Induction and Recursion

Chapter 5

Mathematical Induction

Section 5.1

Video 44: Mathematical Induction

- Principle of Mathematical Induction
- Validity of Mathematical Induction

Principle of Mathematical Induction

Assume you would like to prove that a propositional function P(n) is true for all positive integers.

To prove this, you complete these steps:

- Step 1: Show that $P(1)$ is true (**basis step**)
- Step 2: Assuming that $P(k)$ holds for an arbitrary integer k (**inductive hypothesis**), show that $P(k + 1)$ must be true (**inductive step**)

- It seems perfectly reasonable to assume that then P(n) is true for all positive integers
- This is the principle of **mathematical induction**

Example

Theorem: $n < 2^n$ for all positive integers n.

Proof: Let $P(n)$ be the proposition that $n < 2^n$.

BASIS STEP: $P(1)$ is true since $1 < 2^1 = 2$.

INDUCTIVE STEP: Assume $P(k)$ holds, i.e., $k < 2^k$, for an arbitrary positive integer k.

Show that $P(k + 1)$ holds.

Since by the inductive hypothesis, $k < 2^k$, it follows that:

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

Therefore $n < 2^n$ holds for all positive integers n. ◀

Mathematical Induction as Rule of Inference

Mathematical induction can be expressed as the rule of inference

$$(P(1) \land \forall k\, (P(k) \to P(k+1))) \to \forall n\, P(n)$$

where the domain is the set of positive integers.

Basis Step: Show that P(1) is true

Inductive Step: Show that $P(k) \to P(k+1)$ is true for all positive integers k.

Inference: $\forall n\, P(n)$ is true

Two Important Points

In a proof by mathematical induction, we don't assume that $P(k)$ is true for all positive integers!

- We rather show that if we assume that $P(k)$ is true, then $P(k + 1)$ must also be true.

Proofs by mathematical induction do not always start at the integer 1.

- In such a case, the basis step begins at a starting point b where b is an integer.

Validity of Mathematical Induction

Mathematical induction is valid because of the well-ordering property (an axiom for the set of positive integers):

> Every nonempty subset of the set of positive integers has a least element.

Mathematical induction is actually equivalent to the well-ordering property.

Correctness of Mathematical Induction

Suppose that $P(1)$ is true and $P(k) \rightarrow P(k+1)$ is true for all positive integers k.

- Assume there is at least one positive integer n for which $P(n)$ is false
- Then the set S of positive integers for which $P(n)$ is false is nonempty
- By the well-ordering property, S has a least element, say m
- m cannot be 1 since $P(1)$ is true
- Since m is positive and greater than 1, $m - 1$ must be a positive integer
- Since $m - 1 < m$, it is not in S, so $P(m - 1)$ must be true
- But then, since $P(k) \rightarrow P(k+1)$ for every positive integer k holds, $P(m)$ must also be true
- This contradicts $P(m)$ being false
- Hence, $P(n)$ must be true for every positive integer n ◀

Summary

- Mathematical Induction as a widely used proof method
- It’s Validity derives from the well-ordering axiom of natural numbers

Video 45: Proofs by Mathematical Induction

- Proofs of summation formulas
- Inequalities
- Divisibility Results
- Number of Subsets

Summation Formulas

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

Proof:

- BASIS STEP: *P*(1) is true since 1(1 + 1)/2 = 1.
- INDUCTIVE STEP: Assume the formula is true for *P*(*k*).
- The inductive hypothesis is $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$
- Then

$$\sum_{i=1}^{k+1} i = \sum_{i=1}^{k} i + (\mathrm{k}+1) = \frac{k(k+1)}{2} + (\mathrm{k}+1) =$$

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

◀

Inequalities

Show that $2^n < n!$, for every integer $n \geq 4$.

Proof: Let $P(n)$ be the proposition that $2^n < n!$

- BASIS STEP: $P(4)$ is true since $2^4 = 16 < 4! = 24$.
- INDUCTIVE STEP: Assume $P(k)$ holds, i.e., $2^k < k!$ for an arbitrary integer $k \geq 4$. Then:

$$\begin{aligned} 2^{k+1} &= 2 \cdot 2^k \\ &< 2 \cdot k! \qquad \textit{(by the inductive hypothesis)} \\ &< (k+1)\, k! \\ &= (k+1)! \end{aligned}$$

Therefore, $2^n < n!$ holds, for every integer $n \geq 4$. ◀

Note that here the basis step is $P(4)$, since $P(0)$, $P(1)$, $P(2)$, $P(3)$ are all false.

Divisibility Results

Show that $n^3 - n$ is divisible by 3*,* for every positive integer n.

Proof: Let $P(n)$ be the proposition that $n^3 - n$ is divisible by 3.

