

# Algorithmen

WU

23. Dezember 2020

# Gliederung (1)

Algorithmusbegriff

Eigenschaften von Algorithmen

Darstellung von Algorithmen

Programmablaufpläne

Struktogramme

Pseudocode

Anwendungsaufgabe

# Gliederung (2)

## Rekursion und Iteration

Rekursion

Iteration

Effizienz und Komplexität

## Grenzen der Berechenbarkeit

# Algorithmus: Definition

## Ein Algorithmus

... ist eine eindeutige Handlungsvorschrift zur Lösung eines Problems (*oder einer Klasse von Problemen*), die aus einer endlichen Folge von Anweisungen besteht.

Das Wort „Algorithmus“ ist eine Abwandlung des Namens **MUHAMMED AL-CHWARIZMI** (ca. 780 – 850), der in einem seiner Werke viele Rechenverfahren beschrieben hat.

Die Abbildung zeigt ihn auf einer sowjetischen Briefmarke anlässlich seines 1200-jährigen Geburtsjubiläums.

Quelle: [https://de.wikipedia.org/wiki/Datei:1983\\_CPA\\_5426\\_\(1\).png](https://de.wikipedia.org/wiki/Datei:1983_CPA_5426_(1).png)  
(23. 12. 2020); Dieses Werk unterliegt keinem Urheberrecht.



# Algorithmus: Eigenschaften

Algorithmen sind durch einige wesentliche Eigenschaften gekennzeichnet:

## Determiniertheit

Der Algorithmus ist **determiniert**, wenn er bei jeder Ausführung mit gleichen Startbedingungen und Eingaben das gleiche Ergebnis liefert.

## Determinismus

Ein Algorithmus ist **deterministisch**, wenn zu jedem Zeitpunkt der Ausführung der nächste Schritt eindeutig definiert ist.

# Algorithmus: Eigenschaften

## Dabei gilt:

- ① Jeder deterministische Algorithmus ist determiniert,
- ② aber nicht jeder determinierte Algorithmus ist deterministisch.

## Beispiele:

- ▶ Der **euklidische Algorithmus** (Bestimmung des größten gemeinsamen Teilers zweier ganzer Zahlen) und der Sortieralgorithmus **Bubblesort** sind **deterministisch**.
- ▶ Der Sortieralgorithmus **Quicksort** nutzt ein zufällig ausgewähltes Element zum Zerlegen einer Liste in zwei Teillisten. Er ist damit **nicht deterministisch**.

# Algorithmus: Eigenschaften

## Finitheit

- Die Beschreibung des Algorithmus (der Quelltext) besitzt eine endliche Länge (⇒ statische Finitheit).
- Der Algorithmus darf zu jedem Zeitpunkt der Ausführung nur einen begrenzten Speicherplatz belegen (⇒ dynamische Finitheit).
- Der Algorithmus hält bei jeder möglichen Eingabe nach endlich vielen Schritten an (⇒ Terminiertheit).

**Halteproblem** ⇒ bewiesen durch Alan Turing

*Es gibt keinen Algorithmus, der feststellen kann, ob ein anderer Algorithmus terminiert ist.*

# Algorithmus: Eigenschaften

## Effektivität

Der Effekt jeder Anweisung eines Algorithmus muss eindeutig festgelegt sein.

# Darstellung von Algorithmen

In Alltagssituationen werden Algorithmen meist umgangssprachlich beschrieben (z. B. Fahranweisungen bei Navigationssystemen).

In der Informatik nutzt man häufig abstraktere grafische Darstellungen. Dazu gehören:

- ▶ Programmablaufpläne (nach DIN 66001 oder ISO 5807)
- ▶ NASSI-SHNEIDERMAN-Diagramme (bekannt als „Struktogramme“)
- ▶ Pseudocode (ähnelt natürlicher Sprache und höheren Programmiersprachen gemischt mit mathematischer Notation)

