FFHS - Fernfachhochschule Schweiz
Diskrete Mathematik und Lineare Systeme

2024

Discrete Mathematics

David

Table of contents

1 Glossary of Mathematical Symbols

This is a glossary of the mathematical symbols used in this document.

1.1 Set Theory

Symbol	Usage	Interpretation
$\emptyset$	$\{\}$	The empty set
$\{\}$	$\{a, b, c...\}$	A set containing elements a, b, and c (and so on)
$\mid$	$\{a \mid T(a)\}$	The set of all a such that $T(a)$ is true
$:$	$\{a : T(a)\}$	The set of all a such that $T(a)$ is true

1.2 Set Operations

Symbol	Usage	Interpretation
$\cup$	$\{A \cup B\}$	The union of sets A and B
$\cap$	$\{A \cap B\}$	The intersection of sets A and B
$\uplus$	$\{A \uplus B\}$	Union of disjoint sets A and B

1.3 Set Relations

Symbol	Usage	Interpretation
$\in$	$\{a \in A\}$	The element a is in the set A
$\notin$	$\{a \notin A\}$	The element a is not in the set A
$\subset$	$\{A \subset B\}$	The set A is a subset of the set B
$\subseteq$	$\{A \subseteq B\}$	The set A is a subset of or equal to the set B
$\neq$	$\{A \neq B\}$	The set A is not equal to the set B

1.4 Blackboard bold

Symbol	Interpretation
$\mathbb{N}$	The set of natural numbers
$\mathbb{Z}$	The set of integers
$\mathbb{Z}p$	The set of integers where p is a prime number

1.5 Equality, equivalence and similarity

Symbol	Usage	Interpretation
$=$	$a = b$	The elements a and b are equal
$\neq$	$a \neq b$	The elements a and b are not equal
$\equiv$	$a \equiv b$	The elements a and b are equivalent
$\not\equiv$	$a \not\equiv b$	The elements a and b are not equivalent

1.6 Comparison

Symbol	Usage	Interpretation
$<$	$a < b$	The element a is less than b
$>$	$a > b$	The element a is greater than b
$\leq$	$a \leq b$	The element a is less than or equal to b
$\geq$	$a \geq b$	The element a is greater than or equal to b

1.7 Divisibility

Symbol	Usage	Interpretation
$\mid$	$a \mid b$	The element a divides b
$\nmid$	$a \nmid b$	The element a does not divide b

1.8 Relations

Symbol	Usage	Interpretation
$\circ$	$R \circ S$	The composition of relations R and S
$\leq$	$a \leq b$	Order relation between elements a and b
$\sim$	$a \sim b$	Equivalence relation between elements a and b
$[\,]$	$[a]$	The equivalence class of element a
$^{-1}$	R^{-1}	The inverse of relation R
$^{+}$	R^{+}	The transitive closure of relation R
*	R^{*}	The reflexive-transitive closure of relation R

1.9 Logical Operators

Symbol	Usage	Interpretation	Colloquially
$\wedge$	$a \wedge b$	The logical conjunction of a and b	Both a and b
$\vee$	$a \vee b$	The logical disjunction of a and b	Either a or b or both
$\neg$	$\neg a$	The logical negation of a	Not a
$\Leftrightarrow$	$a \Leftrightarrow b$	The logical implication from a to b and b to a	If a then b and if b then a
$\Rightarrow$	$a \Rightarrow b$	The logical implication from a to b	If a then b

1.10 Quantifiers

Symbol	Usage	Interpretation
$\forall$	$\forall a$	For all elements a
$\exists$	$\exists a$	There exists an element a
$\exists!$	$\exists! a$	There exists exactly one element a
$\nexists$	$\nexists a$	There does not exist an element a

2 Relation

2.1 Cartesian product

The Cartesian product of two sets A and B is the set of all ordered pairs (a, b) where a is an element of A and b is an element of B.
$A * B = \{(a, b) \mid a \in A \land b \in B\}$

Definition 0.0.1

The Cartesian product of the sets $A = \{1, 2\}$ and $B = \{3, 4\}$ is:
$A * B = \{(1, 3), (1, 4), (2, 3), (2, 4)\}$.

Example 0.0.1

A relation R from a set A to a set B is a subset of the Cartesian product $A * B$.
$R \subseteq A * B$

Definition 0.0.2

Let $A = \{1, 2\}$ and $B = \{3, 4\}$. The relation $R = \{(1, 3), (2, 4)\}$ is a relation from A to B.

Example 0.0.2

For $(a, b) \in R$, we write aRb, and say that a is in relation R to b.

2.2 Inverse relation

The inverse relation $R^{\{-1\}}$ of a relation R is the relation that contains the ordered pairs of R in reverse order.
$R^{\{-1\}} = \{(b, a) \mid (a, b) \in R\}$

Definition 0.0.3

Let $R = \{(1, 3), (2, 4)\}$. The inverse relation $R^{\{-1\}}$ is:
$R^{\{-1\}} = \{(3, 1), (4, 2)\}$.

Example 0.0.3

2.3 Composition of relations

Given the relation $R \subseteq A * B$ and $S \subseteq B * C$, the composition of $R \circ S$ is the relation from A to C defined by:

$R \circ S = \{(a, c) \mid \exists b \in B, (a, b) \in R \land (b, c) \in S\}$

Definition 0.0.4

2.4 Representation of relations

Relations can be represented in different ways, one way is by using a directed graph.

$R = \{(a,1),(b,1),(b,3),(c,2)\} \subseteq A * B$ when $A = \{a,b,c\}$ and $B = \{1,2,3\}$.

$$\begin{tikzcd}
c \arrow[r] & 2 \\
a \arrow[r] & 1 \\
b \arrow[ru] \arrow[r] & 3
\end{tikzcd}$$

or $R^{\{-1\}} = \{(1,a),(1,b),(3,b),(2,c)\}$.

$$\begin{tikzcd}
2 \arrow[r] & c \\
1 \arrow[r] \arrow[rd] & a \\
3 \arrow[r] & b
\end{tikzcd}$$

Example 0.0.4

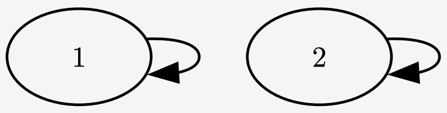
2.5 Relations $R \subseteq A * A$

Relations that are subsets of the Cartesian product of a set with itself are called relations on the set. They can have the following properties:

Reflexive: $(a, a) \in R \forall a \in A$.

Definition 0.0.5

The relation $R = \{(1, 1), (2, 2)\} \subseteq A * A$ is reflexive.

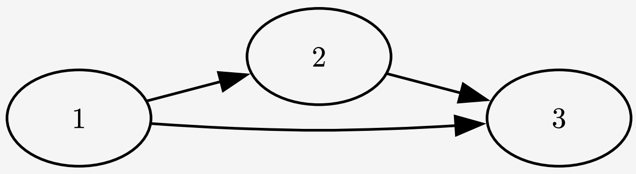


Example 0.0.5

Transitive: $\forall a, b, c \in A, (a, b) \in R \land (b, c) \in R \Rightarrow (a, c) \in R$.

Definition 0.0.6

The relation $R = \{(1, 2), (2, 3), (1, 3)\} \subseteq A * A$ is transitive.

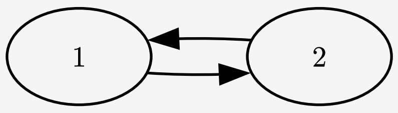


Example 0.0.6

Symmetric: $\forall a, b \in A, (a, b) \in R \Rightarrow (b, a) \in R$.

Definition 0.0.7

The relation $R = \{(1, 2), (2, 1)\} \subseteq A * A$ is symmetric.

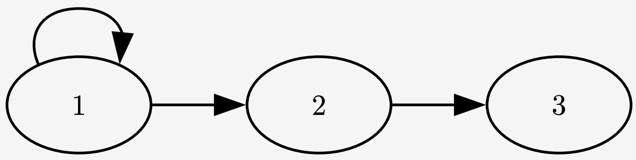


Example 0.0.7

Antisymmetric: $\forall a, b \in A, (a, b) \in R \land (b, a) \in R \Rightarrow a = b$.
or equivalently: $\forall a, b \in A, (a, b) \in R \land a \neq b \Rightarrow (b, a) \notin R$.

Definition 0.0.8

The relation $R = \{(1, 2), (2, 3), (1, 1)\} \subseteq A * A$ is antisymmetric.

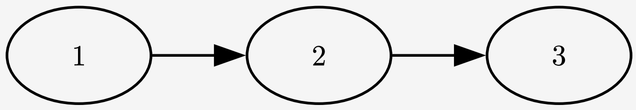


Example 0.0.8

Asymmetric: $\forall a, b \in A, (a,b) \in R \Rightarrow (b,a) \notin R$.

Definition 0.0.9

The relation $R = \{(1,2),(2,3)\} \subseteq A * A$ is asymmetric.



Example 0.0.9

A relation R on a set A is called an equivalence relation if it is **reflexive**, **symmetric**, and **transitive**.
For $(a,b) \in R$, we say that a is **equivalent** to b and write $a \equiv b$.

Statement 0.0.1

2.6 Equivalence classes

Given an equivalence relation R on a set A, the equivalence class of an element $a \in A$ is the set of all elements in A that are equivalent to a.
$[a]_R = \{b \in A \mid a \equiv b\}$

Definition 0.0.10

Given the relation R is an equivalence relation on the set A then the following properties hold:

1. The equivalence classes of R form a partition of A.

2. A partition of a set A is a collection of nonempty, mutually disjoint subsets of A whose union is A.

Statement 0.0.2

2.7 Order Relations

A relation R on a set A is called a order(relation) if it is **reflexive**, **antisymmetric** and **transitive**.
Often denoted by $a \leq b$.

Definition 0.0.11

For each order there also exists a strict order. A strict order is the result of removing the reflexive property from the order relation.

2.7.1 Strict order

A relation R on a set A is called a strict order if it is **antisymmetric** and **transitive** and **not reflexive.**

Definition 0.0.12

From each order relation R there exists a strict order relation S such that $aRb \iff aSb \land a \neq b$. From each strict order a order relation can be derived by adding the reflexive property.

Statement 0.0.3

$A \leq B$ is a order relation on the set A.
$A < B$ is a strict order relation on the set A.

Example 0.0.10

2.8 Comparability

Two elements a and b in a set A are said to be comparable with respect to a relation R if either aRb or bRa.

2.8.1 Total order

A relation R on a set A is called a total order if it is a partial order and for all $a, b \in A$ either aRb or bRa.

Definition 0.0.13

Total means that for any elements a and b in A, they are always related (they can always be compared) with respect to $R \iff aRb \lor bRa$.

Statement 0.0.4

2.8.2 Partial order

A relation R on a set A is called a partial order if it is **reflexive**, **antisymmetric** and **transitive**.

Definition 0.0.14

Partial means that for any elements a and b in A, they are not always related (they can not always be compared) with respect to $R \iff aRb \lor bRa$.

Statement 0.0.5

2.9 closures

Closure of a relation R is the smallest relation that contains R and has a certain property.

2.9.1 Reflexive closure

The reflexive closure of a relation R on a set A is the smallest relation that contains R and is reflexive.
A relation R is reflexive if for all $a \in A$, $(a, a) \in R$.

Definition 0.0.15

The reflexive closure of a relation R is $R \cup \{(a, a) \mid a \in A\}$.
Often denoted by $[R]^{\text{refl}}$.

Statement 0.0.6

Let $R = \{(1, 2), (2, 3)\} \subseteq A * A$.
The reflexive closure of R is $R \cup \{(1, 1), (2, 2), (3, 3)\}$.

Example 0.0.11

2.9.2 Transitive closure

The transitive closure of a relation R on a set A is the smallest relation that contains R and is transitive.
A relation R is transitive if for all $a, b, c \in A$, $(a, b) \in R \wedge (b, c) \in R \Rightarrow (a, c) \in R$.

Definition 0.0.16

The transitive closure of a relation R is the intersection of all transitive relations that contain R.
Often denoted by $[R]^{\text{trans}}$.
$[R]^{\text{trans}} = R \cup \{(a, c) \mid \exists b \in A, (a, b) \in R \wedge (b, c) \in R\}$.

Statement 0.0.7

Let $R = \{(1, 2), (2, 3)\} \subseteq A * A$.
The transitive closure of R is $R \cup \{(1, 3)\}$.

