L10: Recursion

- **Recursively Defined Functions**
- Other Recursively Definitions
- Structural Induction
- Recursive Algorithms

- Reading: Rosen 5.3, 5.4, 8.1

From Induction to Recursion

- Induction
$$\forall P\left(P(0) \wedge \forall k\big(P(k) \rightarrow P(k+1)\big) \rightarrow \forall n P(n)\right)$$
where P is a propositional function.
- We can replace P by a definition
  - $P(k)$: k is a natural number
  - Basis: $P(0)$: 0 is a natural number
  - Inductive step: If k is a natural number, $k+1$ is a natural number
- This is actually the definition of natural numbers using the "$+1$" operator.

Examples

- **Example**
  Give a recursive definition of $n! = 1 \cdot 2 \cdot \dots \cdot n$
- **Solution**
  - $1! = 1$
  - $(n+1)! = (n+1) \cdot n!$
- What does this define:
  - $f(x,0) = x$
  - $f(x,y+1) = f(x,y) + 1$
- What does this define:
  - $g(x,0) = 0$
  - $g(x,y+1) = f(g(x,y),x)$

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**:
  - $f(1) = 2f(0) + 3 = 2 \cdot 3 + 3 = 9$
  - $f(2) = 2f(1) + 3 = 2 \cdot 9 + 3 = 21$
  - $f(3) = 2f(2) + 3 = 2 \cdot 21 + 3 = 45$
  - $f(4) = 2f(3) + 3 = 2 \cdot 45 + 3 = 93$

Recursion and Induction

- **Example**
  Suppose f is defined by:
  $f(0) = 3,$
  $f(n+1) = 2f(n) + 3$
  Show that $f(n) = 3(2^{n+1} - 1)$
- **Proof**
  - Basis: $f(0) = 3$ by definition
  - Inductive step: Assume $f(k) = 3(2^{k+1} - 1)$. Want to show $f(k+1) = 3\left(2^{k+2} - 1\right)$:
    $f(k+1) = 2f(k) + 3 = 2 \cdot 3\left(2^{k+1} - 1\right) + 3 = 3\left(2^{k+2} - 1\right)$

Recurrences

- **Definition**
  A recursively defined function over $\mathbf{N}$ is called a recurrence.
- We can solve recurrences by
  - Guess & induction
  - Direct methods

Solving First-Order Linear Recurrence

- Solve the following recurrence:
  $f(0) = a,$
  $f(n) = bf(n-1) + c$ for $n > 0$
  where a, b, c are constants.
- **Solution**

$$\begin{aligned} f(n) &= bf(n-1) + c \\ &= b(b(f(n-2) + c) + c \\ &= b^2 f(n-2) + bc + c \\ &= b^2(bf(n-3) + c) + bc + c \\ &= b^3 f(n-3) + b^2 c + bc + c \end{aligned}$$

Solving First-Order Linear Recurrence

- Solve the following recurrence:
  $f(0) = a$,
  $f(n) = bf(n-1) + c$ for $n > 0$
  where a, b, c are constants.
- **Solution**

$$
\begin{aligned}
f(n) &= bf(n-1) + c \\
&= b(b(f(n-2) + c) + c \\
&= b^2 f(n-2) + bc + c \\
&= b^2(bf(n-3) + c) + bc + c \\
&= b^3 f(n-3) + b^2 c + bc + c
\end{aligned}
$$

Solving First-Order Linear Recurrence

- Solve the following recurrence:
  $f(0) = a,$
  $f(n) = bf(n-1) + c$ for $n > 0$
  where a, b, c are constants.
- **Solution**

$$\begin{aligned} f(n) &= bf(n-1) + c \\ &= b(b(f(n-2) + c) + c \\ &= b^2 f(n-2) + bc + c \\ &= b^2(bf(n-3) + c) + bc + c \\ &= b^3 f(n-3) + b^2 c + bc + c \\ &\dots \\ &= b^n f(0) + b^{n-1}c + b^{n-2}c + \cdots + c \end{aligned}$$

Solving First-Order Linear Recurrence

- Solve the following recurrence:
  $f(0) = a$,
  $f(n) = bf(n-1) + c$ for $n > 0$
  where a, b, c are constants.
- **Solution**

