

Algorithmen und Datenstrukturen

- Organisation, Einführung und Übersicht -

Prof. Dr. Klaus Volbert

Wintersemester 2018/19
Regensburg, 01. Oktober 2018

Algorithmen in der Informatik

- Welche Programmiersprachen beherrschen Sie?
- Welche Konzepte kennen Sie aus Programmiersprachen?
- Was wissen Sie über Algorithmen?
- Welche Algorithmen kennen Sie?
- Welche Datenstrukturen kennen Sie?
- Was erwarten Sie von dieser Veranstaltung?

- Studiengang Bachelor Informatik (IN3/IN4)

- Vorlesungen:

Mo 13:30-15:00 Uhr Raum K001
Do 08:15-09:45 Uhr Raum U212

- Übungen ([Beginn ab Donnerstag, 04.10.2018](#)):

- Gruppe 1: Di 08:15-09:45 Uhr Raum K223 (IN3IN4[1])
 - Gruppe 2: Di 11:45-13:15 Uhr Raum K140 (IN3IN4[2])
 - Gruppe 3: Mo 15:15-16:45 Uhr Raum K220 (IN3IN4[3])
 - Gruppe 4: Do 10:00-11:30 Uhr Raum K140 (IN3IN4[4])

- Rahmendaten zur Veranstaltung

- Nach Studienplan: 3./4. Semester
 - Mindestens 30 Credits aus dem 1. Studienabschnitt
 - 6 SWS: V4, Ü2, 8 Credits (nach erfolgreich bestandener Prüfung)

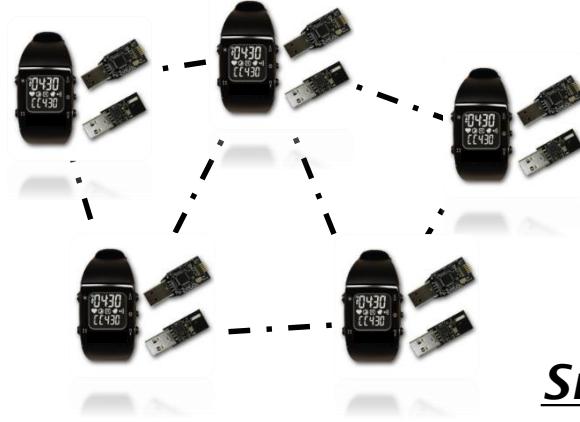
Organisatorisches II

- Begleitende Webseite mit aktuellen Informationen
 - <http://hps.oth-regensburg.de/~vok39696>
 - Lehre Wintersemester 2018/19:
 - Algorithmen und Datenstrukturen IN (→ Kursraum in G.R.I.P.S.)
- Dozent
 - Sprechstunde: Di 10-11 Uhr Raum K214 und nach Vereinbarung

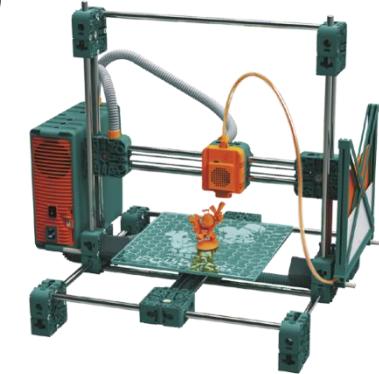
klaus.volbert@oth-regensburg.de
 - Lehrgebiete:
 - Algorithmen und Datenstrukturen, Spezielle Algorithmen
 - Modelle und Algorithmen im Internet der Dinge, ...

Forschungsinteressen

Algorithmen-/Software-/App-Entwicklung für drahtlose, eingebettete Systeme (z. B. **Mobile Ad-hoc-Netze, Sensornetze, IoT, Industrie 4.0**)



Algorithmen
für **3D-Drucker**

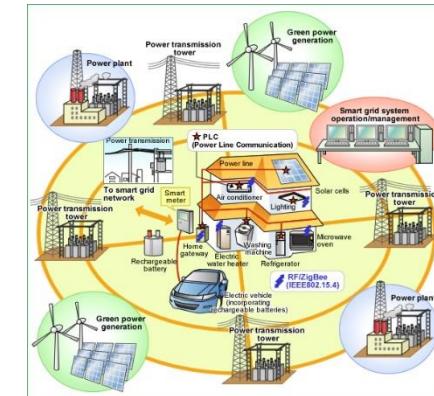


Smarte Algorithmen

OTH Forschungscluster **IKT** und **RAKS**



- Studentische Hilfskräfte
- Themen für Abschlussarbeiten
- Themen für HSP im Master



Energieinformatik (RCER)
(**Smart Metering, Smart Submetering, Smart Grid**)

Spielregeln in der Vorlesung

- Seminaristischer Unterricht
 - Aktive Teilnahme erwünscht
- Verwendung elektronischer Geräte
 - Die **kontextunabhängige** Verwendung von Tablets, Laptops, Mobiltelefonen und ähnlichen elektronischen Geräten während der Vorlesung ist **nicht** zugelassen



Bitte ausschalten!

Hinweise zum Übungsbetrieb

- Übungsbetrieb
 - Übungsgruppenauswahl im Kursraum (Zuordnung randomisiert)
 - Jede Woche am **Mittwoch**: 1 Übungsblatt mit 4 Aufgaben
 - Bearbeitung der Aufgaben
 - Vorbereitung/Bearbeitung frühzeitig starten
 - Bearbeitung in den Übungen fortsetzen (Dozent kann unterstützen)
 - Fertigstellung nach den Übungen (**nacharbeiten und üben!!!**)
 - Spätestens bis zum folgenden **Dienstag (23:00 Uhr)**:
 - Votieren der Aufgaben und elektronische Abgabe Ihrer Lösungen durch Hochladen einer Datei (**Pflichtabgabe**)
 - Nutzen Sie die Chance zum **Fragen und zum Präsentieren** Ihrer Lösung!

Anmerkungen zur Prüfung

- Art der Prüfung
 - 90 min. Klausur am Ende des Semesters
- Zugelassene Hilfsmittel
 - Ein beidseitig handgeschriebenes DIN-A4 Blatt
 - Taschenrechner oder andere elektronische Hilfsmittel sind **nicht** erlaubt
- Relevante Themen
 - Gesamter Stoff der Veranstaltung
 - Klausuraufgaben angelehnt an Übungsaufgaben
- Zulassungsvoraussetzung
 - Regelmäßige und aktive Teilnahme an den Übungen
 - **50 % der Aufgaben positiv votiert und abgegeben**

Übungsabgabe/Votieren

- Spätestens vor Erscheinen des neuen Übungsblatts (bis Di 23 Uhr)
...Aufgabe votieren und Lösungen elektronisch abgeben!
- Votieren
 - Zu jeder Aufgabe muss jeder Teilnehmer angeben, ob die Aufgabe gelöst werden konnte
 - 50 % der Aufgaben müssen gelöst werden (**positiv votiert werden**)
- Abgabe
 - Ihre Lösungen zu positiv votierten Aufgaben müssen elektronisch durch Hochladen einer Datei abgegeben werden
 - Die Datei kann ein Zip-Archiv oder eine PDF-/JPG-Datei sein
- Wichtig:
 - Nehmen Sie sich genügend Zeit zur Bearbeitung der Aufgaben!
 - Eine Aufgabe, die schwierig wirkt, kann auch einfach zu lösen sein
 - Lösungen müssen nicht perfekt sein

Einordnung der Veranstaltung

Angewandte Informatik

Wirtschaftliche, kommerzielle Anwendungen
Technisch-wissenschaftliche Anwendungen

Technische Informatik

Mikroprozessortechnik, Rechnerarchitektur
Rechnerkommunikation

Praktische Informatik

Programmiersprachen, Compiler/Interpreter,
Algorithmen und Datenstrukturen,
Betriebssysteme, Netzwerke, Datenbanken

Theoretische Informatik

Formale Sprachen und Automaten, Berechenbarkeit, Komplexität

Kerninformatik

Anmerkung: Informatik \neq Programmierung

Zentraler Begriff: Algorithmus

- Persischer Mathematiker und Astronom (~ 825 n. Chr.)
Abu Ja'far Muhammad Ibn Musa al-Khwarizmi

- Lehrbuch über die **Rechenregeln** in dem aus Indien stammenden dezimalen Stellenwertsystem

„Kitab al jabr w'al mugabala“



- Intuitive, informale Definition eines Algorithmus:
 - Ein Algorithmus ist eine Vorschrift zur Lösung eines Problems, die für eine Realisierung in Form eines Programms auf einem Computer geeignet ist (Taschenbuch der Informatik, 2004)
 - Ein Algorithmus ist eine präzise Handlungsvorschrift, um aus vorgegebenen **Eingaben** in endlich vielen Schritten eine bestimmte **Ausgabe** zu ermitteln (Eingabe-Verarbeitung-Ausgabe, EVA-Prinzip)

Sind Rezepte Algorithmen?

