

# Algorithmen

Lukas Abelt

`lukas.abelt@airbus.com`

DHBW Ravensburg  
Wirtschaftsinformatik

Ravensburg  
31. März 2019

# Outline

- 1 Allgemeines
  - Begriffsklärung
  - Ziele des Moduls
- 2 Beschreibung
  - Formale Eigenschaften
  - Darstellungsformen
- 3 Analyse
  - Korrektheit eines Algorithmus
  - Komplexitätsanalyse

# Inhalt

## 1 Allgemeines

- Begriffsklärung
- Ziele des Moduls

## 2 Beschreibung

- Formale Eigenschaften
- Darstellungsformen

## 3 Analyse

- Korrektheit eines Algorithmus
- Komplexitätsanalyse

# Inhalt

## 1 Allgemeines

- Begriffsklärung
- Ziele des Moduls

## 2 Beschreibung

- Formale Eigenschaften
- Darstellungsformen

## 3 Analyse

- Korrektheit eines Algorithmus
- Komplexitätsanalyse

# Begriffklärung

## Etymologie

- Leitet sich ursprünglich vom persischen Astronomen „Muhammad Ibn-Musa al-Hwarizmi“ ab
  - Schrieb Bücher über das indische Zahlensystem (um 800 n. Chr.)
  - Im 12. Jh übersetzt ins lateinische
  - Dabei wurde der Namensbestandteil „al-Hwarizmi“ in „Algorismi“ lateinisiert
- Durch spätere Überlieferungen wurde der Begriff später als Zusammensetzung betrachtet aus...
  - Dem Namen „Algos-“...
  - und dem aus dem griechisch entlehnten „-rismus“ (Zahl)

Vgl. [3]

# Begriffsklärung

Was bedeutet das jetzt

## Formale Definition (Nach [3])

*Eine Berechnungsvorschrift zur Lösung eines Problems heißt genau dann Algorithmus, wenn eine zu dieser Berechnungsvorschrift äquivalente Turingmaschine existiert, die für jede Eingabe, die eine Lösung besitzt, stoppt.*

## Oder auch

*Ein Algorithmus ist eine domänenunabhängige Beschreibung einer Handlungsvorschrift zur Lösung eines Problems. Eine bestimmte Eingabe wird in eine bestimmte Ausgabe überführt.*

# Begriffsklärung

## Also

- Ist also die Beschreibung eines Programmes oder einer Funktion
  - Unabhängig von der verwendeten Programmiersprache!
  - Source Code direkt ist also kein Algorithmus...
  - ...aber aus diesem lässt sich der verwendete Algorithmus ableiten und beschreiben
- Algorithmen können in verschiedenen Formen dargestellt werden (Mehr dazu im nächsten Kapitel)

# Inhalt

## 1 Allgemeines

- Begriffsklärung
- Ziele des Moduls

## 2 Beschreibung

- Formale Eigenschaften
- Darstellungsformen

## 3 Analyse

- Korrektheit eines Algorithmus
- Komplexitätsanalyse



# Ziele

- Am Ende des Moduls könnt ihr...
  - Einen Algorithmus in eine Implementierung umsetzen
  - Aus einer Implementierung den Algorithmus ableiten
  - Die formalen Eigenschaften von Algorithmen kennen
  - Algorithmen anhand der kennengelernten Methoden zu analysieren