Example 0.0.12

2.9.3 Symmetric closure

The symmetric closure of a relation R on a set A is the smallest relation that contains R and is symmetric.
A relation R is symmetric if for all $a, b \in A$, $(a, b) \in R \Rightarrow (b, a) \in R$.

Definition 0.0.17

The symmetric closure of a relation R is $R \cup \{(b, a) \mid (a, b) \in R\}$.
Often denoted by $[R]^{\text{sym}}$.

Statement 0.0.8

Let $R = \{(1,2),(2,3)\} \subseteq A * A$.
The symmetric closure of R is $R \cup \{(2,1),(3,2)\}$.

Example 0.0.13

3 Modular arithmetic

Modular arithmetic is a system of arithmetic for integers, where numbers “wrap around” upon reaching a certain value called the modulus.

Definition 0.0.18

A common example of modular arithmetic is the 12-hour clock, where the hours are represented by numbers from 1 to 12. When the clock reaches 12, it wraps around to 1. If the time now is 10 o’clock and we add 5 hours, the result is 10 + 5 = 3, because 10 + 5 = 15, and 15 modulo 12 is 3.

Example 0.0.14

Two integers a and b are said to be congruent modulo m if m divides their difference. This is denoted as $a \equiv b \operatorname{mod}(m)$.

In other words, a and b leave the same remainder when divided by m.

Statement 0.0.9

$7 \equiv 19 \operatorname{mod}(6)$ because 6 divides $19 - 7 = 12$.
or equivalently, $7 \operatorname{mod}(6) = 1$ and $19 \operatorname{mod}(6) = 1$. So, $7 \equiv 19 \operatorname{mod}(6)$.

Example 0.0.15

3.1 Ring of integers modulo m

The ring of integers modulo m, denoted as $\mathbb{Z}/m\mathbb{Z}$, is the set of integers from 0 to $m - 1$.

Definition 0.0.19

The ring of integers modulo 3, denoted as $\mathbb{Z}/3\mathbb{Z}$, is the set $\{0, 1, 2\}$. Since the possible numbers are limited to $0, 1, 2$ its easy to create a table of addition and multiplication for all possible combinations.

+	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

Table 1: Addition table

*	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

Table 2: Multiplication table

Example 0.0.16

3.2 Digit sum

The digit sum of a number is the sum of its digits.

Definition 0.0.20

The digit sum of 123 is $1 + 2 + 3 = 6$.

Example 0.0.17

We can use the digit sum to determine if a number is divisible by $3 \vee 9$.

3.2.1 Digital root

The digital root of a number is the single-digit number obtained by repeatedly summing the digits of the number until a single-digit number is obtained.

Definition 0.0.21

The digital root of 123 is $1 + 2 + 3 = 6$.
The digital root of 12345 is $1 + 2 + 3 + 4 + 5 = 15$, and $1 + 5 = 6$.

Example 0.0.18

3.3 Neutral element

The neutral element is an element, that when combined with another element using a binary operation, leaves the other element unchanged.

Definition 0.0.22

The neutral element for addition is 0, because $a + 0 = a \forall a \in \mathbb{Z}$.
The neutral element for multiplication is 1, because $a * 1 = a \forall a \in \mathbb{Z}$.

Statement 0.0.10

3.3.1 Inverse element

The inverse element is an element, that when combined with another element using a binary operation, results in the neutral element.

Definition 0.0.23

The inverse element for addition is the negative of the element, because $a + (-a) = 0 \forall a \in \mathbb{Z}$.

The inverse element for multiplication is the reciprocal of the element, because $a * \left(\frac{1}{a}\right) = 1 \forall a \in \mathbb{Z}$.

Statement 0.0.11

3.4 Neutral and inverse elements in modular arithmetic

Neutral and inverse elements can also be defined in modular arithmetic.

3.4.1 Subtraction

In order to subtract b from a in the ring of integers modulo m, find the inverse of b and add it to a.

The inverse of b is the element x such that $b + x = 0 \bmod (m)$.

Every element in the ring of integers modulo m has an inverse element therefore **subtraction is always possible.**

Statement 0.0.12

If we take the ring of integers modulo 6 ($\mathbb{Z}/6\mathbb{Z}$), we get the following addition table:

+	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

In this table, each row and column contains all the elements of the ring of integers modulo 6. Figures like this are called Latin squares.

Since subtraction can be defined as addition of the inverse element $a + (-b)$, its possible to subtract by finding the inverse element.

For example: $5 - 4 \in \mathbb{Z}/6\mathbb{Z}$
1. Find the inverse of 4: $4 + x = 0 \bmod (6)$. The inverse of 4 is 2.
2. Add the inverse to 5: $5 + 2 = 1 \bmod (6)$.

Example 0.0.19

3.4.2 Division

In order to divide a by b in the ring of integers modulo m, find the inverse of b and multiply it by a.

The inverse of b is the element x such that $b * x = 1 \operatorname{mod}(m)$.

Not every element in the ring of integers modulo m has an inverse element therefore **division is not always possible**.

Statement 0.0.13

If we take the ring of integers modulo 6 ($\mathbb{Z}/6\mathbb{Z}$), we get the following multiplication table:

*	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	1	2	3	4	5
2	0	2	4	0	2	4
3	0	3	0	3	0	3
4	0	4	2	0	4	2
5	0	5	4	3	2	1

In this table, not every element has a reciprocal element. For example, 2 does not have a reciprocal element.

therefore, division is only possible for the elements $1, 5$ in the ring of integers modulo 6.

Example 0.0.20

In order to find a ring of integers modulo m where division is possible for all elements, m must be a prime number.

7 ($\mathbb{Z}/7\mathbb{Z}$), we get the following multiplication table, that shows that every element has a reciprocal element.

*	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

Example 0.0.21

For smaller numbers, its easy to find the inverse element by trial and error or by writing out the multiplication table. For larger numbers, the Extended Euclidean Algorithm can be used to find the inverse element.

In order for a to have an inverse element x in the ring of integers modulo m, a and m must be coprime. This means that the greatest common divisor of a and m must be 1 $(\gcd(a, m) = 1)$.

Definition 0.0.24

Since a and m are coprime, there exists integers x and y such that $ax + my = 1$.

To find the inverse element x, we can use the Extended Euclidean Algorithm.

Statement 0.0.14

4 Greatest Common Divisor

Each number has at least two divisors: 1 and itself.

Definition 0.0.25

$d \mid a \Rightarrow a = d * k$ for some integer k.

This means the divisors d can not be larger than a itself.

Statement 0.0.15

The divisors of 12 are $1, 2, 3, 4, 6, 12$.

$1 * 12 = 12$, $2 * 6 = 12$, $3 * 4 = 12$, $4 * 3 = 12$, $6 * 2 = 12$, $12 * 1 = 12$.

Example 0.0.22

A common divisor of two numbers is a number that divides both numbers. This means a common divisor of a and b is a number d that divides both a and b.

Definition 0.0.26

The common divisors of 12 and 18 are $1, 2, 3, 6$.

$1 * 12 = 12$, $2 * 6 = 12$, $3 * 4 = 12$, $6 * 2 = 12$.

$1 * 18 = 18$, $2 * 9 = 18$, $3 * 6 = 18$, $6 * 3 = 18$.

Example 0.0.23

The greatest common divisor of two numbers is the largest number that divides both numbers denoted as $\gcd(a, b)$.

Definition 0.0.27

In order to find the greatest common divisor of two numbers, the Euclidean Algorithm can be used.

4.1 Euclidean Algorithm

The Euclidean Algorithm is an efficient method to find the greatest common divisor of two numbers. It is based on the fact that a common divisor of two numbers is also a divisor of their sum and difference.

$a = 42$, $b = 66$.

A common divisor of 42 and 66 is for example 3. $3 * 14 = 42$ and $3 * 22 = 66$.

For sum and difference the following holds:

$108 = 42 + 66 = 3 * 14 + 3 * 22 = 3 * (14 + 22) = 3 * 36$.

$24 = 66 - 42 = 3 * 22 - 3 * 14 = 3 * (22 - 14) = 3 * 8$.

Example 0.0.24

4.1.1 Algorithm

The Euclidean algorithm works as follows:

$\gcd(400, 225)$

$400 - 225$	$400 - 225 = 175$
$225 - 175$	$225 - 175 = 50$
$175 - 50$	$175 - 50 = 125$
$125 - 50$	$125 - 50 = 75$
$75 - 50$	$75 - 50 = 25$
$50 - 25$	$50 - 25 = 25$
$25 - 25$	$25 - 25 = 0$

$\gcd(400, 225) = 25$.

Example 0.0.25

$$d \mid (\alpha * a + \beta * b) \quad \forall \alpha, \beta \in \mathbb{Z}.$$

Every term of the form $\alpha * a + \beta * b$ is a multiple of d if both a and b are multiples of d. Such terms are called **linear combinations** of a and b.

Statement 0.0.16

4.2 Extended Euclidean Algorithm

The Extended Euclidean Algorithm calculates in addition to the greatest common divisor (gcd) of integers a and b, also the coefficients of Bézout's identity, which are integers x and y such that

$$a * x + b * y = \gcd(a, b).$$

Definition 0.0.28

Given the same example as before: $a = 400$, $b = 225$.

The Extended Euclidean Algorithm calculates the coefficients x and y such that $a * x + b * y = \gcd(a, b)$.

Example 0.0.26

$$400 - 225$$
$$225 - 175$$
$$175 - 50$$
$$125 - 50$$
$$75 - 50$$
$$50 - 25$$
$$25 - 25$$

$\gcd(400, 225) = 25$.

Now the coefficients x and y can be calculated by working backwards:

$25 = 50 - 25$

$= 50 - (75 - 50) = 2 * 50 - 75$

$= 2 * 50 - (125 - 50) = 3 * 50 - 125$

$= 3 * 50 - (175 - 50) = 4 * 50 - 175$

$= 4 * (225 - 175) - 175 = 4 * 225 - 5 * 175$

$= 4 * 225 - 5 * (400 - 225)$

$= 9 * 225 - 5 * 400$

Example 0.0.27

5 Graph Theory

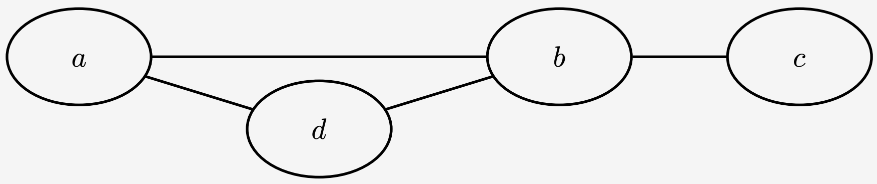
5.1 Basic Definitions

A Graph G is a pair $G = (V, E)$ where V is a set of vertices and E is a set of edges $\{a, b\}$ where $a, b \in V, a \neq b$.

Definition 0.0.29

An edge $e = \{a, b\}$ always connects two vertices a and b. The vertices a and b are called the **endpoints** of the edge e.

A graph $G = (V, E)$ with $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{b, c\}, \{a, d\}\}, \{d, b\}\}$.

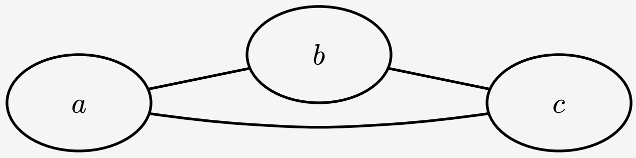


Example 0.0.28

A graph $H = (V, E)$ is a **subgraph** of a graph $G = (V', E')$ if $V \subset V'$ and $E \subset E'$.

Definition 0.0.30

A graph $G = (V, E)$ with $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{b, c\}, \{a, d\}\}, \{d, b\}\}$ has a subgraph $H = (V', E')$ with $V' = \{a, b, c\}$ and $E' = \{\{a, b\}, \{b, c\}\}, \{a, c\}$.

