Lecture 16

Chapter 4. Induction and Recursion

1. Mathematical Induction
2. Strong Induction

Mathematical Induction

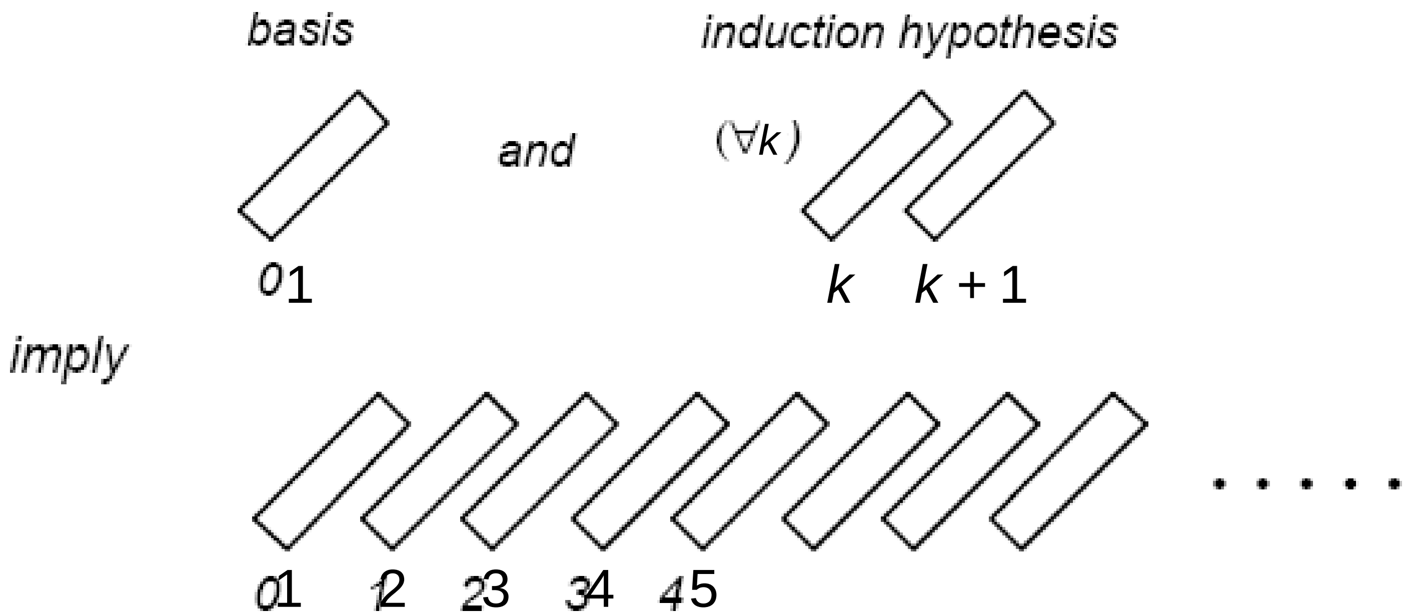
- A powerful, rigorous technique for proving that a statement $P(n)$ is true for ***every*** positive integers n, no matter how large.
- Essentially a "domino effect" principle.
- Based on a predicate-logic inference rule:

$$\frac{\begin{array}{l} P(1) \\ \forall k \geq 1\ [P(k) \rightarrow P(k+1)] \end{array}}{\therefore \forall n \geq 1\ P(n)}$$

"The First Principle of Mathematical Induction"

The “Domino Effect”

- **Premise #1:** Domino #1 falls.
- **Premise #2:** For every $k \in \mathbf{Z}^+$, if domino #k falls, then so does domino #k+1.
- **Conclusion:** All of the dominoes fall down!



Note: this works even if there are infinitely many dominoes!

Mathematical Induction

- **PRINCIPLE OF MATHEMATICAL INDUCTION:**

To prove that a statement $P(n)$ is true for all positive integers n, we complete two steps:

- **BASIS STEP**: Verify that $P(1)$ is true

- **INDUCTIVE STEP**: Show that the conditional statement $P(k) \rightarrow P(k+1)$ is true for all positive integers k

Inductive Hypothesis

Validity of Induction

Proof: that $\forall n \geq 1\ P(n)$ is a valid consequent: Given any $k \geq 1$, the 2nd premise $\forall k \geq 1\ (P(k) \rightarrow P(k+1))$ trivially implies that $(P(1) \rightarrow P(2)) \wedge (P(2) \rightarrow P(3)) \wedge \ldots \wedge (P(n-1) \rightarrow P(n))$.

Repeatedly applying the hypothetical syllogism rule to adjacent implications in this list $n - 1$ times then gives us $P(1) \rightarrow P(n)$; which together with $P(1)$ (premise #1) and *modus ponens* gives us $P(n)$.

Thus $\forall n \geq 1\ P(n)$. ■

Outline of an Inductive Proof

- Let us say we want to prove $\forall n \in \mathbf{Z}^+ \, P(n)$.
  - Do the ***base case*** (or ***basis step***): Prove $P(1)$.
  - Do the ***inductive step***: Prove $\forall k \in \mathbf{Z}^+ \, P(k) \rightarrow P(k+1)$.
    - *E.g.* you could use a direct proof, as follows:
    - Let $k \in \mathbf{Z}^+$, assume $P(k)$. (*inductive hypothesis*)
    - Now, under this assumption, prove $P(k+1)$.
  - The inductive inference rule then gives us $\forall n \in \mathbf{Z}^+ \, P(n)$.

Induction Example

- Show that, for $n \geq 1$

$$\boxed{1+2+\cdots+n=\frac{n(n+1)}{2}}$$

- Proof by induction
- $P(n)$: the sum of the first n positive integers is $n(n+1)/2$, i.e. $P(n)$ is
- ***Basis step***: Let $n = 1$. The sum of the first positive integer is 1, i.e. $P(1)$ is true.

$$1=\frac{1(1+1)}{2}$$

Example *(cont.)*

- ***Inductive step***: Prove $\forall k \geq 1: P(k) \rightarrow P(k+1)$.
  - Inductive Hypothesis, $P(k)$:

$$1 + 2 + \cdots + k = \frac{k(k+1)}{2}$$

  - Let $k \geq 1$, assume $P(k)$, and prove $P(k+1)$, i.e.

$$1 + 2 + \cdots + k + (k+1) = \frac{(k+1)[(k+1)+1]}{2}$$

$$\underbrace{=}_{P(k+1)} \frac{(k+1)(k+2)}{2}$$

Example *(cont.)*

By inductive hypothesis $P(k)$

- ***Inductive step*** continues…

$$1+2+\cdots+k+(k+1)=\frac{k(k+1)}{2}+(k+1)$$

$$=\frac{k(k+1)}{2}+\frac{2(k+1)}{2}$$

$$=\frac{k^2+3k+2}{2}$$

$$=\frac{(k+1)(k+2)}{2}$$

$P(k+1)$

- Therefore, by the principle of mathematical induction $P(n)$ is true for all integers n with $n \geq 1$