# Inhalt

## 1 Allgemeines

- Begriffsklärung
- Ziele des Moduls

## 2 Beschreibung

- Formale Eigenschaften
- Darstellungsformen

## 3 Analyse

- Korrektheit eines Algorithmus
- Komplexitätsanalyse

# Inhalt

## 1 Allgemeines

- Begriffsklärung
- Ziele des Moduls

## 2 Beschreibung

- Formale Eigenschaften
- Darstellungsformen

## 3 Analyse

- Korrektheit eines Algorithmus
- Komplexitätsanalyse

# Eigenschaften von Algorithmen

## Grundlegendes

- **Finitheit** - Ein Algorithmus lässt sich in endlich vielen Schritten eindeutig beschreiben
- **Ausführbarkeit** - Jeder Einzelschritt muss tatsächlich ausführbar sein
- **Platzkomplexität** - Ein Algorithmus benötigt zu jedem Zeitpunkt nur endlich viel Speicherplatz
- **Terminierung** - Der Algorithmus benötigt eine endliche Anzahl von Schritten zur Ausführung
- **Determiniertheit** - Der Algorithmus muss bei gleichen Rahmenbedingungen das gleiche Ergebnis liefern
- **Determinismus** - Der nächste Schritt des Algorithmus ist zu jedem Zeitpunkt genau definiert

# Effizienz von Algorithmen

- Ergibt sich indirekt aus den Grundlegenden Eigenschaften
- Effizienz lässt sich über verschiedene Größen beschreiben:
  - Speicherverbrauch
  - Zeitverbrauch
- Die sind jedoch oft Implementierungs- und Rechnerabhängig
- Deshalb wird mit formalisierten Modellen gearbeitet
- ...Mehr dazu im Kapitel „Analyse“

Vgl. [2], S. 2f

# Inhalt

## 1 Allgemeines

- Begriffsklärung
- Ziele des Moduls

## 2 Beschreibung

- Formale Eigenschaften
- Darstellungsformen

## 3 Analyse

- Korrektheit eines Algorithmus
- Komplexitätsanalyse

# Ein kleines Beispiel

Warum wir das überhaupt brauchen (Algorithmus siehe [4])

```
1 float Q_rsqrt( float number ) {
2     long i;
3     float x2, y;
4     const float threehalfs = 1.5F;
5
6     x2 = number * 0.5F;
7     y  = number;
8     i  = * (long*) &y;    // evil floating point bit level
    ↪   hacking
9     i  = 0x5f3759df - ( i >> 1 );    // what the fuck?
10    y  = *(float*) &i;
11    y  = y*(threehalfs-(x2*y*y));    // 1st iteration
12    //y = y*(threehalfs-(x2*y*y));    // 2nd iteration, this
    ↪   can be removed
13    return y;
14 }
```

# Warum wir das brauchen

- ❑ Source Code ist nicht immer verständlich
- ❑ Benötigt spezielles Wissen über die Sprache
- ❑ Nutzt ggf. Besonderheiten der Sprache aus
- ❑ Nutzt teilweise Workarounds (zum Beispiel aus Effizienzgründen)



# Möglichkeiten der Darstellung

- Zur Definition von Algorithmen gibt es verschiedenste Möglichkeiten
- Mit ganz eigenen Vor- und Nachteilen
- Wir betrachten im Rahmen der Vorlesung:
  - Prosatext
  - Pseudocode
  - Struktogramme
  - Programmablaufplan (PAP)

# Was beschreiben wir?

## Unser Referenzalgorithmus

- Um die verschiedenen Elemente zu vergleichen, wollen wir mit allen den folgenden Algorithmus beschreiben:

### Referenz

Für eine Zahl  $n$  (Wobei gilt:  $n \in \mathbb{N}$ ), soll die Summe aller geraden Zahlen von 0 bis  $n$  berechnet werden.

# Darstellung als Prosatext

## Der simple Weg

- Simpleste Herangehensweise
- Man beschreibt in eigenen Worten, wie man vorgehen würde um die gegebene Problemstellung zu lösen
- **Achtung:** Unterscheiden zwischen Problemstellung und Lösungsbeschreibung!
- Auch in Prosaform sollten die Einzelschritte eindeutig beschrieben sein
- Nicht standardisiert → Beschreibung von Algorithmen inkonsistent

# Prosabeschreibung

Für unseren Algorithmus

## Addiere alle geraden Zahlen

Lese die Zahl  $n$  ein.