- BASIS STEP: $P(1)$ is true since $1^3 - 1 = 0$, which is divisible by 3.
- INDUCTIVE STEP: Assume $P(k)$ holds, i.e., $k^3 - k$ is divisible by 3, for an arbitrary positive integer k. Then:

$$\begin{aligned}(k + 1)^3 - (k + 1) &= (k^3 + 3k^2 + 3k + 1) - (k + 1)\\ &= (k^3 - k) + 3(k^2 + k)\end{aligned}$$

By the inductive hypothesis, the first term $(k^3 - k)$ is divisible by 3 and the second term is divisible by 3 since it is an integer multiplied by 3.
The sum of two numbers divisible by 3 is divisible by 3.

Therefore, $n^3 - n$ is divisible by 3*,* for every integer positive integer n. ◀

Number of Subsets of a Finite Set

Theorem: If *S* is a finite set with n elements, where *n* is a nonnegative integer, then *S* has 2^n subsets.

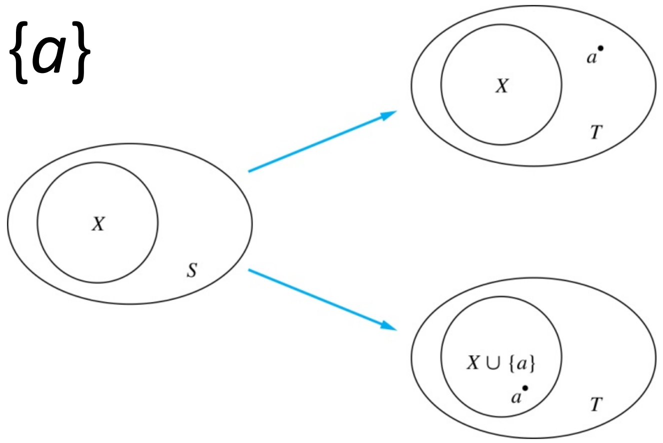
Proof: Let *P*(*n*) be the proposition that a set with *n* elements has 2^n subsets.

- BASIS STEP : *P*(0) is true, because the empty set has only itself as a subset and $2^0 = 1$.
- INDUCTIVE STEP: Assume *P*(*k*) is true for an arbitrary nonnegative integer *k*.