Zutaten: 1 Packung Löffelbiskuit, 500g Mascarpone, 4 Eier, 2 Esslöffel Zucker, 5 Esslöffel Amaretto, etwas Kakaopulver, 1 Tasse Kaffee

REZEPT_TIRAMISU

1. Den Kaffee in einen Suppenteller gießen.
2. Die Biskuits kurz im Kaffee tränken.
3. Den Boden einer Auflaufform mit einer Lage Biskuits belegen.
4. Eigelb und Eiweiß trennen.
5. Das Eigelb mit dem Zucker und Amaretto zu schaumiger Masse schlagen.
6. Mascarpone zugeben und gut mischen, bis es cremig wird.
7. Das Eiweiß in einem anderen Behälter steif schlagen.
8. Das geschlagene Eiweiß zur Crème hinzugeben.
9. Die Biskuits in der Form gleichmäßig mit der Hälfte der Crème überziehen.
10. Nun eine neue Lage kaffeetränkter Biskuits auflegen.
11. Den Rest der Crème gleichmäßig auftragen.
12. Die Form mindestens drei Stunden (besser über Nacht) in den Kühlschrank stellen.
13. Die Crème vor dem Servieren mit Kakao bestreuen.

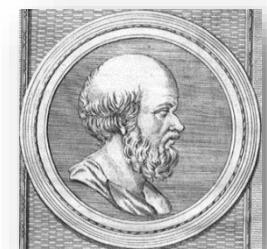


END_REZEPT_TIRAMISU

Ergebnis: Ein leckeres Tiramisu

Sieb des Eratosthenes

- Bestimmung aller Primzahlen bis zu einem gegebenen Wert k
- **Eingabe:** Wert $k \in \mathbb{N}$ (obere Schranke)
- **Ausgabe:** Alle Primzahlen bis zur Schranke k
- **Arbeitsweise** des Algorithmus:



Entfernen der Vielfachen von 2

(2)	3	4	5	6	7	8	9	10	
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48		

Entfernen der Vielfachen von 3

(2)	(3)	4	5	6	7	8	9	10	
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48		

Entfernen der Vielfachen von 5

(2)	(3)	4	(5)	6	7	8	9	10	
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48		

Die Primzahlen bis 48

(2)	(3)	4	(5)	6	(7)	8	9	10	
11	12	13	14	15	16	17	18	(19)	20
21	22	(23)	24	25	26	27	28	(29)	30
31	32	33	34	35	36	(37)	38	39	40
41	42	(43)	44	45	46	(47)	48		

Quelle: Logofatu - Grundlegende Algorithmen mit Java

Eigenschaften von Algorithmen

- Die Abfolge der einzelnen Verarbeitungsschritte muss eindeutig aus einem Algorithmus hervorgehen
- Ein Algorithmus ist unabhängig von einer Notation (z.B. natürliche Sprache, Pseudocode, C, C++, Java, C#, ...)
- Es gibt viele Beschreibungen desselben Algorithmus
- Hauptziel beim Entwurf von Algorithmen
 - Korrekte Problemlösung (**totale Korrektheit**):
 - **Terminiertheit**: Der Algorithmus endet für jede spezifizierte Eingabe
 - **Partielle Korrektheit**: Der Algorithmus liefert für jede spezifizierte Eingabe das geforderte Ergebnis
- Nebenziele beim Entwurf von Algorithmen
 - Effiziente Problemlösung hinsichtlich **Zeit** und **Platz**
- Algorithmen sollten effektiv sein

Beispiele für Algorithmen

- Addieren, Subtrahieren, Multiplizieren, Dividieren, etc.
- Kochrezepte, Bastel-/Gebrauchsanleitungen, Spielregeln, ...
- Sortieren, Suchen, Effizienter Umgang mit Datenstrukturen, ...
- Vorsicht: Forderung der Eindeutigkeit fordert Präzision!
- Algorithmus zur Bestimmung des **größten gemeinsamen Teilers (ggT)** zweier natürlicher Zahlen (Euklidischer Algorithmus, ~ 300 v. Chr.):
 - **Eingabe:** $a, b \in \mathbb{N}$ (zwei natürliche Zahlen a und b)
 - **Ausgabe:** $a = \text{ggT}(a,b)$ (ggT von a und b)
 - **Algorithmus:** Wiederhole
 - $r = \text{Rest der ganzzahligen Division von } a/b$
 - $a = b$
 - $b = r$Bis $r = 0$ ist
Gib a aus

Nachweis der totalen Korrektheit?

Vorbedingung: $a, b \in \mathbb{N}$; sei G größter gemeinsame Teiler von a und b , $E = x + y$

Nachbedingung: $x = G$

{ G ist ggT(a, b) \wedge $a \in \mathbb{N}^+$ \wedge $b \in \mathbb{N}^+$ }

x := a; y := b;

{ INV: G ist ggT(x, y) \wedge $x \in \mathbb{N}^+$ \wedge $y \in \mathbb{N}^+$ \wedge $x + y > 0$ }

solange $x \neq y$ **wiederhole**

{ INV \wedge $x \neq y \wedge x + y = k$ }

falls $x > y$:

{ G ist ggT(x, y) \wedge $x > y \wedge x \in \mathbb{N}^+ \wedge y \in \mathbb{N}^+ \wedge x + y = k$ } \Rightarrow

{ G ist ggT($x - y, y$) \wedge $x - y \in \mathbb{N}^+ \wedge y \in \mathbb{N}^+ \wedge x - y + y = k - y < k$ }

x := x - y

{ INV \wedge $x + y < k$ }

sonst

{ G ist ggT(x, y) \wedge $y > x \wedge x \in \mathbb{N}^+ \wedge y \in \mathbb{N}^+ \wedge x + y = k$ } \Rightarrow

{ G ist ggT($x, y - x$) \wedge $x \in \mathbb{N}^+ \wedge y - x \in \mathbb{N}^+ \wedge x + y - x = k - x < k$ }

y := y - x

{ INV \wedge $x + y < k$ }

{ INV \wedge $x + y < k$ }

{ INV \wedge $x = y$ } \Rightarrow { $x = G$ }

Zentraler Begriff: Datenstruktur

- Algorithmen verarbeiten Eingaben zu Ausgaben und benutzen dabei ggf. unstrukturierte (skalare) und strukturierte Daten
- Datentyp
 - Menge von Objekten mit darauf definierten Operationen
 - Zwei Varianten:
 - (konkreter) Datentyp (abhängig von Rechner und Implementierung)
 - abstrakter Datentyp (abstrahiert von spezieller Implementierung)
- Datentyp wird durch folgendes Tupel festgelegt:
 - Objektmenge (Werte)
 - Operationen, definiert durch
 - Signatur (Name mit Definitions- und Wertebereich, Syntax)
 - Regeln/Axiome (Wirkung der Operationen, Semantik)
- Datenstruktur (= strukturierte Daten + Operationen)
 - Menge von Datentypen, zwischen denen Beziehungen bestehen

Beispiele für Datentypen in C++ (32-Bit)

Datentyp	Bits	Wertebereich	Typische Operationen
char, signed char	8	-128 ... 127	+,-,*,/,<;>,%
unsigned char	8	0 ... 255	+,-,*,/,<;>,%
short, signed short	16	-32768 ... 32767	+,-,*,/,<;>,%
unsigned short	16	0 ... 65535	+,-,*,/,<;>,%
int, signed int	32	-2.147.483.648 ... 2.147.483.647	+,-,*,/,<;>,%
unsigned, unsigned int	32	0 ... 4.294.967.295	+,-,*,/,<;>,%
long, signed long	32	-2.147.483.648 ... 2.147.483.647	+,-,*,/,<;>,%
unsigned long	32	0 ... 4.294.967.295	+,-,*,/,<;>,%
float	32	$1,2 \cdot 10^{-38}$... $3,4 \cdot 10^{38}$	+,-,*,/,<;>
double	64	$2,2 \cdot 10^{-308}$... $1,8 \cdot 10^{308}$	+,-,*,/,<;>
long double	96	$3,4 \cdot 10^{-4932}$... $1,1 \cdot 10^{4932}$	+,-,*,/,<;>