Induction Example 3

- Prove that $\forall n \geq 1,\ n < 2^n$. Let $P(n) = (n < 2^n)$
- ***Basis step***: $P(1)$: $(1 < 2^1) \equiv (1 < 2)$: **True**.
- ***Inductive step***: For $k \geq 1$, prove $P(k) \rightarrow P(k+1)$.
- Assuming $k < 2^k$, prove $k + 1 < 2^{k+1}$.
- Note $k + 1 < 2^k + 1$ (by inductive hypothesis)

$< 2^k + 2^k$ (because $1 < 2^k$ for $k \geq 1$)

$= 2 \cdot 2^k = 2^{k+1}$

- So $k + 1 < 2^{k+1}$, i.e. $P(k+1)$ is true
- Therefore, by the principle of mathematical induction $P(n)$ is true for all integers n with $n \geq 1$.

Generalizing Induction

- Rule can also be used to prove $\forall n \geq c\ P(n)$ for a given constant $c \in \mathbf{Z}$, where maybe $c \neq 1$.
- In this circumstance, the basis step is to prove $P(c)$ rather than $P(1)$, and the inductive step is to prove $\forall k \geq c\ (P(k) \rightarrow P(k+1))$.

Induction Example 4

- **Example 6**: Prove that $2^n < n!$ for $n \geq 4$ using mathematical induction.
- $P(n)$: $2^n < n!$
- ***Basis step***: Show that $P(4)$ is true
  - Since $2^4 = 16 < 4! = 24$, $P(4)$ is true
- ***Inductive step***: Show that $P(k) \rightarrow P(k+1)$ for $k \geq 4$

$P(k+1)$ is true

$$
\begin{aligned}
2^{k+1} &= 2 \cdot 2^k && \text{(by definition of exponent)} \\
&< 2 \cdot k! && \text{(by the inductive hypothesis } P(k)\text{)} \\
&< (k+1) \cdot k! && \text{(because } 2 < k+1 \text{ for } k \geq 4\text{)} \\
&= (k+1)! && \text{(by definition of factorial function)}
\end{aligned}
$$

- Therefore, by the principle of mathematical induction $P(n)$ is true for all integers n with $n \geq 4$.

Second Principle of Induction

"Strong Induction"

- Characterized by another inference rule:

$$\begin{array}{l} P(1) \\ \forall k \geq 1: (P(1) \wedge \overbrace{P(2) \wedge \cdots}^{P \text{ is true in } all \text{ previous cases}} \wedge P(k)) \rightarrow P(k+1) \\ \hline \therefore \forall n \geq 1: P(n) \end{array}$$

- The only difference between this and the 1st principle is that:
  - the inductive step here makes use of the stronger hypothesis that all of $P(1)$, $P(2)$,…, $P(k)$ are true, not just $P(k)$.

Example of Second Principle

- Show that every integer $n > 1$ can be written as a product $n = p_1 p_2 \ldots p_s = \prod p_i$ of some series of s prime numbers.
- Let $P(n)$ = "n has that property"
- ***Basis step*:** $n = 2$, let $s = 1$, $p_1 = 2$. Then $n = p_1$
- ***Inductive step*:** Let $k \geq 2$. Assume $\forall 2 \leq i \leq k$: $P(i)$.
- Consider $k + 1$. If it's prime, let $s = 1$, $p_1 = k + 1$.
- Else $k + 1 = ab$, where $1 < a \leq k$ and $1 < b \leq k$.

Then $a = p_1 p_2 \ldots p_t$ and $b = q_1 q_2 \ldots q_u$.
(by Inductive Hypothesis) Then we have that $k + 1 =$

$p_1 p_2 \ldots p_t q_1 q_2 \ldots q_u$,

a product of $s = t + u$ primes.

Generalizing Strong Induction

- Handle cases where the inductive step is valid only for integers greater than a particular integer
- $P(n)$ is true for $\forall n \geq b$ (b: fixed integer)
- **BASIS STEP**: Verify that $P(b), P(b+1), \ldots, P(b+j)$ are true (j: a fixed positive integer)
- **INDUCTIVE STEP**: Show that the conditional statement $[P(b) \wedge P(b+1) \wedge \cdots \wedge P(k)] \rightarrow P(k+1)$ is true for all positive integers $k \geq b + j$

2nd Principle example

- Prove that every amount of postage of 12 cents or more can be formed using just 4-cent and 5-cent stamps.
- $P(n)$ = “postage of n cents can be formed using 4-cent and 5-cent stamps” for $n \geq 12$.
- ***Basis step*:**
- $12 = 3 \cdot 4$
- $13 = 2 \cdot 4 + 1 \cdot 5$
- $14 = 1 \cdot 4 + 2 \cdot 5$
- $15 = 3 \cdot 5$
- So $\forall 12 \leq i \leq 15, P(i)$.

Example *(cont.)*

- ***Inductive step***:
- Let $k \geq 15$, assume $\forall 12 \leq i \leq k, P(i)$.
- Note $12 \leq k - 3 \leq k$, so $P(k - 3)$.

(by inductive hypothesis) This means we can form postage of k – 3 cents
using just 4-cent and 5-cent stamps.

- Add a 4-cent stamp to get postage for $k + 1$,

i.e. $P(k + 1)$ is true (postage of k + 1 cents
can be formed using 4-cent and 5-cent stamps).

- Therefore, by the 2nd principle of mathematical induction $P(n)$ is true for all integers n with $n \geq 12$.

Another 2nd Principle example

- Prove by the 1st Principle.
- $P(n)$ = "postage of n cents can be formed using 4-cent and 5-cent stamps", $n \geq 12$.
- ***Basis step***: $P(12)$: $12 = 3 \cdot 4$.
- ***Inductive step***: $P(k) \rightarrow P(k+1)$
  - Case 1: At least one 4-cent stamp was used for $P(k)$
    - $k + 1 = k - 4 + 5$ (i.e. replace the 4-cent stamp with a 5-cent stamp to form a postage of $k + 1$ cents)

Example *Continues*…

- ***Inductive step*: $P(k) \rightarrow P(k+1)$**
- Case 2: No 4-cent stamps were used for $P(k)$
- Since $k \geq 12$, at least three 5-cent stamps are needed to form postage of k cents
- $k + 1 = k - 3 \cdot 5 + 4 \cdot 4$ (i.e. replace three 5-cent stamps with four 4-cent stamps to form a postage of $k + 1$ cents)

- Therefore, by the principle of mathematical induction $P(n)$ is true for all integers n with $n \geq 12$.