Anschließend setze die Zählvariable  $i$  sowie die Ergebnisvariable  $res$  auf 0.

Wenn  $i$  gerade ist, addiere  $i$  auf die Ergebnisvariable. Erhöhe anschließend  $i$  um 1. Wiederhole die letzten zwei Schritte bis  $i$  größer ist als  $n$ .

Gebe  $res$  aus

# Darstellung als Pseudocode

## Der Zwischenweg

- ❑ Mischung aus Prosa und tatsächlichem Code
- ❑ Orientiert sich an den in Programmiersprachen vorhandenen Strukturen (If-then-else, Schleifen...)
- ❑ Nutzt dabei aber leicht verständliche und programmiersprachenunabhängige Begriffe
- ❑ Wie Code in der Regel zeilenweise auf atomare Operationen beschränkt
- ❑ Keine formale Standardisierung, dadurch auch hier Inkonsistenzen möglich → Aber weniger als bei Prosabeschreibung

# Pseudocode

## Für unser Pseudoproblem

```
1  LESE  n
2  SETZE res=0
3  FUER  i=0 BIS  n
4      WENN  istGerade(i) DANN
5          res+=i
6      ENDE WENN
7  ENDE FUER
8  GEBE  res  AUS
```

# Struktogramme

## Der erste Standard

- Entwickelt durch *Nassi Shneidermann*
- Grafische Darstellung von Algorithmen
- Standardisiert nach **DIN 66261**
- Zerlegt den Algorithmus in elementare Grundstrukturen
- Die über die definierten Blöcke dargestellt werden
- Werden (lückenlos) von oben nach unten aneinander gereiht

# Elemente von Struktogrammen

## Anweisung

- Einzelne Anweisung:

Einzelne Anweisung
--------------------

- Mehrere aufeinanderfolgende Anweisungen:

Anweisung 1
-------------

Anweisung 2
-------------

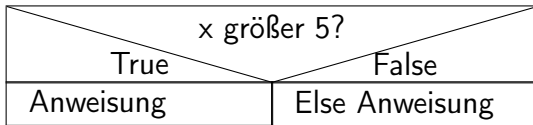
Anweisung 3
-------------



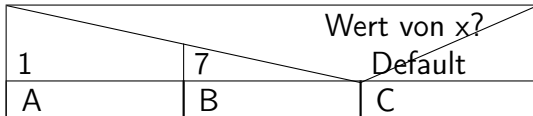
# Elemente von Struktogrammen

## Verzweigungen

- Einfache Verzweigung(if-then-else):



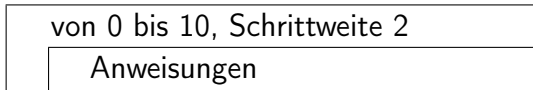
- Mehrfache Verzweigung(switch-case):



# Elemente von Struktogrammen

## Zählschleifen

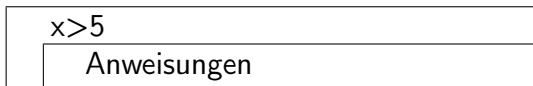
- Einfache Zählschleifen(for-loop):



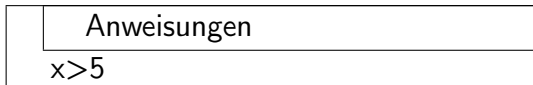
# Elemente von Struktogrammen

## Schleifen

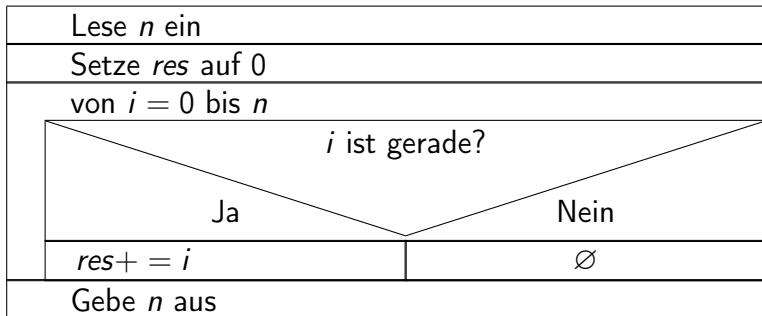
- Kopfgeprüfte Schleifen(while):