- **Lernziele**

- Grundlegende Algorithmen und Datenstrukturen für Standard-Probleme kennenlernen und implementieren können
- Effizienz von Algorithmen und Datenstrukturen bewerten und vergleichen können
- Effiziente Algorithmen und Datenstrukturen (anhand von kennengelernten Entwurfsprinzipien) entwerfen können

- **Inhalt**

- Grundlagen
 - Begriffe, Komplexität, Landau-Notation, Rekursionsgleichungen
- Algorithmen
 - Einfügen, Entfernen, Suchen und Sortieren
 - Entwurfsmethoden, Ausgewählte Algorithmen
- Datenstrukturen
 - Listen, Stapel und Schlangen
 - Bäume und Graphen

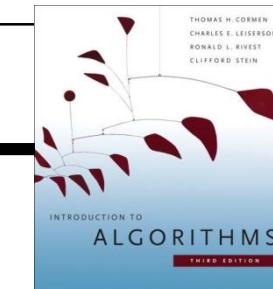


- Schwerpunkt liegt auf **algorithmischen Aspekten**
- Beispielsprachen:
 - Pseudocode, C, C++, Java, C#
(abstrakte, möglichst klare und präzise Beschreibung der Ideen)
- Softwaretechnische Aspekte werden vernachlässigt, z.B.
 - Vollständigkeit
 - Fehlerbehandlung
 - Objekt-orientierter Entwurf, Modularität
 - C, C++, Java, C# - spezifische Konstrukte
 - ...

Programmieren im Kleinen

Literatur zur Vorlesung

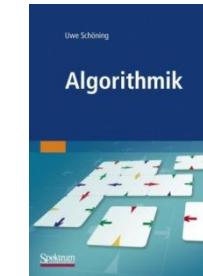
Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.:
Introduction to Algorithms, 3rd Ed., MIT Press, 2009



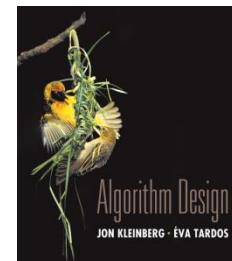
Sedgewick, R.:
Algorithmen in C++, Pearson Studium, 3. Auflage, 2002



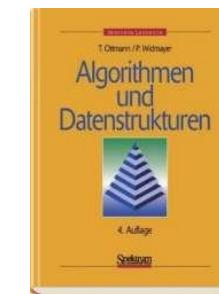
Schöning, U.:
Algorithmik, Spektrum Akademischer Verlag, 2001



Kleinberg, J., Tardos, É.:
Algorithm Design, Addison Wesley, 2005



Ottmann, T., Widmayer, P.:
Algorithmen und Datenstrukturen,
 Spektrum Akademischer Verlag, 4. Auflage, 2002



Pomberger, G., Dobler, H.:
Algorithmen und Datenstrukturen, Pearson Studium 2008



Regensburger Katalog

The screenshot shows the search results page of the Regensburger Katalog plus. The search query is "Freie Suche = algorithmen und datenstrukturen". The results are filtered by the "Regensburger Katalog (59)" category. There are 59 results listed, each with a checkbox, a thumbnail, the title, the author, and download links for Volltext and Inhaltsverzeichnis.

Rank	Title	Author	Actions
1	Algorithmen und Datenstrukturen	Ottmann, Thomas	Volltext
2	Programmierung, Algorithmen und Datenstrukturen	Gumm, Heinz-Peter	Volltext
3	Programmierung, Algorithmen und Datenstrukturen	Gumm, Heinz-Peter	Volltext
4	Algorithmen und Datenstrukturen	Blum, Norbert	Volltext
5	Algorithmen und Datenstrukturen	Weicker, Karsten	Volltext
6	Algorithmen und Datenstrukturen	Ottmann, Thomas . - 2012	Volltext
7	Algorithmen und Datenstrukturen	Ottmann, Thomas . - 2012	Volltext

<https://www.regensburger-katalog.de>

Algorithmen und Datenstrukturen

- Grundlagen (Begriffe und Beispiele) -

Prof. Dr. Klaus Volbert

Wintersemester 2018/19
Regensburg, 04. Oktober 2018

Zentraler Begriff: Algorithmus

- Persischer Mathematiker und Astronom (~ 825 n. Chr.)
Abu Ja'far Muhammad Ibn Musa al-Khwarizmi

- Lehrbuch über die **Rechenregeln** in dem aus Indien stammenden dezimalen Stellenwertsystem

„Kitab al jabr w'al mugabala“



- Intuitive, informale Definition eines Algorithmus:
 - Ein Algorithmus ist eine Vorschrift zur Lösung eines Problems, die für eine Realisierung in Form eines Programms auf einem Computer geeignet ist (Taschenbuch der Informatik, 2004)
 - Ein Algorithmus ist eine präzise Handlungsvorschrift, um aus vorgegebenen **Eingaben** in endlich vielen Schritten eine bestimmte **Ausgabe** zu ermitteln (Eingabe-Verarbeitung-Ausgabe, EVA-Prinzip)

Eigenschaften von Algorithmen

- Die Abfolge der einzelnen Verarbeitungsschritte muss eindeutig aus einem Algorithmus hervorgehen
- Ein Algorithmus ist unabhängig von einer Notation (z.B. natürliche Sprache, Pseudocode, C, C++, Java, C#, ...)
- Es gibt viele Beschreibungen desselben Algorithmus
- Hauptziel beim Entwurf von Algorithmen
 - Korrekte Problemlösung (**totale Korrektheit**):
 - **Terminiertheit**: Der Algorithmus endet für jede spezifizierte Eingabe
 - **Partielle Korrektheit**: Der Algorithmus liefert für jede spezifizierte Eingabe das geforderte Ergebnis
- Nebenziele beim Entwurf von Algorithmen
 - Effiziente Problemlösung hinsichtlich **Zeit** und **Platz**
- Algorithmen sollten effektiv sein

Beispiele für Algorithmen

- Addieren, Subtrahieren, Multiplizieren, Dividieren, etc.
- Kochrezepte, Bastel-/Gebrauchsanleitungen, Spielregeln, ...
- Sortieren, Suchen, Effizienter Umgang mit Datenstrukturen, ...
- Vorsicht: Forderung der Eindeutigkeit fordert Präzision!
- Algorithmus zur Bestimmung des **größten gemeinsamen Teilers (ggT)** zweier natürlicher Zahlen (Euklidischer Algorithmus, ~ 300 v. Chr.):
 - **Eingabe:** $a, b \in \mathbb{N}$ (zwei natürliche Zahlen a und b)
 - **Ausgabe:** $a = \text{ggT}(a,b)$ (ggT von a und b)
 - **Algorithmus:** Wiederhole
 - $r = \text{Rest der ganzzahligen Division von } a/b$
 - $a = b$
 - $b = r$Bis $r = 0$ ist
Gib a aus

Nachweis der totalen Korrektheit?