The Well-Ordering Property

- Another way to prove the validity of the inductive inference rule is by using the *well-ordering property*, which says that:
- Every non-empty set of non-negative integers has a minimum (smallest) element.
- $\forall \varnothing \subset S \subseteq \mathbf{N}: \exists m \in S$ such that $\forall n \in S, m \leq n$
- This implies that $\{n | \neg P(n)\}$ (if non-empty) has a minimum element m, but then the assumption that $P(m-1) \rightarrow P((m-1)+1)$ would be contradicted.

Lecture

Chapter 4. Induction and Recursion

4.3 Recursive Definitions and

Structural Induction

Recursive Definitions

- In induction, we *prove* all members of an infinite set satisfy some predicate *P* by:
  - proving the truth of the predicate for larger members in terms of that of smaller members.
- In ***recursive definitions***, we similarly *define* a function, a predicate, a set, or a more complex structure over an infinite domain (universe of discourse) by:
  - defining the function, predicate value, set membership, or structure of larger elements in terms of those of smaller ones.

Recursion

- ***Recursion*** is the general term for the practice of defining an object in terms of *itself*
  - or of part of itself.
  - This may seem circular, but it isn't necessarily.
- An inductive proof establishes the truth of $P(k+1)$ *recursively* in terms of $P(k)$.
- There are also recursive *algorithms*, *definitions*, *functions*, *sequences*, *sets*, and other structures.

Recursively Defined Functions

- Simplest case: One way to define a function $f:\mathbf{N}\rightarrow S$ (for any set S) or series $a_n = f(n)$ is to:
  - Define $f(0)$
  - For $n > 0$, define $f(n)$ in terms of $f(0),\ldots,f(n-1)$

- **Example**: Define the series $a_n = 2^n$ where n is a nonnegative integer recursively:
  - a_n looks like $2^0, 2^1, 2^2, 2^3,\ldots$
  - Let $a_0 = 1$
  - For $n > 0$, let $a_n = 2 \cdot a_{n-1}$

Another Example

- Suppose we define $f(n)$ for all $n \in \mathbf{N}$ recursively by:
  - Let $f(0) = 3$
  - For all $n > 0$, let $f(n) = 2 \cdot f(n-1) + 3$
- What are the values of the following?
  - $f(1) =$ $2 \cdot f(0) + 3 = 2 \cdot 3 + 3 = 9$
  - $f(2) =$ $2 \cdot f(1) + 3 = 2 \cdot 9 + 3 = 21$
  - $f(3) =$ $2 \cdot f(2) + 3 = 2 \cdot 21 + 3 = 45$
  - $f(4) =$ $2 \cdot f(3) + 3 = 2 \cdot 45 + 3 = 93$

Recursive Definition of Factorial

- Give an inductive (recursive) definition of the factorial function,

  $$F(n) = n! = \prod_{1 \le i \le n} i = 1 \cdot 2 \cdots n$$

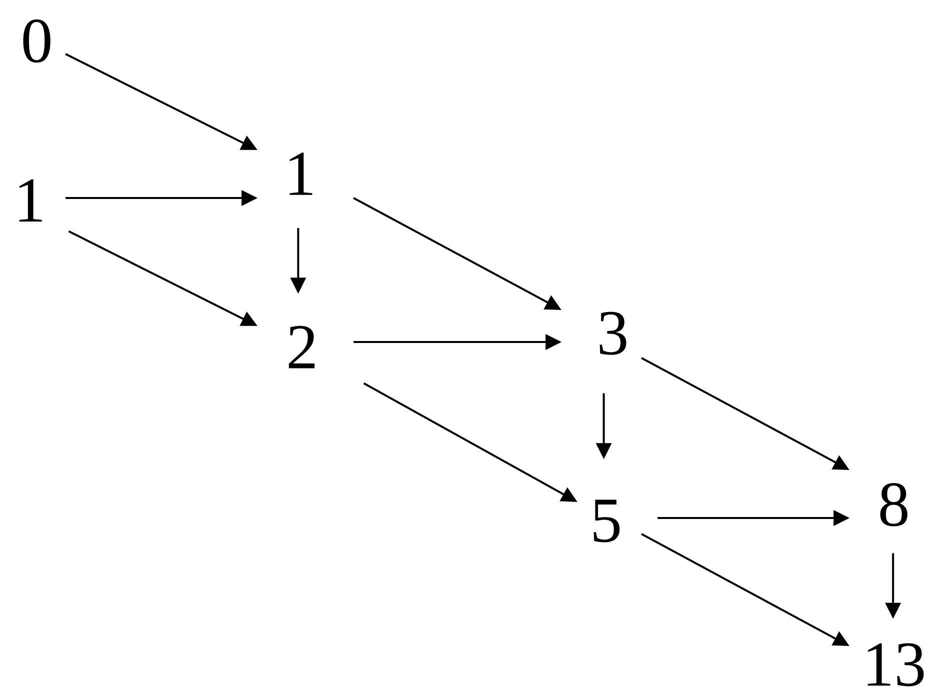
  - Basis step: $F(1) = 1$
  - Recursive step: $F(n) = n \cdot F(n-1)$ for $n > 1$
    - $F(2) = 2 \cdot F(1) = 2 \cdot 1 = 2$
    - $F(3) = 3 \cdot F(2) = 3 \cdot \{2 \cdot F(1)\} = 3 \cdot 2 \cdot 1 = 6$
    - $F(4) = 4 \cdot F(3) = 4 \cdot \{3 \cdot F(2)\} = 4 \cdot \{3 \cdot 2 \cdot F(1)\}$

      $= 4 \cdot 3 \cdot 2 \cdot 1 = 24$

The Fibonacci Numbers

- The ***Fibonacci numbers*** $f_{n\geq 0}$ is a famous series defined by:

$$f_0 = 0, \quad f_1 = 1, \quad f_{n\geq 2} = f_{n-1} + f_{n-2}$$



Inductive Proof about Fibonacci Numbers

- **Theorem:** $f_n < 2^n$. ← Implicitly for all $n \in \mathbf{N}$
- **Proof:** By induction
  - Basis step: $f_0 = 0 < 2^0 = 1$
    $f_1 = 1 < 2^1 = 2$
    Note: use of base cases of recursive definition
  - Inductive step: Use 2nd principle of induction (strong induction).

