UNIVERSITY AT ALBANY
State University of New York

Lecture 2: Induction I and II

Dr. Chengjiang Long
Computer Vision Researcher at Kitware Inc.
Adjunct Professor at SUNY at Albany.
Email: **clong2@albany.edu**

Recap Previous Lecture

1. Propositional Logical
2. Predicate Logical and Quantifiers
3. Rules of inferences
4. Proofs

Identities (Equivalences)	Name
$p \wedge T \equiv p$ $p \vee F \equiv p$	Identity laws
$p \vee T \equiv T$ $p \wedge F \equiv F$	Domination laws
$p \vee p \equiv p$ $p \wedge p \equiv p$	Idempotent laws
$\neg(\neg p) \equiv p$	Double negation law
$p \vee q \equiv q \vee p$ $p \wedge q \equiv q \wedge p$	Commutative laws
$(p \vee q) \vee r \equiv p \vee (q \vee r)$ $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$	Associative laws
$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$ $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$	Distributive laws
$\neg(p \wedge q) \equiv \neg p \vee \neg q$ $\neg(p \vee q) \equiv \neg p \wedge \neg q$	De Morgan's laws
$p \vee (p \wedge q) \equiv p$ $p \wedge (p \vee q) \equiv p$	Absorption laws
$p \vee \neg p \equiv T$ $p \wedge \neg p \equiv F$	Negation laws

Logical Equivalences Involving Conditional Statements.
$p \to q \equiv \neg p \vee q$ $p \to q \equiv \neg q \to \neg p$ $p \vee q \equiv \neg p \to q$ $p \wedge q \equiv \neg(p \to \neg q)$ $\neg(p \to q) \equiv p \wedge \neg q$ $(p \to q) \wedge (p \to r) \equiv p \to (q \wedge r)$ $(p \to r) \wedge (q \to r) \equiv (p \vee q) \to r$ $(p \to q) \vee (p \to r) \equiv p \to (q \vee r)$ $(p \to r) \vee (q \to r) \equiv (p \wedge q) \to r$

Statement	True when...	False when...
$\neg\exists x\, P(x) \equiv \forall x\, \neg P(x)$	$P(x)$ is false for every x	There is an x for which $P(x)$ is true
$\neg\forall x\, P(x) \equiv \exists x\, \neg P(x)$	There is an x for which $P(x)$ is false	$P(x)$ is true for every x

Recap Previous Lecture

- Rules of Inferences
  - Modus Ponens $(p \wedge (p \rightarrow q)) \rightarrow q$
  - Addition $p \rightarrow (p \vee q)$
  - Simplification $(p \wedge q) \rightarrow p$
  - Conjunction $((p) \wedge (q)) \rightarrow (p \wedge q)$
  - Modus Tollens $(\neg q \wedge (p \rightarrow q)) \rightarrow \neg p$
  - Contrapositive $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$
  - Hypothetical Syllogism $((p \rightarrow q) \wedge (q \rightarrow r)) \rightarrow (p \rightarrow r)$
  - Disjunctive Syllogism $((p \vee q) \wedge \neg p) \rightarrow q$
  - Resolution $((p \vee q) \wedge (\neg p \vee r)) \rightarrow (q \vee r)$

Recap Previous Lecture

- Proofs
  - Trivial proof (without using the premise)
  - Vacuous proof (the premise p is false)
  - Direct proof ($p \rightarrow q$)
  - Indirect / Contrapositive proof ($\neg q \rightarrow \neg p$)
  - Proof by contradiction ($\neg p \rightarrow (r \land \neg r)$, where r is a known result)
  - Proof by cases (break the statements down into cases and prove each one separately)

Outline

- Mathematical Induction
- Strong Induction and Well-Ordering
- Structural Recursion and Induction
- Recursive Algorithms

Mathematical Induction

Odd Power are Odd

Fact: If m is odd and n is odd, then nm is odd.

Proposition: for an odd number m, m^k is odd for all non-negative integer k.

$$\forall k \in N \;\; odd(m^k)$$

Let P(i) be the proposition that m^i is odd.

$$\forall k \in N \;\; P(k)$$

Idea of induction.

- P(1) is true by definition.
- P(2) is true by P(1) and the fact.
- P(3) is true by P(2) and the fact.
- P(i+1) is true by P(i) and the fact.
- So P(i) is true for all i.

Divisibility by a Prime

Theorem. Any integer $n > 1$ is divisible by a prime number.

- Let n be an integer.
- If n is a prime number, then we are done.
- Otherwise, $n = ab$, both are smaller than n.
- If a or b is a prime number, then we are done.
- Otherwise, $a = cd$, both are smaller than a.
- If c or d is a prime number, then we are done.
- Otherwise, repeat this argument, since the numbers are getting smaller and smaller, this will eventually stop and we have found a prime factor of n.

Idea of induction.

Idea of Induction

Objective: Prove $\forall n \geq 0 \; P(n)$

This is to prove

$$\begin{tikzcd}
\underline{P(0)} \arrow[rr, bend right] & \wedge & \underline{P(1)} \arrow[rr, bend right] & \wedge & \underline{P(2)} \arrow[r, bend right] & \wedge \ldots \arrow[rr, bend right] & \wedge & \underline{P(n)} \arrow[r, bend right] & \ldots
\end{tikzcd}$$

The idea of induction is to first prove P(0) unconditionally,

then use P(0) to prove P(1)

then use P(1) to prove P(2)

and repeat this to infinity…

Idea of Induction

0 and (from n to n +1),

proves 0, 1, 2, 3,….

Very easy to prove

Much easier to prove with P(n) as an assumption.

$$\frac{P(0),\ P(n) \rightarrow P(n+1)}{\forall m \in \underline{N}.\ P(m)}$$

For any n>=0

Like domino effect…

Proof by Induction

Let's prove:

$$\forall r \neq 1.\ 1 + r + r^2 + \cdots + r^n = \frac{r^{n+1} - 1}{r - 1}$$

Statements in green form a template for inductive proofs.

Proof: (by induction on *n*)

The induction hypothesis, *P*(*n*), is:

$$\forall r \neq 1.\ 1 + r + r^2 + \cdots + r^n = \frac{r^{n+1} - 1}{r - 1}$$

Proof by Induction

Induction Step: Assume $P(n)$ for some $n \geq 0$ and prove $P(n + 1)$:

$$\forall r \neq 1.\ 1 + r + r^2 + \cdots + r^{n+1} = \frac{r^{(n+1)+1} - 1}{r - 1}$$

Have $P(n)$ by assumption:

So let r be any number $\neq 1$, then from $P(n)$ we have

$$1 + r + r^2 + \cdots + r^{n} = \frac{r^{n+1} - 1}{r - 1}$$

How do we proceed?

Proof by Induction

adding r^{n+1} to both sides,

$$
\begin{aligned}
1+\cdots+r^n+r^{\mathbf{n+1}} &= \frac{r^{n+1}-1}{r-1}+r^{n+1} \\
&= \frac{r^{n+1}-1+r^{n+1}(r-1)}{r-1} \\
&= \frac{r^{(\mathbf{n+1})+1}-1}{r-1}
\end{aligned}
$$

$$\forall r \neq 1.\ \ 1+r+r^2+\cdots+r^{\mathbf{n+1}} = \frac{r^{(\mathbf{n+1})+1}-1}{r-1}$$

which is $P(n+1)$. This completes the induction proof.

Proving an Equality

$$\forall n \geq 1 \qquad 1^3 + 2^3 + \ldots + n^3 = (\frac{n(n+1)}{2})^2$$

Let P(n) be the induction hypothesis that the statement is true for n.

Base case: P(1) is true

Induction step: assume P(n) is true, prove P(n+1) is true.

$$1^3 + 2^3 + \ldots + n^3 + (n+1)^3$$

$$= (\frac{n(n+1)}{2})^2 + (n+1)^3$$

by induction

$$= (n+1)^2(n^2/4 + n + 1)$$

$$= (n+1)^2(\frac{n^2+4n+4}{4}) = (\frac{(n+1)(n+2)}{2})^2$$

Proving a Property

$$\forall n \geq 1, \quad 2^{2n} - 1 \text{ is divisible by } 3$$

Base Case ($n = 1$): $2^{2n} - 1 = 2^2 - 1 = 3$

Induction Step: Assume $P(i)$ for some $i \geq 1$ and prove $P(i + 1)$:

Assume $2^{2i} - 1$ is divisible by 3, prove $2^{2(i+1)} - 1$ Is divisible by 3.

$$\begin{aligned} 2^{2(i+1)} - 1 &= 2^{2i+2} - 1 \\ &= 4 \cdot 2^{2i} - 1 \\ &= \underbrace{3 \cdot 2^{2i}}_{\text{Divisible by 3}} + \underbrace{2^{2i} - 1}_{\text{Divisible by 3 by induction}} \end{aligned}$$

Proving a Property

$$\forall n \geq 2, \quad n^3 - n \text{ is divisible by } 6$$

Base Case ($n = 2$): $2^3 - 2 = 6$

Induction Step: Assume $P(i)$ for some $i \geq 2$ and prove $P(i + 1)$:

Assume $n^3 - n$ is divisible by 6

Prove $(n+1)^3 - (n+1)$ is divisible by 6.

$$\begin{aligned}(n+1)^3 - (n+1) &= (n^3 + 3n^2 + 3n + 1) - (n+1) \\ &= \underbrace{(n^3 - n)}_{\text{Divisible by 6 by induction}} + 3\underbrace{(n^2 + n)}_{\text{Divisible by 2 by case analysis}}\end{aligned}$$

Proving an Inequality

$$\forall n \geq 3, \quad 2n + 1 < 2^n$$

Base Case ($n = 3$): $2n + 1 = 7 < 2^n = 2^3 = 8$

Induction Step: Assume $P(i)$ for some $i \geq 3$ and prove $P(i + 1)$:

Assume $2i + 1 < 2^i$, prove $2(i+1) + 1 < 2^{(i+1)}$

$$\begin{aligned} 2(i+1)+1 &= 2i + 1 + 2 \\ &< 2^i + 2 && \text{by induction} \\ &< 2^i + 2^i && \text{since i >= 3} \\ &= 2^{(i+1)} \end{aligned}$$

Proving an Inequality

$$\forall n \geq 2, \quad \frac{1}{\sqrt{1}} + \frac{1}{\sqrt{2}} + \ldots + \frac{1}{\sqrt{n}} > \sqrt{n}$$

Base Case ($n = 2$): is true

Induction Step: Assume $P(i)$ for some $i \geq 2$ and prove $P(i + 1)$:

$$\frac{1}{\sqrt{1}} + \frac{1}{\sqrt{2}} + \ldots + \frac{1}{\sqrt{n}} + \frac{1}{\sqrt{n+1}}$$

$$> \sqrt{n} + \frac{1}{\sqrt{n+1}}$$

by induction

$$= \frac{\sqrt{n}\sqrt{n+1} + 1}{\sqrt{n+1}}$$

$$> \frac{\sqrt{n}\sqrt{n} + 1}{\sqrt{n+1}} = \frac{n+1}{\sqrt{n+1}}$$

$$= \sqrt{n+1}$$

Application: Gray Code

Can you find an ordering of all the n-bit strings in such a way that two consecutive n-bit strings differed by only one bit?

This is called the Gray code and has many applications.

How to construct them?

Think inductively! (or recursively!)

2 bit	3 bit
00	000
01	001
11	011
10	010
	110
	111
	101
	100

Can you see the pattern?

How to construct 4-bit gray code?

Application: Gray Code

3 bit	3 bit (reversed)
000	100
001	101
011	111
010	110
110	010
111	011
101	001
100	000

Every 4-bit string appears exactly once.

4 bit	
0000	
0001	
0011	differed by 1 bit
0010	by induction
0110	
0111	
0101	
0100	differed by 1 bit
1100	by construction
1101	
1111	
1110	
1010	differed by 1 bit
1011	by induction
1001	
1000	

Application: Gray Code

n bit	n bit (reversed)
000…0	100…0
…	…
…	…
…	…
…	…
…	…
…	…
100…0	000…0

n+1 bit	
0 000…0	
0 …	
0 …	differed by 1 bit by induction
0 …	
0 …	
0 …	
0 …	
0 100…0	differed by 1 bit by construction
1 100…0	
1 …	
1 …	
1 …	
1 …	differed by 1 bit by induction
1 …	
1 …	
1 000…0	

Every (n+1)-bit string appears exactly once.

So, by induction,
Gray code exists for any n.

Application: Hadamard Matrix (Optional)

Can you construct an $n \times n$ matrix with all entries ± 1 and all the rows are orthogonal to each other?

Two rows are orthogonal if their inner product is zero.

That is, let $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$,

their inner product $ab = a_1b_1 + a_2b_2 + \ldots + a_nb_n$

This matrix is famous and has many applications.

To think inductively, first we come up with small examples.

$$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Application: Hadamard Matrix (Optional)

Then we use the small examples to build larger examples.

Suppose we have an nxn Hadamard matrix H_n.

We can use it to construct an 2nx2n Hadamard matrix as follows.

$$\begin{bmatrix} H_n & H_n \\ H_n & -H_n \end{bmatrix}$$

Check this!

So by induction there is a 2^k x 2^k Hardmard matrix for any k.

Application: Inductive Construction

This technique is very useful.

We can use it to construct:

- codes
- graphs
- matrices
- circuits
- algorithms
- designs
- proofs
- buildings
- …

Paradox

Theorem: All horses are the same color.

Proof: (by induction on n)

Induction hypothesis:

$P(n)$ ::= any set of n horses have the same color

Base case (n=0):

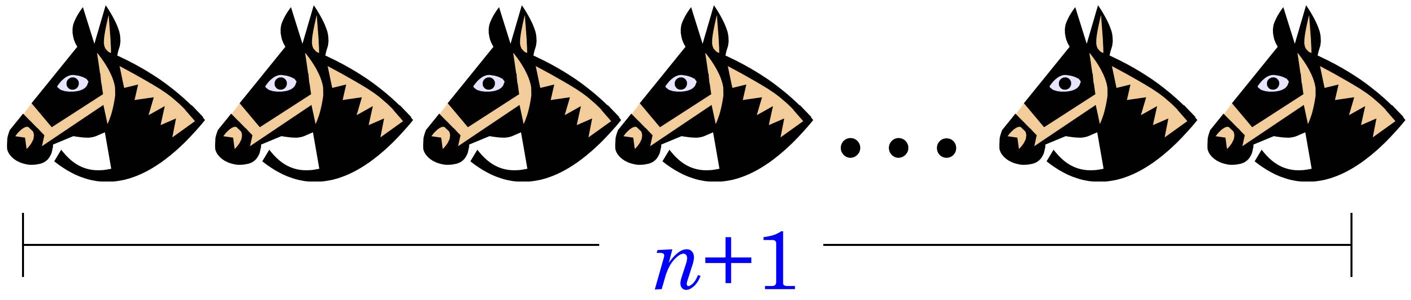
No horses so *obviously* true!

Paradox

(Inductive case)

Assume any n horses have the same color.

Prove that any $n+1$ horses have the same color.

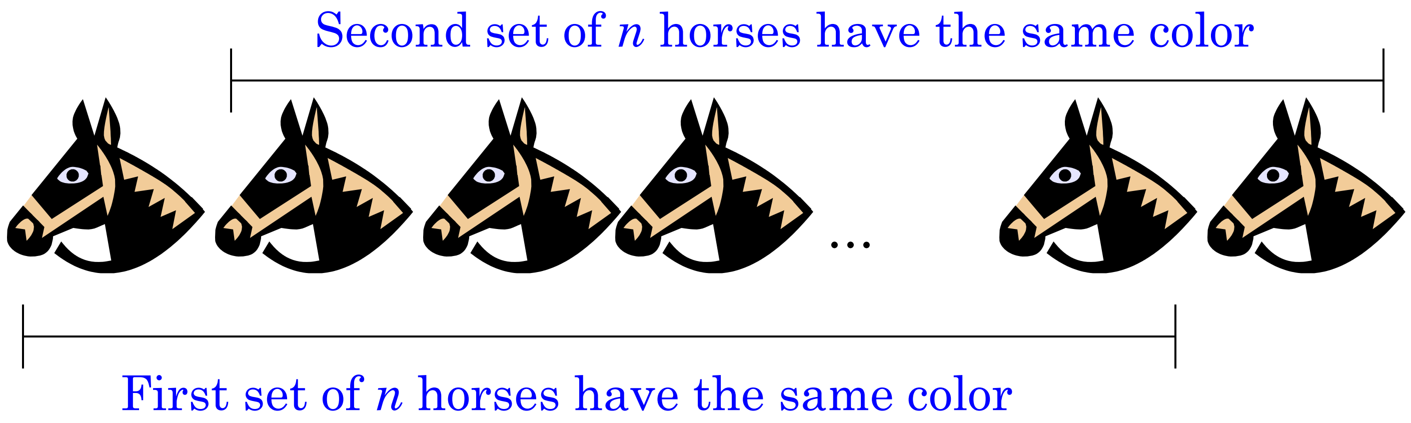


Paradox

(Inductive case)

Assume any n horses have the same color.

Prove that any $n+1$ horses have the same color.