Vorbedingung: $a, b \in \mathbb{N}$; sei G größter gemeinsame Teiler von a und b , $E = x + y$

Nachbedingung: $x = G$

{ G ist ggT(a, b) \wedge $a \in \mathbb{N}^+$ \wedge $b \in \mathbb{N}^+$ }

x := a; y := b;

{ INV: G ist ggT(x, y) \wedge $x \in \mathbb{N}^+$ \wedge $y \in \mathbb{N}^+$ \wedge $x + y > 0$ }

solange $x \neq y$ **wiederhole**

{ INV \wedge $x \neq y \wedge x + y = k$ }

falls $x > y$:

{ G ist ggT(x, y) \wedge $x > y \wedge x \in \mathbb{N}^+ \wedge y \in \mathbb{N}^+ \wedge x + y = k$ } \Rightarrow

{ G ist ggT($x - y, y$) \wedge $x - y \in \mathbb{N}^+ \wedge y \in \mathbb{N}^+ \wedge x - y + y = k - y < k$ }

x := x - y

{ INV \wedge $x + y < k$ }

sonst

{ G ist ggT(x, y) \wedge $y > x \wedge x \in \mathbb{N}^+ \wedge y \in \mathbb{N}^+ \wedge x + y = k$ } \Rightarrow

{ G ist ggT($x, y - x$) \wedge $x \in \mathbb{N}^+ \wedge y - x \in \mathbb{N}^+ \wedge x + y - x = k - x < k$ }

y := y - x

{ INV \wedge $x + y < k$ }

{ INV \wedge $x + y < k$ }

{ INV \wedge $x = y$ } \Rightarrow { $x = G$ }

Zentraler Begriff: Datenstruktur

- Algorithmen verarbeiten Eingaben zu Ausgaben und benutzen dabei ggf. unstrukturierte (skalare) und strukturierte Daten
- Datentyp
 - Menge von Objekten mit darauf definierten Operationen
 - Zwei Varianten:
 - (konkreter) Datentyp (abhängig von Rechner und Implementierung)
 - abstrakter Datentyp (abstrahiert von spezieller Implementierung)
- Datentyp wird durch folgendes Tupel festgelegt:
 - Objektmenge (Werte)
 - Operationen, definiert durch
 - Signatur (Name mit Definitions- und Wertebereich, Syntax)
 - Regeln/Axiome (Wirkung der Operationen, Semantik)
- Datenstruktur (= strukturierte Daten + Operationen)
 - Menge von Datentypen, zwischen denen Beziehungen bestehen

Beispiele für Datentypen in C++ (32-Bit)

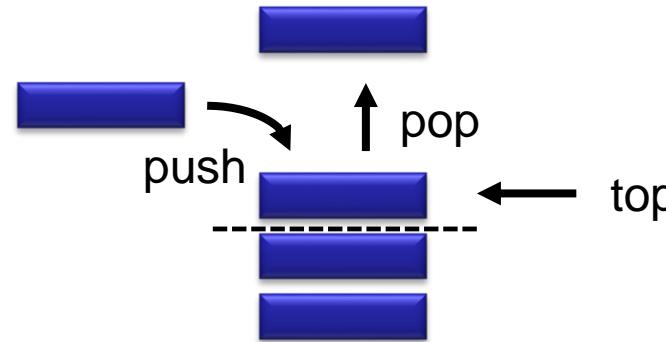
Datentyp	Bits	Wertebereich	Typische Operationen
char, signed char	8	-128 ... 127	+,-,*,/,<;>,%
unsigned char	8	0 ... 255	+,-,*,/,<;>,%
short, signed short	16	-32768 ... 32767	+,-,*,/,<;>,%
unsigned short	16	0 ... 65535	+,-,*,/,<;>,%
int, signed int	32	-2.147.483.648 ... 2.147.483.647	+,-,*,/,<;>,%
unsigned, unsigned int	32	0 ... 4.294.967.295	+,-,*,/,<;>,%
long, signed long	32	-2.147.483.648 ... 2.147.483.647	+,-,*,/,<;>,%
unsigned long	32	0 ... 4.294.967.295	+,-,*,/,<;>,%
float	32	$1,2 \cdot 10^{-38}$... $3,4 \cdot 10^{38}$	+,-,*,/,<;>
double	64	$2,2 \cdot 10^{-308}$... $1,8 \cdot 10^{308}$	+,-,*,/,<;>
long double	96	$3,4 \cdot 10^{-4932}$... $1,1 \cdot 10^{4932}$	+,-,*,/,<;>

Beispiel: Abstrakter Datentyp Boolean (bool)

- Boolean = (Objekte O_B , Funktionen F_B) mit
 - $O_B = \{ w, f \}$ // oder: {wahr, falsch} = {true, false}
 - $F_B = \{ \neg, \wedge, \vee \}$ // oder: {nicht, und, oder} = {not, and, or}
- Signaturen
 - $w: \rightarrow O_B$
 - $f: \rightarrow O_B$
 - $\neg: O_B \rightarrow O_B$ // Negation
 - $\wedge: O_B \times O_B \rightarrow O_B$ // Konjunktion
 - $\vee: O_B \times O_B \rightarrow O_B$ // Disjunktion
- Regeln/Axiome (x, y, z seien vom ADT Boolean)
 - $x \wedge y = y \wedge x$, $x \vee y = y \vee x$ (Kommutativgesetze)
 - $(x \wedge y) \wedge z = x \wedge (y \wedge z)$, $(x \vee y) \vee z = x \vee (y \vee z)$ (Assoziativgesetze)
 - $(x \wedge y) \vee z = (x \vee z) \wedge (y \vee z)$, $(x \vee y) \wedge z = \dots$ (Distributivgesetze)
 - $x \wedge w = x$, $x \vee f = x$, $x \wedge \neg x = f$, $x \vee \neg x = w$ (Neutralität, Komplement)
 - $\neg w = f$, $\neg f = w$ (Dualität)

Beispiel: Datenstruktur Stapel (Keller, Stack)

- Ein Stapel ist eine Datenstruktur, die eine begrenzte Menge von Objekten eines Datentyps aufnehmen kann und folgende Operationen unterstützt (Last-In-First-Out, LIFO-Prinzip):
 - create: Erzeugt einen leeren Keller
 - push: Legt ein Objekt auf den Stapel
 - pop: Holt u. entfernt das oberste Objekt vom Stapel
 - top: Holt das oberste Objekt vom Stapel (ohne entfernen)
 - empty: Liefert wahr, wenn der Stapel leer ist, sonst falsch



- Ein Stapel kann mit Hilfe eines Feldes implementiert werden

Implementierung eines Stapels in C++ (I)

```
typedef int object; // O.B.d.A. seien die Objekte vom Typ int

class stack1 {

    private:

        object *o; // Zeiger auf ein dynamisches Feld
        int size; // Groesse des Stapels
        int tp; // Oberstes Element

    public:

        stack1(int n); // Erzeugt leeren Stapel, n Objekte
        ~stack1(); // Gibt einen Stapel wieder frei
        void push(object o); // Legt Objekt auf den Stapel
        object pop(); // Holt u. entfernt oberstes Objekt
        object top(); // Liefert das oberste Objekt
        bool IsEmpty(); // Liefert, ob der Stapel leer ist
        bool IsFull(); // Liefert, ob der Stapel voll ist
};
```

Implementierung eines Stapels in C++ (II)

```
stack1::stack1(int n) {  
    size=n;  
    tp=-1; •  
    o=new object[size]; } •
```

```
stack1::~stack1() {  
    delete[] o; }
```

Was ändert sich, wenn
hier **tp=0** steht?

```
void stack1::push(object o) {  
    if (!IsFull()) this->o[++tp]=o; }
```

```
object stack1::pop() {  
    if (!IsEmpty()) return(o[tp--]);  
    else ... Fehlerbehandlung ... }
```

```
object stack1::top() {  
    if (!IsEmpty()) return(o[tp]);  
    else ... Fehlerbehandlung ... }
```

Welchen **Aufwand** haben
die Operationen?

```
bool stack1::IsEmpty() {  
    return (tp > -1 ? false : true); }
```

```
bool stack1::IsFull() {  
    return (tp ≥ size-1 ? true : false);  
}
```