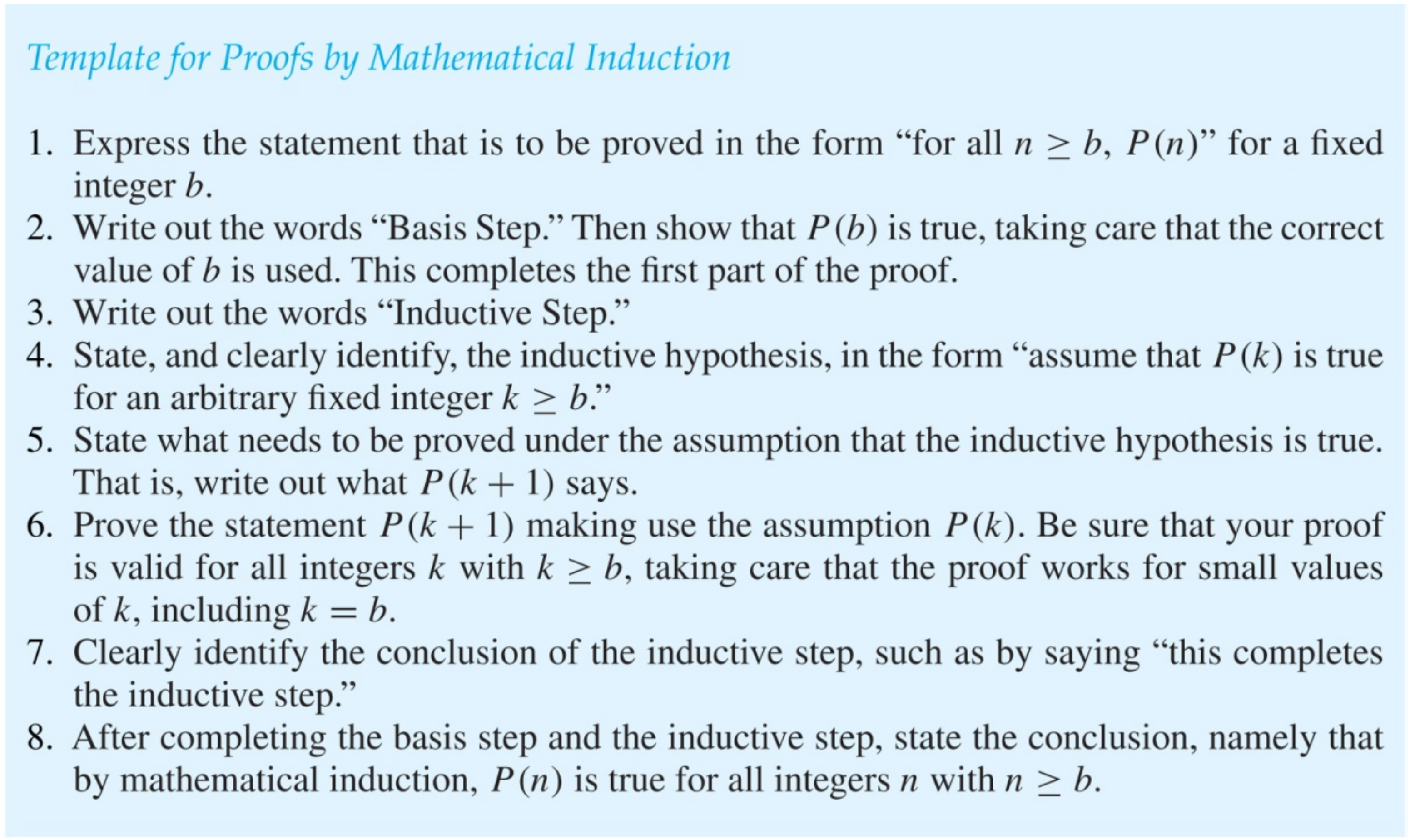
Number of Subsets – Inductive Step

Inductive Hypothesis: For an arbitrary nonnegative integer k, every set with k elements has 2^k subsets.

- Let T be a set with $k + 1$ elements.
- Then, for some a, $T = S \cup \{a\}$, where $a \in T$ and $S = T - \{a\}$
- Hence $|S| = k$.
- For each subset X of S, there are exactly two subsets of T, i.e., X and $X \cup \{a\}$.
- By the inductive hypothesis S has 2^k subsets.
- Since there are two subsets of T for each subset of S, the number of subsets of T is $2 \cdot 2^k = 2^{k+1}$.



Mathematical Induction Proofs Guideline



Template for Proofs by Mathematical Induction

1. Express the statement that is to be proved in the form “for all $n \geq b$, $P(n)$” for a fixed integer b.
2. Write out the words “Basis Step.” Then show that $P(b)$ is true, taking care that the correct value of b is used. This completes the first part of the proof.
3. Write out the words “Inductive Step.”
4. State, and clearly identify, the inductive hypothesis, in the form “assume that $P(k)$ is true for an arbitrary fixed integer $k \geq b$.”
5. State what needs to be proved under the assumption that the inductive hypothesis is true. That is, write out what $P(k + 1)$ says.
6. Prove the statement $P(k + 1)$ making use the assumption $P(k)$. Be sure that your proof is valid for all integers k with $k \geq b$, taking care that the proof works for small values of k, including $k = b$.
7. Clearly identify the conclusion of the inductive step, such as by saying “this completes the inductive step.”
8. After completing the basis step and the inductive step, state the conclusion, namely that by mathematical induction, $P(n)$ is true for all integers n with $n \geq b$.

Summary

- Proofs of summation formulas
- Inequalities
- Divisibility Results
- Number of Subsets

Strong Induction and Well-Ordering

Section 5.2

Video 46: Strong Induction

- Principle of Strong Induction
- Examples of Strong Induction

Strong Induction

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

Basis Step: Show that P(1) is true

Inductive Step: Show that $\forall k\ ([P(1) \land P(2) \land \cdots \land P(k)] \rightarrow P(k + 1))$ is true for all positive integers k.

- Strong Induction is sometimes called the *second principle of mathematical induction* or *complete induction*.

Properties of Strong Induction

- We can always use strong induction instead of mathematical induction.
- But there is no reason to use it if it is simpler to use mathematical induction.
- In fact, the principles of mathematical induction, strong induction, and the well-ordering property are all equivalent.
- Sometimes it is clear how to proceed using one of the three methods, but not the other two.

Example of Strong Induction

Theorem: Every positive integer n can be written as a sum of distinct powers of two, that is, there exists a set of integers S = {k_1,…,k_m} such that $n = \sum_{j=1}^{m} 2^{k_j}$.

Proof:

BASIS STEP: for n = 1 chose the set S = {0}, such that 1 = 2^0

INDUCTIVE STEP: Assume *P*(*k*) is true for 1,…,*k.* Proof by cases

If k+1 is odd, then k is even. There exists a set S such that $\mathrm{k} = \sum_{j=1}^{m} 2^{k_j}$. Since k is even, $0 \notin S$, otherwise the sum is odd. Therefore we can add 0 to S and $\mathrm{k} + 1 = 2^0 + \sum_{j=1}^{m} 2^{k_j}$.

If k+1 is even, then k+1/2 is an integer. There exists a set S such that $\frac{k+1}{2} = \sum_{j=1}^{m} 2^{k_j}$ which implies $\mathrm{k} + 1 = \sum_{j=1}^{m} 2^{k_j+1}$.

Therefore the set {k_1+1,…,k_m+1} allows to write k+1 as sum of distinct powers of two.

◀

Proof using Strong Induction

Prove that every amount of postage of 12 cents or more can be formed using just 4-cent and 5-cent stamps.