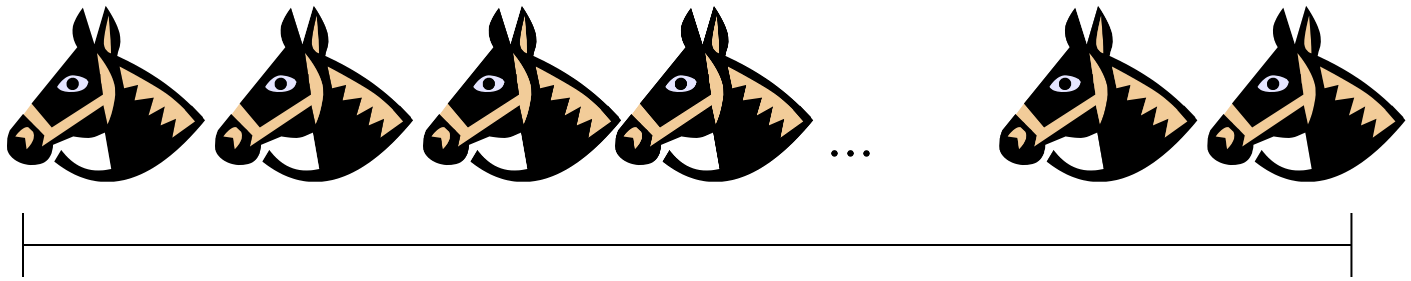

Second set of n horses have the same color
…
First set of n horses have the same color

Paradox

(Inductive case)

Assume any n horses have the same color.

Prove that any $n+1$ horses have the same color.

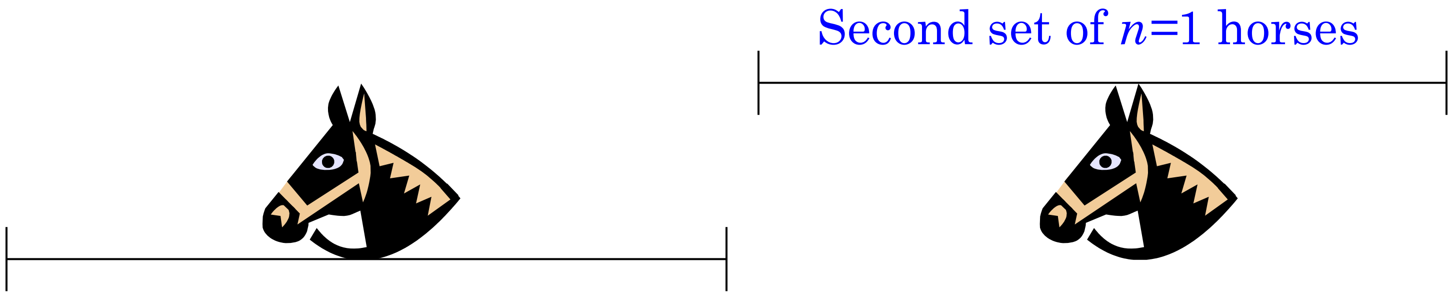


Therefore the set of $n+1$ have the same color!

Paradox

What is wrong? $n=1$

Proof that $P(n) \rightarrow P(n+1)$ is false if $n = 1$, because the two horse groups *do not overlap*.



(But proof works for all $n \neq 1$)

Summary

You should understand the principle of mathematical induction well,

and do basic induction proofs like

- proving equality
- proving inequality
- proving property

Mathematical induction has a wide range of applications in computer science.

In the next lecture we will see more applications and more techniques.

Strong Induction and Well-Ordering

Proofs

- **Basic proof methods**:
  - Direct, Indirect, Contradiction, By Cases,
- **Equivalences Proof of quantified statements**:
  - **There exists x with some property P(x).**
    – It is sufficient to find one element for which the property holds.
  - **For all x some property P(x) holds.**
    – Proofs of 'For all x some property P(x) holds' must cover all x and can be harder.
  - **Mathematical induction** is a technique that can be applied to prove the universal statements for sets of positive integers or their associated sequences.

Mathematical induction

- Used to prove statements of the form ∀ x P(x) where $x \in Z^+$

- **Mathematical induction proofs** consists of two steps:

  1) **Basis**: The proposition P(1) is true.

  2) **Inductive Step**: The implication P(n) → P(n+1), is true for all positive n.

  Therefore we conclude ∀ x P(x).

- Based on the **well-ordering property**: Every nonempty set of nonnegative integers has a least element

Correctness of Mathematical Induction

- Suppose **P(1) is true** and **P(n) → P(n+1)** is true for all positive integers n. Want to show **∀ x P(x).**
- Assume there is at least one n such that P(n) is false. Let S be the set of nonnegative integers where P(n) is false. Thus S≠ ∅.
- **Well-Ordering Property**: Every nonempty set of nonnegative integers has a least element.
- **By the Well-Ordering Property**, S has a least member, say k. k > 1, since P(1) is true. This implies k - 1 > 0 and P(k-1) is true (since k is the smallest integer where P(k) is false).
- **Now**: P(k-1) → P(k) is true thus, P(k) must be true (a contradiction).
- **Therefore ∀ x P(x).**

Strong induction

- The **regular induction**:
  - uses the basic step P(1) and
  - inductive step P(n-1) → P(n)

- **Strong induction**:
  - Uses the basis step P(1) and
  - inductive step P(1) and P(2) … P(n-1) → P(n)

Fundamental Theorem of Arithmetic

- **Problem**: Prove that if *n* is an integer greater than 1, then *n* can be written as $\prod_{i=1}^{l} p_i$ where p_i is a prime number.

- **Solution:**

I. Let *P*(*n*) be the predicate that *n* can be written as "a product of primes."

II. Base Case: *P*(2) is true since 2 itself is a prime.

III. Inductive Step: The inductive hypothesis is *P*(*j*) is true for all integers *j* with $2 \leq j \leq k$.
Show that *P*(*k* + 1) must be true.

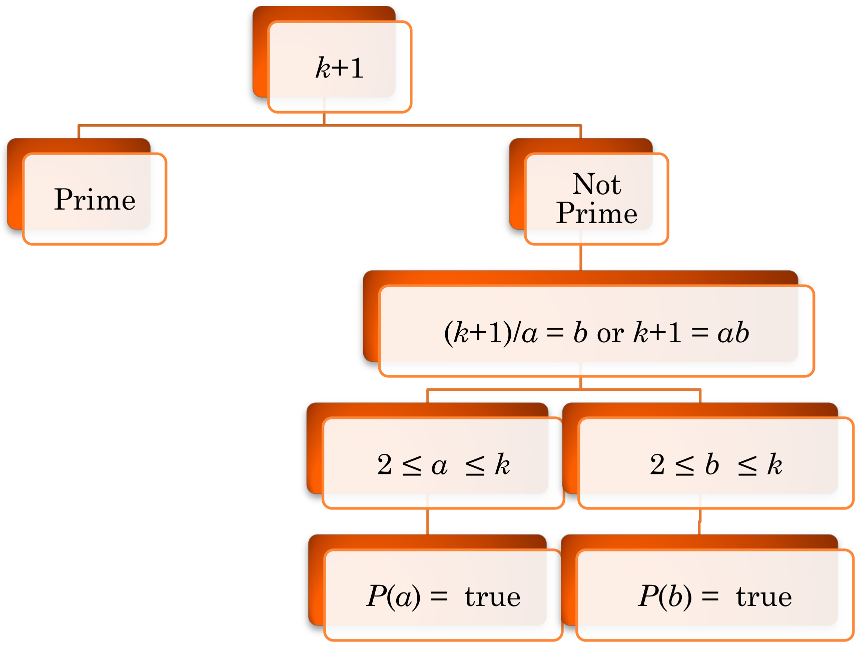
Fundamental Theorem of Arithmetic, cont'd

Consider $P(k + 1)$:

- If $k + 1$ is prime, then $P(k + 1)$ is true.

- Otherwise, $k + 1$ is composite and can be written as the product of two positive integers a and b with $2 \le a \le b < k + 1$. By the inductive hypothesis a and b can be written as the product of primes and therefore $k + 1$ can also be written as the product of those primes.

Hence, it has been shown that every integer greater than 1 can be written as the product of primes.



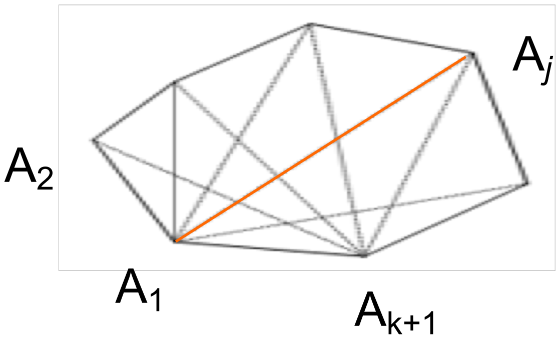
Polygon of *n* Sides – Strong Induction

- **Problem:** Prove that any polygon of *n* sides could be triangulated into *n*-2 triangles.
- **Solution:**
  - Base Case: $P(3)$ is true as 3-2 = 1
  - Inductive hypothesis: $P(k)$ is true for all j where $3 \leq j \leq k$.
  - Proof: Consider (k+1)-gon. A diagonal A_1A_j splits $(k+1)$-gon into two polygons:
    - j-gon
    - $((k + 1) – j + 2)$-gon.