- Fußgeprüfte Schleifen(do-while):



# Struktogramm für unseren Algorithmus



# Programmablaufplan

## Der zweite Standard

- Bildet einen linearen Programmfluss aber
- Standardisiert nach **DIN 66001**
- Wie beim Struktogramm gibt es fest definierte Grundblöcke
- Diese werden hier jedoch über Pfeile verbunden

# Elemente von Programmablaufplänen

Start, Stop, Anweisungsblock, Ein- und Ausgaben

TODO: Abbildung Grundelemente

# Elemente von Programmablaufplänen

## Verzweigungen

TODO: Verzweigung

# Elemente von Programmablaufplänen

## Zählschleifen

TODO: Abbildung Zählschleife



# Elemente von Programmablaufplänen

## Schleifen

TODO: Abbildung Schleifen

# Programmablaufplan

...für unseren Algorithmus

TODO: PAP Algorithmus

# Zusammenfassung

- Keine der dargestellten Formen ist optimal
- Verwendung kommt auf Anforderungen und persönliche Vorlieben an
- Keine der hier vorgestellten Methoden zur Abbildung komplexerer objektorientierter Zusammenhänge möglich
- Weitere Darstellungsformen:
  - Aktivitätsdiagramm
  - Petrinetze
  - Interaktionsdiagramme

# Inhalt

## 1 Allgemeines

- Begriffsklärung
- Ziele des Moduls

## 2 Beschreibung

- Formale Eigenschaften
- Darstellungsformen

## 3 Analyse

- Korrektheit eines Algorithmus
- Komplexitätsanalyse

# Inhalt

## 1 Allgemeines

- Begriffsklärung
- Ziele des Moduls

## 2 Beschreibung

- Formale Eigenschaften
- Darstellungsformen

## 3 Analyse

- Korrektheit eines Algorithmus
- Komplexitätsanalyse

# Korrektheit von Algorithmen

Allgemeines (Vgl. [3])

- Jeder Algorithmus sollte auch in allen Fällen das korrekte Ergebnis liefern...
- Klingt simpel, aber eindeutiger Beweis für alle Eingaben oft schwierig
- Testen an ausgewählten Beispielen **nicht** ausreichend
  - Jedoch verringern umfangreiche Tests natürlich das Risiko eines unentdeckten Fehler
- Korrektheit lässt sich im Grunde nur durch formalen Beweis zeigen
  - Wie zum Beispiel Induktionsbeweis
  - Diese sind häufig sehr umfangreich und komplex...
  - ...und deshalb auch nicht Teil der Vorlesung

# Korrektheit von Algorithmen



*„Program testing can be used to show the presence of bugs, but never to show their absence!“*

Edsger W. Dijkstra

Quelle: [1]

# Inhalt

## 1 Allgemeines

- Begriffsklärung
- Ziele des Moduls

## 2 Beschreibung

- Formale Eigenschaften
- Darstellungsformen

## 3 Analyse

- Korrektheit eines Algorithmus
- Komplexitätsanalyse



# Speicherkomplexität

Wie lässt sich diese messen?