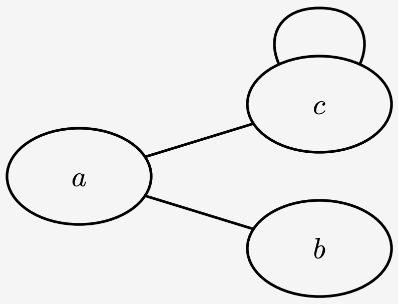


Example 0.0.29

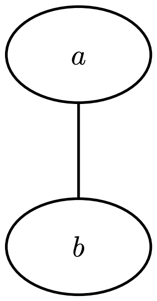
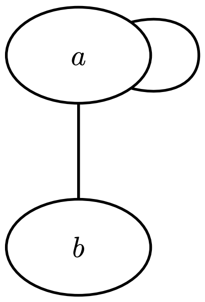
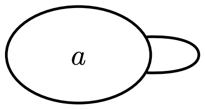
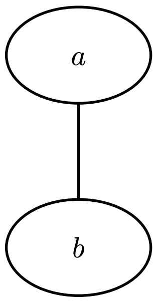
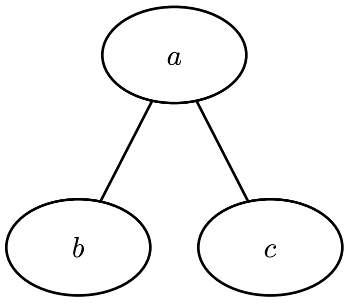
To reduce confusion, graphs are often drawn so that the edges do not cross each other. For all graphs that this is possible, the graph is called a **planar graph**.

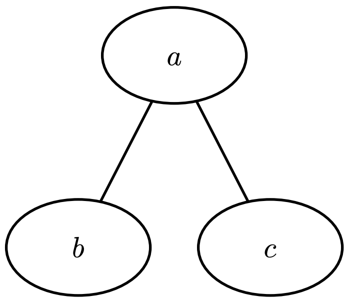
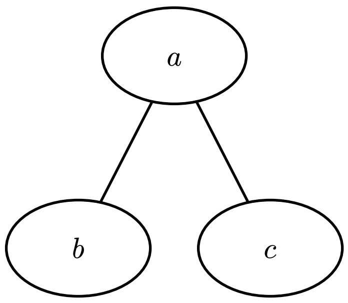
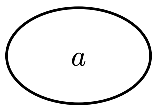
According to our previous definition, a graph can not have an edge that connects a vertex to itself. However, in some cases, it is useful to allow such edges. A graph that allows edges to connect a vertex to itself or multiple edges between the same vertices is called a **multigraph.**

A graph $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{\{a, c\}, \{a, b\}, \{c, c\}\}$.



Example 0.0.30

Term	Definition	Example
Vertex	A point in a graph	a
Edge	A connection between two vertices	$\{a, b\}$
Subgraph	A graph that is a subset of another graph	$H = (V', E')$
Planar Graph	A graph that can be drawn without edges crossing	
Multigraph	A graph that allows edges to connect a vertex to itself or multiple edges between the same vertices	 $E = \{\{a, a\}, \{a, b\}, \{b, a\}\}$
Loop	An edge that connects a vertex to itself	
Adjacent	Two vertices are adjacent if they are connected by an edge	 a and b are adjacent
Incident (edges)	Two edges are incident if they share a vertex	

Term	Definition	Example
		$\{a, b\}$ and $\{a, c\}$ are incident
Incident (vertex)	An edge is incident to a vertex if the vertex is an endpoint of the edge	 a is incident to $\{a, b\}$
Degree	The number of edges incident to a vertex	 The degree of a is 2
Isolated Vertex	A vertex with degree 0	 a is an isolated vertex

Given a graph $G = (V, E)$, the sum of the degrees of all vertices is equal to twice the number of edges.

$$\sum_{v \in V} \deg(v) = 2\ |E|$$

Statement 0.0.17

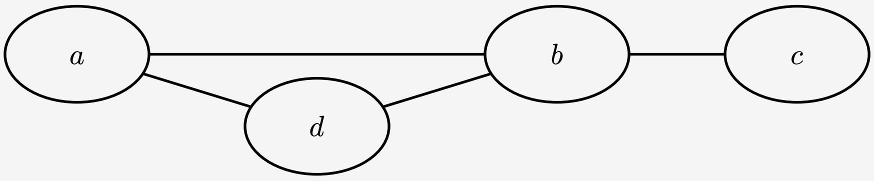
It is important to note that in a multigraph with loops, each loop contributes $+2$ to the degree of the vertex it is incident to.

In each graph, the number of vertices with odd degree is even.

Statement 0.0.18

This must be the case because the sum of the degrees of all vertices is equal to twice the number of edges. Since the sum of the degrees is even (due to the factor of 2), the number of vertices with odd degree must be even (odd + odd = even).

In the graph $G = (V, E)$ with $V = \{a, b, c, d\}$ and $E = \{\{a, b\}, \{b, c\}, \{a, d\}\}, \{d, b\}\}$,



the vertices a and d have an even degree, while b and c have an odd degree.

Example 0.0.31

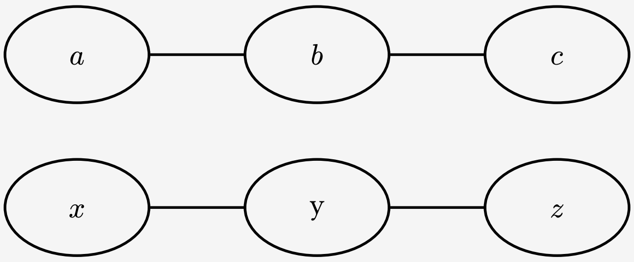
5.1.1 Equivalency of Graphs

Two graphs are considered equivalent if they have the same number of vertices and edges, and the edges are connected in the same way.

Two graphs $G = (V, E)$ and $H = (V', E')$ are **isomorphic** if there is a bijection $f : V \to V'$ such that $\{a, b\} \in E$ if and only if $\{f(a), f(b)\} \in E'$.

Definition 0.0.31

The graphs $G = (V, E)$ and $H = (V', E')$ with $V = \{a, b, c\}$, $V' = \{x, y, z\}$, $E = \{\{a, b\}, \{b, c\}\}$, and $E' = \{\{x, y\}, \{y, z\}\}$ are isomorphic.



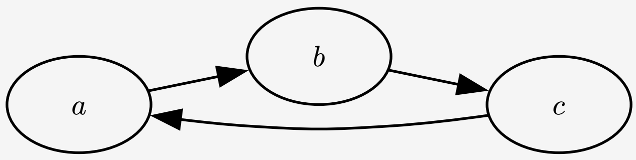
The bijection f is defined as $f(a) = x$, $f(b) = y$, and $f(c) = z$.

Example 0.0.32

5.1.2 Directed Graphs

In a directed graph, the edges have a direction. An edge $e = \{a, b\}$ is directed from a to b. The vertex a is called the **tail** of the edge, and the vertex b is called the **head** of the edge.

A directed graph $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}, \{c, a\}\}$.



Example 0.0.33

5.2 Representation of Graphs in Computers

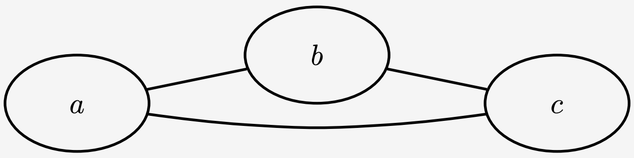
5.2.1 Matrix Representation

Graphs can be represented in computers in different ways. One common representation is the **adjacency matrix**.

The **adjacency matrix** of a graph $G = (V, E)$ is a matrix A of size $|V| \times |V|$ where $A_{ij} = 1$ if there is an edge between vertices v_i and v_j, and 0 otherwise.

Definition 0.0.32

The adjacency matrix of the graph $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}, \{c, a\}\}$ is:



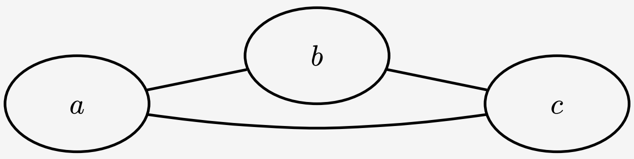
$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

Example 0.0.34

The **Incidence Matrix** of a graph $G = (V, E)$ is a matrix A of size $|V| \times |E|$ where $A_{ij} = 1$ if vertex v_i is incident to edge e_j, and 0 otherwise.

Definition 0.0.33

The incidence matrix of the graph $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}, \{c, a\}\}$ is:



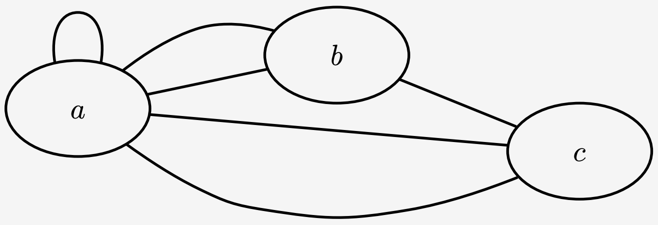
$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

Example 0.0.35

5.2.1.1 Multigraphs

In the adjacency matrix of a multigraph, the value of A_{ij} is the number of edges between vertices v_i and v_j.

The adjacency matrix of the multigraph $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{aa, ab, ba, bc, ca, ac\}$ is:



$$\begin{pmatrix} 2 & 2 & 2 \\ 2 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix}$$

Example 0.0.36

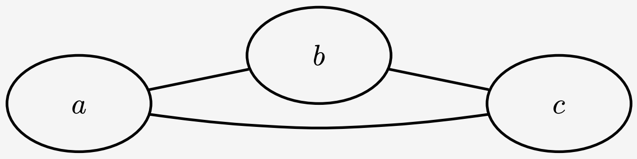
5.2.2 List Representation

Another common representation is the **adjacency list**.

The **adjacency list** of a graph $G = (V, E)$ is a list of size $|V|$ where each element $A[i]$ is a list of vertices adjacent to vertex v_i.

Definition 0.0.34

The adjacency list of the graph $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}, \{c, a\}\}$ is:



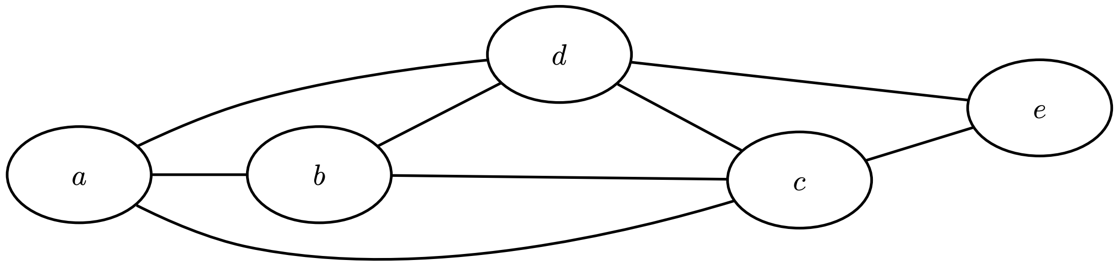
Vertex	Adjacent Vertices
a	b, c
b	a, c
c	a, b

Example 0.0.37

The adjacency list representation is more space-efficient than the adjacency matrix representation for sparse graphs, because the number of elements in an adjacency list is twice the number of edges, while the number of elements in an adjacency matrix is $|V|^2$.

6 Cycles and Paths

Given the following graph $G = (V, E)$
$V = \{a, b, c, d, e\}, E = \{ab, ad, ac, bd, bc, dc, de, ce\}$:



In this graph, there are multiple paths from the vertex a to the vertex e. One such path could be ad, dc, cb, bd, de. This path is called a **walk**.

a walk is a sequence of incident edges in a graph.

Properties of a walk:
- A walk is called **closed** if the start and end vertices are the same.
- The number of edges in a walk is called the **length** of the walk.

Definition 0.0.35

a, d, c, b, d, e is a walk of length 5 in the graph G.

Example 0.0.38

A **trail** is a walk where no edge is repeated.

Definition 0.0.36

a, d, c, b, d, e is a trail in the graph G.

Example 0.0.39

A **path** is a trail where no vertex is repeated.

Definition 0.0.37

a, d, c, e is a path in the graph G.

Example 0.0.40

A **cycle** is a non-empty trail where the start and end vertices are the same.

Definition 0.0.38

a, d, c, b, a is a cycle in the graph G.

Example 0.0.41

6.1 Calculating the Number of Paths

The number of paths of length k between two vertices a and b can be calculated using the adjacency matrix A of the graph.

The number of paths of length k between vertices a and b is the value of A^k_{ab}.