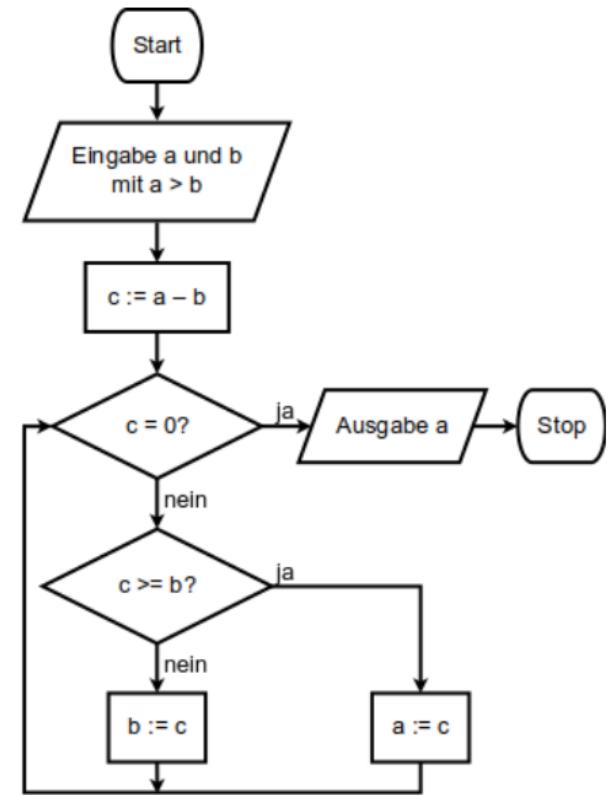
# Programmablaufpläne

## Ein Programmablaufplan (PAP)

ist ein Ablaufdiagramm für ein Computerprogramm. Es wird auch als Flussdiagramm (engl. flowchart) oder Programmstrukturplan bezeichnet.

Es beschreibt die Folge von Operationen zur Lösung einer Aufgabe.

Beispiel →



# Programmablaufpläne – Elemente

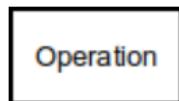
Kontrollpunkt:



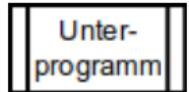
Verbindung:



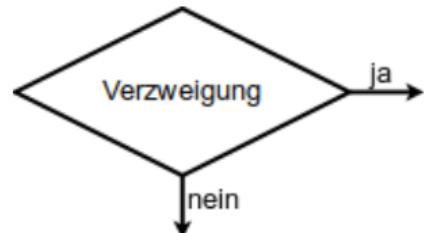
Operation:



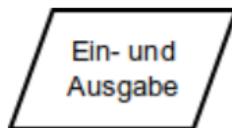
Unterprogramm ausführen:



Verzweigung:



Ein- und Ausgabe:



# NASSI-SHNEIDERMAN-Diagramm

... oder Struktogramm ist ein Diagrammtyp zur Darstellung von Programmentwürfen.

Es wurde 1972/73 von ISAAC NASSI und BEN SHNEIDERMAN entwickelt und ist in der DIN 66261 genormt.

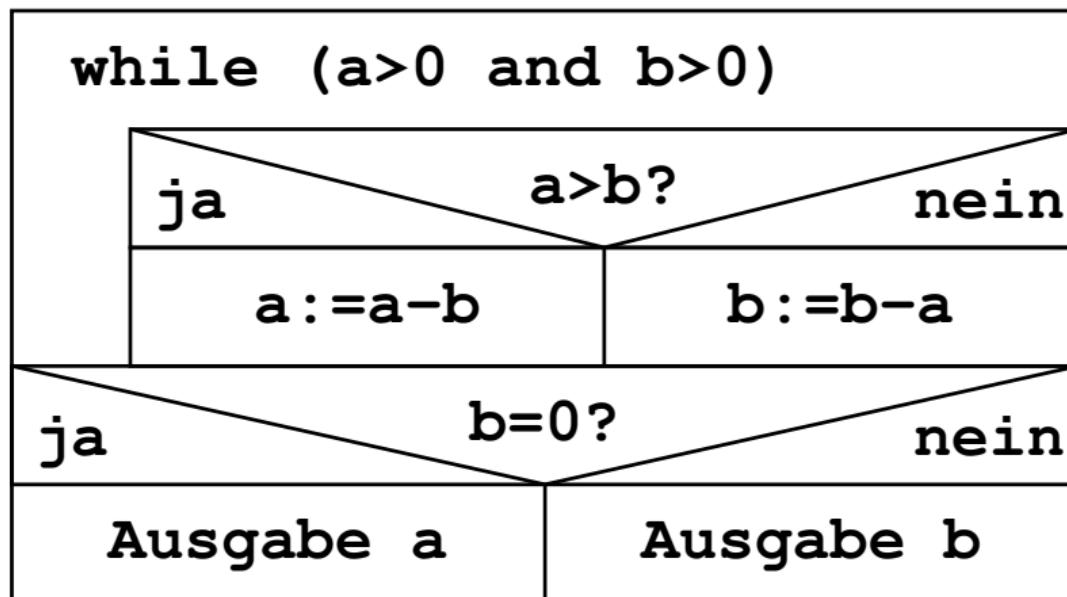
NASSI-SHNEIDERMAN-Diagramme können folgende nach **DIN 66261** definierte Strukturelemente enthalten:

<https://de.wikipedia.org/wiki/Nassi-Shneiderman-Diagramm>

⇒ *siehe auch Tafelwerk!*

# Nassi-Shneiderman-Diagramm

**Beispiel:** Euklidischer Algorithmus zur Berechnung des größten gemeinsamen Teilers zweier Zahlen



# Pseudocode

Mit Pseudocode kann ein Programmablauf unabhängig von einer konkreten Programmiersprache beschrieben werden. Dadurch ist er oft kompakter und leichter verständlich als realer Programmcode.

Häufig verwendete Schlüsselwörter siehe:

<https://de.wikipedia.org/wiki/Pseudocode>

# Pseudocode

**Beispiel:** Pseudocode für eine rekursive Variante des Euklidischen Algorithmus:

**EUCLID (a, b)**

- 1    **wenn**  $b = 0$
- 2        **dann return**  $a$
- 3        **sonst return** **EUCLID**( $b$ ,  $a \bmod b$ )

# Aufgaben

1. Informieren Sie sich über Funktionsweise und Varianten des Euklidischen Algorithmus.  
[http://de.wikipedia.org/wiki/Euklidischer\\_Algorithmus](http://de.wikipedia.org/wiki/Euklidischer_Algorithmus)
2. Wählen Sie eine iterative Variante aus und stellen Sie sie als NASSI-SHNEIDERMAN-Diagramm, Programmablaufplan und Pseudocode dar.
3. Programmieren und testen Sie ihre Variante mit Lazarus (zwei Eingabefelder, ein Ausgabefeld).
4. Ergänzen Sie ihr Programm um eine Ausgabe der Zwischenergebnisse. Verwenden Sie dazu z. B. die Komponente TMemo. Mit `memo1.Lines.Add(' STRING')` können Sie Strings zeilenweise anzeigen.

# Rekursion und Iteration

Algorithmen können beim Programmieren auf unterschiedliche Art und Weise implementiert werden:

- ▶ **rekursiv**, d. h. eine Prozedur, Funktion oder Methode **ruft sich** im Programm **selbst wieder auf**.
- ▶ **iterativ**, d. h. durch **Schleifen** werden Anweisungen oder Anweisungsblöcke **wiederholt ausgeführt**.

*Rekursive Algorithmen können prinzipiell auch iterativ implementiert werden und umgekehrt.*

# Rekursion

## Rekursion

... (von lateinisch „recurrere“ – „zurücklaufen“) ist eine Technik, eine Funktion durch sich selbst zu definieren.

### Beispiel: mathematische Definition der Fakultät

- Die Fakultät der Zahl 0 ist als 1 definiert.
- Die Fakultät einer ganzen Zahl größer Null ist das Produkt dieser Zahl mit der **Fakultät** der nächstkleineren ganzen Zahl.

*Die nächste Folie zeigt die Funktionsweise dieser Definition am Beispiel der Berechnung der Fakultät von 3.*

# Beispiel: Fakultät von 3

für Fakultät von 3 muss man Fakultät von 2 berechnen und mit 3 multiplizieren

    für Fakultät von 2 muss man Fakultät von 1 berechnen und mit 2 multiplizieren

        für Fakultät von 1 muss man Fakultät von 0 berechnen und mit 1 multiplizieren

    Fakultät von 0 ist als 1 definiert

    Fakultät von 1 ist also  $1 \cdot 1 = 1$

    Fakultät von 2 ist also  $1 \cdot 1 \cdot 2 = 2$

    Fakultät von 3 ist also  $1 \cdot 1 \cdot 2 \cdot 3 = 6$

# Implementation (Pascal)

```
function fac(x : Integer) : Integer;  
begin  
  if x = 0 then fac := 1  
    else fac := x * fac(x - 1);  
end;
```

## Aufgaben:

1. Implementieren Sie diesen Algorithmus mit Lazarus.
2. Testen Sie schrittweise größere Eingabewerte.
3. Variieren Sie das Programm mit anderen  
Ganzzahlen-Datentypen und testen Sie erneut.