Both polygons are of sides less than k and not less than 3, so the numbers of triangles are

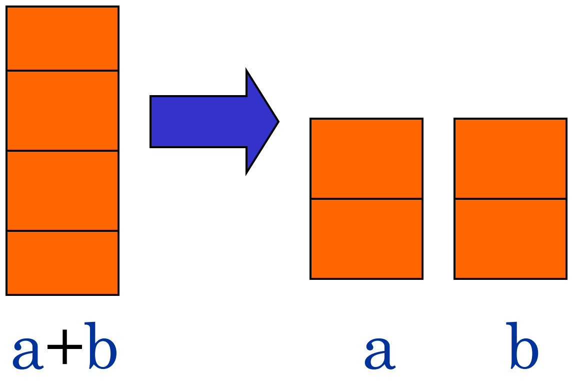
- $j – 2$
- $((k + 1) – j + 2) – 2 = k + 1 – j.$

The total number of triangles for $(k + 1)$-gon is $j – 2 + k + 1 – j = k + 1 – 2.$



Unstacking Game

- Start: a stack of boxes
- Move: split any stack into two stacks of sizes a,b>0
- Scoring: ab points
- Keep moving: until stuck
- Overall score: sum of move scores



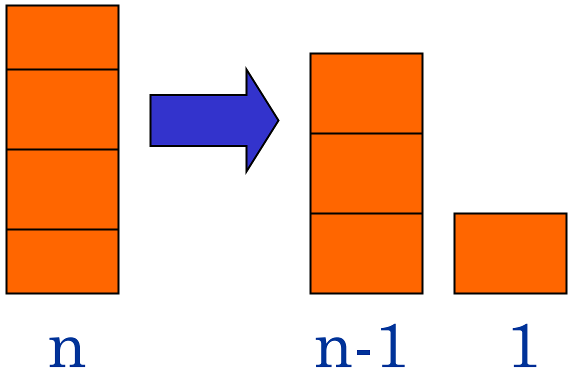
Unstacking Game

What is the best way to play this game?

Suppose there are n boxes.

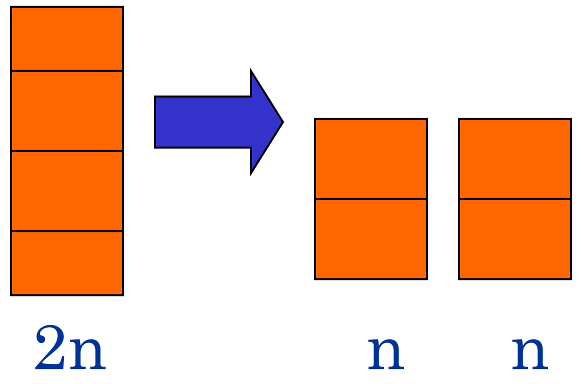
What is the score if we just take the box one at a time?

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$$



Unstacking Game

What is the best way to play this game?



Suppose there are n boxes.

What is the score if we cut the stack into half each time?

Say n=8, then the score is $\underbrace{1\text{x}4\text{x}4}_{\text{first round}} + \underbrace{2\text{x}2\text{x}2}_{\text{second}} + \underbrace{4\text{x}1}_{\text{third}} = 28$

Not better than the first strategy!

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

Say n=16, then the score is 8x8 + 2x28 = 120

Unstacking Game

Claim: Every way of unstacking gives the same score.

Claim: Starting with size n stack, final score will be $\frac{n(n-1)}{2}$

Proof: by Induction with *Claim*(n) as hypothesis

Base case $n = 0$:

score = 0 $= \frac{0(0-1)}{2}$

Claim(0) is okay.

Unstacking Game

Inductive step. assume for n-stack,

and then prove $C(n+1)$:

$(n+1)$-stack score = $\frac{(n+1)n}{2}$

Case $n+1 = 1$. verify for 1-stack:

score = 0 $= \frac{1(1-1)}{2}$

$C(1)$ is okay.

Unstacking Game

Case $n+1 > 1$. So split into an a-stack and b-stack, where $a + b = n + 1$.

$(a + b)$-stack score = ab + a-stack score + b-stack score

by induction:

a-stack score = $\dfrac{a(a-1)}{2}$

b-stack score = $\dfrac{b(b-1)}{2}$

Unstacking Game

$(a + b)$-stack score = ab + a-stack score + b-stack score

$$ab + \frac{a(a-1)}{2} + \frac{b(b-1)}{2} =$$

$$\frac{2ab + a^2 - a + b^2 - b}{2} = \frac{(a+b)^2 - (a+b)}{2} =$$

$$\frac{(a+b)((a+b)-1)}{2} = \frac{(n+1)n}{2}$$

so $C(n+1)$ is okay. We're done!

Well Ordering Principle

Axiom Every nonempty set of *nonnegative integers* has a *least element.*

This is an axiom equivalent to the principle of mathematical induction.

Note that some similar looking statements are not true:

Every nonempty set of *nonnegative rationals* has a *least element.* NO!

Every nonempty set of *negative integers* has a *least element.* NO!

Well Ordering Principle

Thm: $\sqrt{2}$ is irrational

Proof: suppose $\sqrt{2} = \frac{m}{n}$

…can always find such *m*, *n* without common factors…

why always?

By *WOP*, ∃ minimum |*m*| s.t. $\sqrt{2} = \frac{m}{n}$.

so $\sqrt{2} = \frac{m_0}{n_0}$ where $|m_0|$ is minimum.

Well Ordering Principle

but if m_0, n_0 had common factor $c > 1$, then

$$\sqrt{2} = \frac{m_0 / c}{n_0 / c}$$

and $|m_0 / c| < |m_0|$ contradicting minimality of $|m_0|$

The well ordering principle is usually used in “proof by contradiction”.

- Assume the statement is not true, so there is a counterexample.
- Choose the “smallest” counterexample, and find a even smaller counterexample.
- Conclude that a counterexample does not exist.

Well Ordering Principle in Proofs

To prove ``$\forall n \in N.\ P(n)$'' using WOP:

1. Define the set of *counterexamples*

   $$C ::= \{n \in N \mid \neg P(n)\}$$

2. Assume C is not empty.
3. By WOP, have minimum element $m_0 \in C$.
4. Reach a contradiction (*somehow*)
   – usually by finding a member of C that is $< m_0$.
5. Conclude no counterexamples exist. QED

Non-Fermat Theorem

It is difficult to prove there is no positive integer solutions for

$$a^3 + b^3 = c^3$$

Fermat's theorem

But it is easy to prove there is no positive integer solutions for

$$4a^3 + 2b^3 = c^3$$

Non-Fermat's theorem

Hint: Prove by contradiction using well ordering principle…

Non-Fermat Theorem

$$4a^3 + 2b^3 = c^3$$

Suppose, by contradiction, there are integer solutions to this equation.

By the well ordering principle, there is a solution with |a| smallest.

In this solution, a,b,c do not have a common factor.

Otherwise, if a=a'k, b=b'k, c=c'k,

then a',b',c' is another solution with |a'| < |a|,

contradicting the choice of a,b,c.

(*) There is a solution in which a,b,c do not have a common factor.

Non-Fermat Theorem

$$4a^3 + 2b^3 = c^3$$

On the other hand, we prove that every solution must have a,b,c even.

This will contradict (*), and complete the proof.

First, since c^3 is even, c must be even. (because odd power is odd).

Let c = 2c', then $$4a^3 + 2b^3 = (2c')^3$$

$$4a^3 + 2b^3 = 8c'^3$$

$$b^3 = 4c'^3 - 2a^3$$

Non-Fermat Theorem

$$b^3 = 4c'^3 - 2a^3$$

Since b^3 is even, b must be even. (because odd power is odd).

Let b = 2b', then $$(2b')^3 = 4c'^3 - 2a^3$$

$$8b'^3 = 4c'^3 - 2a^3$$

$$a^3 = 2c'^3 - 4b'^3$$

Since a^3 is even, a must be even. (because odd power is odd).

There a,b,c are all even, contradicting (*)

Structural Recursion and Induction

Example

- The sequence is defined by the following algorithm:

1. At the first step there are two numbers: 1, 1.
2. At the next step, insert between two numbers a new number which is the sum of two neighbors:

1, 2, 1

1, 3, 2, 3, 1

1, 4, 3, 5, 2, 5, 3, 4, 1

…

- The sum of the sequence was defined:
  - recursively $S(0) = 1$, $S(n+1) = 3S(n) - 2$,
  - explicitly $S(n) = 3^n + 1$.

Recursive vs. Explicit

Why Recursive Definition is also called Inductive Definition?

Is the sequence in the previous example defined recursively or explicitly?

Can the sequence be defined explicitly?

Recursive Definition :

in some cases is the only possible,

reflects the algorithm,

easier to read when programed,

needs stack (risk of stack overflow)

Explicit Definition:

faster

more reliable

hard to read

Recursion is useful to define sequences, functions, sets, and algorithms.

Recursive or Inductive Function Definition

- Basis Step: Specify the value of the function for the base case.
- Recursive Step: Give a rule for finding the value of a function from its values at smaller integers greater than the base case.

