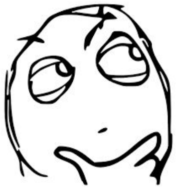
- We give the **base elements** of the set.
- Then we give **recursive rules** for generating new elements of the set from existing elements in the set.

Example: Recursively-Defined Naturals Numbers

The set of natural numbers **N** can be defined as the **smallest** set such that

- $0 \in \mathbb{N}$ **# base elements**
- If $k \in \mathbb{N}$, then $k + 1 \in \mathbb{N}$ **# recursive rule for generating other elements**

This reminds me of simple induction!



The word "smallest" is important because it guarantees that the set has only 0, 1, 2, 3, …, and does NOT have unnecessary elements like 2.5, 3.14, -100, etc.

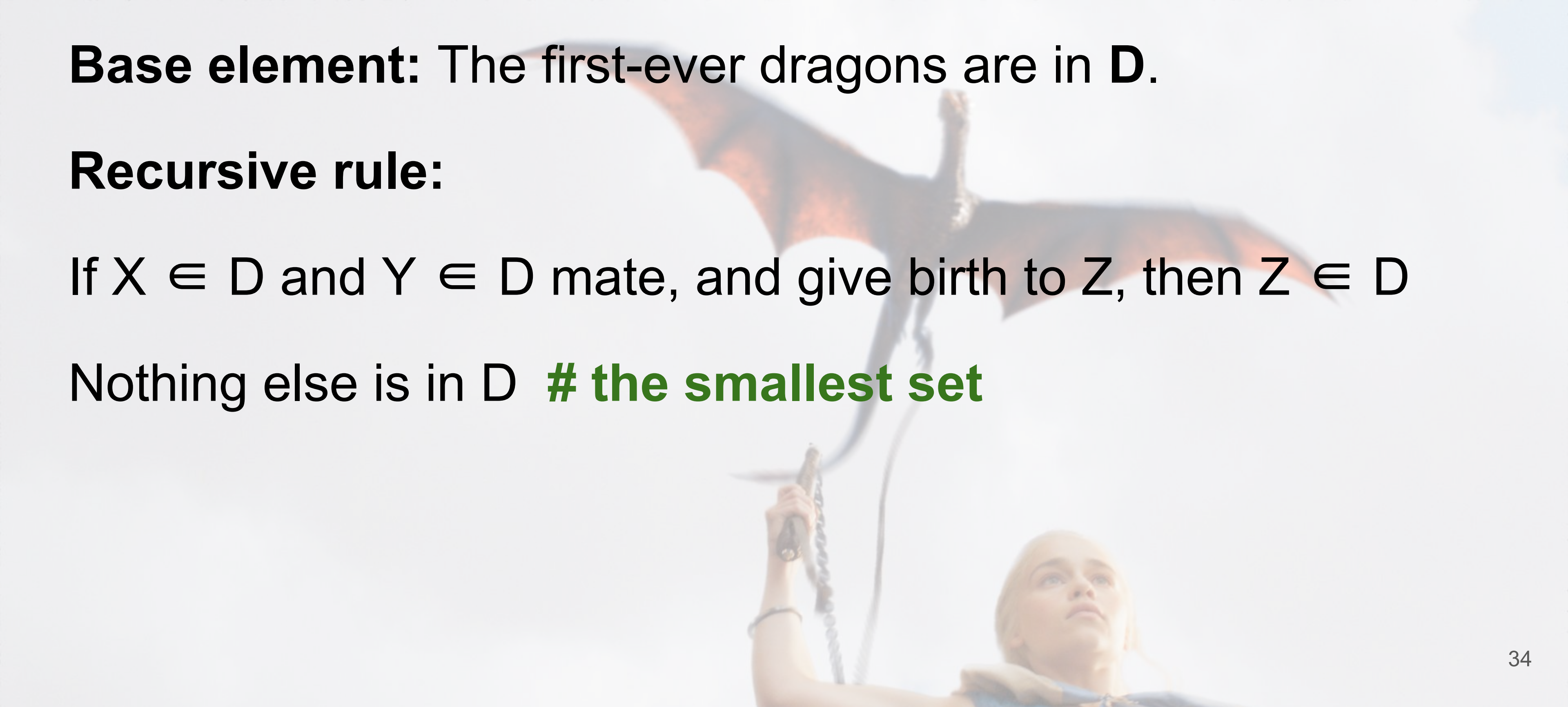
Another example: Define the set of "pure dragons" **D**.

