Functions continued

(still section 2.3)

Properties of Functions

A function $f:A\to B$ with $A,B \subseteq R$ is called *strictly increasing*, if

$\forall x,y\in A\ (x < y \to f(x) < f(y))$,

and *strictly decreasing*, if

$\forall x,y\in A\ (x < y \to f(x) > f(y))$.

Obviously, a function that is either strictly increasing or strictly decreasing is one-to-one.

Properties of Functions

A function $f:A\to B$ is called *onto*, or *surjective*, if and only if for every element $b\in B$ there is an element $a\in A$ with $f(a) = b$.

In other words, f is onto if and only if its range is its entire codomain.

A function $f: A\to B$ is a *one-to-one correspondence*, or a *bijection*, if and only if it is both one-to-one and onto.

Obviously, if f is a bijection and A and B are finite sets, then $|A| = |B|$.

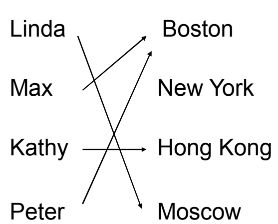
Properties of Functions

Examples:

In the following examples, we use the arrow representation to illustrate functions $f:A\to B$.

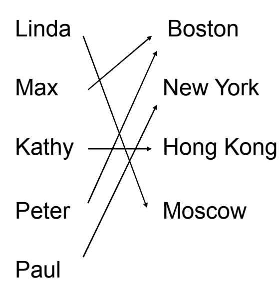
In each example, the complete sets A and B are shown.

Properties of Functions



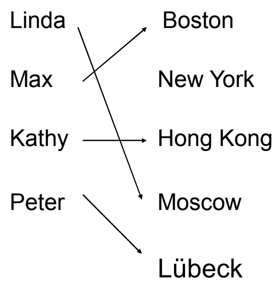
Is f injective?
No.
Is f surjective?
No.
Is f bijective?
No.

Properties of Functions



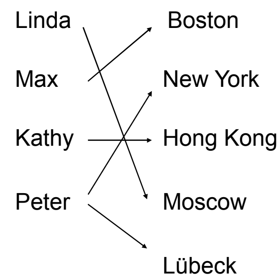
Is f injective?
No.
Is f surjective?
Yes.
Is f bijective?
No.

Properties of Functions



Is f injective?
Yes.
Is f surjective?
No.
Is f bijective?
No.

Properties of Functions



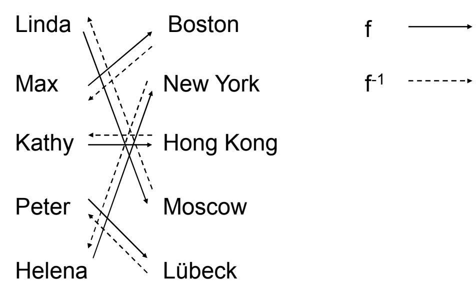
Is f injective?
No! f is not even a function!

Inversion

An interesting property of bijections is that they have an *inverse function*.

The inverse function of the bijection $f:A\rightarrow B$ is the function $f^{-1}:B\rightarrow A$ with
$f^{-1}(b) = a$ whenever $f(a) = b$.

Inversion



Inversion

Example:

f(Linda) = Moscow
f(Max) = Boston
f(Kathy) = Hong Kong
f(Peter) = Lübeck
f(Helena) = New York

Clearly, f is bijective.

The inverse function f^{-1} is given by:

f^{-1}(Moscow) = Linda
f^{-1}(Boston) = Max
f^{-1}(Hong Kong) = Kathy
f^{-1}(Lübeck) = Peter
f^{-1}(New York) = Helena

Inversion is only possible for bijections (= invertible functions)

Composition

The **composition** of two functions $g:A\rightarrow B$ and $f:B\rightarrow C$, denoted by $f\circ g$, is defined by

$(f\circ g)(a) = f(g(a))$

This means that

- **first**, function g is applied to element $a\in A$, mapping it onto an element of B,
- **then**, function f is applied to this element of B, mapping it onto an element of C.
- **Therefore**, the composite function maps from A to C.