Inductive Definitions

- We completely understand the function f(n) = n! right?
    - n! = 1 · 2 · 3 · … · (n-1) · n, n ≥ 1

But equivalently, we could define it like this:

Recursive Case

Base Case

$$n! = \begin{cases} n \cdot (n-1)!, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

Inductive (Recursive) Definition

Inductive Definitions

- The 2nd most common example:
- Fibonacci Numbers

Note why you need two base cases.

$$f(n)=\begin{cases}0 & \text{if} \quad n=0\\ 1 & \text{if} \quad n=1\\ f(n-1)+f(n-2) & \text{if} \quad n>1\end{cases}$$

Base Cases

Recursive Case

Is there a non-recursive definition for the Fibonacci Numbers?

$$f(n)=\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n-\left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

(Prove by induction.)
All linear recursions have a closed form.

Recursively Defined Sets: Inductive Definitions

- Examples so far have been inductively defined functions.
- Sets can be defined inductively, too.

Give an inductive definition of T = {x: x is a positive integer divisible by 3}

$3 \in S$

Base Case

$x,y \in S \rightarrow x + y \in S$

Recursive Case

Exclusion Rule: No other numbers are in S.

Exclusion rule:
The set contains nothing other than those elements specified in the basic step or generated by the recursive step.

How can we prove it's correct?

Recursively Defined Sets: Inductive Definitions

We want to show that the definition of S:

rule 1: $3 \in S$

rule 2: $x, y \in S \to x + y \in S$

Contains the same elements as the set: T={x: x is a positive integer divisible by 3}

To prove S = T, show

$T \subseteq S$

$S \subseteq T$

Perhaps the "trickiest" aspect of this exercise is realizing that there *is* something to prove! ☺

Recursively Defined Sets: Inductive Definitions

First, we prove $T \subseteq S$.

$T = \{x$: x is a positive integer, multiple of 3$\}$

If $x \in T$, then $x = 3k$ for some integer k. We show by induction on $|k|$ that $3k \in S$.

Hypothesis: P(n) – 3 n belongs to S, for all positive integers n.

Inductive Hypothesis: $3k \in S$

Base Case $P(1) = 3 \in S$ since $3 \in S$ by rule 1.

Inductive Step: Assume $3k, \in S$, show that $3(k+1), \in S$.

Recursively Defined Sets: Inductive Definitions

- Inductive Step:
- $3k \in S$ by inductive hypothesis.
- $3 \in S$ by rule 1.
- $3k + 3 = 3(k+1) \in S$ by rule 2.

Recursively Defined Sets: Inductive Definitions

Next, we show that $S \subseteq T$.

That is, if an number x is described by S, then it is a positive multiple of 3.

Observe that the exclusion rule, all numbers in S are created by a finite number of applications of rules 1 and 2. We use the number of rule applications as our induction counter.

For example:

$3 \in S$ by 1 application of rule 1.

$9 \in S$ by 3 applications (rule 1 once and rule 2 twice).

Recursively Defined Sets: Inductive Definitions

- Base Case (k=1): If $x \in S$ by 1 rule application, then it must be rule 1 and x = 3, which is clearly a multiple of 3.

Inductive Hypothesis: Assume any number described by k or fewer applications of the rules in S is a multiple of 3

Inductive Step: Prove that any number described by (k+1) applications of the rules is also a multiple of 3, assuming IH.

Suppose the (k+1)st rule is applied (rule 2), and it results in value $x = a + b$. Then a and b are multiples of 3 by inductive hypothesis, and thus x is a multiple of 3.

Aside --- Message here: in a proof, follow a well-defined sequence of steps. This avoids subtle mistakes.

Structural Induction

- Basic Step: Show that the result holds for all elements specified in the basis step of the recursive definition to be in the set.
- Recursive step: Show that if the statement is true for each of the elements used to construct new elements in the recursive step of the definition, the result holds for these new elements.

Validity of Structural Induction follows Mathematical Induction

- P(n) the claim is true for all elements of the set that are generated by *n* or fewer applications of the rules in the recursive step of the recursive definition.

- So, we will do induction on the number of rules applications.

- We show that P(n) is true whenever n is a nonnegative integer.
- Basis case - we show that P(0) is true (i.e., it's true for the elements specified in the basis step of recursive definition).
- From recursive step, if we assume P(k), it follows that P(k+1) is true.
- Therefore when we complete a structural induction proof we have shown that P(0) is true, and that P(k) $\rightarrow$P(k+1).
- So, by mathematical induction P(n) follows for all nonnegative numbers.

Recursive Definition of Structures

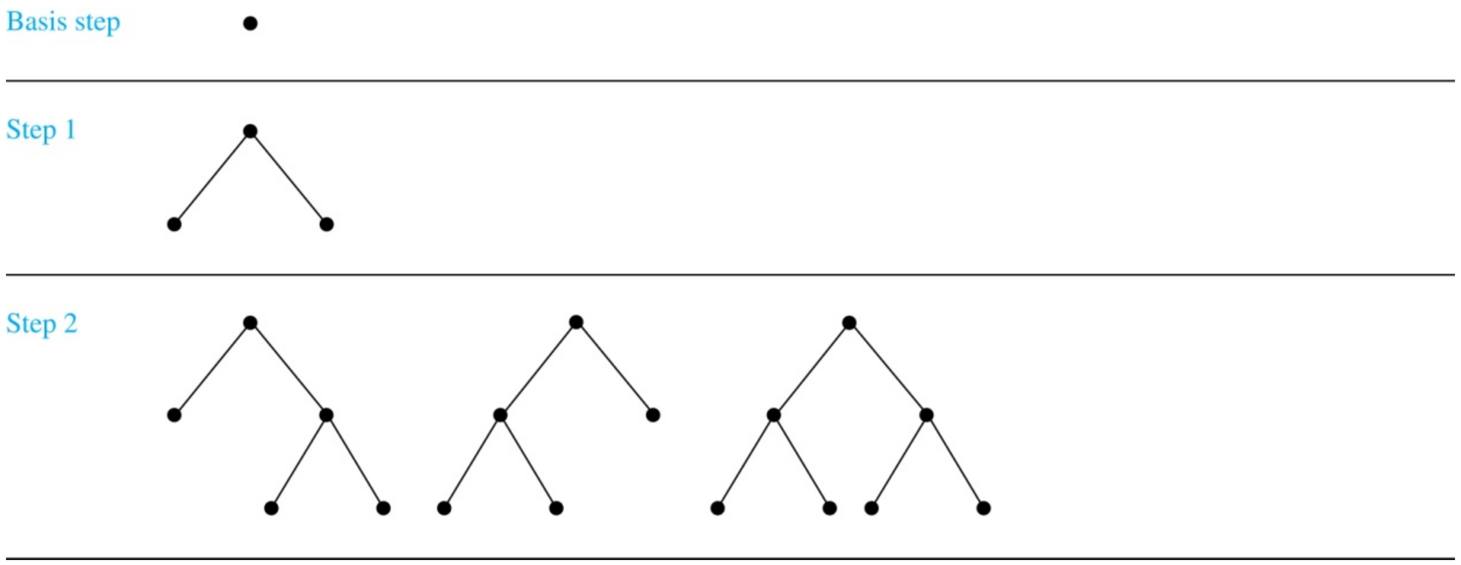
- In the previous lecture we showed that recursive definitions are applicable for functions, sets and other **structures**.

1. Define the “smallest” or “simplest” object (or objects) in the set of structures.
2. Define the ways in which “larger” or “more complex” objects in the set can be constructed out of “smaller” or “simpler” objects in the set.