Base element: The first-ever dragons are in **D**.

Recursive rule:

If X ∈ D and Y ∈ D mate, and give birth to Z, then Z ∈ D

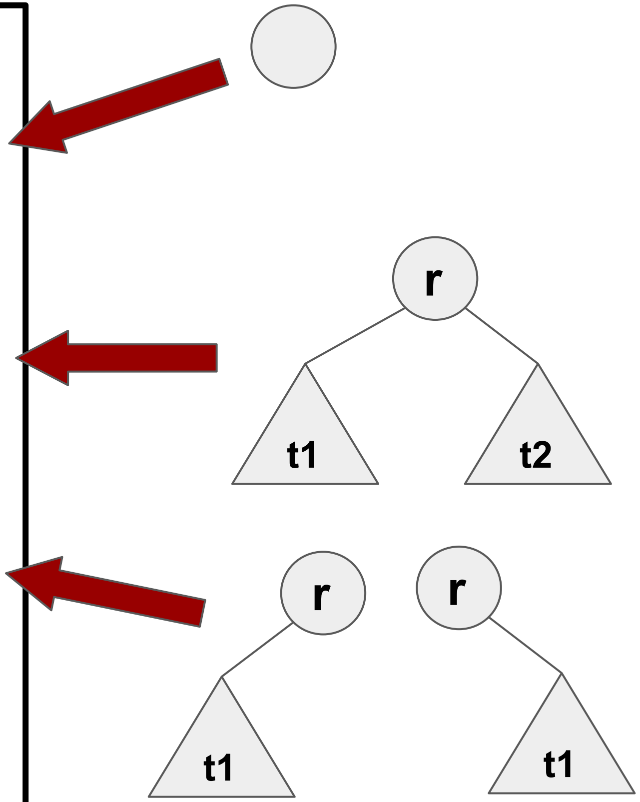
Nothing else is in D **# the smallest set**



Yet Another Example: Recursively-Defined Trees

The set ***T*** of non-empty binary trees is defined as:

- **Base elements:** A single node is an element of ***T***, i.e., a single node is a non-empty binary tree
- **Recursive rule #1:** If **t1** and **t2** are two non-empty binary trees in ***T***, the bigger tree with root **r** connected to the roots of **t1** and **t2** is in ***T***.
- **Recursive rule #2:** If **t1** is a non-empty binary tree in ***T***, then the bigger root with root **r** connected to the root of **t1** on the left or right is in ***T***.
- **Nothing else is in *T*.** **# "smallest set"**



Induction and Recursive Structure

- Recursively-defined sets have the structure (**recursive structure**) that is suitable for structural inductions.

- Simple Induction (or complete induction) is really just a special case of structural induction.

  - when the recursive structure is on “a single line”

Principle of Structural Induction

Suppose that **S** is a recursively-defined set and **P** is some predicate

- **Base case:** If **P** is true for each base element of **S**, and
- **Induction Step:** under the assumption that **P(e)** is true for element **e** of **S**, we find that **each** recursive rule generates an element that satisfies **P**.
- Then **P** is true for all elements of **S**.

We must do the induction step for every recursive rule!

Proof Example 0

Prove: All pure dragons breathe fire

Define the predicate P(X): x breathe fire

Base case: The first-ever dragons breathe fire (as a fact).

Induction step:

Assume X and Y are pure dragons,

By **induction hypothesis**: both X and Y breathe fire.