- Wie schon erwähnt: Der verbrauchte Speicher ist Sprach- und Rechnerabhängig
- Mögliche Lösung über Definition von Referenzsprache und -system
- Messungen sind allerdings nicht repräsentativ
- Deswegen wird in der formalen Informatik mit dem *Random-Access-Machine*(RAM) Modell gearbeitet
  - Besteht im Grunde aus abzählbar unendlich vielen adressierbaren Speicherzellen
  - Für einen Algorithmus wird dann bestimmt, wie viele Speicherzellen genutzt werden müssen
  - Dies entspricht dann der Speicherkomplexität

# Laufzeitkomplexität

## Grundlegendes

- Gleiches Problem wie bei der Speicherkomplexität
- Deswegen hier ähnliches Modell:
  - Man bestimmt die Anzahl von „atomaren Operationen“ des Algorithmus
  - Diese Operationen sind vergleichbar mit Assembler-Befehlsrepertoire
- Beispiele für atomare Operationen:
  - Addition/Subtraktion/Multiplikation/Division zweier Zahlen
  - Lesen einer Variable von einer Speicheradresse
  - Schreiben einer Variable an eine bestimmte Adresse
  - Random Access in Arrays
  - Vergleich zweier Zahlen
- Angabe benötigten Operationen über  $\tau(N) = x$

# Beispiel einer Komplexitätsanalyse

## Vertauschen zweier Zahlen I

```
1 public void swap(int first, int second){  
2     int tmp = first;  
3     first = second;  
4     second = tmp;  
5 }
```

- Laufzeitkomplexität:  $\tau(N) = 3$
- Speicherkomplexität(In Byte):  $\tau(N) = 12$

# Beispiel einer Komplexitätsanalyse

## Vertauschen zweier Zahlen II

```
1 public void swap(int first, int second){  
2     first = first + second;  
3     second = first - second;  
4     first = first - second;  
5 }
```

- Laufzeitkomplexität:  $\tau(N) = 6$
- Speicherkomplexität(In Byte):  $\tau(N) = 8$

# Probleme und Problem instanzen

- Komplexität ist selten statisch
- In der Regel von Problem und der konkreten *Problem instanz* abhängig
  - Problem: z.B. Das sortieren einer Liste
  - Problem instanz: konkrete Liste die sortiert werden soll, z.B.  
(7, 3, 12, -5, 45)
- Die Problem instanz hat meist einen oder mehrere dynamische Faktoren von denen die entgültige Komplexität abhängt
  - Für Sortieren: Länge der Liste
- Angegeben werden in der Komplexität nur noch die skalierenden Faktoren:  $O(N)$

Vgl. [2] S. 3 ff

# Dynamische Komplexität

## Ein Beispiel

```
1 //A-> Array mit Elementen, n->Länge von A
2 int sumList(int[] A, int n){
3     int sum = 0;                //Kosten: 1, Anzahl: 1 mal
4     for(int i=0;i<n;i++){       //Kosten: 2, Anzahl: n+1 mal
5         sum+=A[i]               //Kosten: 2, Anzahl: n mal
6     }
7     return sum;                //Kosten: 1, Anzahl: 1 mal
8 }
```

$$\tau(n) = 1 + 2 \cdot (n + 1) + 2n + 1$$

$$\tau(n) = 1 + 2n + 2 + 2n + 1$$

$$\tau(n) = 4n + 4 \quad \tau(n) = c_1 n + c_2$$

$$\Rightarrow O(n)$$

# Unterschiedliche Laufzeiten von Algorithmen

- Bisher betrachtete Algorithmen hatten (für gegebenes  $N$ ) feste Komplexität
- Algorithmen können jedoch für gleiches  $N$  verschiedene Laufzeiten haben
  - Warum?
    - Zum Beispiel bei Verzweigung innerhalb des Algorithmus (z.B. durch gesonderte Betrachtung besonderer Listenelemente o.Ä.)
- Beispiel: Leicht angepasste Variante unseres Algorithmus zum addieren von geraden Zahlen:

# Addieren aller geraden Zahlen einer Liste

```
1 public int getSquareSum(int[] data, int n){
2     int res = 0;                //Kosten: 1, 1 mal
3     for(int i=0;i<n;i++){       //Kosten: 2, (n+1) mal
4         if(data[i]%2 == 1){     //Kosten: 3, n mal
5             continue;          //Kosten: 1, ? mal
6         }else{
7             //Kosten: 5, ? mal
8             res += data[i]*data[i];
9         }
10    }
11    return res;
12 }
```