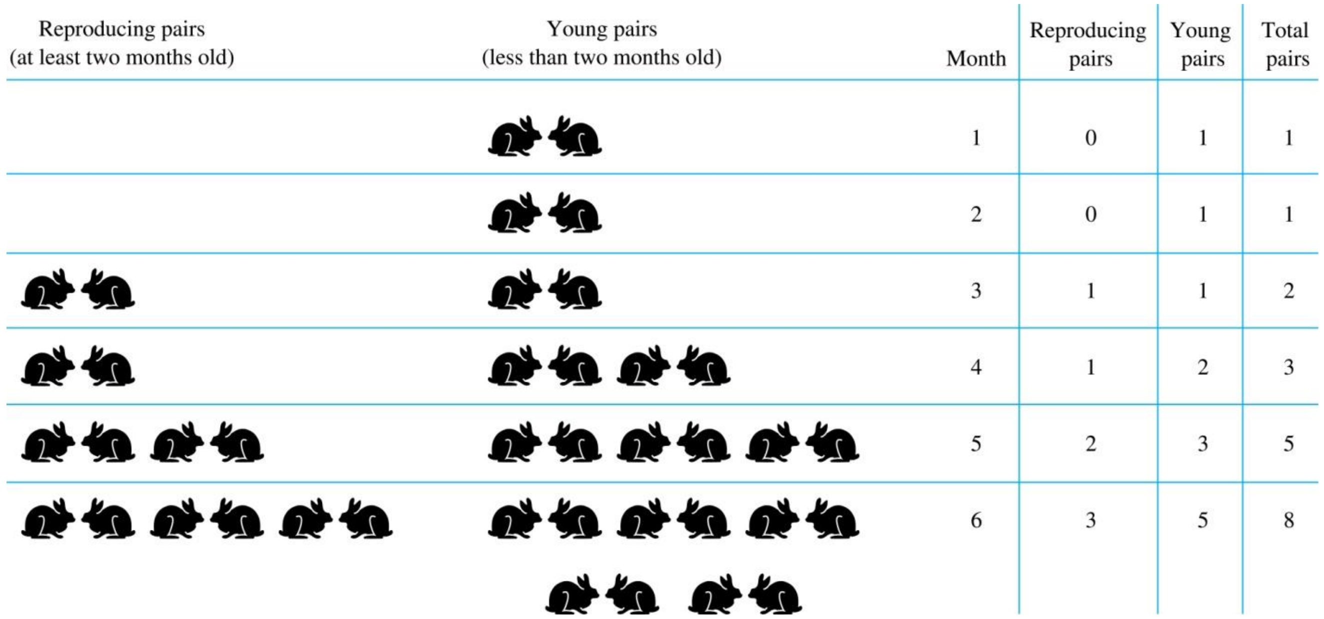
$$\begin{aligned}
f(n) &= bf(n-1) + c \\
&= b(b(f(n-2) + c) + c \\
&= b^2 f(n-2) + bc + c \\
&= b^2(bf(n-3) + c) + bc + c \\
&= b^3 f(n-3) + b^2 c + bc + c \\
&\dots \\
&= b^n f(0) + b^{n-1}c + b^{n-2}c + \cdots + c \\
&= ab^n + c(b^{n-1} + b^{n-2} + \cdots + 1) \\
&= ab^n + c \cdot \frac{b^n - 1}{b - 1}
\end{aligned}$$

Mortgage Calculation

- Initial loan amount: A
- Monthly interest rate: p
  - Note: Banks usually list the p.a. rate. Dividing that by 12 gives the monthly rate
- Monthly Payment: M
- Number of months to clear: n
- Recurrence:
  - $f(0) = A$
  - $f(n) = (1+p)f(n-1) - M$
- Solving the recurrence:
  - $f(n) = A(1+p)^n - M\big((1+p)^n - 1\big)/p$.
  - Want $f(n) = 0$, so $M = \frac{Ap(1+p)^n}{(1+p)^n - 1}$

Counting Rabbits

- **Problem**
  A young pair of rabbits (one of each gender) is placed on an island. A pair of rabbits does not breed until they are 2 months old. After they are 2 months old, each pair of rabbits produces another pair each month. How many pairs of rabbits will be on the island after n months, assuming that rabbits never die?



