Recursive definitions

- The sequence of powers of 2 is given by \( a_n = 2^n \) for \( n=0, 1, 2, \ldots \) 
- Can also be defined by \( a_0 = 1 \), and a rule for finding a term of the sequence from the previous one, i.e., \( a_{n+1} = 2a_n \) 
- Can use induction to prove results about the sequence 
- **Structural induction**: We define a set recursively by specifying some initial elements in a basis step and provide a rule for constructing new elements from those already in the recursive step

Recursively defined functions

- Use two steps to define a function with the set of non-negative integers as its domain 
- **Basis step**: specify the value for the function at zero 
- **Recursive step**: give a rule for finding its value at an integer from its values at smaller integers 
- Such a definition is called a recursive or inductive definition
Example

• Suppose f is defined recursively by
  - f(0)=3
  - f(n+1)=2f(n)+3
Find f(1), f(2), f(3), and f(4)
  - f(1)=2f(0)+3=2*3+3=9
  - f(2)=2f(1)+3=2*9+3=21
  - f(3)=2f(2)+3=2*21+3=45
  - f(4)=2f(3)+3=2*45+3=93

Example

• Give an inductive definition of the factorial function f(n)=n!
  • Note that (n+1)=(n+1)·n!
  • We can define f(0)=1 and f(n+1)=(n+1)f(n)
  • To determine a value, e.g., f(5)=5!, we can use the recursive function
    f(5)=5·f(4)=5·4·f(3)=5·4·3·f(2)=5·4·3·2·f(1)
    =5·4·3·2·1·f(0)=5·4·3·2·1·1=120

Recursive functions

• Recursively defined functions are well defined
• For every positive integer, the value of the function is determined in an unambiguous way
• Given any positive integer, we can use the two parts of the definition to find the value of the function at that integer
• We obtain the same value no matter how we apply two parts of the definition

Example

• Given a recursive definition of a^n, where a is a non-zero real number and n is a non-negative integer
  • Note that a^{n+1}=a·a^n and a^0=1
  • These two equations uniquely define a^n for all non-negative integer n
Example

• Given a recursive definition of \( \sum_{k=0}^{n} a_k \)
• The first part of the recursive definition \( \sum_{k=0}^{n} a_k = a_i \)
• The second part is \( \sum_{k=0}^{n} (x_k + a_{k+1}) \)

Example – Fibonacci numbers

• Fibonacci numbers \( f_0, f_1, f_2, \ldots \) are defined by the equations, \( f_0 = 0, f_1 = 1 \), and \( f_n = f_{n-1} + f_{n-2} \) for \( n = 2, 3, 4, \ldots \)
• By definition
  \[
  f_2 = f_1 + f_0 = 1 + 0 = 1 \\
  f_3 = f_2 + f_1 = 1 + 1 = 2 \\
  f_4 = f_3 + f_2 = 2 + 1 = 3 \\
  f_5 = f_4 + f_3 = 3 + 2 = 5 \\
  f_6 = f_5 + f_4 = 5 + 3 = 8 
  \]

Recursively defined sets and structures

• Consider the subset \( S \) of the set of integers defined by
  – Basis step: 3 \( \in S \)
  – Recursive step: if \( x \in S \) and \( y \in S \), then \( x + y \in S \)
• The new elements formed by this are 3 + 3 = 6, 3 + 6 = 9, 6 + 6 = 12, ...
• We will show that \( S \) is the set of all positive multiples of 3 (using structural induction)

String

• The set \( \Sigma^* \) of strings over the alphabet \( \Sigma \) can be defined recursively by
  – Basis step: \( \lambda \in \Sigma^* \) (where \( \lambda \) is the empty string containing no symbols)
  – Recursive step: if \( \omega \in \Sigma^* \) and \( \sigma \in \Sigma \), then \( \omega \sigma \in \Sigma^* \)
• The basis step defines that the empty string belongs to string
• The recursive step states new strings are produced by adding a symbol from \( \Sigma \) to the end of strings in \( \Sigma^* \)
• At each application of the recursive step, strings containing one additional symbol are generated
Example

- If \( \Sigma = \{0, 1\} \), the strings found to be in \( \Sigma^* \), the set of all bit strings, are
- \( \lambda \), specified to be in \( \Sigma^* \) in the basis step
- 0 and 1 found in the 1st recursive step
- 00, 01, 10, and 11 are found in the 2nd recursive step, and so on

Concatenation

- Two strings can be combined via the operation of concatenation
- Let \( \Sigma \) be a set of symbols and \( \Sigma^* \) be the set of strings formed from symbols in \( \Sigma \)
- We can define the concatenation for two strings by recursive steps
  - Basis step: if \( w \in \Sigma^* \), then \( w \cdot \lambda = w \), where \( \lambda \) is the empty string
  - Recursive step: if \( w, w_1, w_2 \in \Sigma^* \) and \( x \in \Sigma \), then \( w_1 \cdot (w_2 \cdot x) = (w_1 \cdot w_2) \cdot x \)
- Oftentimes \( w_1 \cdot w_2 \) is rewritten as \( w_1 w_2 \)
- e.g., \( w_1 = \text{abra} \), and \( w_2 = \text{cadabra} \), \( w_1 w_2 = \text{abracadabra} \)

Length of a string

- Give a recursive definition of \( l(w) \), the length of a string \( w \)
- The length of a string is defined by
  - \( l(\lambda) = 0 \)
  - \( l(wx) = l(w) + 1 \) if \( w \in \Sigma^* \) and \( x \in \Sigma \)