    Assume $\forall 0 \le i \le k,\ f_i < 2^i$. Then

    $$f_{k+1} = f_k + f_{k-1} \text{ is}$$
    $$< 2^k + 2^{k-1}$$
    $$< 2^k + 2^k = 2^{k+1}. \ \blacksquare$$

A Lower Bound on Fibonacci Numbers

- **Theorem:** For all integers $n \geq 3$, $f_n > \alpha^{n-2}$, where $\alpha = (1 + 5^{1/2})/2 \approx 1.61803$.
- **Proof.** (Using strong induction.)
  - Let $P(n) = (f_n > \alpha^{n-2})$.
  - **Basis step:**
    For $n = 3$, note that $\alpha^{n-2} = \alpha < 2 = f_3$.
    For $n = 4$, $\alpha^{n-2} = \alpha^2$

$$= (1 + 2 \cdot 5^{1/2} + 5)/4$$
$$= (3 + 5^{1/2})/2$$
$$\approx 2.61803 \qquad (= \alpha + 1)$$
$$< 3 = f_4.$$

A Lower Bound on Fibonacci

- **Inductive step:** For $k \geq 4$, assume $P(j)$ for $3 \leq j \leq k$, prove $P(k+1)$.
  - $f_{k+1} = f_k + f_{k-1} > \alpha^{k-2} + \alpha^{k-3}$ (by inductive hypothesis, $f_{k-1} > \alpha^{k-3}$ and $f_k > \alpha^{k-2}$).
  - Note that $\alpha^2 = \alpha + 1$.
    since $(3 + 5^{1/2})/2 = (1 + 5^{1/2})/2 + 1$
  - Thus, $\alpha^{k-1} = \alpha^2 \alpha^{k-3} = (\alpha + 1)\alpha^{k-3}$
    $= \alpha\alpha^{k-3} + \alpha^{k-3} = \alpha^{k-2} + \alpha^{k-3}$.
  - So, $f_{k+1} = f_k + f_{k-1} > \alpha^{k-2} + \alpha^{k-3} = \alpha^{k-1}$.
  - Thus $P(k+1)$. ■

Recursively Defined Sets

- An infinite set *S* may be defined recursively, by giving:
  - A small finite set of *base* elements of *S*.
  - A rule for constructing new elements of *S* from previously-established elements.
  - Implicitly, *S* has no other elements but these.

- **Example:** Let $\underbrace{3 \in S}_{\text{base element (basis step)}}$, and let $\underbrace{x+y \in S \text{ if } x,y \in S}_{\text{construction rule (recursive step)}}$.
  What is *S*?

Example *cont.*

- Let $3 \in S$, and let $x+y \in S$ if $x,y \in S$. What is S?
  - $3 \in S$ (*basis step*)
  - 6 (= 3 + 3) is in S (*first application of recursive step*)
  - 9 (= 3 + 6) and 12 (= 6 + 6) are in S (*second application of the recursive step*)
  - 15 (= 3 + 12 or 6 + 9), 18 (= 6 + 12 or 9 + 9), 21 (= 9 + 12), 24 (= 12 + 12) are in S (*third application of the recursive step*)
  - … so on
  - Therefore, **S = {3,** 6, 9, 12, 15, 18, 21, 24,…}
    = set of *all positive multiples of* 3

The Set of All Strings

- Given an alphabet **Σ**, the set **Σ*** of all strings over **Σ** can be recursively defined by:
  - <u>**Basis step**</u>: $\lambda \in \Sigma^*$ (λ : empty string)
  - <u>**Recursive step**</u>: $(w \in \Sigma^* \wedge x \in \Sigma) \rightarrow wx \in \Sigma^*$

- **Example**: If $\Sigma = \{0, 1\}$ then
  - $\lambda \in \Sigma^*$ (*basis step*)
  - 0 and 1 are in Σ^* (*first application of recursive step*)
  - 00, 01, 10, and 11 are in Σ^* (*second application of the recursive step*)
  - … so on
  - Therefore, Σ^* consists of all finite strings of 0's and 1's together with the empty string

String: Example

- Show that if $\Sigma = \{a, b\}$ then *aab* is in Σ^*.

  Proof: We construct it with a finite number of applications of the basis and recursive steps in the definition of Σ^*:

1. $\lambda \in \Sigma^*$ by the basis step.
2. By step 1, the recursive step in the definition of Σ^* and the fact that $a \in \Sigma$, we can conclude that $\lambda a = a \in \Sigma^*$.

Proof *cont.*

3. Since $a \in \Sigma^*$ from step 2, and $a \in \Sigma$, applying the recursive step again we conclude that $aa \in \Sigma^*$.

4. Since $aa \in \Sigma^*$ from step 3 and $b \in \Sigma$, applying the recursive step again we conclude that $aab \in \Sigma^*$.

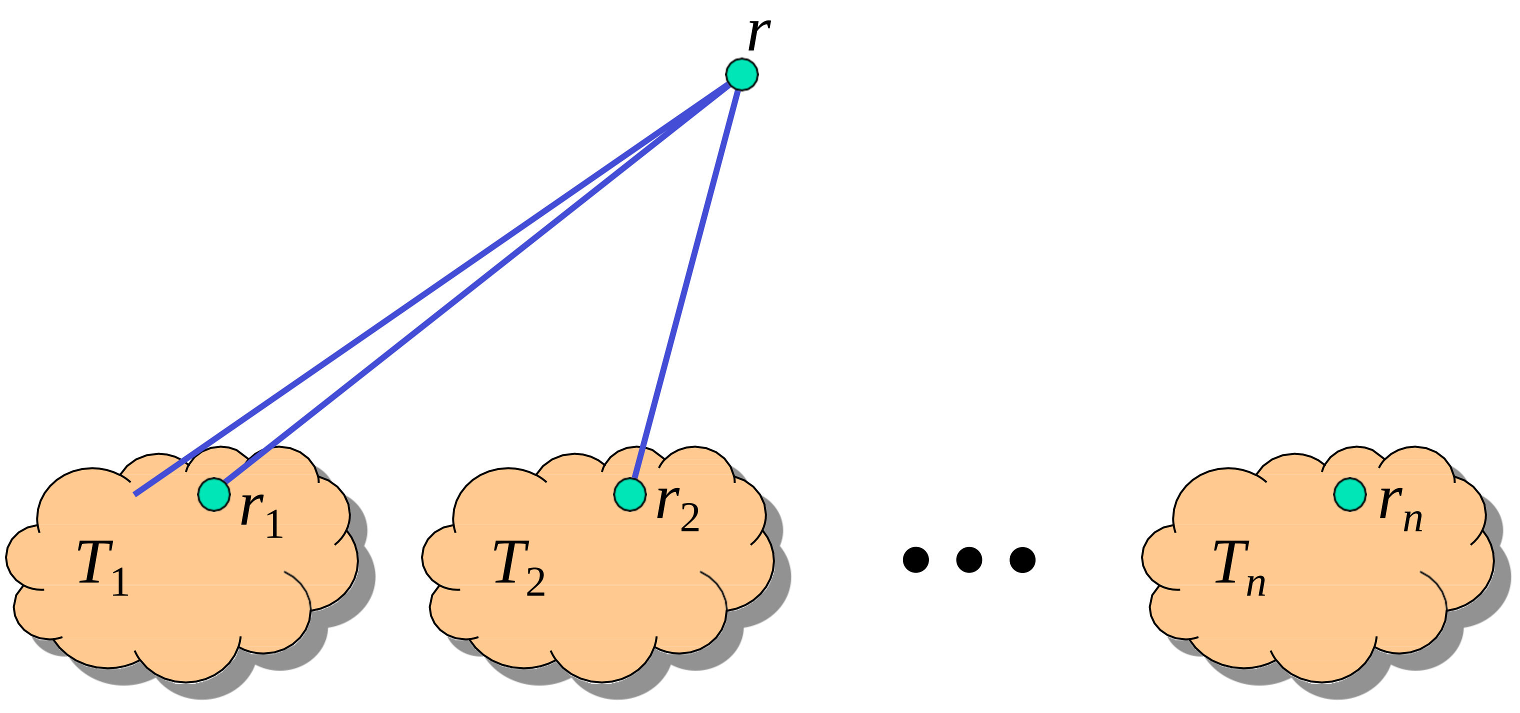
- Since we have shown $aab \in \Sigma^*$ with a finite number of applications of the basis and recursive steps in the definition we have finished the proof.