Proof: Let $P(n)$ be the proposition that postage of n cents can be formed using 4-cent and 5-cent stamps.

- BASIS STEP: $P(12)$, $P(13)$, $P(14)$, and $P(15)$ hold.
  - $P(12)$ uses three 4-cent stamps.
  - $P(13)$ uses two 4-cent stamps and one 5-cent stamp.
  - $P(14)$ uses one 4-cent stamp and two 5-cent stamps.
  - $P(15)$ uses three 5-cent stamps.
- INDUCTIVE STEP: The inductive hypothesis states that $P(j)$ holds for $12 \leq j \leq k$, where $k \geq 15$. Assuming the inductive hypothesis, it can be shown that $P(k + 1)$ holds.
- Using the inductive hypothesis, $P(k - 3)$ holds since $k - 3 \geq 12$. To form postage of $k + 1$ cents, add a 4-cent stamp to the postage for $k - 3$ cents.

Hence, $P(n)$ holds for all $n \geq 12$. ◀

Summary

- Principle of Strong Induction
- Proofs can be sometimes simpler with strong induction

Recursive Definitions and Structural Induction

Section 5.3

Video 47: Recursively Defined Functions

- Recursively Defined Functions
- Fibonacci Numbers

Recursively Defined Functions

Definition: A **recursive** or **inductive definition** of a function *f* nonegative integers as domain 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 a sequence a_0, a_1, … , where $f(i) = a_i$.

Example

Suppose f is defined by

$f(0) = 3,$

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

Then

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

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

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

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

Example

Recursive definition of the factorial function *n*!

$f(0) = 1$

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

Give a recursive definition of $\displaystyle\sum_{k=0}^{n} a_k.$

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

$$\sum_{k=0}^{n+1} a_k = \left(\sum_{k=0}^{n} a_k\right) + a_{n+1}.$$

Fibonacci Numbers

The Fibonacci numbers are defined as follows:

$f_0 = 0$

$f_1 = 1$

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

Then

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

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

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

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

Property of Fibonacci Numbers

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

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

We 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 $\alpha^2 = (3 + \surd 5)/2 < 3 = f_4$.

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 $\alpha^2 = \alpha + 1$,

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

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

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

Therefore, it follows that

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

Hence, $P(k + 1)$ is true.

Summary

- Recursively Defined Functions
- Fibonacci Numbers
- Proving properties of Recursively Defined Functions using induction

Video 48: Recursively Defined Sets and Structures

- Recursive definitions of sets
- Natural Numbers and Strings
- Recursively defined functions
- Well-formed Formulae

Recursively Defined Sets and Structures

Recursion can be also used to define sets

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.

- Only elements generated in the basis and recursive steps belong to the set (**exclusion rule**)

Examples

Recursive definition of the set of 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.

A subset of Integers *S3*:

BASIS STEP: 3 ∈ S3.

RECURSIVE STEP: If *x* ∈ *S3* and *y* ∈ *S3*, then *x + y* is in *S3.*

- Initially 3 is in *S3*, then 3 + 3 = 6, then 3 + 6 = 9, etc.

Strings

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

BASIS STEP: $\lambda \in \Sigma^*$ (λ is the empty string)

RECURSIVE STEP: If *w* is in Σ^* and *x* is in Σ, then $wx \in \Sigma^*$.

The alphabet Σ is a finite set, e.g.

- $\Sigma = \{0, 1\}$
- $\Sigma = \{a, b, \ldots., z\}$

Examples

If $\Sigma = \{0, 1\}$, the strings in in Σ^* are the set of all bit strings:

$\lambda, 0, 1, 00, 01, 10, 11, \ldots$

If $\Sigma = \{a, b\}$, showing that aab is in Σ^*:

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 $b \in \Sigma$, $aab \in \Sigma^*$.

Recursively defined functions on recursively defined sets

We can define functions by recursion on recursively defined sets

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

The length of a string can be recursively defined by:

$l(\lambda) = 0$

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

String Concatenation

Definition: The **concatenation** of two strings w_1 and w_2, denoted by $w_1 \cdot w_2$, is defined 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 $w_1 w_2$.

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

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 $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ is recursively defined as

- 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 $(\neg E)$, $(E \wedge F)$, $(E \vee F)$, $(E \rightarrow F)$, $(E \leftrightarrow F)$, are well-formed formulae.

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

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

$p \wedge q \wedge q$ is not a well formed formula.

Summary

- Recursive definitions of sets
- Natural Numbers and Strings
- Recursively defined functions
- Well-formed Formulae

Video 49: Structural Induction

- Principle of structural induction
- Examples

Structural Induction

To prove a property of the elements of a recursively defined set, we use **structural induction**.

BASIS STEP: Show that the result holds for all elements specified in the basis step of the recursive definition.

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.

