Discrete Mathematics
Lec7: Induction and Recursion2

馬誠佑

Recursively Defined Functions

- To define a function f: $N \rightarrow S\ (for\ any\ set\ S)\ or\ series\ a_n = f(n)$,

We can
  - Give f(0).
  - For n > 0, define f(n) in terms of $f(0), \ldots, f(n-1)$.
- For example, the series $a_n = 2^n can\ be\ recursively\ defined\ by$
  - Let $a_0 = 1$.
  - For n > 0, let $a_n = 2a_{n-1}$

Recursively Defined Functions[1]

Definition: A *recursive* or *inductive definition* of a function consists of two steps.

- BASIS STEP: Specify the value of the function at zero.
- RECURSIVE STEP: Give a rule for finding its value at an integer from its values at smaller integers.

A function $f(n)$ is the same as a sequence $a_0, a_1, \ldots$, where a_i, where $f(i) = a_i$. This was done using recurrence relations in Section 2.4.

Recursively Defined Functions

Example: Suppose *f* is defined by:

$$f(0) = 3,$$

$$f(n+1) = 2f(n) + 3$$

Find *f*(1), *f*(2), *f*(3), *f*(4)

Solution:

$$f(1) = 2f(0) + 3 = 2\cdot 3 + 3 = 9$$

$$f(2) = 2f(1) + 3 = 2\cdot 9 + 3 = 21$$

$$f(3) = 2f(2) + 3 = 2\cdot 21 + 3 = 45$$

$$f(4) = 2f(3) + 3 = 2\cdot 45 + 3 = 93$$

Example: Give a recursive definition of the factorial function *n*!:

Solution:

$$f(0) = 1$$

$$f(n+1) = (n+1)\cdot f(n)$$

Recursive Definition of Factorial

- Given an inductive definition of the factorial function

$$F(n) = n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n.$$

  - Base case: F(0) = 1
  - Recursive part: For $n \in Z^+, F(n) = n \cdot F(n-1)$.
    - $F(1) = 1 \cdot F(0) = 1 \cdot 1 = 1$
    - $F(2) = 2 \cdot F(1) = 2 \cdot 1 = 2$
    - $F(3) = 3 \cdot F(2) = 3 \cdot 2 = 6$
    - …

Recursively Defined Functions₃

Example: Give a recursive definition of:

$$\sum_{k=0}^{n} a_k .$$

Solution: The first part of the definition is

$$\sum_{k=0}^{0} a_k = a_0 .$$

The second part is $\sum_{k=0}^{n+1} a_k = \left(\sum_{k=0}^{n} a_k\right) + a_{n+1}$

Fibonacci Numbers[1]

Fibonacci
(1170- 1250)

Example : The Fibonacci numbers are defined as follows:

$$f_0 = 0$$

$$f_1 = 1$$

$$f_n = f_{n-1} + f_{n-2}$$

Find f_2, f_3, f_4, f_5 .

$$f_2 = f_1 + f_0 = 1 + 0 = 1$$

$$f_3 = f_2 + f_1 = 1 + 1 = 2$$

$$f_4 = f_3 + f_2 = 2 + 1 = 3$$

$$f_5 = f_4 + f_3 = 3 + 2 = 5$$

In Chapter 8, we will use the Fibonacci numbers to model population growth of rabbits. This was an application described by Fibonacci himself.

Next, we use strong induction to prove a result about the Fibonacci numbers.

Inductive Proof about Fibonacci Series

- Theorem: For all $n \in N, f_n < 2^n$

Pf: Prove the theorem by induction.

- Basis step:
  - $f_0 = 0 < 2^0 = 1$
  - $f_1 = 1 < 2^1 = 2$
- Inductive step:

To apply the strong induction, assume $\forall k < n, f_k < 2^k. Then,$

$$f_n = f_{n-1} + f_{n-2} < 2^{n-1} + 2^{n-2} < 2^{n-1} + 2^{n-1} = 2^n.$$

- So, $f_n < 2^n$ $is\ proved$.