Analyse und Design von Algorithmen

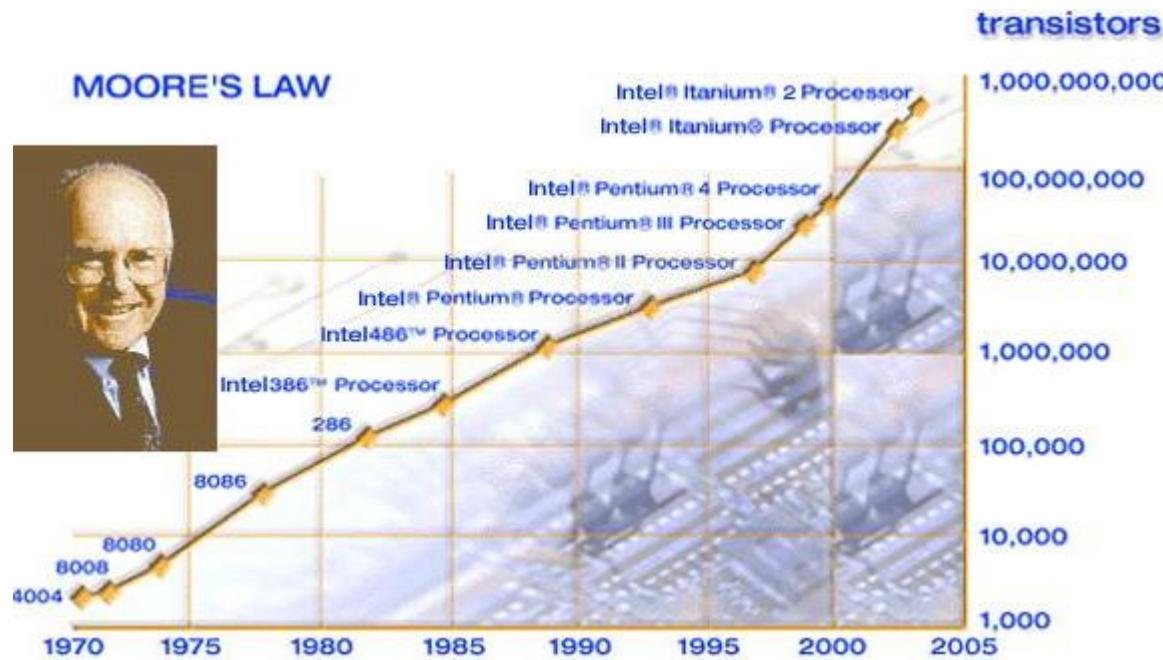
- Algorithmen stehen im Mittelpunkt der Informatik
- Hauptziel beim Entwurf von Algorithmen
 - Korrekte Problemlösung (totale Korrektheit)
 - Terminiertheit: Der Algorithmus endet für jede spezifizierte Eingabe
 - Partielle Korrektheit: Der Algorithmus liefert für jede spezifizierte Eingabe das geforderte Ergebnis
- Nebenziel beim Entwurf von Algorithmen
 - Effiziente Problemlösung hinsichtlich Zeit und Platz
- Interessant sind meist nur effiziente Algorithmen
- Wesentliche Effizienzmaße
 - Rechenzeitbedarf
 - Zeitkomplexität: Zählen der atomaren Schritte
 - Speicherplatzbedarf
 - Platzkomplexität: Zählen der verwendeten Speicherzellen

Komplexität von Algorithmen

- Es gibt unendlich viele Algorithmen zur Lösung von Problemen (in Wissenschaft, Technik, Wirtschaft, ...)
- Funktional gleichwertige Algorithmen können sich erheblich in der Effizienz/Komplexität unterscheiden
- Ein Algorithmus ist umso effizienter, je weniger er von den beiden Ressourcen Rechenzeit und Speicherplatz verwendet
- Komplexität hängt u.a. von der Eingabe für das Programm ab
- Faktoren zur Bestimmung der Komplexität
 - Messungen auf einer konkreten Maschine
 - Aufwandsermittlung in einem idealisierten Rechnermodell (z.B. RAM)
 - Asymptotische Komplexitätsabschätzung durch ein abstraktes Komplexitätsmaß in Abhängigkeit der Problem-/Eingabegröße

Moore's Gesetz

- Dr. Gordon E. Moore: Mitgründer der Firma Intel



- Verdoppelung der Prozessorleistung alle ~ 18-24 Monate
- Computer werden immer kleiner, schneller und billiger

Idealisierter Rechner: Registermaschine

- Registermaschine (engl. Random Access Machine, RAM)
 - Zentrale Recheneinheit inkl. Rechen- und Steuerwerk
 - Akkumulator (Adresse 0 im Datenspeicher)
 - Befehlsregister
 - Befehlszähler (Programm Counter, PC)
 - Programmspeicher (Program Memory, PM, Adressen 1, 2, ...)
 - Datenspeicher (Data Memory, DM, Adressen 1, 2, ...)
 - Vereinfachte Ein-/Ausgabe durch direkte Befehle
- Inhalt des Datenspeichers sei beschrieben durch $f: \mathbb{N}_0 \rightarrow \mathbb{N}_0$
 - $f(0)$ ist der **Inhalt des Akkumulators**
 - $f(\text{adresse})$ ist der **Inhalt des Datenspeichers** an der Stelle *adresse*
- Befehlsstruktur
 - <Adresse PM> <Befehl> <Adresse PM/DM>

8 Bit

8 Bit

Befehle der Registermaschine I

- Verarbeitungs-, Transport- und Ein-/Ausgabebefehle:

Codierung	Syntax	Semantik	Beschreibung
0x01	ADD <i>adresse</i>	$f(0) = f(0) + f(\text{adresse})$	Addieren
0x02	SUB <i>adresse</i>	$f(0) = f(0) - f(\text{adresse})$	Subtrahieren
0x03	MUL <i>adresse</i>	$f(0) = f(0) * f(\text{adresse})$	Multiplizieren
0x04	DIV <i>adresse</i>	$f(0) = f(0) / f(\text{adresse})$	Dividieren
0x05	LDA <i>adresse</i>	$f(0) = f(\text{adresse})$	Laden
0x06	LDK <i>zahl</i>	$f(0) = \text{zahl}$	Konstante Laden
0x07	STA <i>adresse</i>	$f(\text{adresse}) = f(0)$	Speichern
0x08	INP <i>adresse</i>	$f(\text{adresse}) = \langle \text{Eingabe} \rangle$	Eingeben
0x09	OUT <i>adresse</i>	$\langle \text{Ausgabe} \rangle = f(\text{adresse})$	Ausgeben
0x0A	HLT 99		Programmende



Warum ist hier eine Zahl sinnvoll?