- The validity of structural induction can be shown to follow from the principle of mathematical induction.

Example

Theorem: $l(xy) = l(x) + l(y)$, where x and y belong to Σ^*, the set of strings over the alphabet Σ.

Proof: Let $P(y)$ be the statement that $l(xy) = l(x) + l(y)$

BASIS STEP: We show that $P(\lambda)$ is true. We have that $l(x\lambda) = l(x) + l(\lambda)$ for all $x \in \Sigma$, because $l(x\lambda) = l(x) = l(x) + 0 = l(x) + l(\lambda)$ for $x \in \Sigma^*$

RECURSIVE STEP: We assume that $P(y)$ is true and show that this implies that $P(ya)$ is true whenever $a \in \Sigma$.

We need to show $l(xya) = l(x) + l(ya)$ for every $a \in \Sigma$.

We have $l(xya) = l(xy) + 1$ and $l(ya) = l(y) + 1$ (definition of l).

Since the inductive hypothesis is $l(xy) = l(x) + l(y)$ we conclude that $l(xya) = l(x) + l(y) + 1 = l(x) + l(ya)$ ◀

Example

Theorem: Every well-formed formula for compound propositions contains an equal number of left and right parentheses.

Proof:

BASIS STEP*:* Each of the formula **T**, **F**, and *s* contains no parentheses, so they contain an equal number of left and right parentheses.

RECURSIVE STEP*:* Assume *p* and *q* are well-formed formulae, each containing an equal number of left and right parentheses.

For the two propositions *p* and *q,* let l_p and l_q be the number of left parenthesis and r_p and r_q the number of right parenthesis. The inductive hypothesis is that $l_p = r_p$ and $l_q = r_q$

We need to show that each of $(\neg p)$, $(p \vee q)$, $(p \wedge q)$, $(p \rightarrow q)$, and $(p \leftrightarrow q)$ also contains an equal number of left and right parentheses.

The number of left parentheses in $(\neg p)$ equals $l_p + 1$ and the number of right parentheses $r_p + 1$.

For the other compound propositions the number of left parentheses equals $l_p + l_q + 1$ and the number of right parentheses equals $r_p + r_q + 1$

Since $l_p = r_p$ and $l_q = r_q$ in all cases the these compound expressions contains the same number of left and right parentheses.

This completes the proof by structural induction. ◀

Summary

- Principle of structural induction
- Structural induction on strings
- Structural induction on well-formed formulae

Recursive Algorithms

Section 5.4

Video 50: Recursive Algorithms

- Definition of recursive algorithms
- Examples of recursive algorithms

Recursive Algorithms

Definition: An algorithm is called **recursive** if it solves a problem by reducing it to an instance of the same problem with smaller input.

For the algorithm to terminate, the instance of the problem must eventually be reduced to some initial case for which the solution is known.

Recursive Factorial Algorithm

A recursive algorithm for computing *n*!, where *n* is a nonnegative integer.

```
factorial(n) :=
    if  n = 0
    then return 1
    else return $n \cdot factorial(n - 1)$
```

Recursive Computation of Factorial

factorial(4) =
*4*factorial(3) =*
4(3*factorial(2)) =*
4(3*(2*factorial(1))) =*
4(3*(2*(1*factorial(0)))) =*
4(3*(2*(1*1))) =*
4(3*(2*1))) =*
4(3*2)) =*
*4*6 =*
24

Recursive Exponentiation Algorithm

A recursive algorithm for computing a^n, where a is a nonzero real number and n is a nonnegative integer.

```
power(a, n) :=
        if n <= 0
        then return 1
        else return $a \cdot power(a, n-1)$
```

This is a bad algorithm!

Recursive Computation of Exponentiation

power(a, 6) =
*a * power(a, 5) =*
*a * (a * power(a, 4)) =*
*a * (a * (a * power(a, 3))) =*
*a * (a * (a * (a * power(a, 2)))) =*
*a * (a * (a * (a * (a * power(a, 1))))) =*
*a * (a * (a * (a * (a * (a* power(a, 0)))))) =*
*a * (a * (a * (a * (a * (a* 1))))) =*
*a * (a * (a * (a * (a * a)))) =*
*a * (a * (a * (a * a^2))) =*
*a * (a * (a * a^3)) =*
*a * (a * a^4) =*
*a * a^5 =*
a^6

Better Recursive Exponentation

```
$fast\_power(a, n)$ :=
    if $n \leq 0$
    then 1
    else $a^{n\&1} \cdot (fast\_powerl(a, \lfloor n/2 \rfloor))^2$
```