Composition

Example:

$f(x) = 7x – 4$, $g(x) = 3x$,
$f:\mathbf{R}\to\mathbf{R}$, $g:\mathbf{R}\to\mathbf{R}$

$(f\circ g)(5) = f(g(5)) = f(15) = 105 – 4 = 101$

$(f\circ g)(x) = f(g(x)) = f(3x) = 21x - 4$

Composition

Composition of a function and its inverse:

$(f^{-1}\circ f)(x) = f^{-1}(f(x)) = x$

The composition of a function and its inverse is the ***identity function*** $i(x) = x$.

Graphs

The ***graph*** of a function $f:A\to B$ is the set of ordered pairs $\{(a, b) \mid a\in A \text{ and } f(a) = b\}$.

The graph is a subset of $A\times B$ that can be used to visualize f in a two-dimensional coordinate system.

Floor and Ceiling Functions

The **floor** and **ceiling (or roof)** functions map the real numbers onto the integers ($\mathbf{R}\to\mathbf{Z}$).

The ***floor*** function assigns to $r\in\mathbf{R}$ the largest $z\in\mathbf{Z}$ with $z\le r$, denoted by $\lfloor r\rfloor$.

Examples: $\lfloor 2.3\rfloor = 2$, $\lfloor 2\rfloor = 2$, $\lfloor 0.5\rfloor = 0$, $\lfloor -3.5\rfloor = -4$

The ***ceiling*** function assigns to $r\in\mathbf{R}$ the smallest $z\in\mathbf{Z}$ with $z\ge r$, denoted by $\lceil r\rceil$.

Examples: $\lceil 2.3\rceil = 3$, $\lceil 2\rceil = 2$, $\lceil 0.5\rceil = 1$, $\lceil -3.5\rceil = -3$

Partial Functions

A *partial function* f from A to B
is a function
$f:C\to B$
where C is a subset of A.

Exercises

I recommend Exercises 1, 5, and 17 in Section 2.3.

It may also be useful to study the graph displays in that section.

Another question: What do all graph displays for any function $f:\mathbf{R}\to\mathbf{R}$ have in common?

… and now for…

Sequences

(Section 2.4)

Sequences

Sequences represent **ordered lists** of elements.

A ***sequence*** is defined as a function from a subset of **N** to a set S. We use the notation a_n to denote the image of the integer n. We call a_n a term of the sequence.

Example:

subset of **N**:	1	2	3	4	5	…
	↓	↓	↓	↓	↓	
S:	2	4	6	8	10	…

Sequences

We use the notation $\{a_n\}$ to describe a sequence.

Important: Do not confuse this with the {} used in set notation.

It is convenient to describe a sequence with a **formula**.

For example, the sequence on the previous slide can be specified as $\{a_n\}$, where $a_n = 2n$.

The Formula Game

What are the formulas that describe the following sequences $a_1, a_2, a_3, \ldots$?

1, 3, 5, 7, 9, …	$a_n = 2n - 1$
-1, 1, -1, 1, -1, …	$a_n = (-1)^n$
2, 5, 10, 17, 26, …	$a_n = n^2 + 1$
0.25, 0.5, 0.75, 1, 1.25 …	$a_n = 0.25n$
3, 9, 27, 81, 243, …	$a_n = 3^n$

Strings

A ***String*** can be thought of as a finite sequence of characters, denoted by $a_1a_2a_3\ldots a_n$.

The ***length*** of a string S is the number of terms that it consists of.

The ***empty string*** contains no terms at all. It has length zero.

Summations

What does $\sum_{j=m}^{n} a_j$ stand for?

It represents the sum $a_m + a_{m+1} + a_{m+2} + \ldots + a_n$.