# Best- und Worst-Case Execution Time

- Für diese Fälle gibt es prinzipiell drei Betrachtungsweisen
  - ▣ Best-Case Execution Time
  - ▣ Worst-Case Execution Time
  - ▣ Average Execution Time
  - ▣ Frage: Welche ist für die Komplexitätsbetrachtung relevant?
- Average Execution Time lässt sich nur schwer bestimmen
  - ▣ Reiner Durchschnitt aus Best- und Worst-Case nicht praktikabel
  - ▣ Beschaffenheit der Problem instanzen und deren Verteilung müsste bekannt sein
  - ▣ Ist jedoch selten der Fall

# Best- und Worst-Case Execution Time

- In der Regel ist die Worst-Case Execution Time relevant
  - Simpel zu bestimmen
  - Dadurch kann sichergestellt werden, dass der Algorithmus **maximal** die angegebene Zeit benötigt
  - insbesondere relevant für Echtzeitsysteme

Vgl. [2] S. 3ff

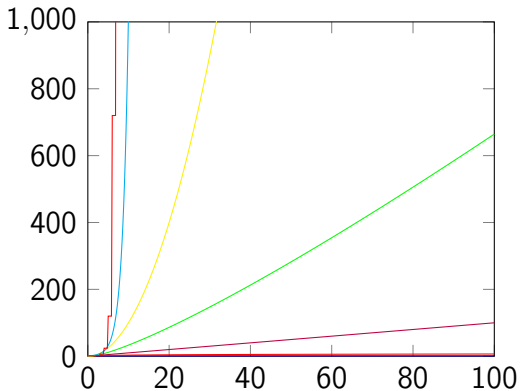
# Gängige Komplexitätsfaktoren

Vgl. [2]

- In der Regel begegnet man den folgenden Komplexitäten bei der Analyse(In Abhängigkeit von  $N$ ):
  - Kein Wachstum:  $O(1)$
  - Logarithmisches Wachstum:  $O(\log N)$
  - Lineares Wachstum:  $O(N)$
  - $N$ -log  $N$ -Wachstum:  $O(N \cdot \log N)$
  - Polynomiales Wachstum:  $O(N^2)$ ,  $O(N^3)$ ...
  - Exponentielles Wachstum:  $O(2^N)$ ,  $O(3^N)$ ...
  - Faktorielles Wachstum:  $O(N!)$
- Praktikabel sind maximal Algorithmen mit polynomialen Wachstum
- Exponentielle und faktorielle Algorithmen wachsen zu schnell an

# Komplexität von Algorithmen

## Visueller Vergleich



# Beispiele für Komplexität

## Kein Wachstum

```
1 public void swap(int[] data, int first, int second){  
2     int tmp = data[first];  
3     data[first] = data[second];  
4     data[second] = tmp;  
5 }
```

- Algorithmus ist in keiner Weise von der Länge von data abhängig (Abgesehen von eventueller Fehlerbetrachtung)

# Beispiele für Komplexität

## Logarithmisches Wachstum

```
1 public void logNComplexity(int n){  
2     for(int i=1; i<=n; i = i * 2){  
3         System.out.println(i);  
4     }  
5 }
```

- Indikator für Logarithmisches Wachstum:
  - Zählvariable steigt multiplikativ/verringert sich durch Division
  - Größe der Problemistanz verringert sich in jedem Schritt mit bestimmtem Faktor
- Basis des Logarithmus ist nicht von Relevanz, da nur konstanter Faktor
- In der Regel handelt es sich aber um  $\log_2$
- Beispiel: Binäre Suche

# Beispiele für Komplexität

## Lineares Wachstum

```
1 public int sum(int[] data, int n){  
2     int res=0;  
3     for(int i=0;i<n;i++){  
4         res+=data[i];  
5     }  
6     return res;  
7 }
```