# Rekursion

## Vorteile:

- ▶ ermöglicht elegante Problemlösungen durch Zurückführen einer Aufgabe auf eine einfachere gleichartige Aufgabe
- ▶ → dadurch einfach und leicht verständlich

## Nachteile

- ▶ geringe Performance und hoher Arbeitsspeicherverbrauch (durch Speichern aller Zwischenergebnisse)
- ▶ Stapel- oder Pufferüberläufe (buffer overflow)
- ▶ fehlerhafte Abbruchbedingung erzeugt infiniten Regress (→ Endlosrekursion)

# Iteration

## Iteration

... (von englisch „to iterate“ – „wiederholen“) verwendet Schleifen anstelle von Selbstaufrufen (➡ *Rekursion*).

## Vorteile

- ▶ Stapelüberläufe (➡ *Nachteil der Rekursion*) werden verhindert
- ▶ schnellere Programmabarbeitung, da die Methodenaufrufe entfallen

## Nachteile

- ▶ aufwändiger als Rekursion
- ▶ ➡ dadurch evtl. schwerer verständlich

# Beispiel: Fakultät

```
function fac(x : integer) : integer;
var i, ergebn : integer;
begin
  ergebn := 1;
  for i := 1 to x do ergebn := ergebn * i;
  fac := ergebn;
end;
```

## Aufgabe

Implementieren Sie diesen Algorithmus mit Lazarus. Nutzen Sie eine analoge Oberfläche wie bei der Rekursion. Testen Sie ebenfalls mit verschiedenen Eingaben und Ganzzahlen-Datentypen.

# Komplexaufgabe

Implementieren Sie den Algorithmus zur Berechnung der Fibonacci-Folge mit Lazarus.

Das Programm soll eine rekursive sowie eine iterative Implementierung beinhalten.

Das Programm kann auf Wunsch benotet werden. Dazu sollten folgende Anforderungen erfüllt werden:

- ▶ übersichtliche Oberfläche mit Eingabe der Elementnummer
- ▶ Buttons für rekursive und iterative Berechnung
- ▶ Ausgabe des Elements und der gesamten Zahlenfolge (getrennt für Rekursion und Iteration)
- ▶ übersichtlicher und kommentierter Quelltext

# Effizienz und Komplexität

## Aufgabe

Testen Sie das Programm für unterschiedliche Elemente.

Welchen Unterschied können Sie bei größeren Elementen zwischen rekursivem und iterativem Aufruf feststellen?

*Die maximale Elementnummer liegt bei Benutzung des Datentyps „int64“ für das Berechnungsergebnis bei 92!*

## Ursache:

- die Anzahl der rekursiven Funktionsaufrufe nimmt mit steigender Elementzahl **exponentiell** zu
- bei der iterativen Berechnung wächst die Anzahl der Berechnungen nur **linear** bei steigender Elementzahl

# Fibonacci, rekursiv

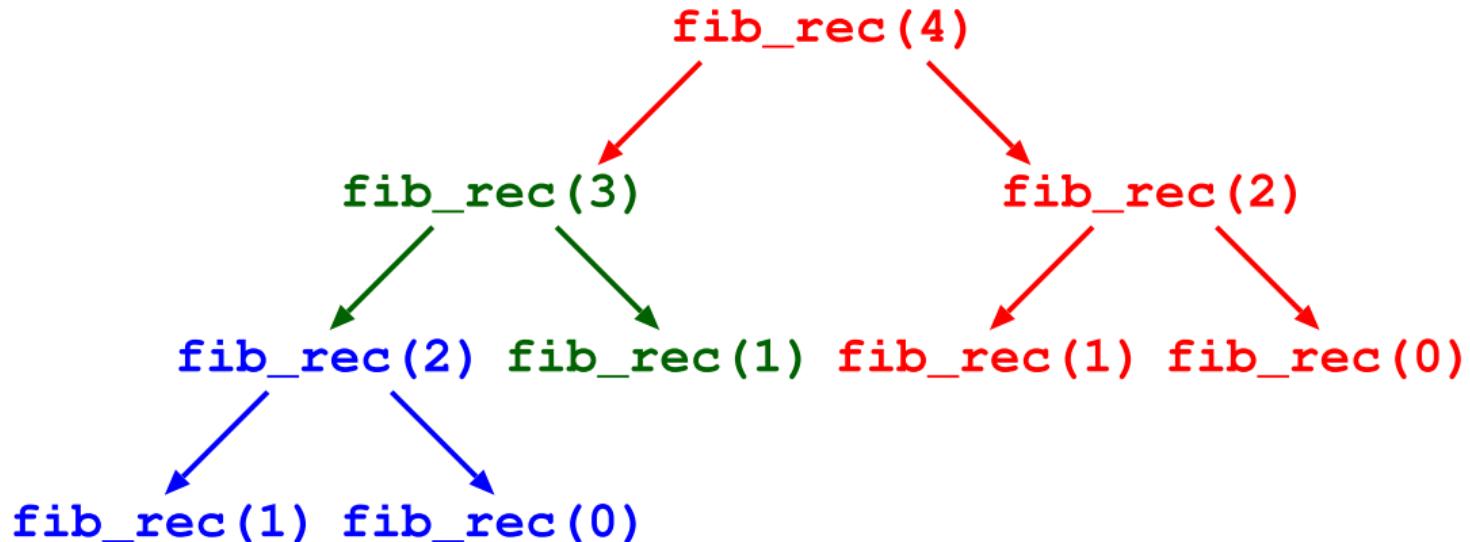
## möglicher Algorithmus:

```
function fib_rek (n:integer):int64;  
begin  
  if (n = 0) or (n = 1) then result := n  
  else result := fib_rek (n - 1) + fib_rek (n - 2)  
end;
```