X and Y gives birth to Z, then **# recursive rule**

…. (some genetic argument) ….

Then Z must breathe fire.

Proof Example 1

Given the recursive definition of **non-empty binary trees**

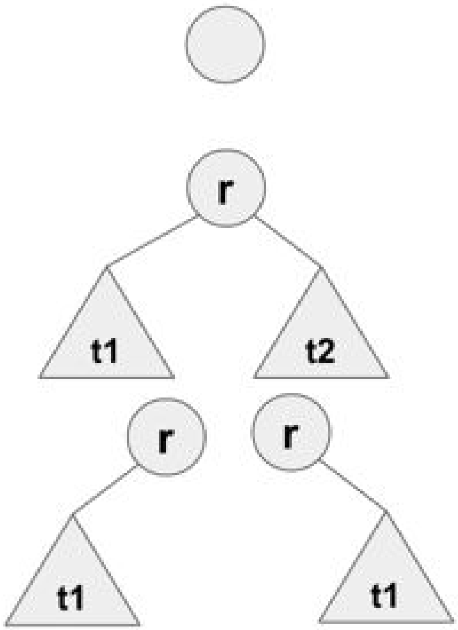
P(t): tree ***t*** has **one more** node than edge

Use structural induction to prove that ***P*** is true for all non-empty binary trees

Let **V(t)** the number of nodes in **t**, and **E(t)** be the number of edges in **t**, we want to prove:

$$\mathbf{V(t) = E(t) + 1}$$

for all non-empty binary tree **t**



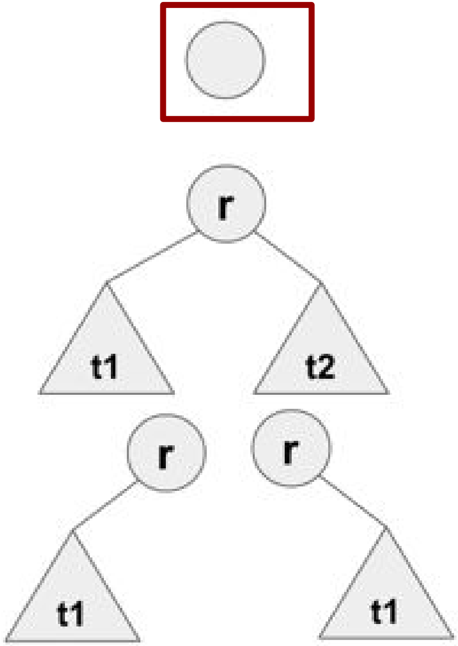
P(t): tree ***t*** has **one more** node than edge

Base Case:

Tree ***t*** has only **one** node

Then ***t*** has 1 node and 0 edge

P(t) is true.



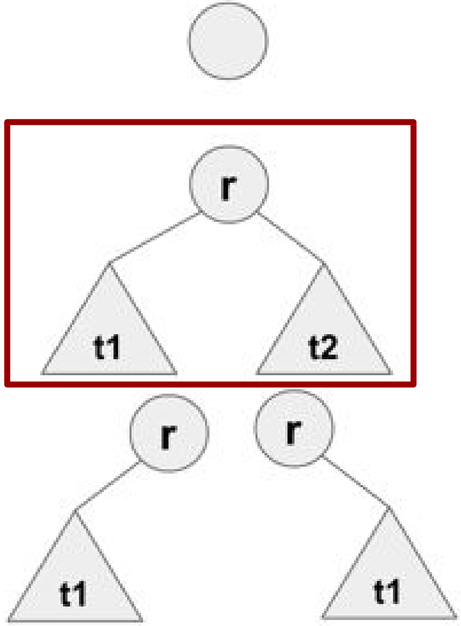
P(t): tree *t* has **one more** node than edge

Induction Step:

Case 1: root of new tree t connects to t1 and t2

- Assume that **t1** and **t2** are two non-empty binary trees.
- By induction hypothesis, **P(t1)** and **P(t2)** hold, i.e.,
  - **V(t1) = E(t1) + 1**
  - **V(t2) = E(t2) + 1**
- **t** adds one new node (the new root),
- and add two new edges (connecting to **t1** and **t2**)

$$V(t) = V(t_1) + V(t_2) + 1$$
$$E(t) = E(t_1) + E(t_2) + 2$$



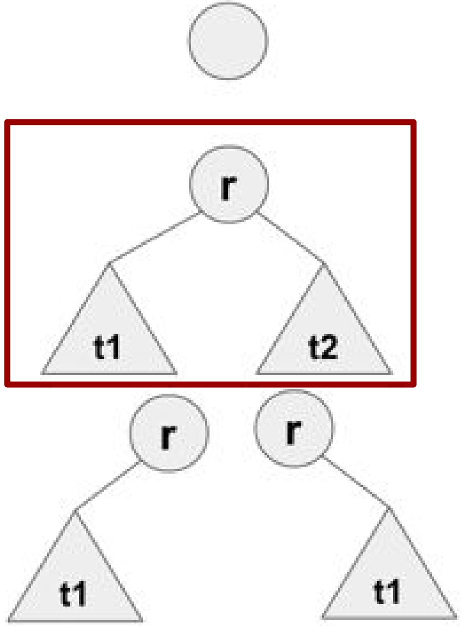
P(t): tree ***t*** has **one more** node than edge

Induction Step:

Case 1: new root connects to t1 and t2

- **V(t1) = E(t1) + 1**
- **V(t2) = E(t2) + 1**

$$V(t) = V(t_1) + V(t_2) + 1$$
$$E(t) = E(t_1) + E(t_2) + 2$$



$$\begin{aligned} V(t) &= V(t_1) + V(t_2) + 1 \\ &= E(t_1) + 1 + E(t_2) + 1 + 1 \\ &= (E(t_1) + E(t_2) + 2) + 1 \\ &= E(t) + 1 \end{aligned}$$

Case 1 done.

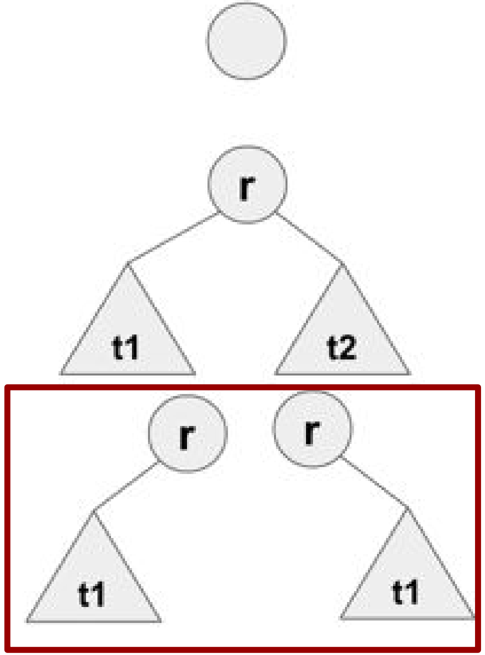
P(t): tree *t* has **one more** node than edge

Induction Step:

Case 2: root of new tree t connects to t1 (left or right)

- Assume **t1** is a non-empty binary tree
- By induction hypothesis, **P(t1)** holds.
- i.e., **V(t1) = E(t1) + 1**
- **t** adds one new node, and one new edge, i.e.,