- Grundlegend alle Schleifen, die von 0 bis  $N$  iterieren (Mit Schrittweite 1)
- Beispiele:
  - Summen von Listen
  - Finden von Mini-/Maxima in unsortierten Listen

# Beispiele für Komplexität

## $N \cdot \log N$ -Wachstum

```
1 public void nlogNComplexity(int n){
2     for(int i=0;i<n;i++){
3         for(int j=0;j<n;j=j*2){
4             System.out.println(i);
5             System.out.println(j);
6         }
7     }
8 }
```

- Entsteht durch die Kombination von linearem und logarithmischem Wachstums
- Beispiele:
  - Heap Sort
  - Quick Sort



# Beispiele für Komplexität

## Polynomiales Wachstum

```
1 //Unter der Annahme, dass data ein n*n Array ist
2 public void print2DArray(int[][] data, int n){
3     for(int i=0;i<n;i++){
4         for(int j=0;j<n;j++){
5             System.out.println(data[i][j]);
6         }
7     }
8 }
```

- Entsteht durch die Verschachtelung mehrerer  $O(n)$  Algorithmen
- Beispiele:
  - Insertion Sort
  - Traversieren von N-Dimensionalen Arrays

# Beispiele für Komplexität

## Polynomiales Wachstum

```
1 //Unter der Annahme, dass data ein n*n Array ist
2 public void print2DArray(int[][] data, int n){
3     for(int i=0;i<n;i++){
4         for(int j=0;j<n;j++){
5             System.out.println(data[i][j]);
6         }
7     }
8 }
```

- Entsteht durch die Verschachtelung mehrerer  $O(n)$  Algorithmen
- Beispiele:
  - Insertion Sort
  - Traversieren von N-Dimensionalen Arrays

# Beispiele für Komplexität

## Exponentielles Wachstum

- Hier ist mir leider kein leichtes Codebeispiel eingefallen
- Beispiele:
  - Bruteforce von Passwörtern
  - Damenproblem (mit naiver Implementierung)

# Beispiele für Komplexität

## Faktorielles Wachstum

```
1 public void factorial(int n){  
2     for(int i=0;i<n;i++){  
3         System.out.println(i);  
4         factorial(n-1);  
5     }  
6 }
```

- Häufig in rekursiven Algorithmen
- Beispiele:
  - Finden aller Permutationen in einem Array
  - Travelling Salesman Problem (Primitiver Ansatz)

# Quellen I

- [1] Wikimedia Commons. *File:Edsger Wybe Dijkstra.jpg* — *Wikimedia Commons, the free media repository*. 2017. URL: [https://commons.wikimedia.org/w/index.php?title=File:Edsger\\_Wybe\\_Dijkstra.jpg&oldid=244763264](https://commons.wikimedia.org/w/index.php?title=File:Edsger_Wybe_Dijkstra.jpg&oldid=244763264) (besucht am 29.03.2019).
- [2] T. Ottmann und P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, 2017. ISBN: 9783662556498.
- [3] Wikipedia. *Algorithmus* — *Wikipedia, Die freie Enzyklopädie*. 2019. URL: <https://de.wikipedia.org/w/index.php?title=Algorithmus&oldid=186838998> (besucht am 29.03.2019).

# Quellen II

- [4] Wikipedia contributors. *Fast inverse square root* — *Wikipedia, The Free Encyclopedia*. 2019. URL: [https://en.wikipedia.org/w/index.php?title=Fast\\_inverse\\_square\\_root&oldid=887505872](https://en.wikipedia.org/w/index.php?title=Fast_inverse_square_root&oldid=887505872) (besucht am 29.03.2019).

# Kontakt

- E-Mail: `lukas.abelt@airbus.com`
- GitHub: `https://www.github.com/LuAbelt`
- GitLab: `https://www.gitlab.com/LuAbelt`
- Telefon(Firma): 07545 - 8 8895
- Telegram: LuAbelt