Die Anzahl der Funktionsaufrufe nimmt hier **exponentiell** zu. Außerdem werden viele **redundante** Berechnungen ausgeführt. Dadurch kann es bei größeren n zu **Laufzeit-** und **Stackproblemen** kommen.

Die folgende Folie zeigt das für die Aufrufe **fib\_rec(0)** bis **fib\_rec(4)**.

# Fibonacci, rekursiv



- ▶  $n = 0$  und  $n = 1 \Rightarrow$  je ein Aufruf
- ▶  $n = 2$  (blau)  $\Rightarrow$  drei Aufrufe
- ▶  $n = 3$  (grün und blau)  $\Rightarrow$  fünf Aufrufe
- ▶  $n = 4$  (rot, grün und blau)  $\Rightarrow$  neun Aufrufe

# Fibonacci, rekursiv

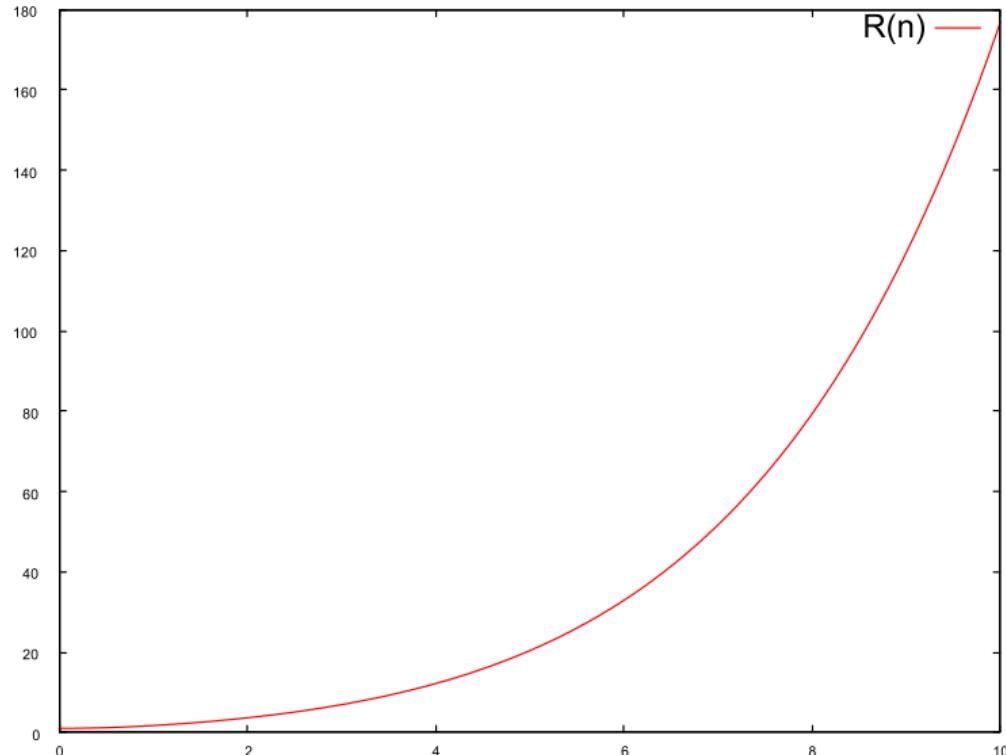
Die Anzahl  $R(n)$  für die Aufrufe der  $n$ -ten Fibonacci-Zahl berechnet sich aus der Summe der Anzahl der vorherigen Aufrufe, vermehrt um 1:

- ▶  $R(0) = 1$
- ▶  $R(1) = 1$
- ▶  $R(n) = R(n - 1) + R(n - 2) + 1$



<b>n</b>	0	1	2	3	4	5	6	7	8	9	10
<b>fib_rec(n)</b>	0	1	1	2	3	5	8	13	21	34	55
<b>R(n)</b>	1	1	3	5	9	15	25	41	67	109	177

# Funktionsplot für $R(n)$ mit gnuplot



# Fibonacci, iterativ

## möglicher Algorithmus:

```
function fib_it (n:integer):int64;
var i : integer;
    f0, f1, f2 : int64;
begin
    f0 := 0; f1 := 1; f2 := 0; i := 0;
    while i < n do
        begin
            i := i + 1;
            f0 := f1; f1 := f2;
            f2 := f1 + f0
        end;
    result := f2
end;
```

Die Schleife wird für jede Erhöhung von  $n$  einmal mehr durchlaufen  
⇒ **lineare Abhängigkeit** von  $R(n)!$

# Grenzen der Berechenbarkeit

## theoretische Berechenbarkeit

Eine mathematische Funktion  $f$  ist **theoretisch berechenbar**, wenn es einen Algorithmus gibt, der für eine beliebige Eingabe  $x$  aus dem Definitionsbereich den Funktionswert  $f(x)$  berechnet.

## praktische Berechenbarkeit

Eine Funktion  $f$  ist **praktisch berechenbar**, wenn es ein Computerprogramm gibt, das für ein beliebiges  $x$  aus dem Definitionsbereich den Funktionswert  $f(x)$  auf einem Rechner in **polynomialer Zeit** berechnet.

# polynomialer Aufwand

Algorithmen, deren Zeitverhalten als Polynom ausgedrückt werden kann, haben polynomialen Aufwand.

## polynomialer Aufwand:

$$T(n) = a_r n^r + a_{r-1} n^{r-1} + \dots + a_1 n^1 + a_0$$

mit  $r \in \mathbb{N}$ ,  $a_r \dots a_0 \in \mathbb{R}$ ,  $a_r \neq 0$

$r$  ist dabei der Grad des Polynoms:

- ▶  $r = 1 \Rightarrow$  linearer Aufwand
- ▶  $r = 2 \Rightarrow$  quadratischer Aufwand
- ▶  $r = 3 \Rightarrow$  kubischer Aufwand

Sortieralgorithmen haben polynomialen Aufwand.

# exponentieller Aufwand

Algorithmen, deren Zeitverhalten als Exponentialfunktion ausgedrückt werden kann, haben exponentiellen Aufwand.

## exponentieller Aufwand:

$$T(n) = c \cdot z^n \quad \text{mit} \quad z, c \in \mathbb{R}, \quad c \neq 0, \quad z > 1$$

Algorithmen mit exponentiellem Aufwand können auch mit modernen Computern sehr lange Rechenzeiten ergeben (☞ mehrere Jahre).

Probleme mit exponentiellem Aufwand gelten daher zwar als **theoretisch lösbar**, sind aber häufig **praktisch unlösbar**.

# Aufgaben (1/2)

## Aufgabe 1

Bearbeiten Sie in Gruppen ein Problem mit exponentiellem Aufwand (Problemstellung, Lösungsalgorithmus oder -algorithmen, Betrachtungen zur praktischen Lösbarkeit).

Stellen Sie Ihre Ergebnisse unter Verwendung des Beamers den Mitschülern vor.

Erzeugen Sie außerdem ein druckbares Dokument (PDF, A4, maximal eine Seite) mit wesentlichen Informationen zum ausgewählten Problem.

*Beispiele für Probleme mit exponentiellem Aufwand siehe nächste Folie!*

# Aufgaben (2/2)

Probleme mit exponentiellem Aufwand:

- ▶ Problem des Handlungsreisenden oder Rundreiseproblem (TSP = Traveling Salesman Problem)
- ▶ Damenproblem (schachmathematische Aufgabe)
- ▶ Rucksackproblem (Optimierungsproblem)
- ▶ ...

## Aufgabe 2

Festigen und ergänzen Sie Ihre Kenntnisse mit folgender  
Webseite: <https://www.inf-schule.de/grenzen>