Example of Structural Recursion

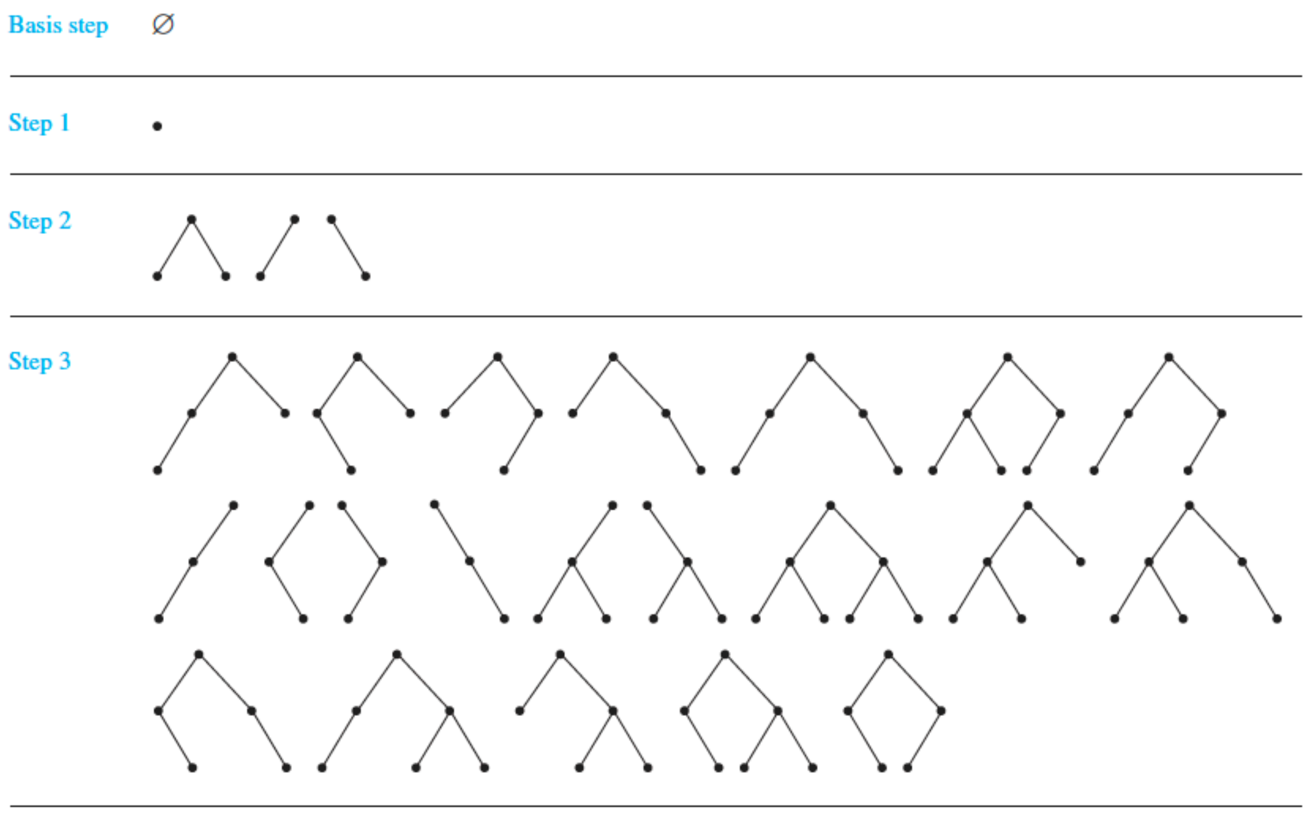
Full binary trees can be defined recursively as follows (*from the textbook*):

1. BASIS STEP: There is a full binary tree consisting of only a single node *r*.
2. RECURSIVE STEP: If T_1 and T_2 are disjoint full binary trees, there is a full binary tree, denoted by $T_1 \cdot T_2$, consisting of a root *r* together with edges connecting the root to each of the roots of the left subtree T_1 and the right subtree T_2.



Extended Binary Tree

- Find the difference between full and extended binary trees.



Definition of Extended Binary Trees

1. BASIS STEP: $\Lambda \cup V$ is in T. The set V represents the set of single nodes and Λ represents the empty binary tree in which **nodes**(Λ) = **edges**(Λ) = ∅ and **root**(Λ) is undefined. We use **nodes**(), **edges**(), and **root**() to specify sets of nodes, edges and roots of a tree.
2. RECURSIVE STEP: Let T_1 and T_2 be elements of T such that **nodes**(T_1) ∩ **nodes**(T_2) = ∅, and
   r ∉ **nodes**(T_1) ∪ **nodes**(T_2). Then the ordered triple $t = (r, T_1, T_2)$ is also in T. Furthermore, we define
   root(T) = r
   nodes(T) = $\{r\}$ ∪ **nodes**(T_1) ∪ **nodes**(T_2)
   edges(T) = E ∪ **edges**(T_1) ∪ **edges**(T_2)
   where E is the set that contains (r, **root**(T_1)) if $T_1 \neq \Lambda$, (r, **root**(T_2)) if $T_2 \neq \Lambda$, and nothing else.

- How to change this recursive definition so that it becomes the definition of full binary trees?

Definition of Full Binary Trees

1. BASIS STEP: V is in T. The set V represents the set of single nodes.
2. RECURSIVE STEP: Let T_1 and T_2 be elements of T such that **nodes**$(T_1) \cap$ **nodes**$(T_2) = \emptyset$, and r $\notin$ **nodes**$(T_1) \cup$ **nodes**(T_2). Then the ordered triple $t = (r, T_1, T_2)$ is also in T. Furthermore, we define
   root$(T) = r$
   nodes$(T) = \{r\} \cup$ **nodes**$(T_1) \cup$ **nodes**(T_2) , and nothing
   edges$(T) = E \cup$ **edges**$(T_1) \cup$ **edges**(T_2) else.
   where E is the set that contains $(r,$ **root**$(T_1))$, $(r,$ **root**$(T_2))$.

- Like this?
- What is missed?

Importance of "Nothing Else"

- **Example:** The set $S = \{n \in \mathbf{N} : n \geq 2\}$ is recursively defined:

1. BASIS STEP: $2 \in S$
2. RECURSIVE STEP: if $n \in S$, then $n + 1 \in S$
3. *S* contains nothing else.

- Why is this definition ambiguous without statement 3?
- Otherwise **N** and **Z** can also satisfy properties because it is structural recursion that defines a set. Sets are not ordered and basis step means entrance point, but does not mean the first point.

Mathematical and Structural Recursions

- As you can see from the previous example *mathematical recursion* over the natural numbers is an instance of the more general concept of *structural recursion* over values of an *recursively-defined sets*.
- The natural numbers themselves is a recursively defined set. Because **N** may be defined as:

1. 0 is an element of **N**.
2. If *m* is an element of **N**, then so is *m+1*.
3. Nothing else is an element of **N**.

Mathematical and Structural Inductions

- We used mathematical induction to prove properties of functions mapped onto the set of natural numbers. And we saw that mathematical induction repeats recursive steps.
- To prove that a property *P(n)* holds of every *n* in **N**, it suffices to demonstrate the following facts:
  - Show that $P(0)$ holds.
  - Assuming that $P(m)$ holds, show that $P(m+1)$ holds.
- Structural recursion is more universal notion and the approach that is used to prove properties of structures is called **structural induction**.
- The pattern of reasoning using structural induction follows the recursive definition of structures.

Structural Induction and Binary Trees

- **Prove**: If T is a full binary tree, then $n(T) \le 2^{h(T)+1} - 1$ where $n(T)$ is size of tree, $h(T)$ is height of a tree.

Solution: Use structural induction.

- BASIS STEP: The result holds for a full binary tree consisting only of a root, $n(T) = 1$ and $h(T) = 0$. Hence, $n(T) = 1 \le 2^{0+1} - 1 = 1$.
- INDUCTION STEP: Assume $n(T_1) \le 2^{h(T_1)+1} - 1$ and also $n(T_2) \le 2^{h(T_2)+1} - 1$ whenever T_1 and T_2 are full binary trees.

- $n(T) = 1 + n(T_1) + n(T_2)$ *(by recursive formula of n(T), see text)*
- $\le 1 + (2^{h(T1)+1} - 1) + (2^{h(T2)+1} - 1)$ *(by inductive hypothesis)*
- $\le 2 \cdot \max(2^{h(T1)+1}, 2^{h(T2)+1}) - 1$
- $= 2 \cdot 2^{\max(h(T1), h(T2))+1} - 1$ $(\max(2^x, 2^y) = 2^{\max(x,y)})$
- $= 2 \cdot 2^{h(T)} - 1$ *(by recursive definition of h(T), see text)*
- $= 2^{h(T)+1} - 1$

Principle of Structural Induction

- Let *R* be a recursive definition.
- Let *P* be a statement (property) about the elements defined by *R*.

- If the following hypotheses hold:
  - P is **True** for every element $b_1,\ldots,b_m$ in the base case of the definition R.
  - For every element E constructed by the recursive definition from some elements $e_1,\ldots,e_n$: P is **True** for $e_1,\ldots,e_n \Rightarrow P$ is true for E.
- Then we can conclude that:
- *P* is **True** for every element *E* defined by the recursive definition *R*.

Example: Set of Strings

A. Recursive definition of a set of valid strings :

- BASE CASE: $b \in S$, where b is the empty string, S is a set of valid strings.
- RECURSIVE STEP: asa, $s \in S$, where a is any letter from the alphabet A ($a \in A$).

B. Explicit formula for a valid string:

- $a^n b a^n$, where $n \geq 0$ is a number of letters.

- Prove that
  - Recursive Definition $\Leftrightarrow$ Explicit Definition

Proof A: Recursive → Explicit

- Property *P*(*s*): "Every element *s* constructed recursively is of the form $a^n b a^n$."

- By **Structural Induction**.
  - Base Case: $b = a^0 b a^0$.
  - Induction Step: Suppose $s = a^n b a^n$.
  - Structural Induction is to prove: if $P(b) \vee (\forall s\, P(s) \rightarrow P(asa))$, then $\forall s\, P(s)$.

  By recursive step next element $asa = a(a^n b a^n)a = a^{n+1} b a^{n+1}$

Proof B: Explicit → Recursive

- Predicate *P*(*n*): "Every element of the form $a^n b a^n$ can be constructed recursively."
- By **Mathematical Induction**.
  - Base Case: $n = 0 \rightarrow a^0 b a^0 = b$.
  - Induction Hypothesis: Every element of the form $a^n b a^n$ can be constructed recursively.
  - Prove by mathematical induction must show: Every element of the form $a^{n+1} b a^{n+1}$ can be constructed recursively.

$$a^{n+1} b a^{n+1} = a(a^n b a^n)a = asa.$$

By the inductive hypothesis: $a^n b a^n$ satisfies the recursive definition; hence by the recursive step, so does $a^{n+1} b a^{n+1}$.