Reproducing pairs (at least two months old)	Young pairs (less than two months old)	Month	Reproducing pairs	Young pairs	Total pairs
		1	0	1	1
		2	0	1	1
		3	1	1	2
		4	1	2	3
		5	2	3	5
		6	3	5	8

Fibonacci Numbers

- The Fibonacci numbers are defined as follows:
  $f_1 = 1$
  $f_2 = 1$
  $f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$
- This definition corresponds to strong induction
- Find f_3, f_4, f_5, f_6
  - $f_3 = f_2 + f_1 = 1 + 1 = 2$
  - $f_4 = f_3 + f_2 = 2 + 1 = 3$
  - $f_5 = f_4 + f_3 = 3 + 2 = 5$
  - $f_6 = f_5 + f_4 = 5 + 3 = 8$
- This is a second-order linear recurrence
  - See textbook Sec 8.2 for methods to solve it exactly

Fibonacci Numbers

- Show that whenever $n \geq 3$, $f_n > \alpha^{n-2}$, where $\alpha = \frac{1+\sqrt{5}}{2}$.
- **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: $P(3)$ holds: $f_3 = 2 > \alpha$
  $P(4)$ holds: $f_4 = 3 > \alpha^2$
- Inductive step: Assume that $P(j)$ holds for all $3 \leq j \leq k$. Want to show that $P(k+1)$ is true, i.e., $f_{k+1} > \alpha^{k-1}$.
- Note that $\alpha^2 = \alpha + 1$ (α is a solution of $x^2 - x - 1 = 0$)

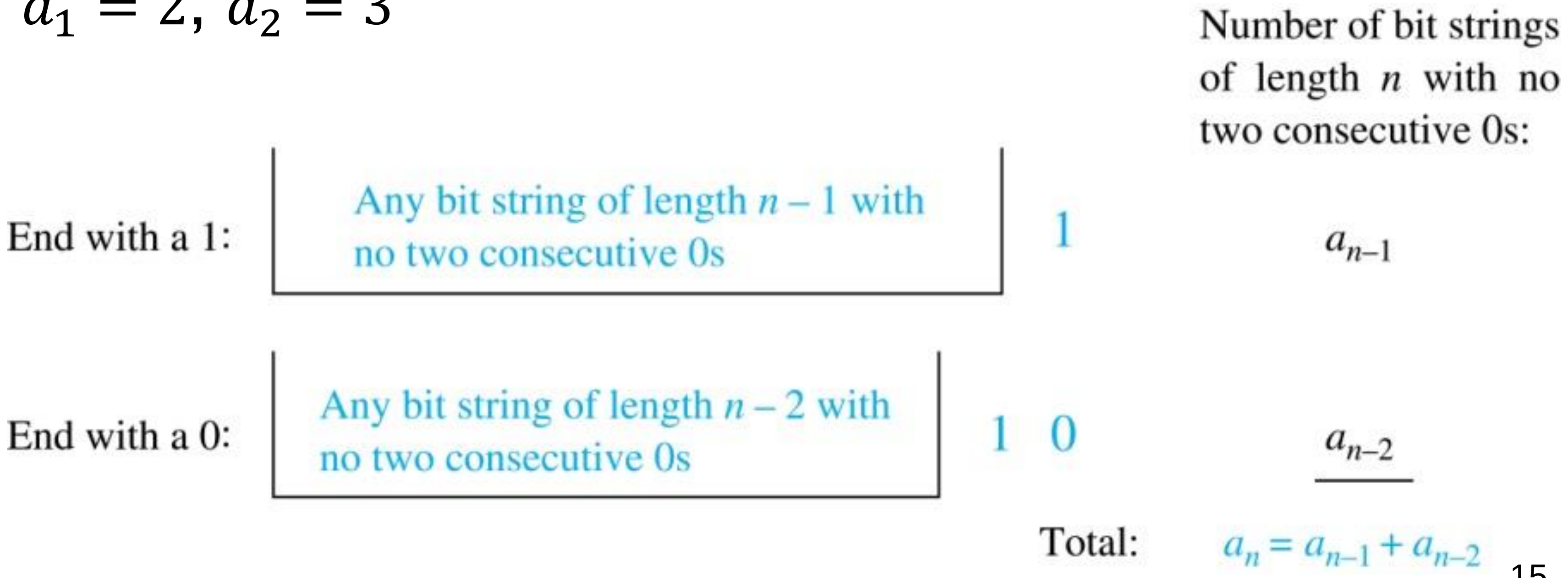
$$\alpha^{k-2} + \alpha^{k-3} = (\alpha + 1)\alpha^{k-3} = \alpha^2 \cdot \alpha^{k-3} = \alpha^{k-1}$$