$\lfloor n/2 \rfloor$ is the Integer part of n/2

n&1 is 0 if n is even and 1 if n is odd

Recursive Computation of Power

fast_power(a, 6) =

a^0 * *fast_power(a, 3)*^{2} =

a^0 * (a^1 * *fast_power(a, 1)*^{2})^2 =

a^0 * (a^1 * (a^1 * *fast_power(a, 0)*^{2})^2)^2 =

$a^0 * (a^1 * (a)^2)^2 =$

$a^0 * (a^3)^2 =$

a^6

Recursive Linear Search

```
procedure recursive_linear_search($i$, $j$, $x$: integers, $1 \le i \le j \le n$)
        if $a_i = x$ then return $i$
        else    if $i = j$ then return 0
                else return recursive_linear_search($i + 1$, $j$, $x$)
```

```
procedure linear search($x$: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \le n$ and $x \ne a_i$)
   $i := i + 1$
   if $i \le n$ then location := $i$ else location := 0
return location
```

Recursive Binary Search Algorithm

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.

```
recursive_binary_search(i, j, x ) :=
        $m := \lfloor (i + j)/2 \rfloor$
        if  $x = a_m$ then return $m$
        else  if  ($x < a_m$ and $i < m$) then return recursive_binary_search($i$, $m-1$, $x$)
                else  if  ($x > a_m$ and  $j > m$) then return recursive_binary_search($m+1$, j, $x$)
                        else return 0
```

Summary

- Recursive algorithms for
  - Factorial function
  - Exponentation
  - Search

Video 51: Recursion, Induction and Iteration

- Recursion and induction
- Recursion and iteration

Recursion and Induction

Induction and recursion are different approaches to proving results and solving problems

- They have in common that they both in the first place rely on the ability to achieve the desired result for the smallest possible version of the problem at hand
- Induction **extends** this ability to problems of any size
- Recursion **reduces** a problem of any size to the smallest possible ones

Induction can be used to prove correctness of recursive algorithms

Proving Recursive Algorithms Correct

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

```
power(a, n) :=
        if n <= 0  then return 1 else return a · power(a, n-1)
```

Proof: 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

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

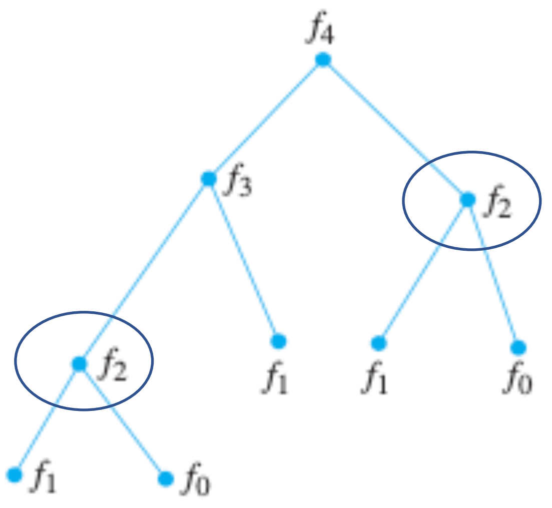
Recursion and Iteration

A recursively defined function can be evaluated in two different ways

- **Recursively**: for a value apply directly the recursive definition, till a base case is reached, a recursive algorithm
- **Iteratively**: start with base cases, and apply the recursive definition to compute the function for larger values

Fibonacci Function

```
fibonacci(n) :=
        if n ≤ 1 then return n
        else return fibonacci(n-1) + fibonacci(n-2)
```



Note that f_2 is computed multiple times
Blindy using recursion can be inefficient!

Iterative Algorithm

```
iterative_fibonacci(n) :=
    if n = 0 then return 0
    else
        previous := 0
        current := 1
        for i = 1 to n - 1
            next := previous  + current
            previous := current
            current := next
        return current
```

Complexity is $\Theta(n)$

Iterative computation

iterative_fibonacci(4)

previous = 0, current = 1

i = 1: next = 1, previous = 1, current = 1

i = 2: next = 2, previous = 1, current = 2

i = 3: next = 3, previous = 2, current = 3

Keep memory of last two Fibonacci numbers

Summary

- Proving correctness of recursive algorithms using induction
- Iterative algorithm for Fibonacci numbers