Statement 0.0.19

Given is the following adjacency matrix A that does not contain weights:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

We are looking for the number of paths of length 3 between vertices 2 and 3. For this, we calculate A^3:

$$A^2 = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$A^3 = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

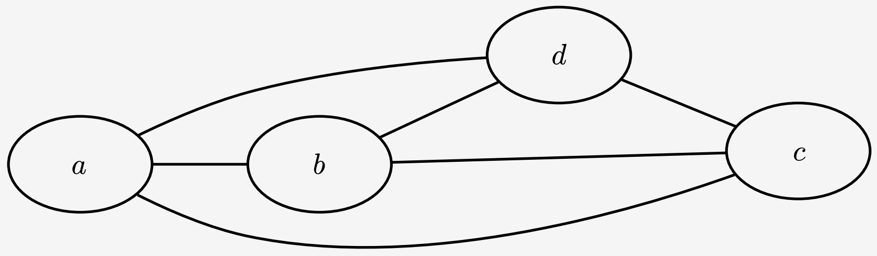
The value of A^3_{23} is 2, which means that there are 2 paths of length 3 between vertices 2 and 3.

Example 0.0.42

In an undirected graph $G = (V, E)$, two vertices a and b are called **connected** if G contains a path from a to b. If a graph contains a path between every pair of vertices, it is called **connected.**

Definition 0.0.39

The graph $G = (V, E)$ with $V = \{a, b, c, d\}$ and $E = \{ab, ad, ac, bd, bc, dc\}$ is connected.

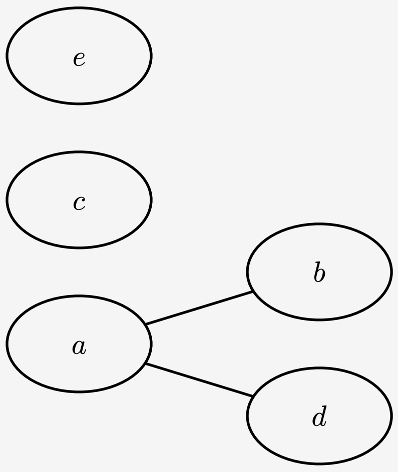


Example 0.0.43

A maximal connected subgraph of a graph G is called a **connected component.**

Definition 0.0.40

The graph $G = (V, E)$ with $V = \{a, b, c, d, e\}$ and $E = \{ab, ad\}$ has three connected components.



The connected components are $\{a, b, d\}$, $\{c\}$, and $\{e\}$.

Example 0.0.44

A graph is called **disconnected** if it contains more than one connected component.

Definition 0.0.41

6.2 Breadth-First Search (BFS)

Breadth-First Search (BFS) is an algorithm used to traverse graphs. It starts at a given vertex and explores all of its neighbors at the present depth before moving on to the vertices at the next depth level.

Algorithm:

1. Determine the starting vertex s, mark it as visited, and add it to the queue.
2. Take the first vertex v from the queue:
   - If v is the target vertex, the search is complete.
   - Otherwise, add all unvisited neighbors of v to the queue and mark them as visited.
3. Repeat step 2 until the queue is empty.
4. If the queue is empty and the target vertex has not been found, the target vertex is not reachable from the starting vertex.

To check if a graph is connected, we can start a BFS from any vertex and check if all vertices are visited at the end.

A connected graph with n vertices has at least $n - 1$ edges.

A graph with more than $\frac{(n-1)(n-2)}{2}$ edges is connected.

Statement 0.0.20

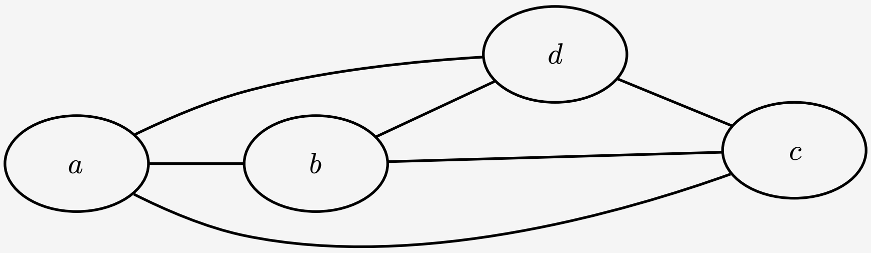
Another way to check if a graph is connected is by checking the adjacency matrix. If the matrix A^k contains only non-zero values, the graph is connected.

6.2.1 Eulerian Paths and Cycle

An **Eulerian path** is a path that visits every edge exactly once.

Definition 0.0.42

The graph $G = (V, E)$ with $V = \{a, b, c, d\}$ and $E = \{ab, ad, ac, bd, bc, dc\}$ has an Eulerian path a, b, d, c, a, d, b, c.



Example 0.0.45

A connected (Multi)graph has an Eulerian cycle if and only if all vertices have an even degree.

Statement 0.0.21

6.2.2 Fleury's Algorithm

Fleury's algorithm is an algorithm used to find an Eulerian cycle in a graph.

Algorithm:

1. Start at any vertex v.
2. While there are unvisited edges in the graph:
   - If v has no neighbors, add v to the cycle and remove it from the graph.
   - If v has neighbors, choose the neighbor u that is not a bridge (an edge whose removal would disconnect the graph) if possible.
   - Add the edge $v - u$ to the cycle and remove it from the graph.
   - Set $v = u$.
3. The cycle is complete when all edges have been visited.

6.2.3 Hamiltonian Cycle

A **Hamiltonian cycle** is a cycle that visits every vertex exactly once.

Definition 0.0.43

In a graph with n vertices, a Hamiltonian cycle must exists if a minimum of $\frac{1}{2}(n-1)(n-2)+2$ edges are present.

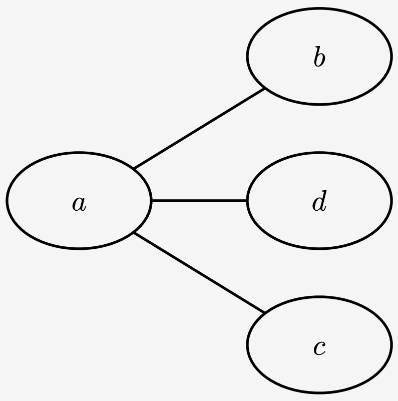
Statement 0.0.22

7 Trees

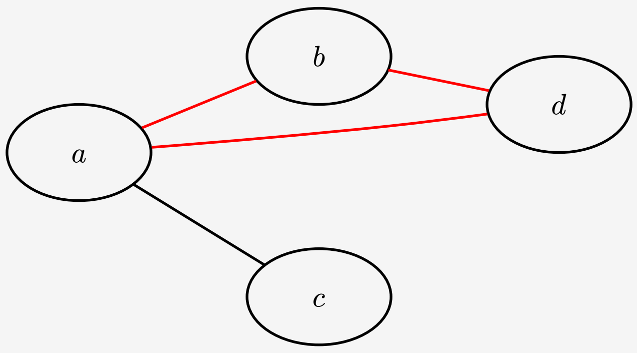
A **tree** is a connected graph with no cycles.

Definition 0.0.44

The graph $G = (V, E)$ with $V = \{a, b, c, d\}$ and $E = \{ab, ad, ac\}$ is a tree.



The graph $H = (V, E)$ with $V = \{a, b, c, d\}$ and $E = \{ab, ad, ac, bd\}$ is not a tree because it contains a cycle.



Example 0.0.46

If all connected components of a graph are trees, the graph is called a **forest**.

Definition 0.0.45

A connected graph G with n vertices is a tree if and only if:
- G has $n - 1$ edges.
- If one edge is removed from G, it becomes disconnected.
- Between any two vertices in G, there is exactly one path.

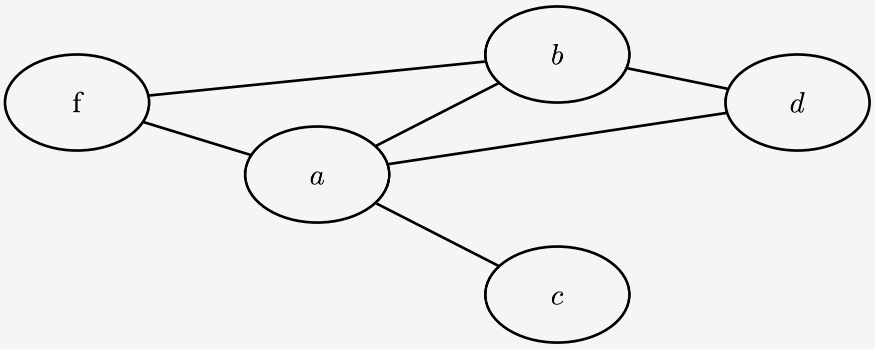
Statement 0.0.23

A **spanning tree** of a graph G is a subgraph that is a tree and contains all vertices of G.

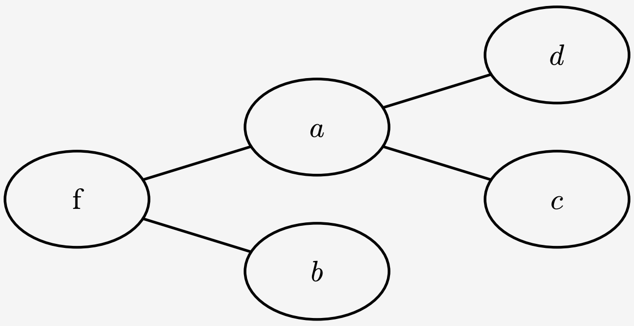
Definition 0.0.46

The graph $G = (V, E)$ with $V = \{a, b, c, d, e, f\}$ and $E = \{ab, ad, ac, bd, fa, fb\}$ has a spanning tree $T = (V, E')$ with $V = \{a, b, c, d, e, f\}$ and $E' = \{ad, ac, fa, fb\}$.

G:

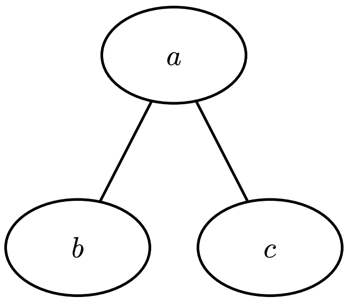


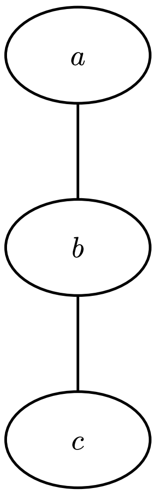
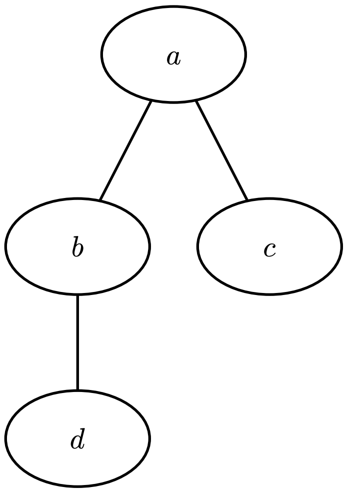
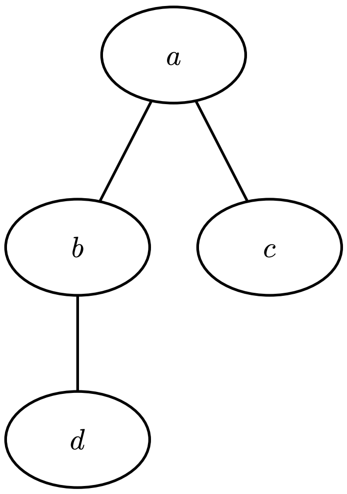
T:

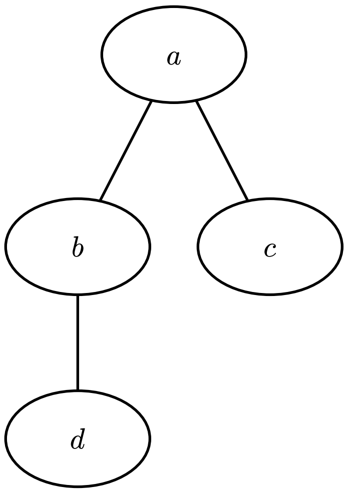
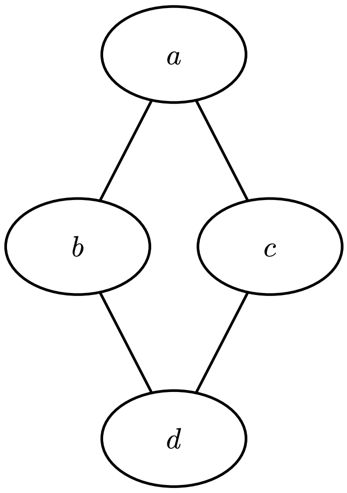
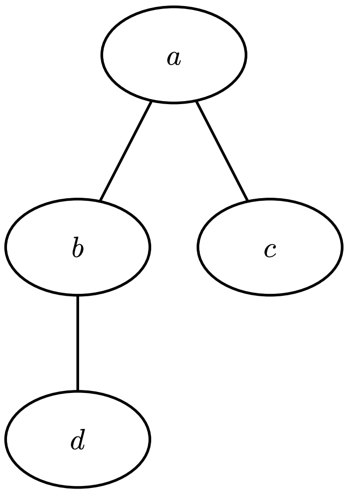