- By the induction hypothesis: $f_k > \alpha^{k-2}$, $f_{k-1} > \alpha^{k-3}$. So

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

Example: Counting Bit Strings

- **Problem**
  How many bit strings have length n and no two consecutive 0's?
- **Solution**
- Let a_n be the number of such strings
- $a_1 = 2,\ a_2 = 3$



Number of bit strings of length n with no two consecutive 0s:
End with a 1:
Any bit string of length $n-1$ with no two consecutive 0s
1
a_{n-1}
End with a 0:
Any bit string of length $n-2$ with no two consecutive 0s
1 0
a_{n-2}
Total:
$a_n = a_{n-1} + a_{n-2}$

Revisiting Euclid’s GCD Algorithm

- Given a, b with $a > b$, how many steps (mod’s) does Euclid’s algorithm take to compute $\gcd(a, b)$?
- **Lemma**
  If the algorithm takes n steps to compute $\gcd(a, b)$, then $a \geq f_{n+2},\ \ b \geq f_{n+1}$.
- It follows that

$$b \geq f_{n+1} > \alpha^{n-1}$$
$$n < 1 + \log_\alpha b = O(\log b)$$

Proof of Lemma

- Proof by induction: Let $P(n)$ be the statement of the lemma.
- Basis: $P(1)$ is true
  - If the algorithm takes 1 step, then $b \neq 0$ and $b \mid a$.
  - The smallest such a, b with $a > b$ is $a = 2, b = 1$
  - $f_2 = 1 \leq b, f_3 = 2 \leq a$
- Inductive step:
- Assume $P(k)$ is true. Will show $P(k+1)$ is true, i.e., if $\gcd(a, b)$ takes $k+1$ steps, then $a \geq f_{k+3}, b \geq f_{k+2}$
- After first step, we get $\gcd(b, r)$ where $r = a \bmod b$.
- Then it takes k steps to compute $\gcd(b, r)$
- By induction hypothesis, $b \geq f_{k+2}, r \geq f_{k+1}$
- And $a \geq b + r \geq f_{k+1} + f_{k+2} = f_{k+3}$.

Outline

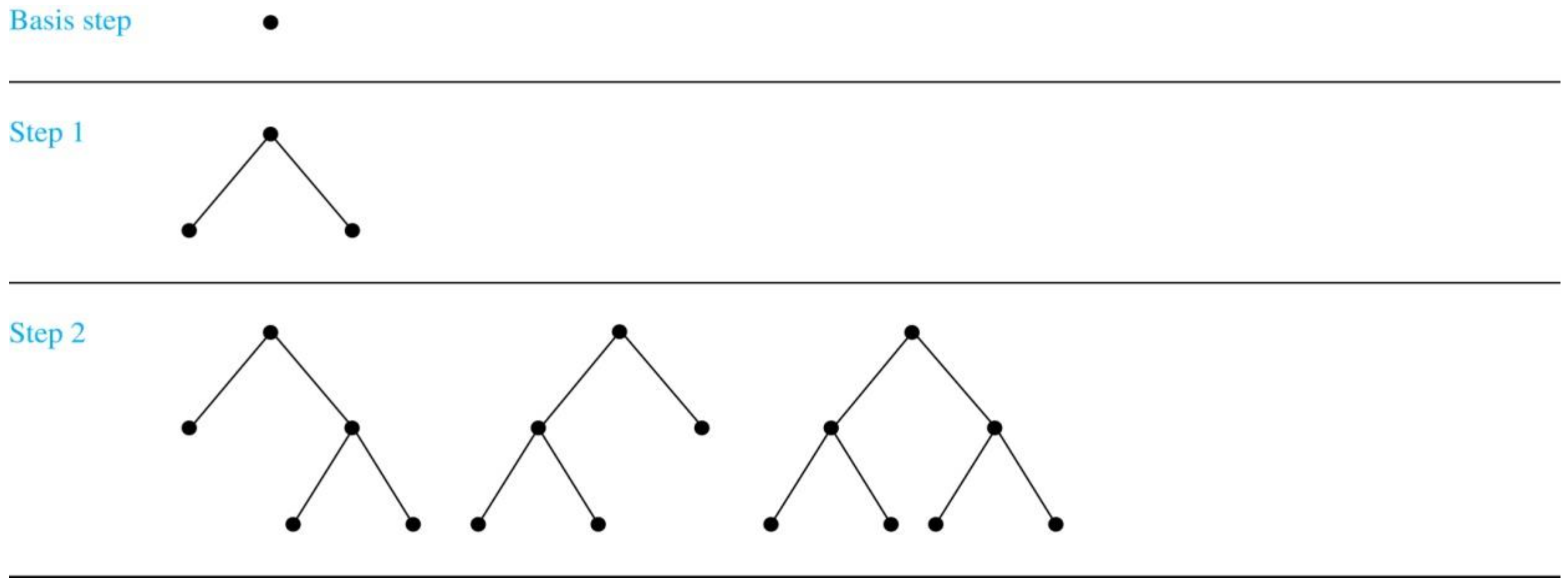
- Recursively Defined Functions
- **Other Recursively Definitions**
- Structural Induction
- Recursive Algorithms