Video 52: Recursive Sorting

- Merge Sort
- Complexity of Merge Sort

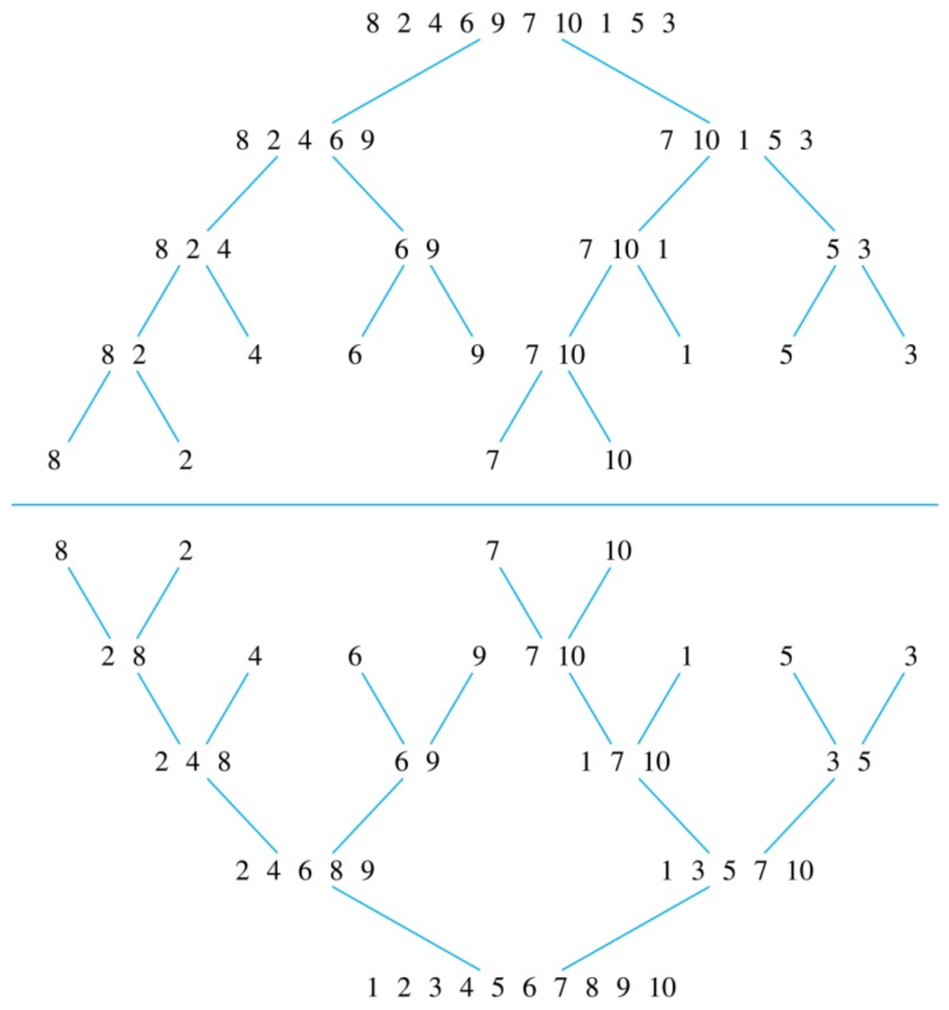
Recursive Sorting

Sorting algorithms, like Bubble Sort, had complexity $\Theta(n^2)$

Merge Sort is a recursive sorting algorithm that performs significantly better

- Merge Sort works by iteratively splitting a list into two sublists of equal length until each sublist has one element.
- 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.