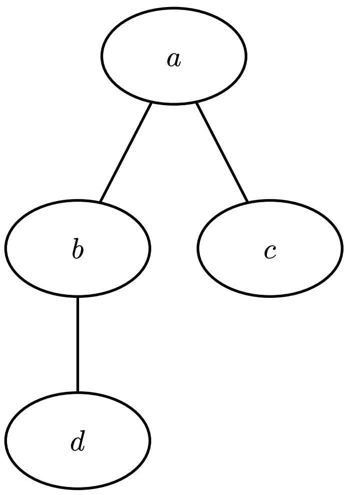
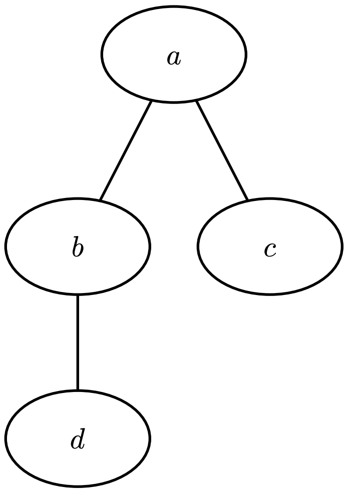
Example 0.0.47

7.1 Naming

Term	Definition	Example
root	A vertex with no incoming edges	 a is the root
leaf	A vertex with no outgoing edges	

Term	Definition	Example
		 c is a leaf
level	The distance from the root	 a is level 0, b is level 1, c is level 1, d is level 2
parent	A vertex v is the parent of vertex u if there is an edge $v - u$	 a is the parent of b

Term	Definition	Example
child	A vertex u is a child of vertex v if there is an edge $v - u$	 b is a child of a
sibling	Two vertices that share the same parent	 b and c are siblings
ancestor	A vertex v is an ancestor of vertex u if there is a path from v to u	 a is an ancestor of d

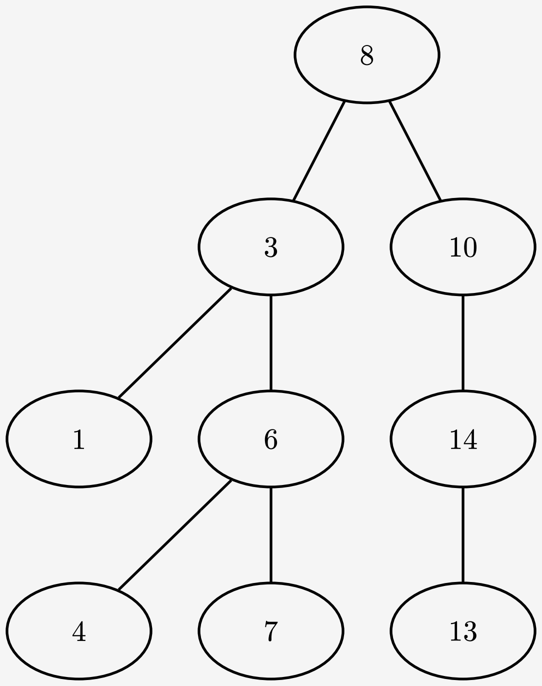
Term	Definition	Example
descendant	A vertex u is a descendant of vertex v if there is a path from v to u	 d is a descendant of a
binary tree	A tree where each vertex has at most two children	

7.2 Binary Search Trees

A **binary search tree** (BST) is a binary tree where each vertex has at most two children, and the left child of a vertex contains a value less than the vertex, while the right child contains a value greater than the vertex.

Definition 0.0.47

The following binary tree is a binary search tree:



Example 0.0.48

7.2.1 Operations

Search: To find a value in a binary search tree, start at the root and compare the value with the current vertex. If the value is less than the current vertex, move to the left child. If it is greater, move to the right child. Repeat this process until the value is found or corresponding subtree is empty, then the value is not in the tree.

Insert: To insert a value into a binary search tree, search for the value. If the value is not found, insert it as a leaf at the position where the search ended.

Delete: To delete a value from a binary search tree, search for the value. If the value is found:
- If the vertex has no children, remove the vertex.
- If the vertex has one child, replace the vertex with its child.
- If the vertex has two children, replace the vertex with the smallest value in the right subtree.

Definition 0.0.48

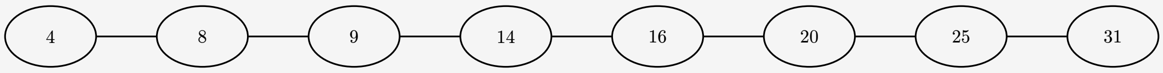
The maximum amount of comparisons needed to search for a value in a binary search tree with length n is $n + 1$.

Statement 0.0.24

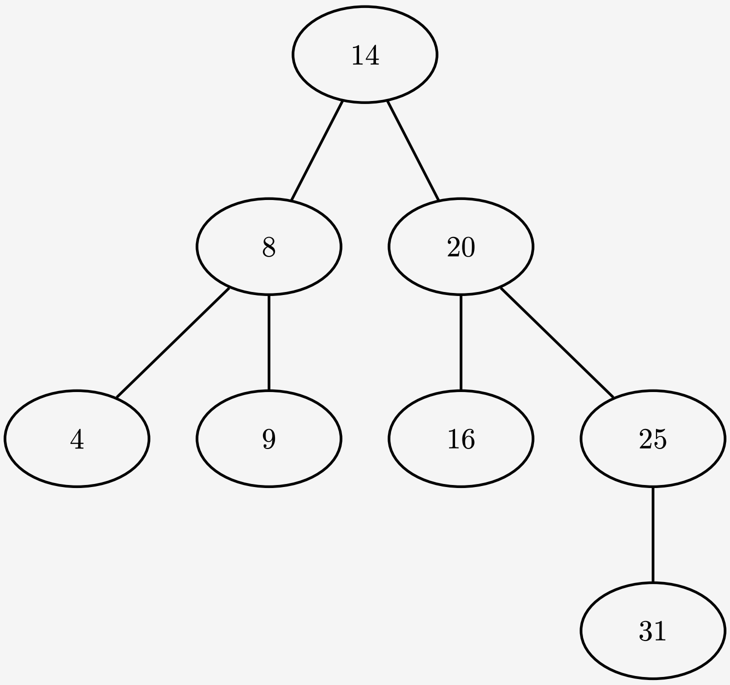
The time complexity of searching, inserting, and deleting in a binary search tree is $O(h)$, where h is the height of the tree. This makes it crucial to ensure the height is minimized. The height of a tree is highly dependent on the order of insertion and deletion operations.

A Binary Search Tree with the values $\{4, 8, 9, 14, 16, 20, 25, 31\}$:

worst case height: 7



compact height: 3



Example 0.0.49

One method to try to minimize the height of a binary search tree is a **devided and conquer** approach. This approach involves inserting the median value of the list as the root, then recursively inserting the median value of the left and right sublists as the left and right children of the root.

In a binary search tree with length l, is the maximum data that can be stored:

$$n = \sum_{k=0}^{l} 2^k = 2^{l+1} - 1$$

Resulting in a search time of $O(\log_2(n))$.

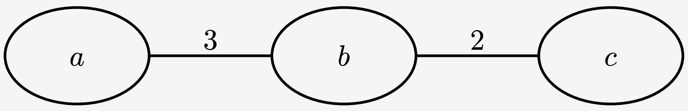
Statement 0.0.25

8 Weighted Graphs

A **weighted graph** is a graph $G = (V, E)$ where each edge $e = \{a, b\}$ has a weight $w(e) \geq 0$.

Definition 0.0.49

The graph $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}\}$ has the weights $w(\{a, b\}) = 3$ and $w(\{b, c\}) = 2$.



or as an adjacency matrix:

$$\begin{pmatrix} 0 & 3 & 0 \\ 3 & 0 & 2 \\ 0 & 2 & 0 \end{pmatrix}$$

Example 0.0.50

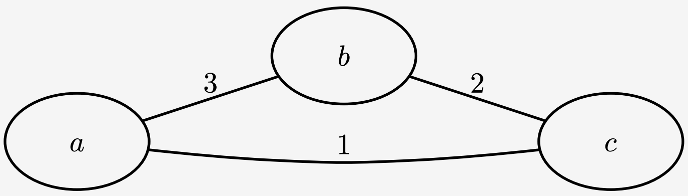
A **complete graph** is a graph where every pair of vertices is connected by a distinct edge. A complete graph with n vertices has $\frac{n(n-1)}{2}$ edges and is denoted as K_n.

Definition 0.0.50

8.1 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a problem where a salesman must visit all cities in a list exactly once and return to the starting city. The goal is to find the shortest path that visits all cities.

Given the following weighted graph $G = (V, E)$ with $V = \{a, b, c\}$ and $E = \{\{a, b\}, \{b, c\}, \{a, c\}\}$:



The shortest path that visits all cities is a, c, b with a total weight of 3.

Example 0.0.51

The Traveling Salesman Problem is NP-hard, meaning that there is no known polynomial-time algorithm to solve it.

Statement 0.0.26

9 Minimum Spanning Trees

A **minimum spanning tree** (MST) of a weighted graph $G = (V, E)$ is a spanning tree with the smallest possible sum of edge weights.

Definition 0.0.51

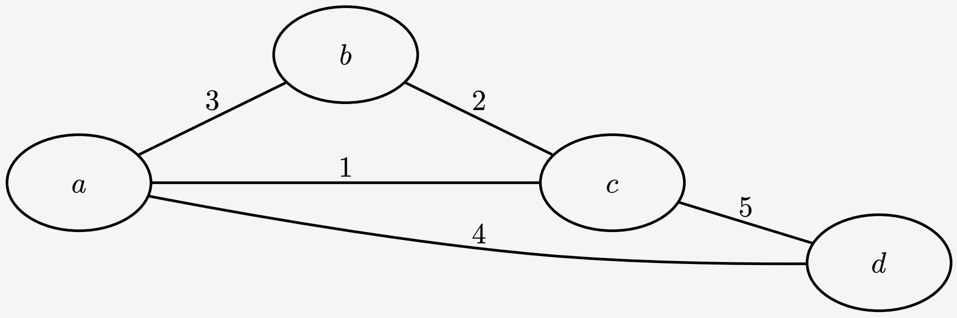
9.1 Kruskal's Algorithm

Kruskal's algorithm is an algorithm used to find the minimum spanning tree of a graph.

Algorithm:

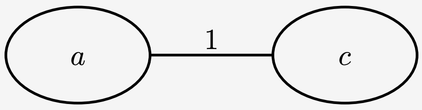
1. Sort all edges by weight.
2. Start with an empty graph T.
3. For each edge e in the sorted list:
   - If adding e to T does not create a cycle (i.e., the endpoints of e are not already connected in T), add e to T.
4. Repeat step 3 until T has $|V| - 1$ edges.

Given the following weighted graph:

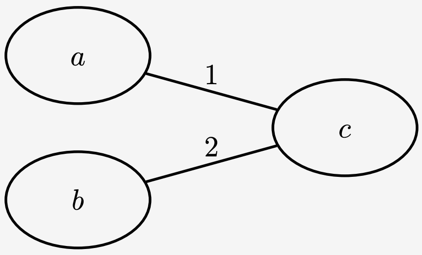


1. sorted edges: $\{\{a,c\},\{b,c\},\{a,b\},\{a,d\},\{c,d\}\}$
2. start with an empty graph T:

3. add $\{a,c\}$ to T:

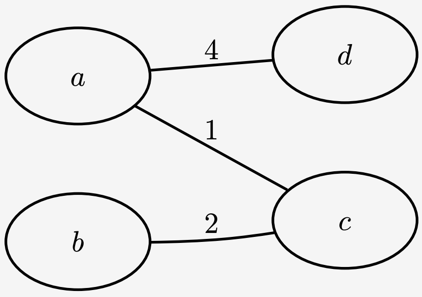


4. add $\{b,c\}$ to T:



do **not** add $\{a,b\}$ to T because it would create a cycle

5. add $\{a,d\}$ to T:



The minimum spanning tree is $\{\{a,c\},\{b,c\},\{a,d\}\}$. The total weight is 7.