Recursively Defined Sets

- **Example**: Subset of integers S:
  - Basis: $3 \in S$
  - Recursive step: If $x \in S$ and $y \in S$, then $x + y \in S$.
- This defines all positive multiples of 3.
- **Example**:
  - Basis: For any $x \in A, \{x\} \in S$
  - Recursive step: If $x \in S$ and $y \in S$, then $x \cup y \in S$.
- This defines $P(A) - \{\emptyset\}$

Full Binary Trees

- A single vertex r is a full binary tree with root r.
- Let T_1 and T_2 be two disjoint full binary trees with roots r_1 and r_2, respectively. Create a new vertex r, and make r_1 and r_2 the left and right child of r, respectively. The resulting structure is also a full binary tree T with root r. T_1 and T_2 are called the subtrees of T.



Outline

- Recursively Defined Functions
- Other Recursively Definitions
- **Structural Induction**
- Recursive Algorithms

Structural Induction

- **Example**
  Show that the S defined below is the set of all positive integers that are multiples of 3.
  - $3 \in S$
  - If $x \in S$ and $y \in S$, then $x + y \in S.$
- **Proof**
  - Let A be the set of all positive integers divisible by 3. We need to show $A \subseteq S$ and $S \subseteq A$.
  - $A \subseteq S$: Let $P(n)$ be the statement that $3n \in S$.
    - Basis: $P(1)$ is true since $3 \in S$
    - Inductive step: Assume $P(k)$ is true, i.e., $3k \in S$. $3(k+1) = 3k + 3.$ By recursive definition with $y = 3$, we have $3k + 3 \in S$, i.e., $P(k + 1)$ is true.

Proof (cnt'd)

- $S \subseteq A$:
  - Basis: The element defined in the basis step of the definition belongs to A.
  - Inductive step: Assume each of the elements used to define the new element in the inductive step of the definition belongs to A, i.e., $3 \mid x, 3 \mid y$. We have $3 \mid (x + y)$. So the new element also belongs to A.
- Proof completed
- The proof of $S \subseteq A$ is an example of structural induction.
- What is the $P(n)$ used implicitly?

Structural Induction Framework

- The validity of structural induction follows from the principle of mathematical induction.
  - The $P(n)$ used implicitly: The statement holds for any element constructed after n steps of the recursive step.
- Used to show that a statement holds for all elements defined by a recursive definition
- Basis: Show that the statement holds for any element defined in the basis step of the recursive definition
- Recursive step: Show that if the statement holds for each of the elements used to construct new elements in the recursive step of the definition, then the statement holds for the new elements.

Example: Full Binary Trees

- Definitions of the height $h(T)$ and size $n(T)$ of a full binary tree T:
  - If T consists of a single vertex, then
$$h(T) = 0$$
$$n(T) = 1$$
  - If r has two subtrees T_1, T_2, then
$$h(T) = \max\{h(T_1), h(T_2)\} + 1$$
$$n(T) = n(T_1) + n(T_2) + 1$$
- Theorem
If T is a full binary tree, then
$$n(T) \leq 2^{h(T)+1} - 1$$

Proof

- Basis: For a full binary tree consisting of a single vertex, the theorem holds.
- Inductive step:
  - Consider the inductive step in the definition
  - Assume the theorem holds for T_1 and T_2, i.e.,

    $$n(T_1) \leq 2^{h(T_1)+1} - 1$$
    $$n(T_2) \leq 2^{h(T_2)+1} - 1$$

    Want to show that it holds on T.