Fibonacci Numbers 2

Example : Show that whenever $n \geq 3, f_n > \alpha^{n-2}$, where $\alpha = \left(1+\sqrt{5}\right)/2$.

Solution: Let *P*(*n*) be the statement $f_n > \alpha^{n-2}$.

Use strong induction to show that *P*(*n*) is true whenever $n \geq 3$.

- BASIS STEP: *P*(3) holds since $\alpha < 2 = f_3$

  P(4) holds since $\alpha^2 = \left(3+\sqrt{5}\right)/2 < 3 = f_4$.
- INDUCTIVE STEP: Assume that *P*(*j*) holds, i.e., $f_j > \alpha^{j-2}$ for all integers *j* with $3 \leq j \leq k$, where $k \geq 4$. Show that *P*(*k* + 1) holds, i.e., $f_{k+1} > \alpha^{k-1}$.
  - Since $\alpha^2 = \alpha + 1$ (because α is a solution of $x^2 - x - 1 = 0$),

$$\alpha^{k-1} = \alpha^2 \cdot \alpha^{k-3} = (\alpha+1)\cdot\alpha^{k-3} = \alpha\cdot\alpha^{k-3} + 1\cdot\alpha^{k-3} = \alpha^{k-2} + \alpha^{k-3}$$

  - By the inductive hypothesis, because $k \geq 4$ we have

$$f_{k-1} > \alpha^{k-3}, \qquad f_k > \alpha^{k-2}.$$

  - Therefore, it follows that

$$f_{k+1} = f_k + f_{k-1} > \alpha^{k-2} + \alpha^{k-3} = \alpha^{k-1}.$$

Why does this equality hold?

  - Hence, *P*(*k* + 1) is true.

Lamé’s Theorem 1

Gabriel Lamé (1795-1870)

Lamé’s Theorem: Let *a* and *b* be positive integers with $a \geq b$. Then the number of divisions used by the Euclidian algorithm to find gcd(*a*,*b*) is less than or equal to five times the number of decimal digits in *b*.

Proof: When we use the Euclidian algorithm to find gcd(*a*,*b*) with $a \geq b$,

- *n* divisions are used to obtain (with $a = r_0, b = r_1$):

$$\begin{aligned} r_0 &= r_1 q_1 + r_2 & 0 \leq r_2 < r_1, \\ r_1 &= r_2 q_2 + r_3 & 0 \leq r_3 < r_2, \\ & \text{M} & \\ r_{n-2} &= r_{n-1} q_{n-1} + r_n & 0 \leq r_n < r_{n-1}, \\ r_{n-1} &= r_n q_n. & \end{aligned}$$

- Since each quotient $q_1, q_2, \ldots, q_{n-1}$ is at least 1 and $q_n \geq 2$: $\because r_n < r_{n-1}$

$$\begin{aligned} r_n &\geq 1 = f_2, \\ r_{n-1} &\geq 2r_n \geq 2f_2 = f_3, \\ r_{n-2} &\geq r_{n-1} + r_n \geq f_3 + f_2 = f_4, \\ & \text{M} \\ r_2 &\geq r_3 + r_4 \geq f_{n-1} + f_{n-2} = f_n, \\ b = r_1 &\geq r_2 + r_3 \geq f_n + f_{n-1} = f_{n+1}. \end{aligned}$$

Lamé's Theorem 2

It follows that if n divisions are used by the Euclidian algorithm to find gcd(a,b) with $a \geq b$, then $b \geq f_{n+1}$. By last example, $f_{n+1} > \alpha^{n-1}$, for $n > 2$, where $\alpha = (1 + \sqrt{5})/2$. Therefore, $b > \alpha^{n-1}$.

Because $\log_{10} \alpha \approx 0.208 > 1/5$, $\log_{10} b > (n-1) \log_{10} \alpha > (n-1)/5$. Hence,

$$n - 1 < 5 \cdot \log_{10} b.$$

Suppose that b has k decimal digits. Then $b < 10^k$ and $\log_{10} b < k$. It follows that $n - 1 < 5k$ and since k is an integer, $n \leq 5k$.