Illustration Merge Sort



List to be sorted

Splitting into two sublists

Splitting the sublist into sublists

Splitting the sublists into sublists

All list of length 1

Merging the lists in the right order

Merging the merged lists in the right order

Merging the merged lists in the right order

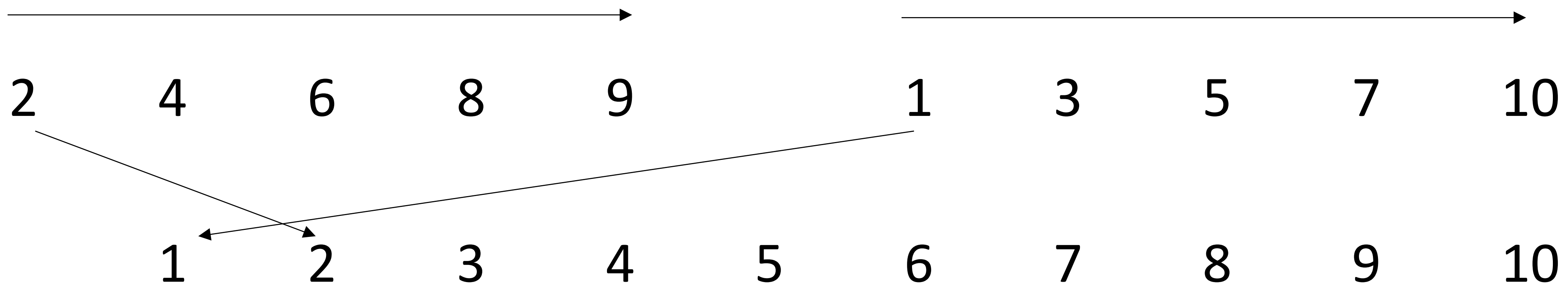
Merging 2 lists in the right order

Resulting ist is sorted

Illustration Merging Two Lists

Example: merging the two sorted lists

Traverse the two lists in parallel from left to right



Always take the smaller element at the left of the two lists

Comparisons for Mergin Two Lists

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

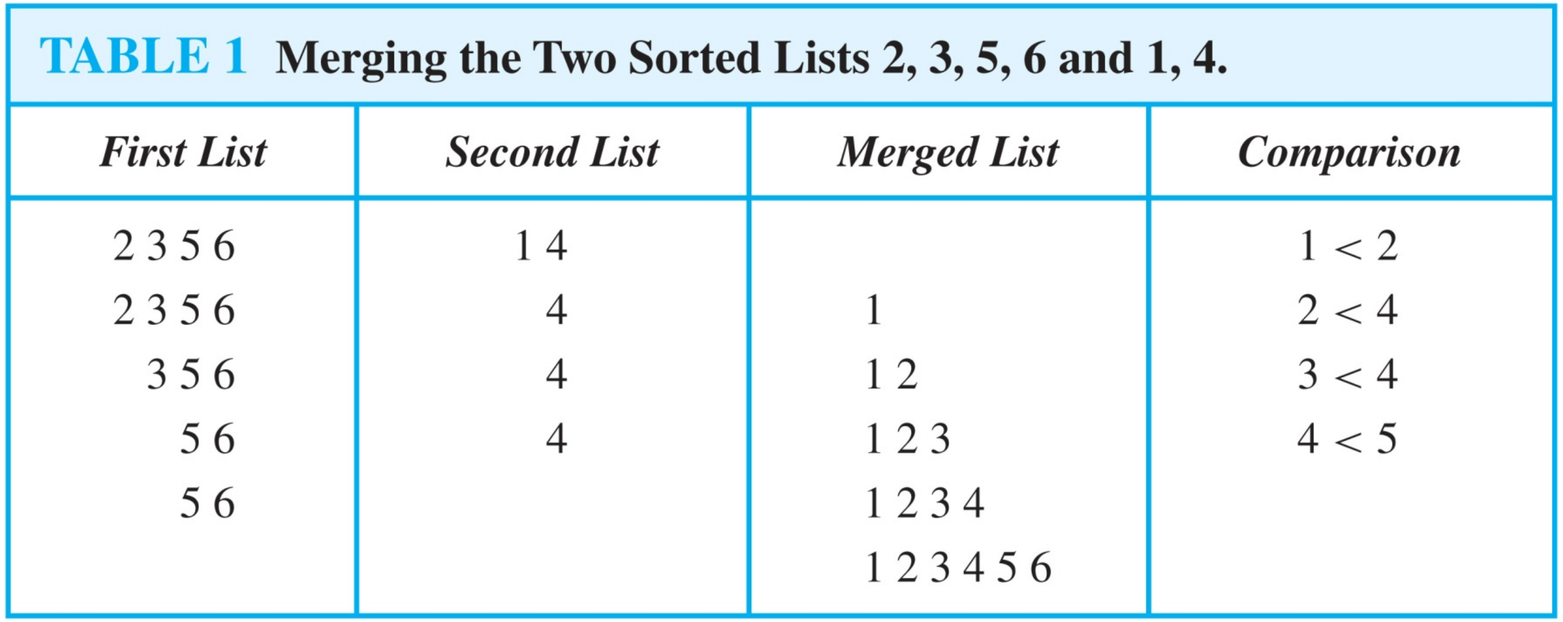


TABLE 1 **Merging the Two Sorted Lists 2, 3, 5, 6 and 1, 4.**

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	

Algorithm for Merging 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 it at the end of $L$
    if this removal makes one list empty
    then remove all elements from the other list and append them to L
return $L$
```

Complexity of Merge: at most $|L_1| + |L_2| - 1$ comparisons

Recursive Merge Sort

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

When mergesort terminates *L* is sorted into elements in increasing order

Complexity of Merge Sort

For simplicity, assume that *n* is a power of 2, say 2^m.

- At each invocation of procedure *mergesort*
  - The number of lists doubles and the length of the lists half
- After *m* invocations there are $n = 2^m$ lists of length 1.
- The merge procedure is now executed *m* times; at each execution
  - The length of lists doubles and their number halves
  - The cost of the merge procedure is the sum of the length of the lists minus 1
- The total cost is thus at most

$$(2^0 * (2^m - 1)) + (2^1 * (2^{m-1} - 1)) + \ldots + (2^{m-1} * (2^1 - 1)) = \sum_{k=1}^{m} \underbrace{\boxed{2^{k-1}}}_{\text{Number of merges}} \underbrace{\boxed{(2^{m-k+1} - 1)}}_{\text{Cost of merge}}$$

Complexity of Merge Sort

Using

$$\sum_{k=1}^{m} 2^{k-1} = 2^m - 1$$

we obtain

$$\sum_{k=1}^{m} 2^{k-1}(2^{m-k+1} - 1) = \sum_{k=1}^{m} 2^m - \sum_{k=1}^{m} 2^{k-1} = m2^m - (2^m - 1) = n \log n - n + 1,$$

Therefore the algorithm has complexity O(n log n), i.e. linearithmic complexity, which is the best complexity that can be obtained for sorting.

Summary

- Merge Sort is a recursive sorting algorithm
- Complexity of Merge Sort is linearithmic