$$V(t) = V(t_1) + 1$$
$$E(t) = E(t_1) + 1$$



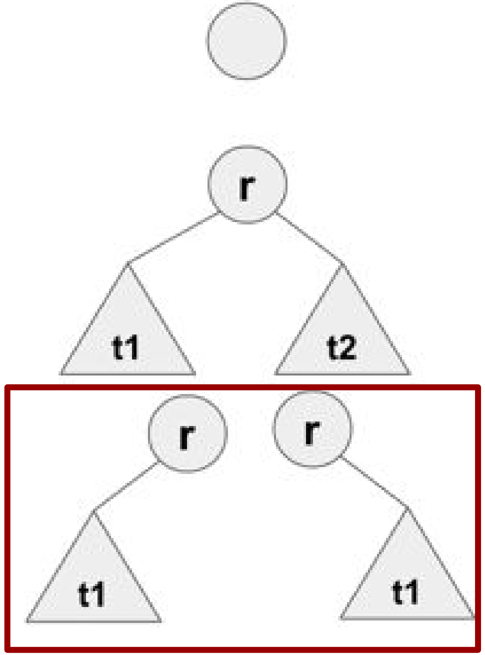
P(t): tree ***t*** has **one more** node than edge

Induction Step:

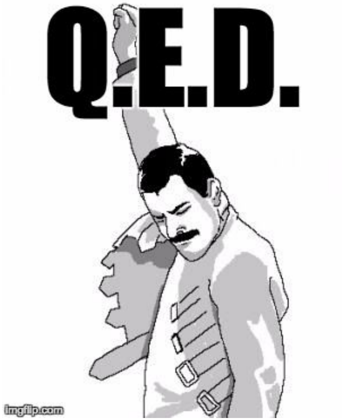
Case 2: root of new tree t connects to t1 (left or right)

- **V(t1) = E(t1) + 1 # I.H.**

$$V(t) = V(t_1) + 1$$
$$E(t) = E(t_1) + 1$$



$$\begin{aligned} V(t) &= V(t_1) + 1 \\ &= (E(t_1) + 1) + 1 \\ &= E(t) + 1 \end{aligned}$$



Case 2 done.

Summary

To get structural induction right

- Make sure to prove for **all base cases**
  - There could be more than one base cases
- Make sure to prove for **all recursive rules**

More exercises in the tutorial!

So far we have learned

- Simple induction
- Complete induction
- Structural induction

These are the basic mathematical tools that we will use a lot in the study of computer science.

Now let’s move on to something more “CS”.

NEW TOPIC

Asymptotic Notations

Big-Oh, Big-Omega, Big-Theta

Computer scientists talk like...

"The worst-case runtime of bubble-sort is in $O(n^2)$."

"I can sort it in $n \log n$ time."

"That's too slow, make it linear-time."

"That problem cannot be solved in polynomial time."

compare two sorting algorithms

bubble sort

merge sort

demo at *http://www.sorting-algorithms.com/*

Observations

→ **merge** is faster than **bubble**

→ with larger input size, the advantage of **merge** over **bubble** becomes larger

compare two sorting algorithms

	20	**40**
bubble	8.6 sec	38.0 sec
merge	5.0 sec	11.2 sec

*when input size **grows** from 20 to 40…*

→ *the **"running time"** of merge roughly doubled*

→ *the **"running time"** of bubble roughly quadrupled*

what does “running time” really mean in computer science?

- ➔ It does **NOT** mean how many **seconds** are spent in running the algorithm.
- ➔ It means **the number of steps** that are taken by the algorithm.
- ➔ So, the running time is **independent of the hardware** on which you run the algorithm.
- ➔ It only depends on the algorithm itself.

You can run **bubble** on a supercomputer and run **merge** on a slow IBM PC-286, that has nothing to do with the fact that **merge** is a faster sorting algorithm than **bubble**.

describe algorithm running time in steps

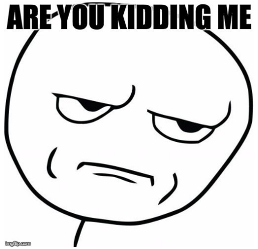
	20	40
bubble	***200 steps***	***800 steps***
merge	***120 steps***	***295 steps***