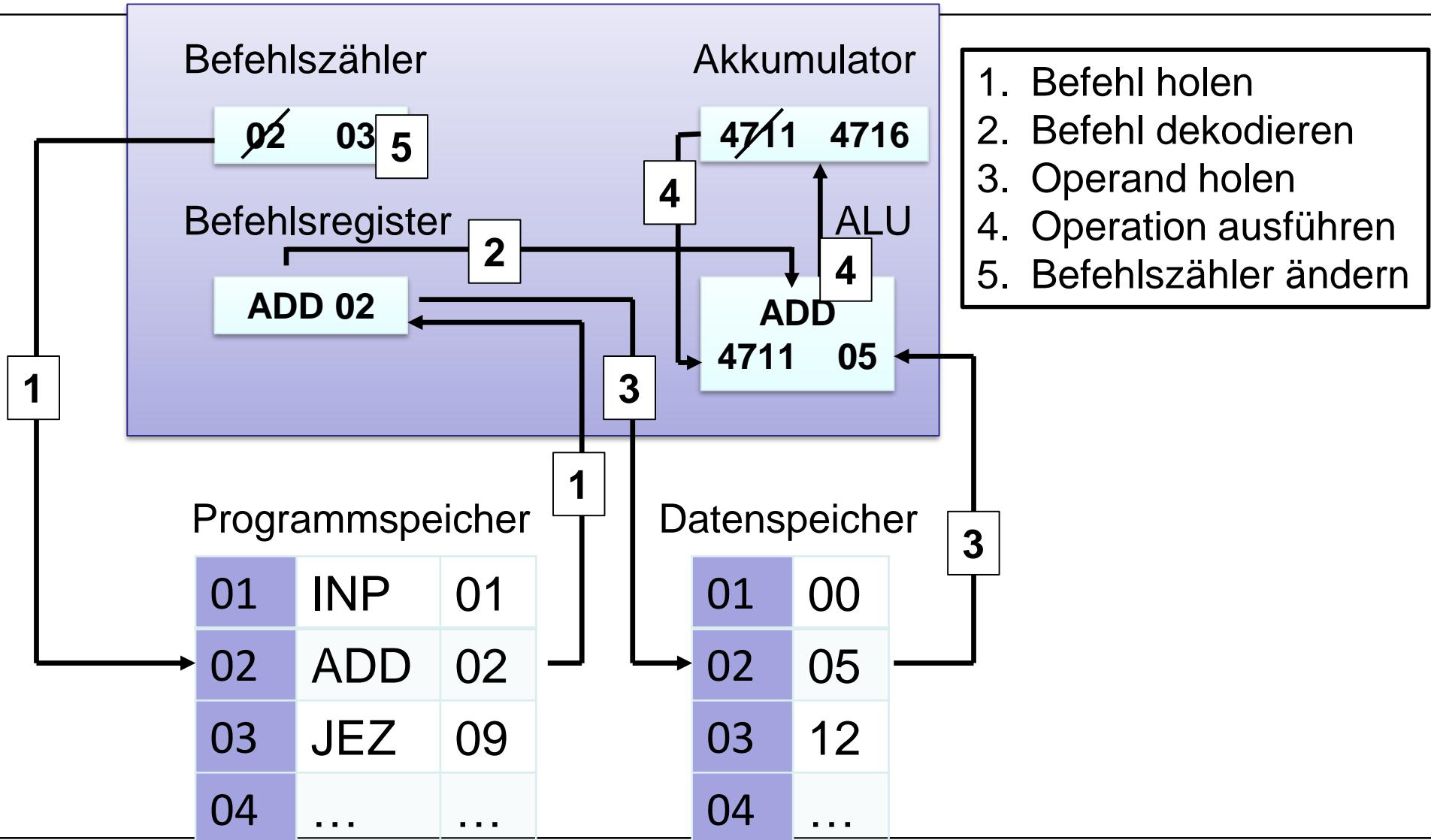
Befehle der Registermaschine II

- Sprungbefehle:

Codierung	Syntax	Semantik	Beschreibung
0x0B	JMP <i>adresse</i>	PC = <i>adresse</i>	Verzweigen
0x0C	JEZ <i>adresse</i>	Falls $f(0) = 0$, dann PC = <i>adresse</i>	
0x0D	JNE <i>adresse</i>	Falls $f(0) \neq 0$, dann PC = <i>adresse</i>	
0x0E	JLZ <i>adresse</i>	Falls $f(0) < 0$, dann PC = <i>adresse</i>	
0x0F	JLE <i>adresse</i>	Falls $f(0) \leq 0$, dann PC = <i>adresse</i>	
0x10	JGZ <i>adresse</i>	Falls $f(0) > 0$, dann PC = <i>adresse</i>	
0x11	JGE <i>adresse</i>	Falls $f(0) \geq 0$, dann PC = <i>adresse</i>	

- Nach jedem Befehl, wenn es **keine Verzweigung** gab:
 - $PC = PC + 1$

Befehlszyklus der Registermaschine



Beispielprogramm

- Was tut folgendes Programm?
 - 06000702080105010C09010207020B0309020A99

01	06	00
02	07	02
03	08	01
04	05	01
05	0C	09
06	01	02
07	07	02
08	0B	03
09	09	02
10	0A	99

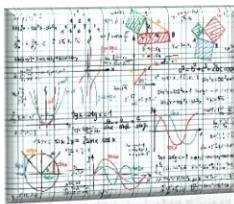


01	LDK	00
02	STA	02
03	INP	01
04	LDA	01
05	JEZ	09
06	ADD	02
07	STA	02
08	JMP	03
09	OUT	02
10	HLT	99

Akku auf 0
Akku → Adr. 2
Eingabe → Adr. 1
Adr. 1 → Akku
Akku = 0 ? Ja: 09
Akku + Adr. 2
Akku → Adr. 2
Weiter bei 03
Ausgabe Adr. 2
Programmende

- These: Die Klasse der Turing-berechenbaren Funktionen ist genau die Klasse der intuitiv berechenbaren Funktionen
- Alternative Formulierungen/Folgerungen
 - Jede Funktion, die überhaupt in irgendeiner Weise berechenbar ist, kann durch eine Turingmaschine berechnet werden
 - Jedes Problem, das überhaupt maschinell lösbar ist, kann von einer Turingmaschine gelöst werden
- Anmerkungen
 - Church'sche These ist nicht beweisbar, da „intuitiv berechenbare Funktionen“ nicht formalisiert werden können
 - Anerkanntes Rechenmodell: Von-Neumann-Rechner
 - Idealisierte Von-Neumann-Rechner: Registermaschinen
 - Registermaschinen sind äquivalent zu Turingmaschinen
- Church'sche These gilt als allgemein akzeptiert

Church'sche These (grafische Interpretation)



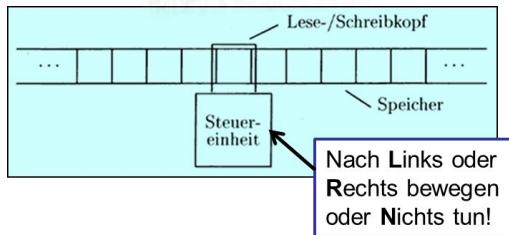
```

public class MeineErsteKlasse {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}

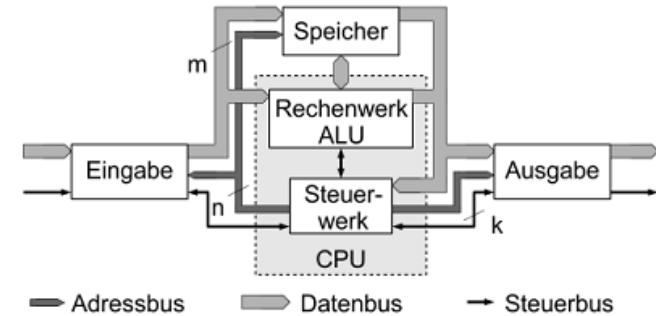
```

Console output: Hello World!

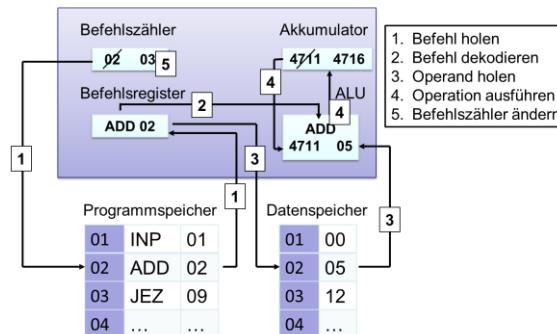
C, C++, C#, Java Programm



Turingmaschine (TM)



Von-Neumann-Rechner



Registermaschine (RAM)

Algorithmus-Begriff

- Interpretation der Church-Turing-Theorie:
 - Bisher und in Zukunft vorgenommene „vernünftige“ Definitionen von Algorithmus sind gleichwertig und haben die gleiche Bedeutung wie die bisher bekannten Definitionen!
 - Algorithmus = Programm für eine TM
 - = Programm für eine RAM
 - = Programm für andere Modelle
(Goto, While, μ -Rekursion)
 - = Programm in C/C++
 Pascal
 Java
 C#, ...

- Uniforme Komplexität (Einheitsmaß)
 - Jeder Befehl, der ausgeführt wird, entspricht einem Schritt (**konstant**)
 - Sei A ein Algorithmus für eine RAM mit Eingabe x
 - $T_A(x)$: Anzahl der Schritte, die A bei Eingabe von x durchführt
 - $S_A(x)$: Anzahl der Speicherzellen, die A bei Eingabe von x benutzt
- Nachteil der uniformen Komplexität
 - Befehle mit unterschiedlicher **Bit-Komplexität** werden gleich bewertet
- Logarithmische Komplexität (Logarithmisches Maß)
 - Jeder Befehl, der ausgeführt wird, wird mit der Bit-Länge der Operanden des Befehls gewichtet (entspricht der Bit-Komplexität)
 - Anstelle einer Speicherzelle wird die Bit-Länge der in einer Speicherzelle abgespeicherten größten Zahl verwendet
 - Kennzeichnung analog zu oben: $T_A^{\log}(x)$ bzw. $S_A^{\log}(x)$
- Beispiel: Addition und Multiplikation von zwei n -Bit Zahlen

Worst Case, Average Case, Best Case I

- Komplexität wird **über alle Eingaben** unterschieden nach
 - Worst Case: Laufzeit im schlechtesten Fall (Wesentlich!)
 - Average Case: Laufzeit im Mittel (Mittelwert)
 - Best Case: Laufzeit im besten Fall (Eher uninteressant)
- Beispiel: Addition um eins im Binärsystem
 - Eingabe: $\text{bin}(a) = (a_{n-1}, \dots, a_0)$, $a \in \mathbb{N}$, $a_i \in \{0,1\}$
(Binärdarstellung einer natürlichen Zahl)
 - Ausgabe: $\text{bin}(a) + 1$
 - Algorithmus:
 - Falls $(a_{n-1}, \dots, a_0) = (1, \dots, 1)$
 Gib $(1,0 \dots, 0)$ aus // n Nullen
 - Sonst
 - Ermittle i mit $a_i = 0$ und $a_j \neq 0$ für alle $j < i$
 - Gib $(a_{n-1}, \dots, a_{i+1}, 1, 0, \dots, 0)$ aus // i Nullen
 - Laufzeit:
 - Worst Case: $n + 1$ Schritte // Wann?
 - Best Case: 1 Schritt // Wann?