As a consequence of Lamé's Theorem, $O(\log b)$ divisions are used by the Euclidian algorithm to find gcd(a,b) whenever $a > b$.

- By Lamé's Theorem, the number of divisions needed to find gcd(a,b) with $a > b$ is less than or equal to 5 $(\log_{10} b + 1)$ since the number of decimal digits in b (which equals $\lfloor \log_{10} b \rfloor + 1$) is less than or equal to $\log_{10} b + 1$.

Lamé's Theorem was the first result in computational complexity

Recursively Defined Sets and Structures 1

Recursive definitions of sets have two parts:

- The *basis step* specifies an initial collection of elements.
- The *recursive step* gives the rules for forming new elements in the set from those already known to be in the set.

Sometimes the recursive definition has an *exclusion rule*, which specifies that the set contains nothing other than those elements specified in the basis step and generated by applications of the rules in the recursive step.

We will always assume that the exclusion rule holds, even if it is not explicitly mentioned.

We will later develop a form of induction, called *structural induction*, to prove results about recursively defined sets.

Recursively Defined Sets and Structures 2

Example : Subset of Integers *S*:

BASIS STEP: 3 ∊ S.

RECURSIVE STEP: If $x \in S$ and $y \in S$, then $x + y$ is in *S.*

Initially 3 is in *S*, then 3 + 3 = 6, then 3 + 6 = 9, etc.

$$S = \{3k | k \in Z^+\}$$

Example: The natural numbers **N**.

BASIS STEP: 0 ∊ **N.**

RECURSIVE STEP: If *n* is in **N**, then $n + 1$ is in **N**.

Initially 0 is in N, then 0 + 1 = 1, then 1 + 1 = 2, etc.

Strings

Definition: The set Σ* of *strings* over the alphabet Σ:

BASIS STEP: λ ∈ Σ* (λ is the empty string)

RECURSIVE STEP: If *w* is in Σ* and *x* is in Σ, then *wx* ∈ Σ*.

$$\Sigma^* = \bigcup_{n \in N} \Sigma^n$$

Example: If Σ = {0,1}, the strings in in Σ* are the set of all bit strings, λ,0,1, 00,01,10, 11, etc.

Example: If Σ = {*a*,*b*}, show that *aab* is in Σ*.

- Since λ ∈ Σ* and *a* ∈ Σ, *a* ∈ Σ*.
- Since *a* ∈ Σ* and *a* ∈ Σ, *aa* ∈ Σ*.
- Since *aa* ∈ Σ* and *b* ∈ Σ, *aab* ∈ Σ*.

String Concatenation

Definition: Two strings can be combined via the operation of *concatenation*. Let Σ be a set of symbols and Σ* be the set of strings formed from the symbols in Σ. We can define the concatenation of two strings, denoted by ·, recursively as follows.

BASIS STEP: If $w \in \Sigma^*$, then $w \cdot \lambda = w$.

RECURSIVE STEP: If $w_1 \in \Sigma^*$ and $w_2 \in \Sigma^*$ and $x \in \Sigma$, then $w_1 \cdot (w_2 x) = (w_1 \cdot w_2)x$.

Often $w_1 \cdot w_2$ is written as $w_1 w_2$.

If $w_1 = abra$ and $w_2 = cadabra$, the concatenation $w_1 w_2 = abracadabra$.

Length of a String

Example: Give a recursive definition of *l*(*w*), the length of the string *w*.

Solution: The length of a string can be recursively defined by:

$$l(\lambda) = 0;$$

$$l(wx) = l(w) + 1 \text{ if } w \in \Sigma^* \text{ and } x \in \Sigma.$$

Balanced Parentheses

Example: Give a recursive definition of the set of balanced parentheses *P*.

Solution:

BASIS STEP: () ∊ *P*

RECURSIVE STEP: If *w* ∊ *P*, then () *w* ∊ *P*, (*w*) ∊ *P* and *w* () ∊ *P*.

Show that (() ()) is in *P*.

Why is))(() not in *P*?