number of steps as a function of ***n***, the size of input

➔ the running time of bubble could be ***$0.5n^2$*** *(steps)*

➔ the running time of merge could be ***n log n*** *(steps)*

but, we don't really care about the number of steps…



- ➔ what we really care: how the number of steps **grows** as the size of input **grows**
- ➔ we **don't care** about the **absolute** number of steps
- ➔ we care about: "*when input size **doubles**, the running time **quadruples***"
- ➔ so, ***$0.5n^2$*** and ***$700n^2$*** are **no different**!
- ➔ **constant factors do NOT matter!**

constant factor does not matter, when it comes to growth

$$T_1(n) = 0.5\,n^2 \qquad\qquad T_2(n) = 700\,n^2$$

$$\frac{T_1(2n)}{T_1(n)} = \frac{0.5\,(2n)^2}{0.5\,n^2} = \frac{2n^2}{0.5n^2} = 4$$

$$\frac{T_2(2n)}{T_2(n)} = \frac{700\,(2n)^2}{700\,n^2} = \frac{2800\,n^2}{700\,n^2} = 4$$

We care about large input sizes

- ➔ We don’t need to study algorithms in order to sort **two** elements, because different algorithms make no difference
- ➔ We care about algorithm design when the input size ***n*** is very large
- ➔ So, n^2 and n^2+n+2 are no different, because when n is really large, $n+2$ is negligible compared to n^2
- ➔ ***only the highest-order term matters***

low-order terms don't matter

$$T_1(n) = n^2 \qquad T_2(n) = n^2 + n + 2$$

$$T_1(10000) = 100,000,000$$

$$T_2(10000) = 100,010,002$$

$$\text{difference} \approx 0.01\%$$

Summary of running time

- ➔ we count the number of steps
- ➔ constant factors don't matter
- ➔ only the highest-order term matters

*so, the followings functions are **of the same class***

$$n^2 \qquad 2n^2 + 3n \qquad \frac{n^2}{165} + 1130n + 3.14159$$

For example, we could call this class O(n²)

O(n^2) is an asymptotic notation

O(f(n)) *is the asymptotic* ***upper-bound***

➔ *the set of functions that grows* ***no faster*** *than* ***f(n)***

➔ *for example, when we say*

$$5n^2 + 3n + 1 \text{ is in } O(n^2)$$

we mean

$$5n^2 + 3n + 1 \text{ grows no faster than } n^2\text{, asymptotically}$$

O(***f(n)***): *the asymptotic* ***upper-bound***

- *"Grows no faster than f(n)"*

Ω(***f(n)***): *the asymptotic* ***lower-bound***

- *"Grows no slower than f(n)"*

Θ(***f(n)***): *the asymptotic* ***tight-bound***

- *"Grows no faster and no slower than f(n)"*

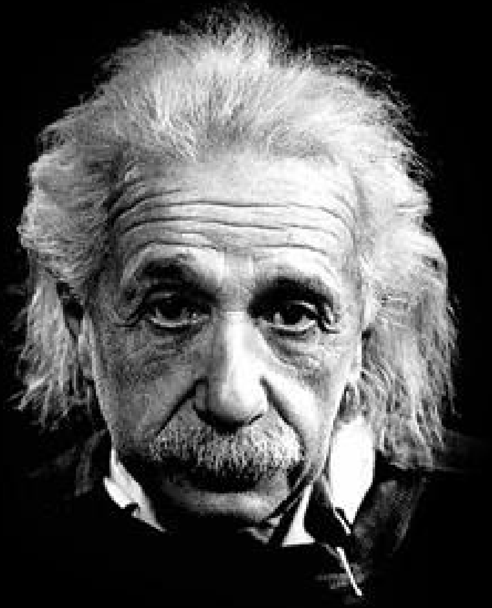
More formal definitions later

a high-level look at asymptotic notations

*It is a **simplification** of the "real" running time*

➔ *it does not tell the whole story about how fast a program runs in real life.*
  - *in real-world applications, constant factor matters! hardware matters! implementation matters!*

➔ *this simplification makes possible the development of the whole **theory of computational complexity**.*
  - ***HUGE idea!***



“Make everything as
simple as possible,
but not simpler.”

—Albert Einstein