Worst Case, Average Case, Best Case II

- Average Case bei Addition um 1 im Binärsystem:
 - Wie viele mögliche Eingaben für a gibt es? 2^n
 - Wie viele Eingaben gibt es für den Worst Case (1. Fall)? 1
 - Laufzeit in diesem Fall: $n + 1$ Schritte
 - Wie viele Eingaben gibt es mit $a_i = 0$ und $a_j \neq 0$ für alle $j < i$? 2^{n-i-1}
 - Laufzeit in diesen Fällen: $(i + 1)$ Schritte
 - Insgesamt ergibt sich für die **durchschnittliche Laufzeit**:

$$\frac{\sum_{i=0}^{n-1} (2^{n-i-1} \cdot (i + 1)) + n + 1}{2^n} \text{ Schritte}$$

- Es gilt:

$$\sum_{i=0}^{n-1} 2^{n-i-1} \cdot (i + 1) = \sum_{i=1}^n 2^{n-i} \cdot i$$

Worst Case, Average Case, Best Case III

- Weiter gilt:

$$\begin{aligned}\sum_{i=1}^n 2^{n-i} \cdot i &= 2^{n-1} \cdot 1 + 2^{n-2} \cdot 2 + \dots + 2^0 \cdot n \\&= 2^{n-1} + 2^{n-2} + \dots + 2^0 \quad (\text{Zeile 1}) \\&\quad + 2^{n-2} + \dots + 2^0 \quad (\text{Zeile 2}) \\&\quad \vdots \quad \vdots \quad \vdots \\&\quad + 2^0 \quad (\text{Zeile } n)\end{aligned}$$

- Erinnerung Geometrische Reihe:

$$\sum_{k=0}^m q^k = \frac{q^{m+1} - 1}{q - 1} \text{ für } q \neq 1$$

- Damit folgt:

$$\sum_{i=1}^n 2^{n-i} \cdot i = \frac{2^{n-1+1} - 1}{2 - 1} + \frac{2^{n-2+1} - 1}{2 - 1} + \dots + \frac{2^{n-n+1} - 1}{2 - 1}$$

Worst Case, Average Case, Best Case IV

- Es folgt schließlich:

$$\begin{aligned}\sum_{i=1}^n 2^{n-i} \cdot i &= (2^n - 1) + (2^{n-1} - 1) + \dots + (2^1 - 1) \\ &= \sum_{i=1}^n 2^i - n = 2^{n+1} - 2 - n\end{aligned}$$

- Für die **durchschnittliche Laufzeit** folgt:

$$\frac{(2^{n+1} - 2 - n) + (n + 1)}{2^n} = 2 - \frac{1}{2^n} \text{ Schritte}$$

- Zusammenfassend:

- Worst Case: $n + 1$ Schritte
- Average Case: 2 Schritte (asymptotisch)
- Best Case: 1 Schritt

- Positiv: Durchschnitt liegt nah am Best Case (nicht immer so!)

Worst Case, Average Case, Best Case V

- Sei A ein Algorithmus für eine RAM mit Eingabe x über einem Alphabet Σ
- Laufzeit im schlechtesten Fall (**Worst Case**)

$$T_A^{wc}(n) = \max_{x \in (\Sigma)^n} \{T_A(x)\}$$

- Laufzeit im besten Fall (**Best Case**)

$$T_A^{bc}(n) = \min_{x \in (\Sigma)^n} \{T_A(x)\}$$

- Laufzeit im Durchschnitt (**Average Case**)

$$T_A^{ac}(n) = \frac{1}{|\Sigma|^n} \sum_{x \in (\Sigma)^n} T_A(x)$$

Zeitkomplexität von *C, C++, C#, Java - Programmen*

- Alle **ausgeführten** Anweisungen werden „mit 1“ gezählt
- Anweisungen sind
 - Zuweisungen
 - Auswahl-Anweisungen (if-then-else, switch-case)
 - Iterationen (for, while-do, do-while)
 - Sprünge (break, continue, goto, return)
 - Funktionsaufrufe/Methodenaufrufe
- Bei **Iterationen und Funktionsaufrufen/Methodenaufrufen**
 - Alle im Rahmen der Iteration oder der Funktion/Methode ausgeführten Anweisungen werden natürlich auch gezählt (iterativ oder rekursiv)
- Vorsicht:
 - Funktions-/Methodenaufrufe in höheren Programmiersprachen sind nicht sofort ersichtlich (Beispiele: Konstruktor, Destruktor, Boolesche Abfragen, ...)
- Einfache und komplexe Zuweisungen werden nicht unterschieden
 - Ziel: Ermittlung des **asymptotischen Laufzeitverhaltens**

Algorithmen und Datenstrukturen

- Grundlagen (Komplexität) -

Prof. Dr. Klaus Volbert

Wintersemester 2018/19
Regensburg, 08. Oktober 2018

- Uniforme Komplexität (Einheitsmaß)
 - Jeder Befehl, der ausgeführt wird, entspricht einem Schritt (**konstant**)
 - Sei A ein Algorithmus für eine RAM mit Eingabe x
 - $T_A(x)$: Anzahl der Schritte, die A bei Eingabe von x durchführt
 - $S_A(x)$: Anzahl der Speicherzellen, die A bei Eingabe von x benutzt
- Nachteil der uniformen Komplexität
 - Befehle mit unterschiedlicher **Bit-Komplexität** werden gleich bewertet
- Logarithmische Komplexität (Logarithmisches Maß)
 - Jeder Befehl, der ausgeführt wird, wird mit der Bit-Länge der Operanden des Befehls gewichtet (entspricht der Bit-Komplexität)
 - Anstelle einer Speicherzelle wird die Bit-Länge der in einer Speicherzelle abgespeicherten größten Zahl verwendet
 - Kennzeichnung analog zu oben: $T_A^{\log}(x)$ bzw. $S_A^{\log}(x)$
- Beispiel: Addition und Multiplikation von zwei n -Bit Zahlen

Worst Case, Average Case, Best Case I

- Komplexität wird **über alle Eingaben** unterschieden nach
 - Worst Case: Laufzeit im schlechtesten Fall (Wesentlich!)
 - Average Case: Laufzeit im Mittel (Mittelwert)
 - Best Case: Laufzeit im besten Fall (Eher uninteressant)
- Beispiel: Addition um eins im Binärsystem
 - Eingabe: $\text{bin}(a) = (a_{n-1}, \dots, a_0)$, $a \in \mathbb{N}$, $a_i \in \{0,1\}$
(Binärdarstellung einer natürlichen Zahl)
 - Ausgabe: $\text{bin}(a) + 1$
 - Algorithmus:
 - Falls $(a_{n-1}, \dots, a_0) = (1, \dots, 1)$
 Gib $(1,0 \dots, 0)$ aus // n Nullen
 - Sonst
 - Ermittle i mit $a_i = 0$ und $a_j \neq 0$ für alle $j < i$
 - Gib $(a_{n-1}, \dots, a_{i+1}, 1, 0, \dots, 0)$ aus // i Nullen
 - Laufzeit:
 - Worst Case: $n + 1$ Schritte // Wann?
 - Best Case: 1 Schritt // Wann?