The variable j is called the ***index of summation***, running from its ***lower limit*** m to its ***upper limit*** n. We could as well have used any other letter to denote this index.

Summations

How can we express the sum of the first 1000 terms of the sequence $\{a_n\}$ with $a_n=n^2$ for $n = 1, 2, 3, \ldots$?

We write it as $\sum_{j=1}^{1000} j^2$.

What is the value of $\sum_{j=1}^{6} j$?

It is 1 + 2 + 3 + 4 + 5 + 6 = 21.

What is the value of $\sum_{j=1}^{100} j$?

It is so much work to calculate this…

Summations

It is said that Friedrich Gauss came up with the following formula:

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

When you have such a formula, the result of any summation can be calculated much more easily, for example:

$$\sum_{j=1}^{100} j = \frac{100(100+1)}{2} = \frac{10100}{2} = 5050$$

Double Summations

Corresponding to nested loops in C or Java, there is also double (or triple etc.) summation:

Example:

$$\sum_{i=1}^{5}\sum_{j=1}^{2} ij$$
$$= \sum_{i=1}^{5}(i+2i)$$
$$= \sum_{i=1}^{5} 3i$$
$$= 3+6+9+12+15 = 45$$

Double Summations

Table 2 in Section 2.4 contains some very useful formulas for calculating sums.

Exercises 31 and 33 are nice exercises, and there are many others as well.

Cardinality (section 2.5)

A set S of n elements has *cardinality* n, |S| = n.

An set is *countable* if it is finite or is in 1-1 correspondence with the natural numbers.

A set that isn’t countable is *uncountable*

Countable and Uncountable sets.

The set of Natural numbers is countable.

The set of integers is countable.

The set of rational numbers is countable.

The set of real numbers is uncountable.

Countable Sets

Theorem. A set S is countable iff its elements can be counted in a finite or infinite sequence. ($S = \{a_1, a_2, a_3, \ldots\}$)

Theorem: Any subset of a countable set is countable.

Theorem. If A and B are countable then $A \cup B$ is countable.

Theorem: A countable union of countable sets is countable.

Countability

Theorem (Cantor): the set of real numbers is uncountable.

Proof: Cantor's diagonalization process shows that no sequence can list every point in the unit interval [0, 1].

See example 5, page 173, for this proof.

Enough Mathematical Appetizers!

Let us look at something more interesting for CS:

Algorithms

(sections 3.1, 3.2)

Algorithms

What is an algorithm?

An *algorithm* is a finite set of precise instructions for performing a computation or for solving a problem.

This is a rather vague definition. You will get to know a more precise and mathematically useful definition when you attend CS420 or CS620.

But this one is good enough for now…

Algorithms

Properties of algorithms:

- **Input** from a specified set,
- **Output** to a specified set (solution),
- **Definiteness** of every step in the computation,
- **Correctness** of output for every possible input,
- **Finiteness** of the number of calculation steps,
- **Effectiveness** of each calculation step and
- **Generality** for a class of problems.

Algorithm Examples

We will use a pseudocode to specify algorithms, which slightly reminds us of Basic and Pascal.

Example: an algorithm that finds the maximum element in a finite sequence

procedure max($a_1, a_2, \ldots, a_n$: integers)
max := a_1
for i := 2 **to** n
    if max < a_i **then** max := a_i
{max is the largest element}

Algorithm Examples

Another example: a linear search algorithm, that is, an algorithm that linearly searches a sequence for a particular element.

procedure linear_search(x: integer; $a_1, a_2, \ldots, a_n$: integers)
i := 1
while (i ≤ n and x ≠ a_i)
    i := i + 1
if i ≤ n **then** location := i
else location := 0
{location is the subscript of the first term that equals x, or is zero if x is not found}

Algorithm Examples

If the terms in a sequence are ordered, a binary search algorithm is more efficient than linear search.