Well-Formed Formulae in Propositional Logic

Definition: The set of *well-formed formulae* in propositional logic involving **T**, **F**, propositional variables, and operators from the set $\{\neg, \land, \lor, \rightarrow, \leftrightarrow\}$.

BASIS STEP: **T**,**F**, and *s*, where *s* is a propositional variable, are well-formed formulae.

RECURSIVE STEP: If *E* and *F* are well formed formulae, then $(\neg E)$, $(E \land F)$, $(E \lor F)$, $(E \rightarrow F)$, $(E \leftrightarrow F)$, are well-formed formulae.

Examples: $((p \lor q) \rightarrow (q \land \mathbf{F}))$ is a well-formed formula.

$pq \land$ is not a well formed formula.

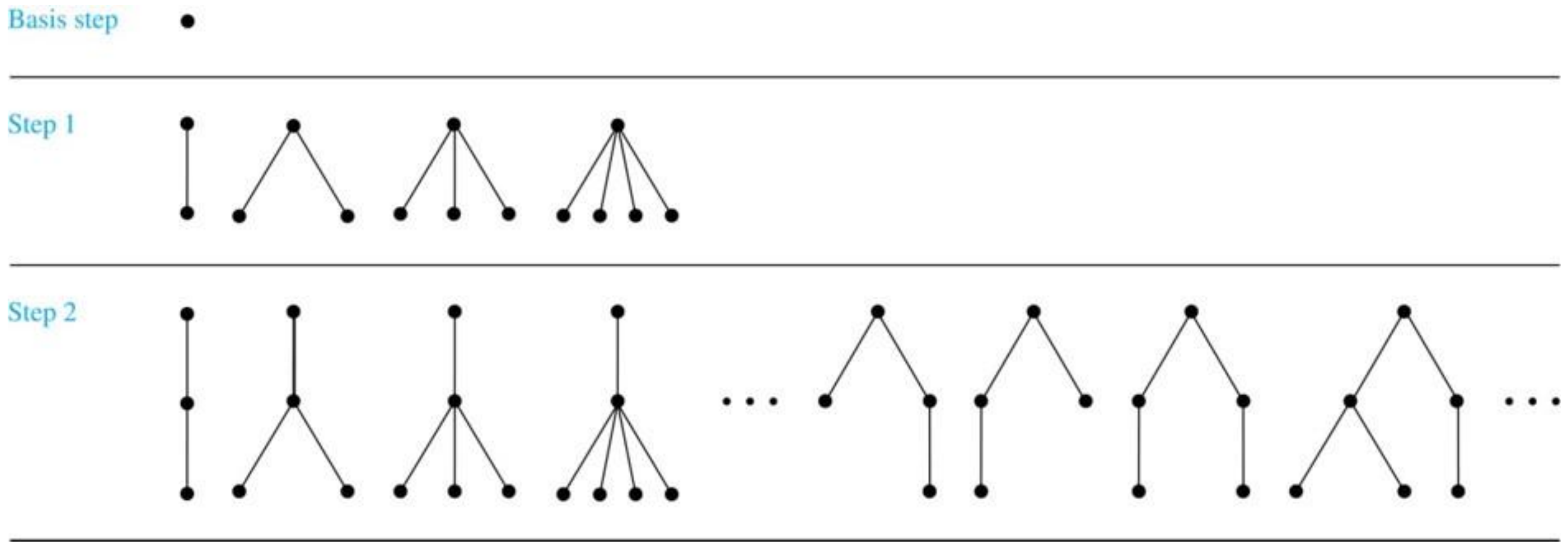
Rooted Trees

Definition: The set of *rooted trees,* where a rooted tree consists of a set of vertices containing a distinguished vertex called the *root*, and edges connecting these vertices, can be defined recursively by these steps:

BASIS STEP: A single vertex r is a rooted tree.

RECURSIVE STEP: Suppose that T_1, T_2, …,T_n are disjoint rooted trees with roots r_1, r_2,…,r_n, respectively. Then the graph formed by starting with a root r, which is not in any of the rooted trees T_1, T_2, …,T_n, and adding an edge from r to each of the vertices r_1, r_2,…,r_n, is also a rooted tree.