Rooted Trees

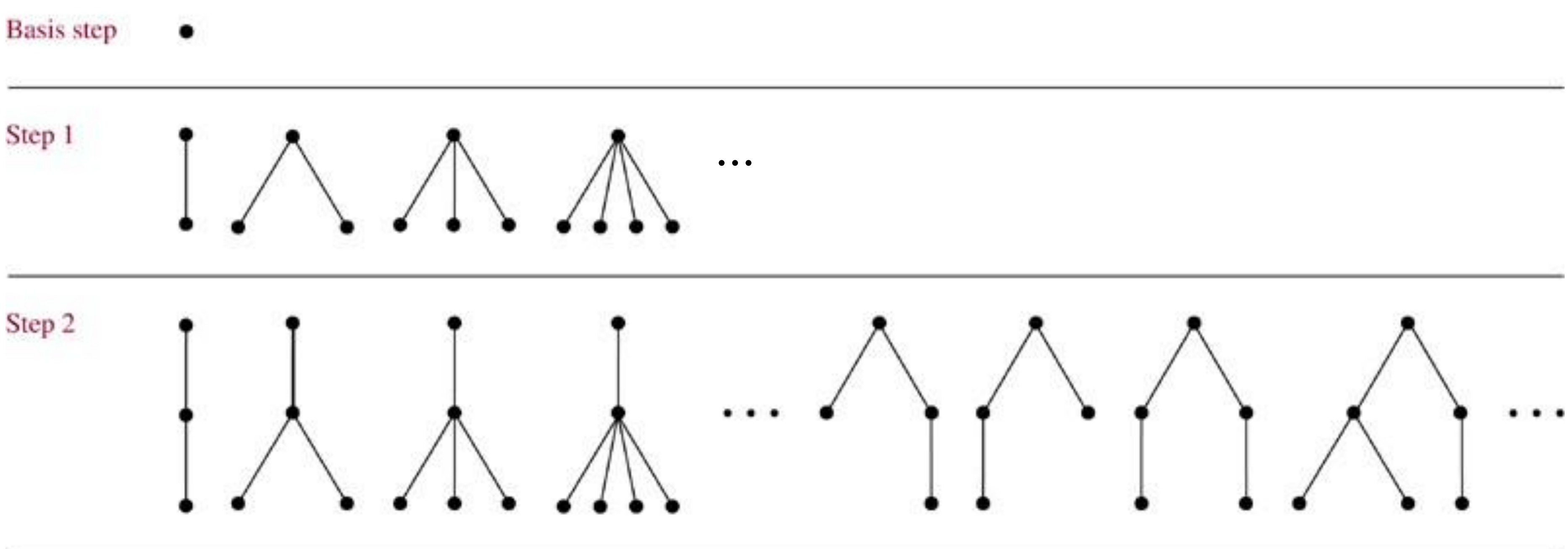
- Trees will be covered in more depth in chapter 10.
  - Briefly, a tree is a graph in which there is exactly one undirected path between each pair of nodes.
  - An undirected graph can be represented as a set of unordered pairs (called *arcs*) of objects called *nodes*.
- Definition of the set of rooted trees:
  - **Basis step**: Any single node r is a rooted tree.
  - **Recursive step**: If $T_1, \ldots, T_n$ are disjoint rooted trees with respective roots $r_1, \ldots, r_n$, and r is a node not in any of the T_i's, then another rooted tree is $\{(r, r_1), \ldots, (r, r_n)\} \cup T_1 \cup \cdots \cup T_n$.