The binary search algorithm iteratively restricts the relevant search interval until it closes in on the position of the element to be located.

Algorithm Examples

binary search for the letter 'j'

search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'

search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'

search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'

search interval

a c d f g h j l m o p r s u v x z

center element

Algorithm Examples

binary search for the letter 'j'

search interval

a c d f g h j l m o p r s u v x z

center element

found !

Algorithm Examples

procedure binary_search(x: integer; $a_1, a_2, \ldots, a_n$: integers)
i := 1 {i is left endpoint of search interval}
j := n {j is right endpoint of search interval}
while (i < j)
begin
    m := $\lfloor (i + j)/2 \rfloor$
    if x > a_m **then** i := m + 1
    else j := m
end
if x = a_i **then** location := i
else location := 0
{location is the subscript of the term that equals x, or is zero if x is not found}

Algorithm Examples

Obviously, on sorted sequences, binary search is more efficient than linear search.

How can we analyze the efficiency of algorithms?

We can measure the

- **time** (number of elementary computations) and
- **space** (number of memory cells) that the algorithm requires.

These measures are called ***computational complexity*** and ***space complexity***, respectively.

Complexity

What is the time complexity of the linear search algorithm?

We will determine the **worst-case** number of comparisons as a function of the number n of terms in the sequence.

The worst case for the linear algorithm occurs when the element to be located is not included in the sequence.

In that case, every item in the sequence is compared to the element to be located.

Complexity

For n elements, the loop

while ($i \leq n$ and $x \neq a_i$)
    i := i + 1

is processed n times, requiring 2n comparisons.

When it is entered for the (n+1)th time, only the comparison $i \leq n$ is executed and terminates the loop.

Finally, the comparison
if $i \leq n$ **then** location := i
is executed, so all in all we have a worst-case time complexity of 2n + 2.

Complexity

What is the time complexity of the binary search algorithm?

Again, we will determine the **worst-case** number of comparisons as a function of the number n of terms in the sequence.

Let us assume there are $n = 2^k$ elements in the list, which means that $k = \log n$.

If n is not a power of 2, it can be considered part of a larger list, where $2^k < n < 2^{k+1}$.

Complexity

In the first cycle of the loop

while (i < j)
begin
  m := $\lfloor (i + j)/2 \rfloor$
  if x > a_m **then** i := m + 1
  else j := m
end

the search interval is restricted to 2^{k-1} elements, using two comparisons.

Complexity

In the second cycle, the search interval is restricted to 2^{k-2} elements, again using two comparisons.

This is repeated until there is only one (2^0) element left in the search interval.

At this point 2k comparisons have been conducted.

Complexity

Then, the comparison

while (i < j)

exits the loop, and a final comparison

if x = a_i **then** location := i

determines whether the element was found.

Therefore, the overall time complexity of the binary search algorithm is $2k + 2 = 2\lceil \log n \rceil + 2$.

Complexity

In general, we are not so much interested in the time and space complexity for small inputs.

For example, while the difference in time complexity between linear and binary search is meaningless for a sequence with n = 10, it is gigantic for $n = 2^{30}$.

Complexity

For example, let us assume we have two algorithms A and B that solve the same class of problems.

And suppose the time complexity of A is 5,000n, the one for B is $\lceil 1.1^n \rceil$ for an input with n elements.

Complexity

Comparison: time complexity of algorithms A and B

Input Size	Algorithm A	Algorithm B
n	5,000n	$\lceil 1.1^n \rceil$
10	50,000	3
100	500,000	13,781
1,000	5,000,000	$2.5 \cdot 10^{41}$
1,000,000	$5 \cdot 10^9$	$4.8 \cdot 10^{41392}$

Complexity

This means that algorithm B cannot be used for large inputs, while running algorithm A is still feasible.

So what is important is the **growth** of the complexity functions.

The growth of time and space complexity with increasing input size n is a suitable measure for the **comparison** of algorithms.