Observations on Structural Induction

- **Proofs by Structural Induction**
- Extends inductive proofs to discrete data structures: strings, lists, trees, etc.
- For every recursive definition there is a corresponding structural induction rule.
- The base case and the recursive step mirror the recursive definition.
  - Prove Base Case
  - Prove Recursive Step

Recursive Algorithms

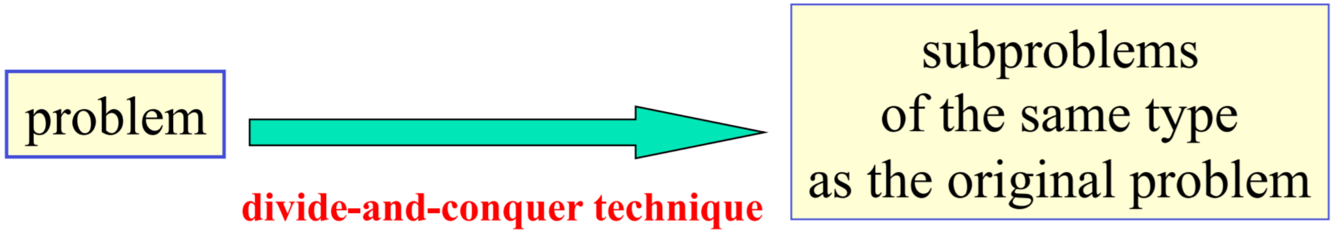
Recursive Algorithms

- Recursive definitions can be used to describe functions and sets as well as algorithms.
- A recursive procedure is a procedure that invokes itself.
- A recursive algorithm is an algorithm that contains a recursive procedure.
- An algorithm is called recursive if it solves a problem by reducing it to an instance of the same problem with smaller input.

Example

- A procedure to compute a^n.

```
procedure power(a≠0: real, n∈N)
if n = 0 then return 1
else return a·power(a, n−1)
```


problem
divide-and-conquer technique
subproblems of the same type as the original problem

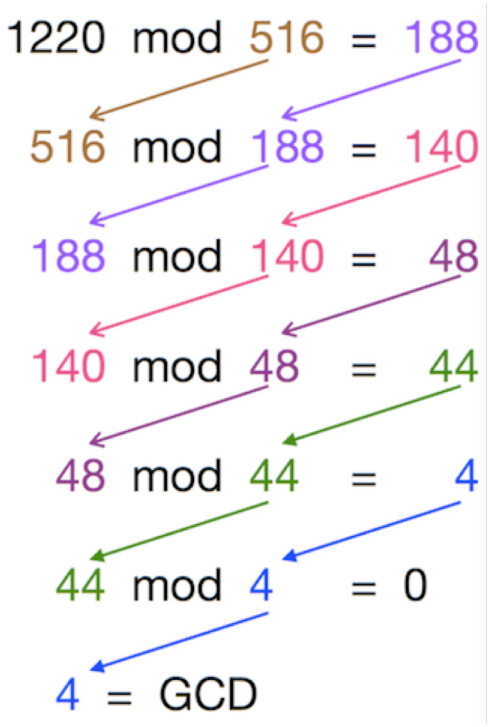
❑ Note recursive algorithms are often simpler to code than iterative ones…

❑ However, they can consume more stack space if your compiler is not smart enough

Recursive Euclid's Algorithm

- gcd(a, b) = gcd((b mod a), a)

```
procedure gcd(a,b∈N with a < b )
if a = 0 then return b
else return gcd(b mod a, a)
```



Recursive Linear Search

- {Finds *x* in series *a* at a location ≥*i* and ≤*j* }

```
procedure search
(a: series; i, j: integer; x: item to find)
if $a_i = x$ return i {At the right item? Return it!}
if $i = j$ return 0 {No locations in range? Failure!}
return search(a, i +1, j, x) {Try rest of range}
```

- Note there is no real advantage to using recursion here over just looping

  for *loc* := *i* to *j*…

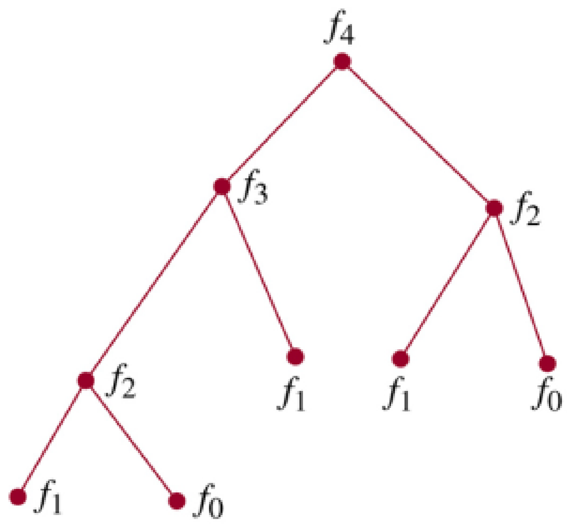
- Recursion is slower because procedure call costs

Recursive Binary Search

```
{Find location of x in a, ≥i and ≤ j}
procedure binarySearch(a, x, i, j)
m := $\lfloor (i + j)/2 \rfloor$ {Go to halfway point}
if $x = a_m$ return m {Did we luck out?}
if $x < a_m \wedge i < m$ {If it's to the left, check that .}
return binarySearch(a, x, i, m−1)
else if $x > a_m \wedge j > m$ {If it's to right, check that .}
return binarySearch(a, x, m+1, j)
else return 0 {No more items, failure.}
```

Recursive Fibonacci Algorithm

```
**procedure** *fibonacci*($n \in \mathbf{N}$)
**if** $n = 0$ **return** 0
**if** $n = 1$ **return** 1
**return** *fibonacci*($n - 1$) + *fibonacci*($n - 2$)
```



- Is this an efficient algorithm?
- How many additions are performed?

Analysis of Fibonacci Procedure

Theorem: The recursive procedure *fibonacci*(n) performs $f_{n+1} - 1$ additions.

- **Proof:** By strong structural induction over n, based on the procedure's own recursive definition.
- **Basis step:**
    - *fibonacci*(0) performs 0 additions, and $f_{0+1} - 1 = f_1 - 1 = 1 - 1 = 0$.
    - Likewise, *fibonacci*(1) performs 0 additions, and $f_{1+1} - 1 = f_2 - 1 = 1 - 1 = 0$.

Analysis of Fibonacci Procedure

Inductive step:

$$\textit{fibonacci}(k+1) = \underbrace{\textit{fibonacci}(k)}_{\boxed{\begin{array}{c}\text{by } P(k)\text{:}\\ f_{k+1}-1 \text{ additions}\end{array}}} + \underbrace{\textit{fibonacci}(k-1)}_{\boxed{\begin{array}{c}\text{by } P(k-1)\text{:}\\ f_{k}-1 \text{ additions}\end{array}}}$$

- For $k > 1$, by strong inductive hypothesis, *fibonacci*(k) and *fibonacci*(k−1) do f_k+1 − 1 and f_k − 1 additions respectively.
- *fibonacci*(k+1) adds 1 more, for a total of