Illustrating Rooted Tree

- How rooted trees can be combined to form a new rooted tree…



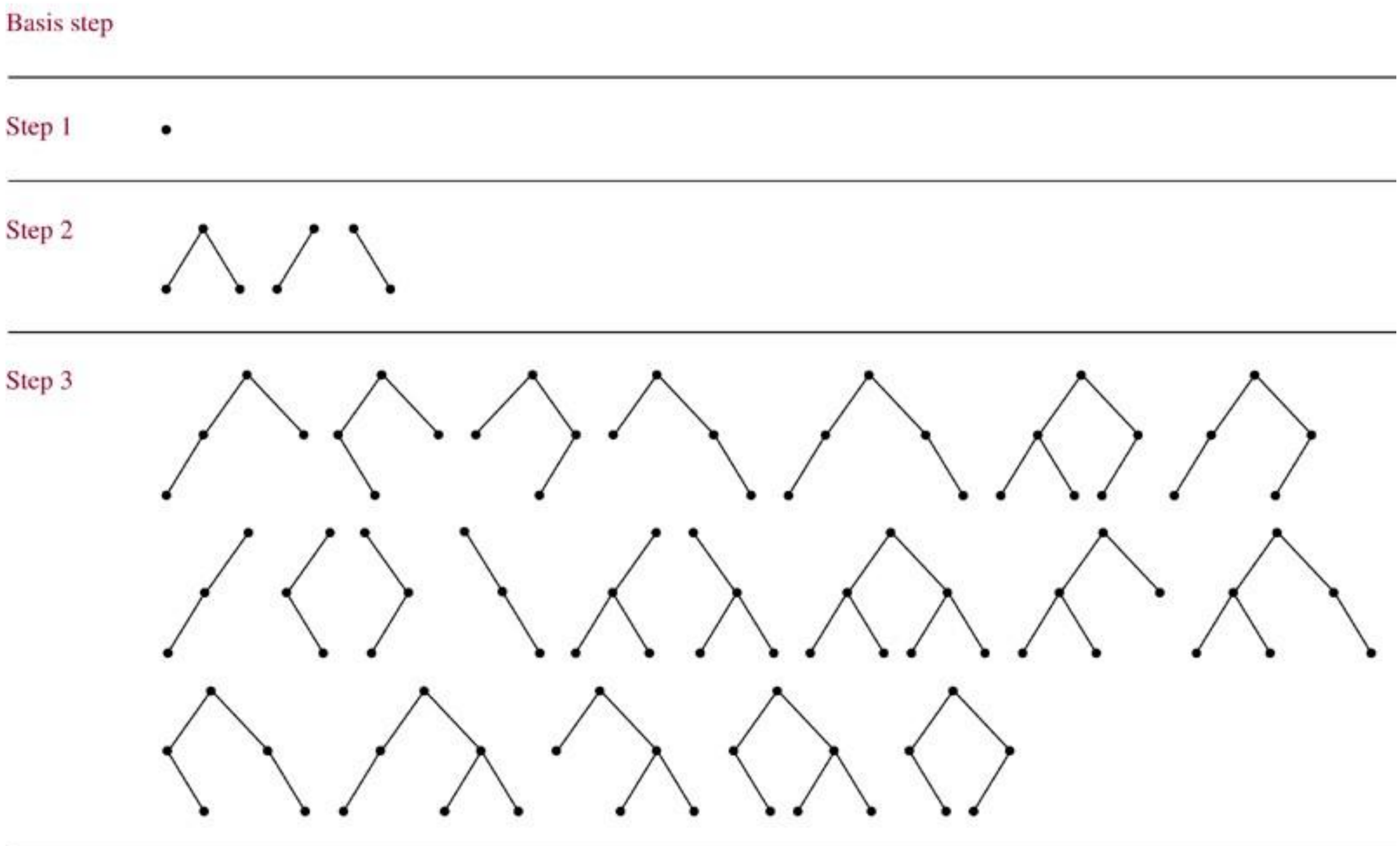
Building Up Rooted Trees



Extended Binary Trees

- A special case of rooted trees.
- Recursive definition of extended binary trees:
  - **Basis step**: The empty set $\emptyset$ is an extended binary tree.
  - **Recursive step**: If T_1, T_2 are disjoint extended binary trees, then $e_1 \cup e_2 \cup T_1 \cup T_2$ is an extended binary tree, where $e_1 = \emptyset$ if $T_1 = \emptyset$, and $e_1 = \{(r, r_1)\}$ if $T_1 \neq \emptyset$ and has root r_1, and similarly for e_2. (T_1 is the left subtree and T_2 is the right subtree.)

Building Up Extended BinaryTrees



Lamé's Theorem

- **Theorem:** $\forall a,b \in \mathbf{N}$, $a \geq b > 0$, and let n be the number of steps Euclid's algorithm needs to compute $\gcd(a,b)$.
  Then $n \leq 5k$, where $k = \lfloor \log_{10} b \rfloor + 1$ is the number of decimal digits in b.
  - Thus, Euclid's algorithm is linear-time in the number of digits in b.
    (or, Euclid's algorithm is $O(\log a)$)
- **Proof:**
  - Uses the Fibonacci sequence! (See next!)

Proof of Lamé's Theorem

- Consider the sequence of division-algorithm equations used in Euclid's alg.:

$$r_0 = r_1 q_1 + r_2 \quad \text{with } 0 \le r_2 < r_1$$

$$r_1 = r_2 q_2 + r_3 \quad \text{with } 0 \le r_3 < r_2$$

$$\ldots$$

$$r_{n-2} = r_{n-1} q_{n-1} + r_n \quad \text{with } 0 \le r_n < r_{n-1}$$

$$r_{n-1} = r_n q_n + r_{n+1} \quad \text{with } r_{n+1} = 0 \text{ (terminate)}$$

Where $a = r_0$, $b = r_1$, and $\gcd(a,b)=r_n$.

- The number of divisions (iterations) is n.

Continued on next slide…

Lamé Proof *cont.*

- Since $r_0 \geq r_1 > r_2 > \ldots > r_n$, each quotient $q_i \equiv \lfloor r_{i-1}/r_i \rfloor \geq 1$.
- Since $r_{n-1} = r_n q_n$ and $r_{n-1} > r_n$, $q_n \geq 2$.
- So we have the following relations between r and f:

$$r_n \geq 1 = f_2$$

$$r_{n-1} \geq 2r_n \geq 2f_2 = f_3$$

$$r_{n-2} \geq r_{n-1} + r_n \geq f_2 + f_3 = f_4$$

$$\ldots$$

$$r_2 \geq r_3 + r_4 \geq f_{n-1} + f_{n-2} = f_n$$

$$b = r_1 \geq r_2 + r_3 \geq f_n + f_{n-1} = f_{n+1}.$$

- Thus, if $n > 2$ divisions are used, then $b \geq f_{n+1} > \alpha^{n-1}$.
  - Thus, $\log_{10} b > \log_{10}(\alpha^{n-1}) = (n-1)\log_{10} \alpha \approx (n-1)0.208 > (n-1)/5$.
  - If b has k decimal digits, then $\log_{10} b < k$, so $n-1 < 5k$, so $n \leq 5k$.