    $$\begin{aligned} n(T) &= n(T_1) + n(T_2) + 1 \\ &\leq 2^{h(T_1)+1} - 1 + 2^{h(T_2)+1} - 1 + 1 \\ &\leq 2 \cdot \max\left\{2^{h(T_1)+1}, 2^{h(T_2)+1}\right\} - 1 \\ &= 2 \cdot 2^{\max\{h(T_1), h(T_2)\}+1} - 1 \\ &= 2 \cdot 2^{h(T)} - 1 = 2^{h(T)+1} - 1 \end{aligned}$$

More examples on recursive definition and structural induction

- **Definition**
  The set of well-formed formulae in propositional logic is defined as follows:
  - T and F are well-formed formulae;
  - For any propositional variable s, s is a well-formed formula;
  - If E_1 and E_2 are well formed formulae, then $(\neg E_1), (E_1 \wedge E_2), (E_1 \vee E_2), (E_1 \rightarrow E_2), (E_1 \leftrightarrow E_2)$ are well-formed formulae.
- **Examples**
  - $\big((p \vee q) \rightarrow (q \wedge F)\big)$ is a well-formed formula
  - $pq \wedge$ is not a well-formed formula
  - Is $\neg p \wedge q$ a well formed formula?

Example: Structural Induction

- **Theorem**: Any well-formed formula contains matching parentheses.
- Basis:
  Each of T, F and s contains no parentheses, so matching parentheses.
- Inductive step:
  - Assume E_1 and E_2 contain matching parentheses
  - $(\neg E_1)$ also contains matching parentheses
    - The first (and the last) match
    - By the induction hypothesis, every parenthesis in E has a match in E_1
  - The cases with $(\neg E_1), (E_1 \wedge E_2), (E_1 \vee E_2), (E_1 \rightarrow E_2), (E_1 \leftrightarrow E_2)$ are similar

Strings

- **Definition**
  If $\Sigma = \{0, 1\}$, Σ^* is the set of all bit strings defined over the alphabet Σ, i.e., λ, 0,1, 00,01,10, 11, etc.
- **Recursive Definition**
  The set Σ^* of strings over the alphabet Σ:
  - Basis: $\lambda \in \Sigma^*$ (λ is the empty string)
  - Recursive step: If $w \in \Sigma^*, x \in \Sigma$, then $wx \in \Sigma^*$.
- **Example**
  If $\Sigma = \{a, b\}$, show that $aab \in \Sigma^*$
  - Since $\lambda \in \Sigma^*$ and $a \in \Sigma$, $a \in \Sigma^*$.
  - Since $a \in \Sigma^*$ and $a \in \Sigma$, $aa \in \Sigma^*$.
  - Since $aa \in \Sigma^*$ and $b \in \Sigma$, $aab \in \Sigma^*$.

Balanced Parentheses

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

  $\lambda \in P$;

  If $w \in P$, then $(w) \in P$.

  If $w_1 \in P$, $w_2 \in P$, then $w_1 w_2 \in P$.
- Show that (() ()) is in P.
- Why is))(() not in P?

String Concatenation

- **Definition**
  Two strings can be combined via the operation of concatenation. Let Σ be the alphabet and Σ^* be the set of strings over Σ. We can define the concatenation of two strings, denoted by $\cdot$, recursively as follows.
  - Basis: If $w \in \Sigma^*$, then $w \cdot \lambda = w$.
  - Recursive step: If $w_1 \in \Sigma^*,\ w_2 \in \Sigma^*,\ x \in \Sigma$,
    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 $w \in \Sigma^*$ can be recursively defined as:
  $l(\lambda) = 0$
  For any $w \in \Sigma^*, x \in \Sigma$, $l(wx) = l(w) + 1$

Example: Structural induction

- Show that for any $u, v \in \Sigma^*$, $l(uv) = l(u) + l(v)$.
- Proof:
  - Let $P(v)$ be the statement $\forall u\big(l(uv) = l(u) + l(v)\big)$.
  - Basis: $P(\lambda)$ is true, since for any u, $l(u\lambda) = l(u) = l(u) + l(\lambda)$.
  - Inductive step: Assume $P(w)$ is true, i.e .$\forall u(l(uw) = l(u) + l(w))$. Need to show $P(wx)$ is true: For any u, $l(uwx) = l(uw) + 1 = l(u) + l(w) + 1 = l(u) + l(wx)$