Building Up Rooted Trees



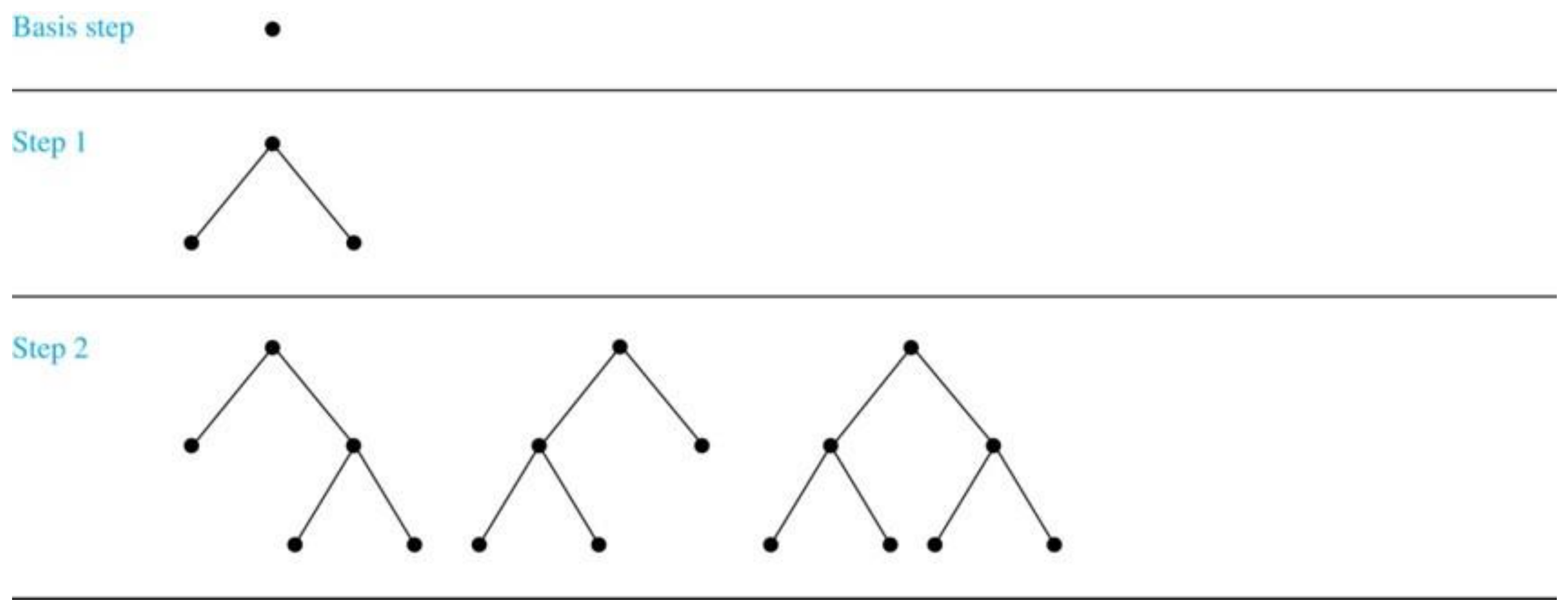
Full Binary Trees 1

Definition: The set of *full binary trees* can be defined recursively by these steps.

BASIS STEP: There is a full binary tree consisting of only a single vertex *r*.

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.

Building Up Full Binary Trees



Recursive Algorithms

Example (A procedure to compute a^n)

```
Procedure power($a \neq 0$: $real, n \in N$)
        if $n = 0$ $\boldsymbol{then\ return}$ $1$
        else return $a \cdot power(a, n-1)$
```

```
Power(5,3)
        return $5 \cdot power(5,2)$
                return $5 \cdot power(5,1)$
                        return $5 \cdot power(5,0)$
                                return 1
```

Recursive Factorial Algorithm

Example: Give a recursive algorithm for computing $n!$, where n is a nonnegative integer.