Example 0.0.52

9.2 Dijkstra’s Algorithm

Dijkstra’s algorithm is an algorithm used to find the shortest path in a weighted graph $G = (V, E)$ between a source vertex s and any other destination vertex v.

Algorithm:

1. Start at the source vertex s.
2. Set the distance to s as 0 and all other distances as ∞.
3. While there are unvisited vertices:
   - Take the unvisited vertex u with the smallest distance.
   - For each neighbor v of u:
     - If the distance to u plus the weight of the edge $u - v$ is smaller than the current distance to v, update the distance to v.
4. Repeat step 3 until all vertices are visited.

10 Computability Theory

10.1 Computability

The field of computability theory is concerned with the study of what can be computed and how efficiently it can be computed. It is a branch of mathematical logic and theoretical computer science that focuses on the limits of computability. Computability theory examines the existence and nature of algorithms and the extent to which problems can be solved algorithmically.

10.1.1 Partial Function

A partial function is a function that is not defined for all possible inputs.

Partial functions can be denoted as:

- $f : X \rightharpoonup Y$
- $f : X \nrightarrow Y$
- $f : X \hookrightarrow Y$

For a partial function $f : X \rightharpoonup Y$, and any $x \in X$, one has either:

- $f(x) = y \in Y$ (its a single element in Y)
- $f(x)$ is undefined (it is not defined for this input)

Definition 0.0.52

The partial function $f : \mathbb{R} \rightharpoonup \mathbb{R}, x \mapsto \frac{1}{x}$ is not defined for $x = 0$.

Example 0.0.53

10.1.2 Register Machine

A register machine is a theoretical model of computation that consists of a set of registers and a set of instructions that operate on these registers.

A register machine over an alphabet $\Gamma = \{a_1, ..., a_L\}$ is a computational model consisting of R registers $R_0, ..., R_{R-1}$, each of which contains a word from Γ^*.

The following operations are possible on the registers:

- Check if a register is empty.
- Read the top character of a register.
- Delete the top character of a register.
- Add a character to the top of a register.

A program for a register machine consists of a sequence of instructions that specify the operations to be performed on the registers.

Command	**Description**
STOP	The program halts execution.
GOTO m	The program jumps to the command with label m.
SWITCH r m0 m1 … mL	If register r is empty, the program jumps to the command with label m0. If the top character of Rr is equal to ai, the program jumps to mi.
POP r m0 m1 … mL	Same as the SWITCH command, but additionally removes the last character from register Rr, if present.

10.1.3 Turing Machine

A Turing machine is a theoretical model of computation that consists of an infinite tape divided into cells, a read/write head that moves along the tape, and a finite set of states.

A formal definition of a Turing machine consists of a 7-tuple $M = (Q, \Gamma, b, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set of states.
- Γ is a finite set of tape symbols.
- $b \in \Gamma$ is the blank symbol.
- $\Sigma \subseteq \Gamma$ is the set of input symbols.
- $\delta : Q \times \Gamma \hookrightarrow Q \times \Gamma \times \{L, R\}$ is the transition function.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of final states.

Given an input string of 0s and 1s, a Turing machine can be designed to check if the number of 1s is even.

Given the input string 10101, we could define the following turning machine:

Current State	**Read Symbol**	**Write Symbol**	**Move**	**Next State**
start	0	0	R	start
start	1	1	R	Odd
Odd	0	0	R	Odd
Odd	1	1	R	Even
Even	0	0	R	Even
Even	1	1	R	Odd
Even	Blank	Blank	Stay	Accept
Odd	Blank	Blank	Stay	Reject

Where:

$Q = \{\text{start}, \text{Odd}, \text{Even}, \text{Accept}, \text{Reject}\}$
$\Gamma = \{0, 1, \text{Blank}\}$
$\Sigma = \{0, 1\}$
$b = \text{Blank}$
δ is the transition function as defined in the table above
$q_0 = \text{start}$
$F = \{\text{Accept}, \text{Reject}\}$

Example 0.0.54

10.2 P and NP

The classes P and NP are fundamental complexity classes in computational complexity theory.

10.2.1 Polynomial Time

A polynomial time algorithm is an algorithm that runs in polynomial time with respect to the size of the input.

A polynomial time algorithm is said to run in polynomial time if there exists a polynomial $p(n)$ such that the algorithm runs in $O(p(n))$ time, where n is the size of the input.

An algorithm runs in polynomial time if there exists a polynomial $p(n)$ such that the algorithm runs in $O(p(n))$ time.

Statement 0.0.27

An algorithm that runs in $O(n^2)$ time is said to run in polynomial time.

Example 0.0.55

10.2.2 P

The class P consists of decision problems that can be solved in polynomial time by a deterministic Turing machine.

A decision problem is in P if there exists an algorithm that solves it in polynomial time, where the running time of the algorithm is bounded by a polynomial in the size of the input.

A decision problem is in P if there exists a deterministic Turing machine that solves it in polynomial time.

Definition 0.0.53

A P problem could be: "Given n Keys and n Locks, find out which key fits which lock."

This problem can be solved in $O(n^2)$ time by trying all possible combinations.

Example 0.0.56

All problems in P also belong to the class NP.

Statement 0.0.28

10.2.3 NP

The class NP consists of decision problems for which a proposed solution can be verified in polynomial time by a deterministic Turing machine, even if the solution itself cannot be found in polynomial time.

An NP problem could be: "Given a graph G and an integer k, is there a clique of size k in G?"

While finding the clique itself may be difficult, verifying that a given set of vertices forms a clique can be done in polynomial time.

Example 0.0.57

10.2.4 NP-Complete

In computational complexity theory, a problem is NP-complete when:

1. It is a decision problem, meaning that for any input to the problem, the output is either "yes" or "no".
2. When the answer is "yes", this can be demonstrated through the existence of a short (polynomial length) solution.
3. The correctness of each solution can be verified quickly (namely, in polynomial time) and a brute-force search algorithm can find a solution by trying all possible solutions.
4. The problem can be used to simulate every other problem for which we can verify quickly that a solution is correct. In this sense, NP-complete problems are the hardest of the problems to which solutions can be verified quickly. If we could find solutions of some NP-complete problem quickly, we could quickly find the solutions of every other problem to which a given solution can be easily verified.

11 Scripts

11.1 Relation

11.1.1 Cartesian product

```py
import itertools

def kartesisches_prdukt(myList):
  """
  Berechnet und gibt das kartesische Produkt der Eingabeliste von Iterablen aus.

  Diese Funktion verwendet `itertools.product`, um das kartesische Produkt aller
  Iterablen in der Eingabeliste `myList` zu berechnen. Jede Kombination wird
  als Tupel ausgegeben.

  Argumente:
    myList (Liste von Iterablen): Eine Liste, die die Iterablen enthält,
            für die das kartesische Produkt berechnet wird.

  Beispiel:
    >>> myList = [[1, 2], ['a', 'b']]
    >>> kartesisches_prdukt(myList)
    (1, 'a')
    (1, 'b')
    (2, 'a')
    (2, 'b')

  Hinweis:
    - Jedes Element in `myList` sollte eine Iterabel sein.
    - Die Funktion gibt die Ergebnisse direkt aus, anstatt sie zurückzugeben.
  """
  for element in itertools.product(*myList):
    print(element)
```

11.1.2 Inverse Relation

```py
def inverse_relation(relation):
  """
  Computes the inverse of a given relation.

  Args:
    relation (set of tuple): A set of ordered pairs representing the relation.

  Returns:
    set of tuple: A set of ordered pairs where each pair (a, b) in the input
       is swapped to (b, a).

  Example:
    >>> relation = {(1, 2), (3, 4), (5, 6)}
    >>> inverse_relation(relation)
    {(2, 1), (4, 3), (6, 5)}
  """
  return {(b, a) for a, b in relation}
```

11.1.3 Compose Relation

```py
def compose_relations(R, S):
    """
    Computes the composition of two relations R and S.

    Args:
      R (set of tuple): The first relation, a set of ordered pairs (a, b).
      S (set of tuple): The second relation, a set of ordered pairs (b, c).

    Returns:
      set of tuple: A set of ordered pairs (a, c) such that there exists a b
          where (a, b) is in R and (b, c) is in S.

    Example:
      >>> R = {(1, 2), (2, 3)}
      >>> S = {(2, 4), (3, 5)}
      >>> compose_relations(R, S)
      {(1, 4), (2, 5)}
    """
    return {(a, c) for (a, b1) in R for (b2, c) in S if b1 == b2}
```

11.1.4 Plot relation graph

```py
import networkx as nx
import matplotlib.pyplot as plt

def plot_relation_graph(relation):
    """
    Plots a directed graph representing the given relation.

    Args:
      relation (set of tuple): A set of ordered pairs representing the relation.

    Example:
      >>> relation = {(1, 2), (2, 3), (1, 3)}
      >>> plot_relation_graph(relation)
    """
    # Create a directed graph
    graph = nx.DiGraph()

    # Add edges for each pair in the relation
    graph.add_edges_from(relation)

    # Draw the graph
    pos = nx.spring_layout(graph) # Position the nodes
    nx.draw(graph, pos, with_labels=True, node_size=2000, node_color="lightblue", arrowsize=20)
    plt.title("Directed Graph Representation of Relation")
    plt.show()
```

11.2 Logic

11.2.1 Create truth table

```py
import sympy as sp
from sympy.logic.boolalg import And, Or, Not, Implies

def create_truth_table(expression, variables):
    """
    Erzeugt eine Wahrheitstabelle für einen gegebenen logischen Ausdruck und druckt diese.

    Diese Funktion generiert alle möglichen Kombinationen von Wahrheitswerten
    für die angegebenen Variablen, berechnet das Ergebnis des logischen
    Ausdrucks für jede Kombination und gibt die Wahrheitstabelle aus.

    Parameter:
    expression (sympy.Expr): Der logische Ausdruck, für den die Wahrheitstabelle erstellt wird.
    variables (list): Eine Liste von sympy-Variablen, die im logischen Ausdruck verwendet werden.

    Beispiel:

    >>> A, B, C = sp.symbols('A B C')
    >>> expression = And(A, Or(B, C))
    >>> create_truth_table(expression, [A, B, C])
    """
    # Erstelle eine Liste aller möglichen Kombinationen der Wahrheitswerte
    num_vars = len(variables)

    # Header für die Tabelle
    print(" ".join([str(var) for var in variables] + ["Result"]))

    # Generiere alle möglichen Kombinationen der Wahrheitswerte
    for i in range(2**num_vars):
        # Erstelle eine Zuordnung von Wahrheitswerten für jede Kombination
        row = [(variables[j], (i >> j) & 1) for j in range(num_vars)]
        values = dict(row)
        # Berechne das Ergebnis des Ausdrucks für die gegebene Kombination
        result = str(expression.subs(values))

        print(" ".join([str(values[var]) for var in variables] + [result]))
```

11.2.2 KNF and DNF

```py
import itertools

def truth_table(variables, func):
    """Generates a truth table for the given Boolean function."""
    n = len(variables)
    table = []
    for values in itertools.product([0, 1], repeat=n):
        table.append((*values, func(*values)))
    return table

def get_dnf(variables, table):
    """Generates the Disjunctive Normal Form (DNF)."""
    terms = []
    for row in table:
        if row[-1] == 1:  # Include rows where the function is True
            term = " ∧ ".join(f"{var}" if val else f"¬{var}" for var, val in zip(variables, row[:-1]))
            terms.append(f"({term})")
    return " ∨ ".join(terms) if terms else "False"

def get_cnf(variables, table):
    """Generates the Conjunctive Normal Form (CNF)."""
    terms = []
    for row in table:
        if row[-1] == 0:  # Include rows where the function is False
            term = " ∨ ".join(f"¬{var}" if val else f"{var}" for var, val in zip(variables, row[:-1]))
            terms.append(f"({term})")
    return " ∧ ".join(terms) if terms else "True"

def boolean_normal_forms(variables, func):
    """Generates both DNF and CNF from a Boolean function."""
    table = truth_table(variables, func)
    dnf = get_dnf(variables, table)
    cnf = get_cnf(variables, table)

    print("Truth Table:")
    for row in table:
        print(row)

    print("\nDisjunctive Normal Form (DNF):", dnf)
    print("Conjunctive Normal Form (CNF):", cnf)

# Example usage
variables = ["A", "B", "C"]
def example_func(A, B, C):
    return (A and not B) or C

boolean_normal_forms(variables, example_func)
```