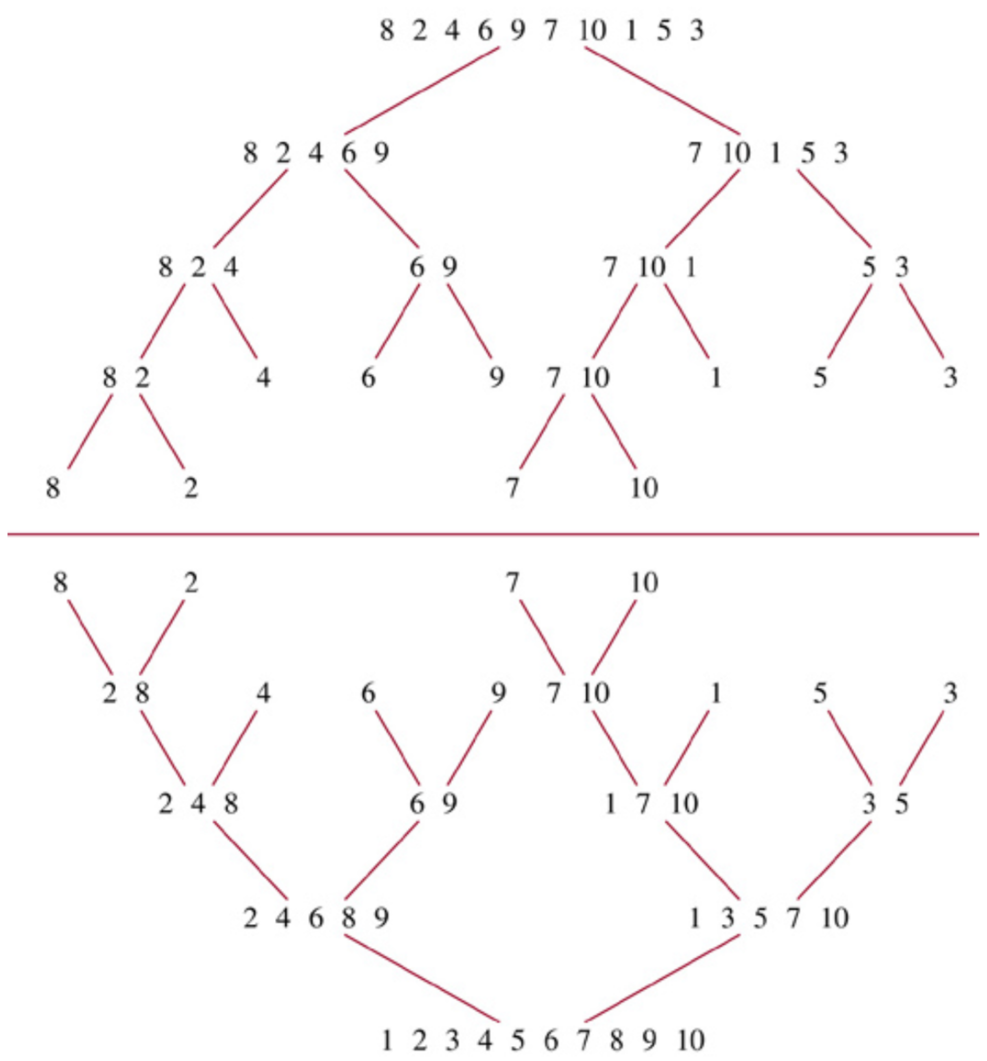
$$(f_k+1 - 1) + (f_k - 1) + 1 = f_k+1 + f_k - 1$$
$$= f_k+2 - 1.$$

Iterative Fibonacci Algorithm

```
• procedure iterativeFib($n \in \mathbf{N}$)
• if $n = 0$ then
• return 0
• else begin
• $x := 0$
• $y := 1$
• for $i := 1$ to $n - 1$ begin
• $z := x + y$
• $x := y$
• $y := z$
• end
• end
• return y {the nth Fibonacci number}
```

Requires only *n* – 1 additions

Recursive Merge Sort Example



Split

Merge

Recursive Merge Sort

```
procedure mergesort(L = 1,…, n)
if n > 1 then
m := ⌊n/2⌋ {this is rough ½-way point}
L1 := 1,…, m
L2 := m+1,…, n
L := merge(mergesort(L1), mergesort(L2))
return L
```

- The merge takes $\Theta(n)$ steps, and therefore the merge-sort takes $\Theta(n \log n)$.

Merging Two Sorted Lists

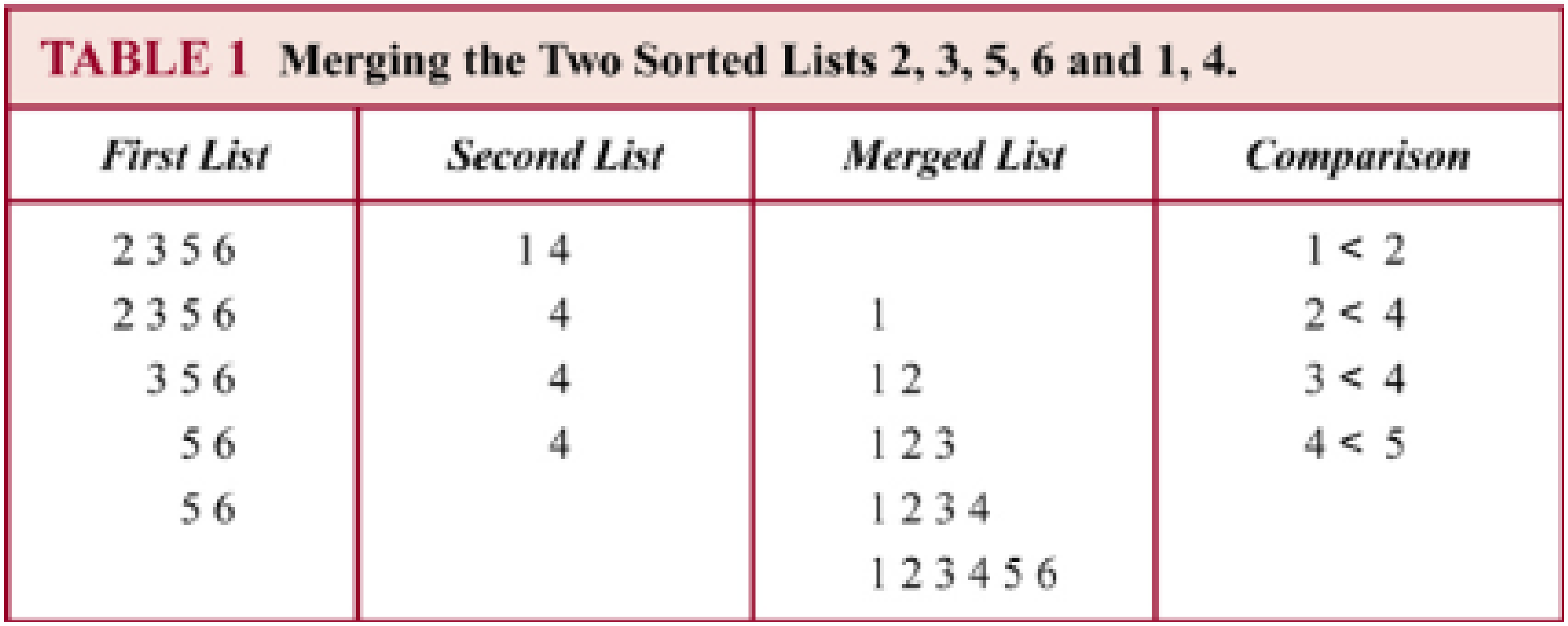


TABLE 1 Merging the Two Sorted Lists 2, 3, 5, 6 and 1, 4.

First List	*Second List*	*Merged List*	*Comparison*
2 3 5 6	1 4		$1 < 2$
2 3 5 6	4	1	$2 < 4$
3 5 6	4	1 2	$3 < 4$
5 6	4	1 2 3	$4 < 5$
5 6		1 2 3 4	
		1 2 3 4 5 6	

Recursive Merge Method

```
{Given two sorted lists $A = (a_1,\ldots, a_{|A|})$,  $B = (b_1,\ldots, b_{|B|})$,
returns a sorted list of all.}
procedure merge($A$, $B$: sorted lists)
if $A$ = empty return $B$ {If $A$ is empty, it's $B$.}
if $B$ = empty return $A$ {If $B$ is empty, it's $A$.}
if $a_1 < b_1$ then
return ($a_1$, merge(($a_2,\ldots, a_{|A|}$), $B$))
else
return ($b_1$, merge($A$, ($b_2,\ldots, b_{|B|}$)))
```

Efficiency of Recursive Algorithms

- The time complexity of a recursive algorithm may depend critically on the number of recursive calls it makes.
- **Example:** *Modular exponentiation* to a power *n* can take log(*n*) time if done right, but linear time if done slightly differently.
- Task: Compute b^n **mod** *m*, where $m \geq 2$, $n \geq 0$, and $1 \leq b < m$.

Modular Exponentiation #1

- Uses the fact that $b^n = b \cdot b^{n-1}$ and that $x \cdot y$ **mod** $m = x \cdot (y$ **mod** $m)$ **mod** m.

  (Prove the latter theorem at home.)

```
{Returns $b^n$ mod $m$.}
procedure mpower
($b$, $n$, $m$: integers with $m \geq 2$, $n \geq 0$, and $1 \leq b < m$)
if $n=0$ then return 1 else
return ($b \cdot mpower(b,n-1,m)$) mod $m$
```

- Note this algorithm takes $\Theta(n)$ steps!

Modular Exponentiation #2

- Uses the fact that $b^{2k} = b^{k \cdot 2} = (b^k)^2$
- Then, b^{2k} **mod** m = $(b^k$ **mod** $m)^2$**mod** m.

```
procedure mpower(b,n,m) {same signature}
if n=0 then return 1
else if 2 | n then
return mpower(b,n/2,m)2 mod m
else return (b ·mpower(b,n−1,m)) mod m
```

- What is its time complexity? **Θ(log *n*) steps**

A Slight Variation

- Nearly identical but takes Θ(*n*) time instead!

```
procedure mpower(b,n,m) {same signature}
if $n=0$ then return 1
else if $2 \mid n$ then
return $(mpower(b,n/2,m) \cdot$
$mpower(b,n/2,m))$ mod $m$
else return $(mpower(b,n-1,m) \cdot b)$ mod $m$
```

The number of recursive calls made is critical!

Next class

- Topic: Basic Structures: Sets, Function, Sequences etc.
- Pre-class reading: Chap 2