Well-formed formulae

- We can define the set of well-formed formulae for compound statement forms involving \( T, F \), proposition 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
- From an initial application of the recursive step, we know that \( (p \land q), (p \rightarrow F), (F \rightarrow q) \) and \( (q \land F) \) are well-formed formulae
- A second application of the recursive step shows that \( ((p \land q) \land (q \land F)), (q \land (p \land q)), \) and \( ((p \rightarrow F) \rightarrow T) \) are well-formed formulae
Rooted trees

- 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
  - Basis step: a single vertex $r$ is a rooted tree
  - Recursive step: suppose that $T_1, T_2, \ldots, T_n$ are disjoint rooted trees with roots $r_1, r_2, \ldots, r_n$, respectively.
  - Then the graph formed by starting with a root $r$, which is not in any of the rooted trees $T_1, T_2, \ldots, T_n$, and adding an edge from $r$ to each of the vertices $r_1, r_2, \ldots, r_n$, is also a rooted tree

Binary trees

- At each vertex, there are at most two branches (one left subtree and one right subtree)
- Extended binary trees: the left subtree or the right subtree can be empty
- Full binary trees: must have left and right subtrees

Extended binary trees

- The set of extended binary trees can be defined by
  - Basis step: the empty set is an extended binary tree
  - Recursive step: if $T_1$ and $T_2$ are disjoint extended binary trees, there is an extended 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 right subtree $T_2$, when these trees are non-empty
Extended binary trees

Full binary trees

The set of full binary trees can be defined recursively:

- **Basis step**: There is a full binary tree consisting only of 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 right subtree $T_2$.

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

The set of extended binary trees can be defined recursively by these steps:

- **Basis step**: The empty set is an extended binary tree.
- **Recursive step**: If $T_1$ and $T_2$ are disjoint extended binary trees, there is an extended 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 $T_2$ when these trees are nonempty.
Structural induction

- Let $p(n)$ be the statement that $3n$ belongs to $S$
- **Basis step:** it holds as the first part of recursive definition of $S$, $3 \cdot 1 = 3 \in S$
- **Inductive step:** assume that $p(k)$ is true, i.e., $3k$ is in $S$. As $3k \in S$ and $3 \in S$, it follows from the 2nd part of the recursive definition of $S$ that $3k + 3 = 3(k+1) \in S$. So $p(k+1)$ is true

Structural induction

- To show that $S \subseteq A$, we use recursive definition of $S$
- The basis step of the definition specifies that $3 \in S$
- As $3 = 3 \cdot 1$, all elements specified to be in $S$ in this step are divisible by 3, and there in $A$
- To finish the proof, we need to show that all integers in $S$ generated using the 2nd part of the recursive definition are in $A$
- This consists of showing that $x \cdot y$ is in $A$ whenever $x$ and $y$ are elements of $S$ also assumed to be in $A$
- If $x$ and $y$ are both in $A$, it follows that $3 \mid x \cdot y$, and thus $3 \mid x \cdot y$, thereby completing the proof
Trees and structural induction

• To prove properties of trees with structural induction
  – **Basis step**: show that the result is true for the tree consisting of a single vertex
  – **Recursive step**: show that if the result is true for the trees $T_1$ and $T_2$, then it is true for $T_1 \cdot T_2$, consisting of a root $r$, which has $T_1$ as its left subtree and $T_2$ as its right subtree

Height of binary tree

• We define the height $h(T)$ of a full binary tree $T$ recursively
  – **Basis step**: the height of the full binary tree $T$ consisting of only a root $r$ is $h(T)=0$
  – **Recursive step**: if $T_1$ and $T_2$ are full binary trees, then the full binary tree $T= T_1 \cdot T_2$ has height $h(T)=1+\max(h(T_1), h(T_2))$

Number of vertices in a binary tree

• If we let $n(T)$ denote the number of vertices in a full binary tree, we observe that $n(T)$ satisfies the following recursive formula:
  – **Basis step**: the number of vertices $n(T)$ of the full binary tree consisting of only a root $r$ is $n(T)=1$
  – **Recursive step**: if $T_1$ and $T_2$ are full binary trees, then the number of vertices of the full binary tree $T= T_1 \cdot T_2$ is $n(T)=1+n(T_1)+n(T_2)$

Theorem

• If $T$ is a full binary tree $T$, then $n(T)\leq 2^{h(T)+1} - 1$
• Use structural induction to prove this
• **Basis step**: for the full binary tree consisting of just the root $r$ the result is true as $n(T)=1$ and $h(T)=0$, so $n(T)=1\leq 2^{0+1}-1=1$
• **Inductive step**: For the inductive hypothesis we assume that $n(T_i) \leq 2^{h(T_i)+1} - 1$, $n(T_j) \leq 2^{h(T_j)+1} - 1$ where $T_1$ and $T_2$ are full binary trees
Theorem

- By the recursive formulae for \( n(T) \) and \( h(T) \), we have
  \[
  n(T) = 1 + n(T_1) + n(T_2) \quad \text{and} \quad h(T) = 1 + \max(h(T_1), h(T_2))
  \]

\[
\begin{array}{l}
  n(T) = 1 + n(T_1) + n(T_2) \\
  \leq 1 + (2^{h(T_1)+1} - 1) + (2^{h(T_2)+1} - 1) \\
  \leq 2 \cdot \max(2^{h(T_1)+1}, 2^{h(T_2)+1}) - 1 \\
  = 2 \cdot 2^{h(T)+1} - 1 \\
  = 2 \cdot 2^{h(T)} - 1 \\
  = 2^{h(T)+1} - 1.
\end{array}
\]

- This completes the inductive step