Lecture

Chapter 4. Induction and Recursion

3. Recursive Definitions and Structural Induction
4. Recursive Algorithms

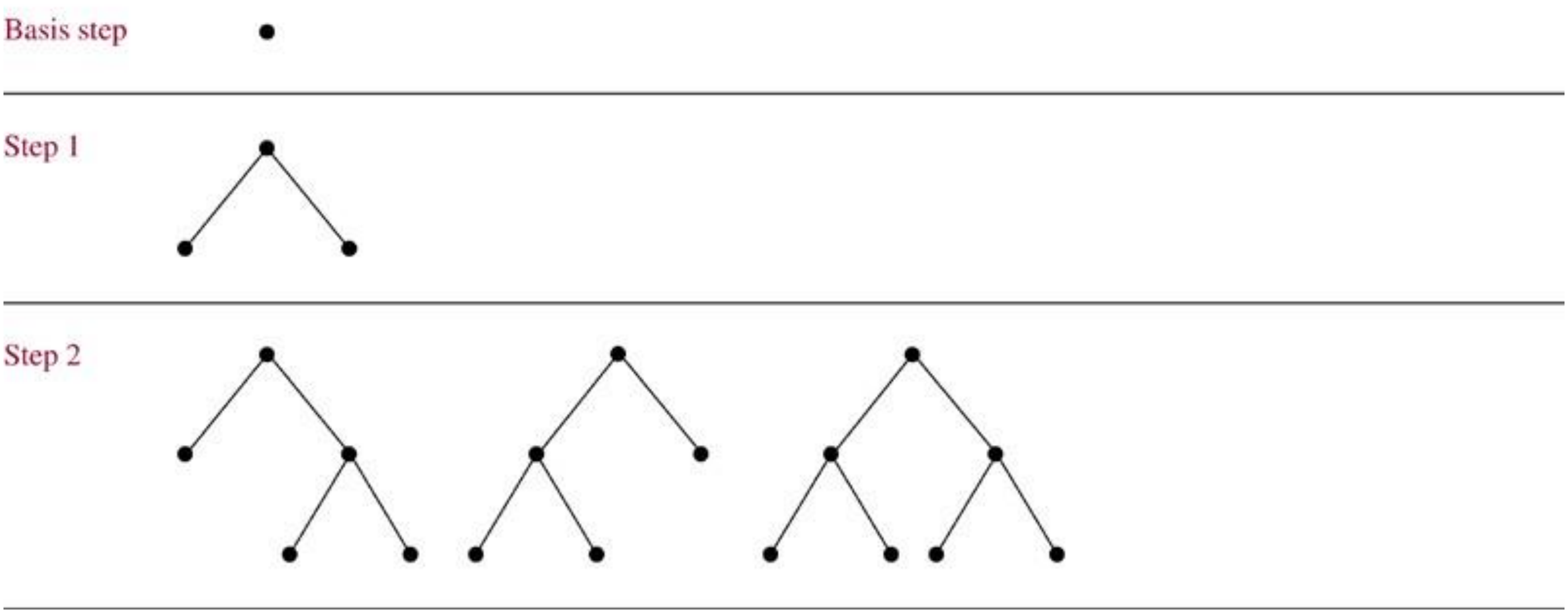
Review: Recursive Definitions

- ***Recursion*** is the general term for the practice of defining an object in terms of *itself* or of part of itself.
- In ***recursive definitions***, we similarly *define* a function, a predicate, a set, or a more complex structure over an infinite domain (universe of discourse) by:
  - defining the function, predicate value, set membership, or structure of larger elements in terms of those of smaller ones.

Full Binary Trees

- A special case of extended binary trees.
- Recursive definition of full binary trees:
  - **Basis step**: A single node r is a full binary tree.
    - Note this is different from the extended binary tree base case.
  - **Recursive step**: If T_1, T_2 are disjoint full binary trees with roots r_1 and r_2, then $\{(r, r_1), (r, r_2)\} \cup T_1 \cup T_2$ is an full binary tree.

Building Up Full Binary Trees



Structural Induction

- Proving something about a recursively defined object using an inductive proof whose structure mirrors the object's definition.
  - **Basis step**: Show that the result holds for all elements in the set specified in the basis step of the recursive definition
  - **Recursive step**: Show that if the statement is true for each of the elements in the new set constructed in the recursive step of the definition, the result holds for these new elements.

Structural Induction: Example

- Let $3 \in S$, and let $x+y \in S$ if $x,y \in S$.
  Show that S is the set of positive multiples of 3.
- Let $A = \{n \in \mathbf{Z}^+ \mid (3|n)\}$. We'll show that $A = S$.
  - **Proof:** We show that $A \subseteq S$ and $S \subseteq A$.
  - To show $A \subseteq S$, show $[n \in \mathbf{Z}^+ \wedge (3|n)] \rightarrow n \in S$.
    - **Inductive proof.** Let $n \in \mathbf{Z}^+$ and $P(n) = 3n \in S$.
      Induction over positive multiples of 3.

      Basis case: $n = 1$, thus $3 \in S$ by definition of S.

      Inductive step: Given $P(k)$, prove $P(k+1)$.
      By inductive hypothesis $3k \in S$, and $3 \in S$,
      so by definition of S, $3(k + 1) = 3k + 3 \in S$.

Example *cont.*

- To show $S \subseteq A$: let $n \in S$, show $n \in A$.
  - **Structural inductive proof.** Let $P(n) = n \in A$.

    Two cases: $n = 3$ (basis case), which is in A, or $n = x + y$ for $x, y \in S$ (recursive step).

    We know x and y are positive, since neither rule generates negative numbers.

    So, $x < n$ and $y < n$, and so we know x and y are in A, by strong inductive hypothesis.

    Since $3|x$ and $3|y$, we have $3|(x+y)$, thus $x + y = n \in A$. ■

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.*

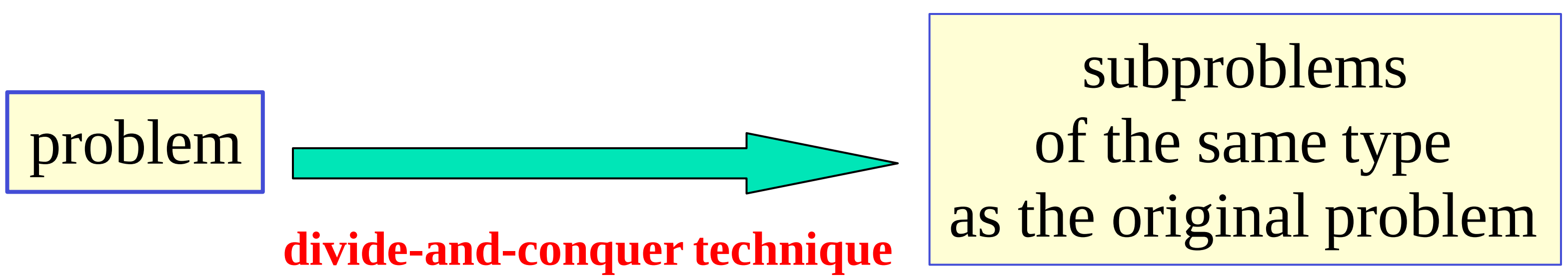
Example

- A procedure to compute a^n.

procedure *power*($a \neq 0$: real, $n \in \mathbf{N}$)

  if $n = 0$ **then return** 1

  else return $a \cdot power(a, n-1)$



Recursive Euclid's Algorithm

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

  procedure *gcd*(a,$b \in \mathbf{N}$ with $a < b$)

  if $a = 0$ **then return** b

  else return *gcd*(b **mod** a, a)