Outline

- Recursively Defined Functions
- Other Recursively Definitions
- Structural Induction
- **Recursive Algorithms**

Recursive Algorithms

- Recursively defined functions naturally give recursive algorithms
- Example:
  - $0! = 1$
  - $n! = n(n-1)!$ for $n > 1$
- Some recursively defined functions can be computed without using recursion

```
Factorial($n$):
if $n = 0$ then return $1$
else return Factorial($n - 1$)*$n$
```

```
Factorial($n$):
$s \leftarrow 1$
for $i = 1$ to $n$
    $s \leftarrow s \cdot i$
return $s$
```

Euclid's GCD Algorithm

- Recursive definition: $\gcd(a, b) = \gcd(b, a \bmod b)$

```
gcd($a$,$b$):
if $b = 0$ then return $a$
else return $\gcd(b, a \bmod b)$
```

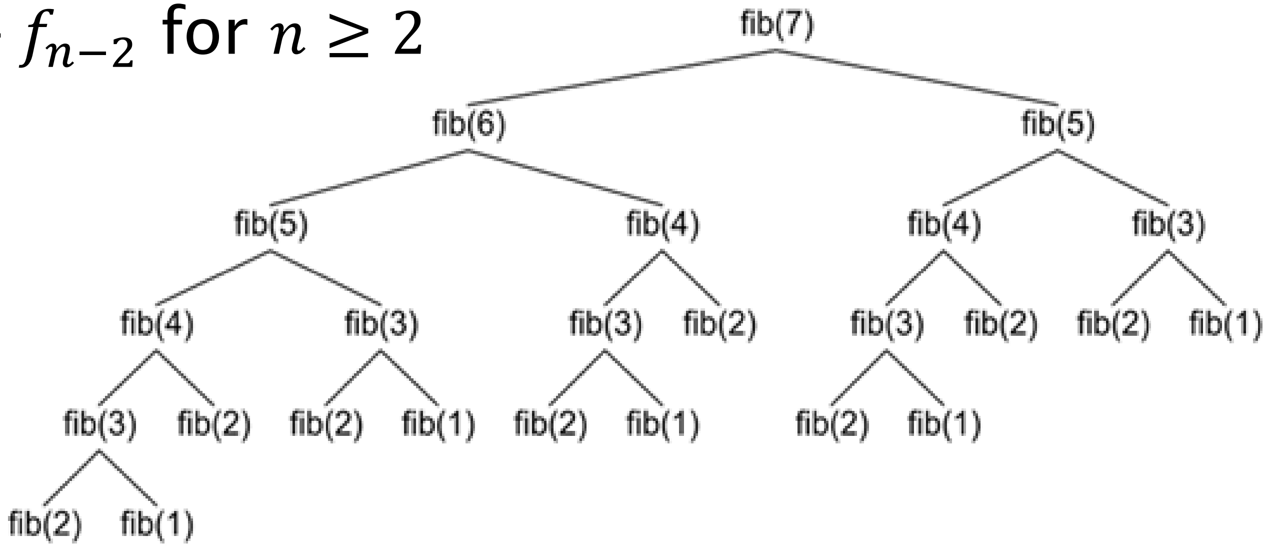
```
gcd($a$,$b$):
$x \leftarrow a$
$y \leftarrow b$
while $y \neq 0$
        $r \leftarrow x \bmod y$
        $x \leftarrow y$
        $y \leftarrow r$
return $x$
```

Example: Fibonacci Numbers

- For some recursively defined functions, using recursion to compute them is very inefficient!

```
Fib($n$):
if $n = 1$ or $n = 2$ then return $1$
else return Fib($n - 1$) + Fib($n - 2$)
```