11.3 Modular arithmetic

11.3.1 additive inverse

```py
def additive_inverse(a, m):
    """
    Computes the additive inverse of a modulo m.

    Args:
        a (int): The number whose additive inverse is to be found.
        m (int): The modulus.

    Returns:
        int: The additive inverse of a modulo m.

    Example:
        >>> additive_inverse(3, 7)
        4
    """
    return (-a) % m
```

11.3.2 Multiplicative inverse

```py
def multiplicative_inverse(a, m):
    """
    Computes the multiplicative inverse of a modulo m using the extended Euclidean algorithm.

    Args:
        a (int): The number whose multiplicative inverse is to be found.
        m (int): The modulus.

    Returns:
        int: The multiplicative inverse of a modulo m, or raises ValueError if no inverse exists.

    Example:
        >>> multiplicative_inverse(3, 7)
        5
    """
    # Extended Euclidean Algorithm
    t, new_t = 0, 1
    r, new_r = m, a

    while new_r ≠ 0:
        quotient = r // new_r
        t, new_t = new_t, t - quotient * new_t
        r, new_r = new_r, r - quotient * new_r

    if r > 1:
        raise ValueError(f"{a} has no multiplicative inverse modulo {m}")
    if t < 0:
        t = t + m

    return t
```

11.4 Number theory

11.4.1 GCD (Euclidean algorithm)

```python
def gcd_with_steps(a, b):
    """
    Computes the Greatest Common Divisor (GCD) of two integers using the Euclidean Algorithm.
    Outputs each step of the algorithm.

    Args:
      a (int): The first integer.
      b (int): The second integer.

    Returns:
      int: The greatest common divisor of a and b.

    Example:
      >>> gcd_with_steps(400, 225)
      Step 1: gcd(400, 225) -> 400 % 225 = 175
      Step 2: gcd(225, 175) -> 225 % 175 = 50
      Step 3: gcd(175, 50) -> 175 % 50 = 25
      Step 4: gcd(50, 25) -> 50 % 25 = 0
      GCD is 25
    """
    step = 1
    while b != 0:
      print(f"Schritt {step}: gcd({a}, {b}) -> {a} % {b} = {a % b}")
      a, b = b, a % b
      step += 1
    print(f"GCD ist {a}")
    return a
```

11.4.2 Extended Euclidean algorithm

```python
def extended_gcd(a, b):
    """
    Computes the Greatest Common Divisor (GCD) of a and b using the extended Euclidean algorithm.
    Outputs each step of the algorithm, including coefficients for Bézout's identity.

    Bézout's identity: ax + by = gcd(a, b)

    Args:
      a (int): The first integer.
      b (int): The second integer.

    Returns:
      tuple: (gcd, x, y), where gcd is the greatest common divisor of a and b,
         and x, y are the Bézout coefficients.

    Example:
      >>> extended_gcd(240, 46)
      Step 1: r = 240, r_next = 46, q = 5, x = 1, x_next = 0, y = 0, y_next = 1
      Step 2: r = 46, r_next = 10, q = 4, x = 0, x_next = 1, y = 1, y_next = -5
      Step 3: r = 10, r_next = 6, q = 1, x = 1, x_next = -4, y = -5, y_next = 21
      Step 4: r = 6, r_next = 4, q = 1, x = -4, x_next = 5, y = 21, y_next = -26
      Step 5: r = 4, r_next = 2, q = 2, x = 5, x_next = -14, y = -26, y_next = 73
      Step 6: r = 2, r_next = 0, q = 2, x = -14, x_next = 33, y = 73, y_next = -172
      GCD is 2, x = -14, y = 73
    """
    # Initial values for the extended algorithm
    x, y, x_next, y_next = 1, 0, 0, 1
    step = 1

    print(f"Initial values: a = {a}, b = {b}")
    while b != 0:
      q = a // b  # Quotient
      r = a % b  # Remainder
      print(f"Step {step}: r = {a}, r_next = {b}, q = {q}, x = {x}, x_next = {x_next}, y = {y}, y_next = {y_next}")

      # Update values
      a, b = b, r
      x, x_next = x_next, x - q * x_next
      y, y_next = y_next, y - q * y_next
      step += 1

    print(f"GCD is {a}, x = {x}, y = {y}")
    return a, x, y
```

11.4.3 Fermat’s little theorem

```py
def fermat_little_theorem(a, p):
    """
    Verifies Fermat's Little Theorem: a^(p-1) ≡ 1 (mod p) where p is a prime number.

    Args:
        a (int): The integer to test.
        p (int): A prime number.

    Returns:
        bool: True if Fermat's Little Theorem holds for a and p, otherwise False.
    """
    if p <= 1:
        raise ValueError("p must be a prime number greater than 1.")

    # Check if p is prime
    if not is_prime(p):
        raise ValueError(f"{p} is not a prime number.")

    # Apply Fermat's Little Theorem: a^(p-1) % p should be 1
    return pow(a, p - 1, p) == 1
```

11.4.4 Is Prime

```py
def is_prime(n):
    """Helper function to check if a number is prime."""
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

11.4.5 Prime factors

```py
def prime_factors(n):
    """
    Berechnet die Primfaktorzerlegung einer Zahl.

    Args:
        n (int): Die Zahl, die zerlegt werden soll.

    Returns:
        list: Eine Liste von Primfaktoren, die n bilden.
    """
    factors = []

    # Zuerst alle 2er Faktoren herausnehmen
    while n % 2 == 0:
        factors.append(2)
        n = n // 2

    # Danach die ungeraden Zahlen von 3 bis √n prüfen
    for i in range(3, int(n**0.5) + 1, 2):
        while n % i == 0:
            factors.append(i)
            n = n // i

    # Wenn n immer noch größer als 2 ist, dann ist n ein Primfaktor
    if n > 2:
        factors.append(n)

    return factors
```

11.5 Graph theory

11.5.1 Sum of degree

```py
def sum_of_degrees(graph):
    """
    Berechnet die Summe der Grade aller Knoten in einem Graphen.

    Der Grad eines Knotens ist die Anzahl der Kanten, die mit diesem Knoten verbunden sind.
    Die Summe der Grade aller Knoten ist gleich der doppelten Anzahl der Kanten.

    Args:
      graph (dict): Ein Graph als Adjazenzliste, wobei die Schlüssel die Knoten sind
          und die Werte Listen von benachbarten Knoten darstellen.

    Returns:
      int: Die Summe der Grade aller Knoten.

    Beispiel:
      >>> graph = {
      >>>   1: [2, 3],
      >>>   2: [1, 3],
      >>>   3: [1, 2]
      >>> }
      >>> sum_of_degrees(graph)
      6
    """
    degree_sum = 0
    for vertex in graph:
      degree_sum += len(graph[vertex]) # Anzahl der Nachbarn (Grad des Knotens)
    return degree_sum
```

11.5.2 number of edges

```py
def number_of_edges(graph):
    """
    Berechnet die Anzahl der Kanten in einem ungerichteten Graphen.

    Die Anzahl der Kanten ist die Hälfte der Summe der Grade aller Knoten,
    da jede Kante in der Adjazenzliste zweimal gezählt wird.

    Args:
      graph (dict): Ein Graph als Adjazenzliste.

    Returns:
      int: Die Anzahl der Kanten im Graphen.

    Beispiel:
      >>> graph = {
      >>>   1: [2, 3],
      >>>   2: [1, 3],
      >>>   3: [1, 2]
      >>> }
      >>> number_of_edges(graph)
      3
    """
    return sum_of_degrees(graph) // 2 # Jede Kante wird zweimal gezählt
```

11.6 Find Cycle

```py
def find_cycle(graph):
    """
    Findet einen Zyklus in einem ungerichteten Graphen und gibt diesen aus, wenn er existiert.

    Der Zyklus wird als Liste von Knoten ausgegeben, beginnend und endend beim gleichen Knoten.

    Args:
        graph (dict): Ein Graph als Adjazenzliste, wobei die Schlüssel die Knoten sind
            und die Werte Listen von benachbarten Knoten darstellen.

    Returns:
        list: Eine Liste von Knoten, die den Zyklus bilden, oder None, wenn kein Zyklus gefunden wird.

    Beispiel:
        >>> graph = {
        >>>     1: [2, 3],
        >>>     2: [1, 3],
        >>>     3: [1, 2]
        >>> }
        >>> find_cycle(graph)
        [3, 2, 1, 3]
    """

    def dfs(v, parent, visited, path):
        visited[v] = True
        path.append(v)

        for neighbor in graph[v]:
            if not visited[neighbor]:
                if dfs(neighbor, v, visited, path):
                    return True
            # Rückkante gefunden
            elif neighbor != parent:
                cycle_index = path.index(neighbor)
                cycle = path[cycle_index:] + [neighbor]
                print(f"Cycle detected: {cycle}")
                return True

        path.pop()
        return False

    visited = {node: False for node in graph}
    for node in graph:
        if not visited[node]:
            path = []
            if dfs(node, None, visited, path):
                return path
    return None
```

11.6.1 BFS

```py
from collections import deque

def bfs(graph, start):
    """
    Perform a Breadth-First Search (BFS) on a graph and print each step.

    Args:
        graph (dict): A graph represented as an adjacency list.
        start (int): The starting node for BFS.
    """
    visited = set()  # Set to keep track of visited nodes
    queue = deque([start])  # Initialize the queue with the start node

    print(f"Starting BFS from node {start}")

    while queue:
        # Dequeue the next node to visit
        node = queue.popleft()
        print(f"Visiting node: {node}, Queue: {list(queue)}")

        if node not in visited:
            visited.add(node)  # Mark the node as visited

            # Add all unvisited neighbors to the queue
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
                    print(f"Adding neighbor {neighbor} to the queue.")

    print("BFS traversal complete.")
```

11.7 Cryptography

11.7.1 Generate RSA Key

```py
from sympy import isprime
# Hilfsfunktion zum Berechnen des größten gemeinsamen Teilers (GCD)
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Funktion zur Berechnung des modularen Inversen von a modulo m
def mod_inverse(a, m):
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

# Funktion zum Generieren von RSA-Schlüsselpaaren mit benutzerdefinierten p, q und e
def generate_rsa_keys_from_input(p, q, e=65537):
    # Sicherstellen, dass p und q Primzahlen sind
    if not isprime(p):
        raise ValueError(f"{p} ist keine Primzahl.")
    if not isprime(q):
        raise ValueError(f"{q} ist keine Primzahl.")

    print(f"Schritt 1: Berechne n = p * q")
    # Berechnung von n = p * q
    n = p * q
    print(f"n = {p} * {q} = {n}")

    # Berechnung von phi(n) = (p-1) * (q-1)
    print(f"Schritt 2: Berechne phi(n) = (p-1) * (q-1)")
    phi_n = (p - 1) * (q - 1)
    print(f"phi(n) = ({p} - 1) * ({q} - 1) = {phi_n}")

    # Sicherstellen, dass e teilerfremd zu phi(n) ist
    print(f"Schritt 3: Überprüfe, ob e = {e} teilerfremd zu phi(n) ist")
    if gcd(e, phi_n) != 1:
        raise ValueError(f"{e} ist nicht teilerfremd zu phi(n). Wählen Sie einen anderen Wert für e.")
    print(f"e = {e} ist teilerfremd zu phi(n).")

    # Berechnung von d, dem privaten Exponenten
    print(f"Schritt 4: Berechne d als das modulare Inverse von e modulo phi(n)")
    d = mod_inverse(e, phi_n)
    print(f"d = {d} (modularer Inverser von {e} mod {phi_n})")

    # Rückgabe des Schlüsselpaares
    public_key = (e, n)
    private_key = (d, n)

    print(f"Öffentlicher Schlüssel: ({e}, {n})")
    print(f"Privater Schlüssel: ({d}, {n})")

    return public_key, private_key
```

11.7.2 RSA encrypt

```py
# RSA Verschlüsselung: Verschlüsselt eine Nachricht mit dem öffentlichen Schlüssel
def encrypt(public_key, message):
  """
  Verschlüsselt eine Nachricht mit dem öffentlichen Schlüssel (e, n).