- 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 Linear Search

{Finds x in series a at a location $\geq i$ and $\leq j$}

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, $\geq i$ and $\leq j$}

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 ½}

return *binarySearch*(a, x, i, $m-1$)

else if $x > a_m \wedge j > m$ {If it's to right, check that ½}

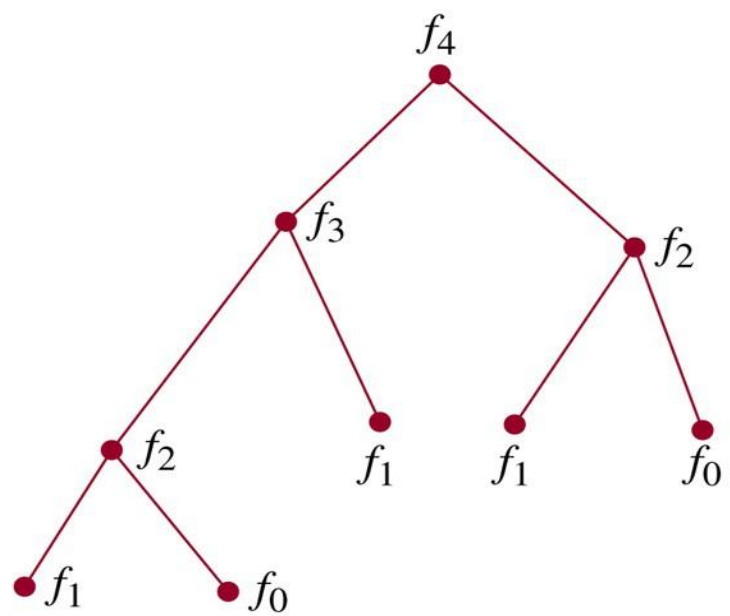
return *binarySearch*(a, x, $m+1$, j)

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**:

  $$\textit{fibonacci}(k+1) = \underbrace{\textit{fibonacci}(k)}_{\substack{\text{by } P(k): \\ f_{k+1} - 1 \text{ additions}}} + \underbrace{\textit{fibonacci}(k-1)}_{\substack{\text{by } P(k-1): \\ f_k - 1 \text{ additions}}}$$

  - 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

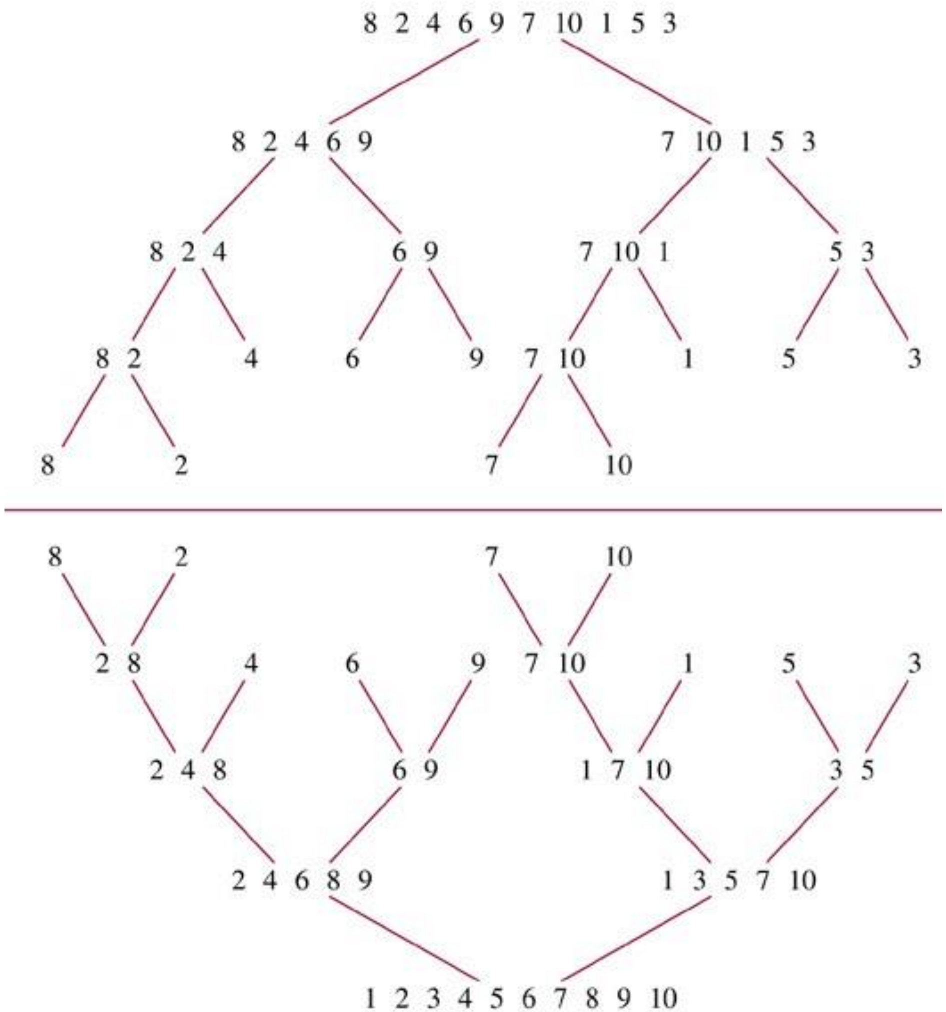
    $$(f_{k+1} - 1) + (f_k - 1) + 1 = f_{k+1} + f_k - 1 = f_{k+2} - 1. \blacksquare$$

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*($L = \ell_1, \ldots, \ell_n$)
  if $n > 1$ **then**
    $m := \lfloor n/2 \rfloor$ {this is rough ½-way point}
    $L_1 := \ell_1, \ldots, \ell_m$
    $L_2 := \ell_{m+1}, \ldots, \ell_n$
    $L := merge(mergesort(L_1), mergesort(L_2))$
  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 $(a_1, \mathit{merge}((a_2, \ldots, a_{|A|}), B))$

else

return $(b_1, \mathit{merge}(A, (b_2, \ldots, b_{|B|})))$

Efficiency of Recursive Algorithm

- 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 $x \cdot y \textbf{ mod } m = x \cdot (y \textbf{ mod } m) \textbf{ mod } m$.
  (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 ($b\cdot$*mpower*(b,$n-1$,m)) **mod** m

- What is its time complexity? $\Theta(\log n)$ steps

A Slight Variation

- Nearly identical but takes $\Theta(n)$ time instead!

procedure *mpower*(b,n,m) {same signature}

if $n=0$ **then return** 1

else if $2|n$ **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!