- $f_1 = 1$
  $f_2 = 1$
  $f_n = f_{n-1} + f_{n-2}$ for $n \geq 2$



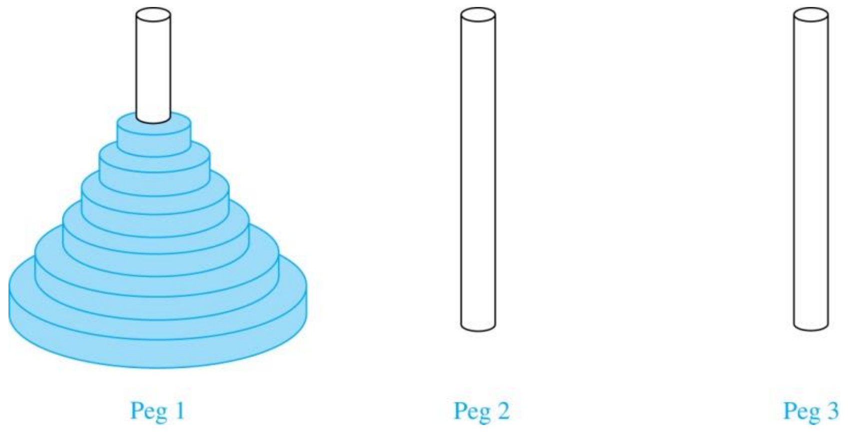
- #calls = size of tree T $\geq f_n \geq \alpha^{n-2}$

Fibonacci Numbers

- More efficient algorithm

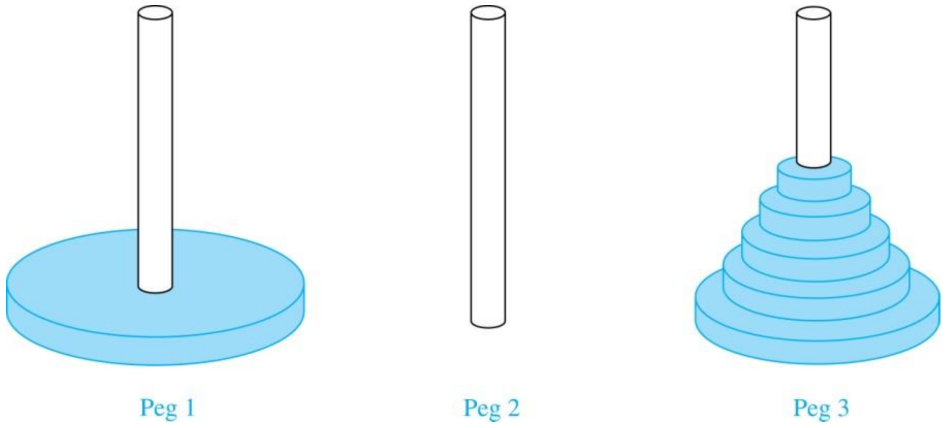
```
Fib($n$):
allocate an array $A$ of size $n$
$A[1] \leftarrow 1$
$A[2] \leftarrow 1$
for $i \leftarrow 3$ to $n$
        $A[i] \leftarrow A[i-1] + A[i-2]$
return $A[n]$
```

The Tower of Hanoi



- Three pegs
  - The first peg has n disks of different sizes
  - The other two legs are empty
- Rule: Move the disks one at a time from one peg to another as long as a larger disk is never placed on a smaller
- Goal: Move all disks to the second peg

The Tower of Hanoi: Solution



- Step 1: Move the top $n - 1$ disks from peg 1 to peg 3
  - This sub-problem is the same as the original problem, but on $n - 1$ disks
- Step 2: Move the largest disk from peg 1 to peg 2
- Step 3: Move the $n - 1$ disks from peg 3 to peg 2
  - Also a sub-problem on $n - 1$ disks

The Tower of Hanoi: Analysis

- How many moves are needed to move n disks?
- The recursive algorithm gives the recurrence
  $f(1) = 1$
  $f(n) = 2f(n-1) + 1$
- Applying our earlier result
  $f(n) = 2^n - 1$
- When $n = 64$, $f(64) > 10^{19}$
  - >500 billon years if one move takes one second
- Why is this algorithm optimal?
- Let $g(n)$ be the # moves of any algorithm
  - $g(1) \geq 1$
  - $g(n) \geq 2g(n-1) + 1$
  - So $g(n) \geq f(n)$

Recursive Algorithms: Summary

- If your recursively algorithm makes one recursive call at the end, it's called a tail recursion and it can be rewritten as an iterative algorithm
  - Good compilers can do this automatically
- If your algorithm makes more than one recursive call, but each call just returns a number, then it can be implemented using dynamic programming
- The running time of a recursive algorithm can be analyzed by solving recurrences
- Much more on this topic in COMP 3711