Solution: Use the recursive definition of the factorial function.

```
procedure factorial(n: nonnegative integer)
if $n = 0$ then return 1
else return $n \cdot factorial(n - 1)$
{output is $n!$}
```

Efficiency of Recursive Algorithms

Modular Exponentiation Alg. #1

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

Example (Returns $b^n\ mod\ m\ by\ using\ b^n = b \cdot b^{n-1}$)

```
procedure $mpower(b \geq 1, n \geq 0, m > b \in N)$
        if $n = 0\ then\ return\ 1$
        else return ($b \cdot mpower(b, n-1, m) mod\ m$)
```

- Draw a diagram to show how the function works.

Recursive Modular Exponentiation Algorithm

Example: Devise a a recursive algorithm for computing b^n **mod** m, where *b, n, and m* are integers with $m \geq 2$, $n \geq 0$, and $1 \leq b \leq m$.

Solution:

(*see text for full explanation*)

```
procedure mpower($b,m,n$: integers with $b > 0$ and $m \geq 2$, $n \geq 0$)
if $n = 0$ then
        return 1
else if $n$ is even then
        return mpower$(b,n/2,m)^2$ mod $m$
else
        return (mpower$(b,\lfloor n/2 \rfloor,m)^2$ mod $m \cdot b$ mod $m$) mod $m$
{output is $b^n$ mod $m$}
```

- power(5,10)
    - Return 5 x $power(5,9)$
        - Return 5 x $power(5,8)$
            - Return 5 x $power(5,7)$
                - Return 5 x $power(5,6)$
                    - Return 5 x power(…
                        - …

- power(5,10)
    - Return $power(5,5)^2$
        - Return 5 x $power(5,4)$
            - Return $power(5,2)^2$
                - Return $power(5,1)^2$
                    - Return power(5,0)

Recursive GCD Algorithm

Example: Give a recursive algorithm for computing the greatest common divisor of two nonnegative integers a and b with $a < b$.

Solution: Use the reduction

$$\gcd(a,b) = \gcd(b \textbf{ mod } a, a)$$

and the condition $\gcd(0,b) = b$ when $b > 0$.

```
procedure gcd(a,b: nonnegative integers
                      with a < b)
if  a = 0 then return b
else  return  gcd (b mod  a, a)
{output is gcd(a, b)}
```

Recursive Binary Search Algorithm

Example: Construct a recursive version of a binary search algorithm.

Solution: Assume we have $a_1, a_2, \ldots, a_n$, an increasing sequence of integers. Initially i is 1 and j is n. We are searching for x.

```
procedure binary search($i, j, x$ : integers, $1 \le i \le j \le n$)
$m := \lfloor (i + j)/2 \rfloor$
if $x = a_m$ then
        return $m$
else if ($x < a_m$ and $i < m$) then
        return binary search($i, m-1, x$)
else if ($x > a_m$ and $j > m$) then
        return binary search($m+1, j, x$)
else return 0
{output is location of $x$ in $a_1, a_2, \ldots, a_n$ if it appears, otherwise 0}
```

Proving Recursive Algorithms Correct

Both mathematical and str0ng induction are useful techniques to show that recursive algorithms always produce the correct output.

Example: Prove that the algorithm for computing the powers of real numbers is correct.

```
procedure power(a: nonzero real number, n: nonnegative integer)
if  n = 0 then return 1
else  return a· power (a, n − 1)
{output is a^n}
```

Solution: Use mathematical induction on the exponent n.

BASIS STEP: $a^0 = 1$ for every nonzero real number a, and $power(a,0) = 1$.

INDUCTIVE STEP: The inductive hypothesis is that $power(a,k) = a^k$, for all $a \neq 0$. Assuming the inductive hypothesis, the algorithm correctly computes a^{k+1}, since

$$power(a,k+1) = a \cdot power(a,k) = a \cdot a^k = a^{k+1}.$$

Merge Sort

Merge Sort works by iteratively splitting a list (with an even number of elements) into two sublists of equal length until each sublist has one element.