  Args:
    public_key: Ein Tupel (e, n), wobei e der öffentliche Exponent und n der Modulus ist.
    message: Die Nachricht (als ganze Zahl), die verschlüsselt werden soll.

  Returns:
    encrypted_message: Die verschlüsselte Nachricht.

  Beispiel:
    public_key = (17, 3233)
    message = 123
    encrypted_message = encrypt(public_key, message)
    print(encrypted_message) # Gibt die verschlüsselte Nachricht aus
  """
  e, n = public_key
  print(f"Verschlüsselung: (M^e) mod n = ({message}^{e}) mod {n}")
  encrypted_message = pow(message, e, n)
  return encrypted_message
```

11.7.3 RSA decrypt

```py
# RSA Entschlüsselung: Entschlüsselt eine Nachricht mit dem privaten Schlüssel
def decrypt(private_key, encrypted_message):
  """
  Entschlüsselt eine Nachricht mit dem privaten Schlüssel (d, n).

  Args:
    private_key: Ein Tupel (d, n), wobei d der private Exponent und n der Modulus ist.
    encrypted_message: Die verschlüsselte Nachricht, die entschlüsselt werden soll.

  Returns:
    decrypted_message: Die entschlüsselte Nachricht.

  Beispiel:
    private_key = (2753, 3233)
    encrypted_message = 2202
    decrypted_message = decrypt(private_key, encrypted_message)
    print(decrypted_message) # Gibt die entschlüsselte Nachricht aus
  """
  d, n = private_key
  print(f"Entschlüsselung: (C^d) mod n = ({encrypted_message}^{d}) mod {n}")
  decrypted_message = pow(encrypted_message, d, n)
  return decrypted_message
```

11.7.4 Diffie-Hellman

```python
# Hilfsfunktion zur Berechnung der Potenzmodulo
def power_mod(base, exponent, modulus):
    return pow(base, exponent, modulus)

# Diffie-Hellman-Schlüsselaustausch
def diffie_hellman(p, g, private_key_a, private_key_b):
    # A und B berechnen jeweils ihre öffentlichen Schlüssel
    public_key_a = power_mod(g, private_key_a, p)
    public_key_b = power_mod(g, private_key_b, p)

    # Austausch der öffentlichen Schlüssel (öffentlich)
    print(f"Öffentlicher Schlüssel A: {public_key_a}")
    print(f"Öffentlicher Schlüssel B: {public_key_b}")

    # A berechnet den gemeinsamen Schlüssel mit B's öffentlichem Schlüssel
    shared_secret_a = power_mod(public_key_b, private_key_a, p)

    # B berechnet den gemeinsamen Schlüssel mit A's öffentlichem Schlüssel
    shared_secret_b = power_mod(public_key_a, private_key_b, p)

    # Überprüfen, ob der gemeinsame Schlüssel korrekt ist
    if shared_secret_a == shared_secret_b:
        print(f"Der gemeinsame geheime Schlüssel ist: {shared_secret_a}")
        return shared_secret_a
    else:
        print("Es gab einen Fehler bei der Berechnung des gemeinsamen Geheimnisses.")
        return None

# Beispiel für den Diffie-Hellman-Schlüsselaustausch
def example_diffie_hellman():
    # Definiere p (Primzahl) und g (Basis)
    p = 23  # Beispiel für eine kleine Primzahl
    g = 5  # Beispiel für eine primitive Basis

    # Zwei private Schlüssel für Alice (A) und Bob (B)
    private_key_a = 6
    private_key_b = 15

    # Berechne das gemeinsame Geheimnis
    shared_secret = diffie_hellman(p, g, private_key_a, private_key_b)
```

11.7.5 ElGamal

```python
import random
from sympy import mod_inverse

# ElGamal-Schlüsselgenerierung
def elgamal_keygen(p, g):
    """
    Schlüsselpaar für ElGamal generieren:
    p: Primzahl
    g: primitive Wurzel modulo p

    Gibt das öffentliche und private Schlüsselpaar zurück.
    """
    # Privater Schlüssel x
    x = random.randint(2, p-2)

    # Öffentlicher Schlüssel y = g^x mod p
    y = pow(g, x, p)

    # Rückgabe des Schlüsselpaares
    return (p, g, y), (p, g, x)

# ElGamal-Verschlüsselung
def elgamal_encrypt(public_key, m):
    """
    Verschlüsselt eine Nachricht m mit dem öffentlichen Schlüssel.

    public_key: (p, g, y)
    m: Die Nachricht (als ganze Zahl)

    Gibt den verschlüsselten Text (c1, c2) zurück.
    """
    p, g, y = public_key

    # Wähle eine zufällige Zahl k
    k = random.randint(2, p-2)

    # Berechne c1 = g^k mod p
    c1 = pow(g, k, p)

    # Berechne c2 = (m * y^k) mod p
    c2 = (m * pow(y, k, p)) % p

    # Rückgabe des verschlüsselten Texts
    return c1, c2

# ElGamal-Entschlüsselung
def elgamal_decrypt(private_key, ciphertext):
    """
    Entschlüsselt den verschlüsselten Text mit dem privaten Schlüssel.

    private_key: (p, g, x)
    ciphertext: (c1, c2)

    Gibt die entschlüsselte Nachricht zurück.
    """
    p, g, x = private_key
    c1, c2 = ciphertext

    # Berechne m = c2 * (c1^x)^-1 mod p
    s = pow(c1, x, p)  # Berechne c1^x mod p
    s_inv = mod_inverse(s, p)  # Berechne das Inverse von s modulo p

    m = (c2 * s_inv) % p  # Entschlüsselte Nachricht

    return m
```

```py
# Beispiel für den ElGamal-Verschlüsselungsprozess
def example_elgamal():
    # Definiere eine Primzahl p und eine primitive Wurzel g
    p = 23  # Beispiel für eine kleine Primzahl
    g = 5  # Beispiel für eine primitive Wurzel

    # Generiere das Schlüsselpaar
    public_key, private_key = elgamal_keygen(p, g)

    print(f"Öffentlicher Schlüssel: {public_key}")
    print(f"Privater Schlüssel: {private_key}")

    # Nachricht, die verschlüsselt werden soll
    message = 15

    print(f"Nachricht: {message}")

    # Verschlüsseln der Nachricht
    ciphertext = elgamal_encrypt(public_key, message)
    print(f"Verschlüsselter Text: {ciphertext}")

    # Entschlüsseln der Nachricht
    decrypted_message = elgamal_decrypt(private_key, ciphertext)
    print(f"Entschlüsselte Nachricht: {decrypted_message}")
```

11.8 Probability

11.8.1 binomial_term_count

```py
import math

def binomial_term_count(n=8, k=3):
    """
    Berechnet, wie oft ein spezifischer Term a^k * b^(n-k) beim Ausmultiplizieren von (a + b)^n vorkommt.

    Algorithmus: Binomialkoeffizient

    n: Exponent (Standard ist 8 für (a + b)^8)
    k: Potenz von a im gesuchten Term (Standard ist 3 für a^3 * b^5)

    Rückgabe:
    Anzahl der Vorkommen des Terms
    """
    return math.comb(n, k)
```

11.8.2 ereignismengen

```py
def ereignismengen():
    """
    Berechnet die Ereignismengen A, B und C basierend auf der Aufgabenstellung.

    Algorithmus: Mengentheorie und Teilbarkeit

    A: Zahlen von 1 bis 100, die durch 8 oder 9 teilbar sind.
    B: Zahlen von 1 bis 100, die durch 9 oder 15 teilbar sind.
    C: Schnittmenge von A und B

    Rückgabe:
    Tuple mit den Mengen A, B und C
    """
    A = {x for x in range(1, 101) if x % 8 == 0 or x % 9 == 0}
    B = {x for x in range(1, 101) if x % 9 == 0 or x % 15 == 0}
    C = A & B

    return A, B, C
```

11.8.3 bayes_wahrscheinlichkeit

```py
def bayes_wahrscheinlichkeit():
    """
    Berechnet die Wahrscheinlichkeit, dass ein kontrollierter Fahrgast Schwarzfahrer ist.

    Algorithmus: Satz von Bayes

    Rückgabe:
    Wahrscheinlichkeit P(S|K)
    """
    # Gegebene Wahrscheinlichkeiten
    P_S = 0.1  # Schwarzfahrer
    P_E = 0.9  # Ehrliche Fahrgäste
    P_K_given_S = 0.7  # Keine Fahrkarte bei Schwarzfahrern
    P_F_given_S = 0.3  # Gefälschte Fahrkarte bei Schwarzfahrern
    P_K_given_E = 0.05  # Keine Fahrkarte bei ehrlichen Fahrgästen

    # Gesamtwahrscheinlichkeit für kontrollierte Fahrgäste ohne Fahrkarte
    P_K = P_K_given_S * P_S + P_K_given_E * P_E

    # Wahrscheinlichkeit P(S|K)
    P_S_given_K = (P_K_given_S * P_S) / P_K

    return P_S_given_K
```

11.8.4 laplace_probability

```py
def laplace_probability(favorable_cases, total_cases):
    """
    Berechnet die Laplace-Wahrscheinlichkeit.
    :param favorable_cases: Anzahl der günstigen Fälle
    :param total_cases: Anzahl der möglichen Fälle
    :return: Wahrscheinlichkeit
    """
    return favorable_cases / total_cases
print(laplace_probability(1, 6)) # Würfel: Wahrscheinlichkeit für eine 6
```

11.8.5 combinations

```py
import math

def combinations(n, k):
    """
    Berechnet die Anzahl der Varianten (Kombinationen) von k Zügen aus n Kugeln ohne Zurücklegen.
    :param n: Gesamtanzahl der Kugeln
    :param k: Anzahl der zu ziehenden Kugeln
    :return: Anzahl der Kombinationen
    """
    if k > n:
        return 0 # Es können nicht mehr Kugeln gezogen werden als vorhanden
    return math.factorial(n) // (math.factorial(k) * math.factorial(n - k))

# Beispiel: Anzahl der Varianten bei 3 Zügen aus 6 Kugeln
print(combinations(6, 3)) # Ausgabe: 20
```

11.8.6 variations_with_replacement

```py
def variations_with_replacement(n, k):
    """
    Berechnet die Anzahl der Varianten (Ziehen mit Zurücklegen und Berücksichtigung der Reihenfolge).
    :param n: Gesamtanzahl der Kugeln
    :param k: Anzahl der Züge
    :return: Anzahl der Varianten
    """
    return n ** k

# Beispiel: Anzahl der Varianten bei 3 Zügen aus 6 Kugeln
print(variations_with_replacement(6, 3)) # Ausgabe: 216
```

11.8.7 conditional_probability

```py
def conditional_probability(P_A_and_B, P_B):
    """
    Berechnet die bedingte Wahrscheinlichkeit.
    :param P_A_and_B: Wahrscheinlichkeit von A und B
    :param P_B: Wahrscheinlichkeit von B
    :return: Bedingte Wahrscheinlichkeit
    """
    return P_A_and_B / P_B
print(conditional_probability(0.3, 0.5)) # Beispielwerte für P(A ∩ B) und P(B)
```

11.8.8 check_independence

```py
def check_independence(P_A, P_B, P_A_and_B):
    """
    Prüft, ob zwei Ereignisse unabhängig sind.
    :param P_A: Wahrscheinlichkeit von A
    :param P_B: Wahrscheinlichkeit von B
    :param P_A_and_B: Wahrscheinlichkeit von A und B
    :return: True, wenn unabhängig, sonst False
    """
    return P_A_and_B == P_A * P_B
print(check_independence(0.5, 0.4, 0.2)) # Prüft Unabhängigkeit
```

11.8.9 sensitivity

```py
def sensitivity(true_positives, actual_positives):
    """
    Berechnet die Sensitivität.
    :param true_positives: Anzahl der richtig positiven Fälle
    :param actual_positives: Anzahl aller tatsächlichen positiven Fälle
    :return: Sensitivität
    """
    return true_positives / actual_positives
print(sensitivity(90, 100)) # Sensitivität mit 90 von 100 erkannten Fällen
```

11.8.10 specificity

```py
def specificity(true_negatives, actual_negatives):
    """
    Berechnet die Spezifität.
    :param true_negatives: Anzahl der richtig negativen Fälle
    :param actual_negatives: Anzahl aller tatsächlichen negativen Fälle
    :return: Spezifität
    """
    return true_negatives / actual_negatives
print(specificity(180, 200)) # Spezifität mit 180 von 200 erkannten Gesunden
```