Each sublist is represented by a balanced binary tree.

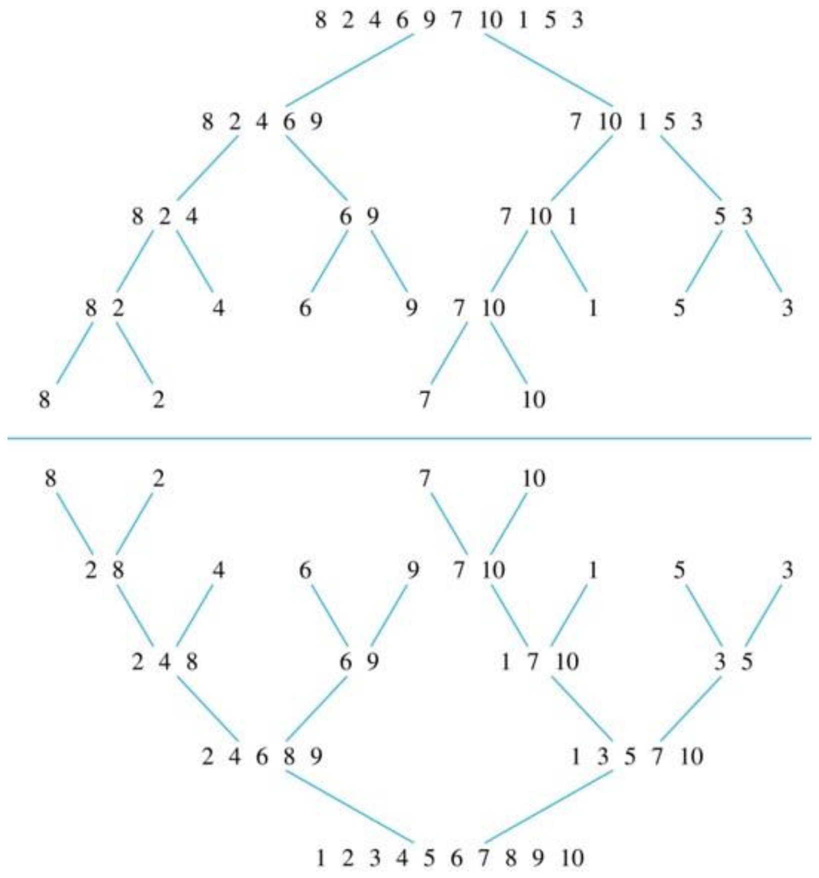
At each step a pair of sublists is successively merged into a list with the elements in increasing order. The process ends when all the sublists have been merged.

The succession of merged lists is represented by a binary tree.

Merge Sort

Example: Use merge sort to put the list
8,2,4,6,9,7,10, 1, 5, 3
into increasing order.

Solution:



Recursive Merge Sort [1]

Example: Construct a recursive merge sort algorithm.

Solution: Begin with the list of *n* elements *L*.

```
procedure mergesort($L = a_1, a_2, \ldots, a_n$ )
if $n > 1$ then
    $m := \lfloor n/2 \rfloor$
    $L_1 := a_1, a_2, \ldots, a_m$
    $L_2 := a_{m+1}, a_{m+2}, \ldots, a_n$
    $L := merge(mergesort(L_1), mergesort(L_2))$
{$L$ is now sorted into elements in increasing order}
```

Recursive Merge Sort 2

Subroutine *merge*, which merges two sorted lists.

```
procedure merge($L_1$, $L_2$ :sorted lists)
$L$ := empty list
while $L_1$ and $L_2$ are both nonempty
    remove smaller of first elements of $L_1$ and $L_2$ from its list;
        put at the right end of $L$
    if this removal makes one list empty
        then remove all elements from the other list and append them to L
return $L$ {$L$ is the merged list with the elements in increasing order}
```

Complexity of Merge: Two sorted lists with m elements and n elements can be merged into a sorted list using no more than $m + n - 1$ comparisons.

Merging Two Lists

Example: Merge the two lists 2,3,5,6 and 1,4.

Solution:

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	