Worst Case, Average Case, Best Case II

- Average Case bei Addition um 1 im Binärsystem:
 - Wie viele mögliche Eingaben für a gibt es? 2^n
 - Wie viele Eingaben gibt es für den Worst Case (1. Fall)? 1
 - Laufzeit in diesem Fall: $n + 1$ Schritte
 - Wie viele Eingaben gibt es mit $a_i = 0$ und $a_j \neq 0$ für alle $j < i$? 2^{n-i-1}
 - Laufzeit in diesen Fällen: $(i + 1)$ Schritte
 - Insgesamt ergibt sich für die **durchschnittliche Laufzeit**:

$$\frac{\sum_{i=0}^{n-1} (2^{n-i-1} \cdot (i + 1)) + n + 1}{2^n} \text{ Schritte}$$

- Es gilt:

$$\sum_{i=0}^{n-1} 2^{n-i-1} \cdot (i + 1) = \sum_{i=1}^n 2^{n-i} \cdot i$$

Worst Case, Average Case, Best Case III

- Weiter gilt:

$$\begin{aligned}\sum_{i=1}^n 2^{n-i} \cdot i &= 2^{n-1} \cdot 1 + 2^{n-2} \cdot 2 + \dots + 2^0 \cdot n \\&= 2^{n-1} + 2^{n-2} + \dots + 2^0 \quad (\text{Zeile 1}) \\&\quad + 2^{n-2} + \dots + 2^0 \quad (\text{Zeile 2}) \\&\quad \vdots \quad \vdots \quad \vdots \\&\quad + 2^0 \quad (\text{Zeile } n)\end{aligned}$$

- Erinnerung Geometrische Reihe:

$$\sum_{k=0}^m q^k = \frac{q^{m+1} - 1}{q - 1} \text{ für } q \neq 1$$

- Damit folgt:

$$\sum_{i=1}^n 2^{n-i} \cdot i = \frac{2^{n-1+1} - 1}{2 - 1} + \frac{2^{n-2+1} - 1}{2 - 1} + \dots + \frac{2^{n-n+1} - 1}{2 - 1}$$

Worst Case, Average Case, Best Case IV

- Es folgt schließlich:

$$\begin{aligned} \sum_{i=1}^n 2^{n-i} \cdot i &= (2^n - 1) + (2^{n-1} - 1) + \dots + (2^1 - 1) \\ &= \sum_{i=1}^n 2^i - n = 2^{n+1} - 2 - n \end{aligned}$$

- Für die **durchschnittliche Laufzeit** folgt:

$$\frac{(2^{n+1} - 2 - n) + (n + 1)}{2^n} = 2 - \frac{1}{2^n} \text{ Schritte}$$

- Zusammenfassend:

- Worst Case: $n + 1$ Schritte
- Average Case: 2 Schritte (asymptotisch)
- Best Case: 1 Schritt

- Positiv: Durchschnitt liegt nah am Best Case (nicht immer so!)

Worst Case, Average Case, Best Case V

- Sei A ein Algorithmus für eine RAM mit Eingabe x über einem Alphabet Σ
- Laufzeit im schlechtesten Fall (**Worst Case**)

$$T_A^{wc}(n) = \max_{x \in (\Sigma)^n} \{T_A(x)\}$$

- Laufzeit im besten Fall (**Best Case**)

$$T_A^{bc}(n) = \min_{x \in (\Sigma)^n} \{T_A(x)\}$$

- Laufzeit im Durchschnitt (**Average Case**)

$$T_A^{ac}(n) = \frac{1}{|\Sigma|^n} \sum_{x \in (\Sigma)^n} T_A(x)$$

Zeitkomplexität von *C, C++, C#, Java - Programmen*

- Alle **ausgeführten** Anweisungen werden „mit 1“ gezählt
- Anweisungen sind
 - Zuweisungen
 - Auswahl-Anweisungen (if-then-else, switch-case)
 - Iterationen (for, while-do, do-while)
 - Sprünge (break, continue, goto, return)
 - Funktionsaufrufe/Methodenaufrufe
- Bei **Iterationen und Funktionsaufrufen/Methodenaufrufen**
 - Alle im Rahmen der Iteration oder der Funktion/Methode ausgeführten Anweisungen werden natürlich auch gezählt (iterativ oder rekursiv)
- Vorsicht:
 - Funktions-/Methodenaufrufe in höheren Programmiersprachen sind nicht sofort ersichtlich (Beispiele: Konstruktor, Destruktor, Boolesche Abfragen, ...)
- Einfache und komplexe Zuweisungen werden nicht unterschieden
 - Ziel: Ermittlung des **asymptotischen Laufzeitverhaltens**

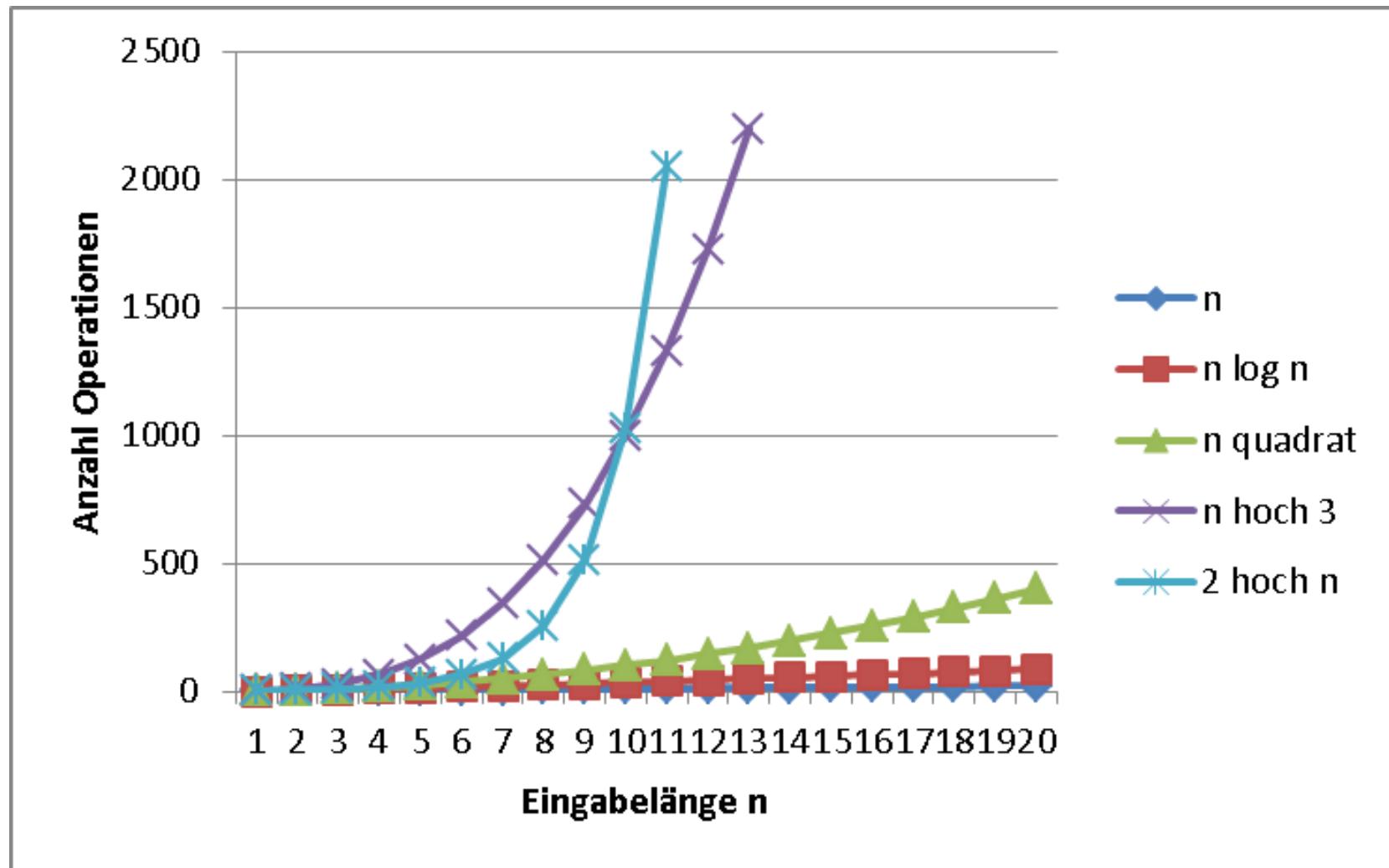
Beispiel: Laufzeitkomplexität

- Annahme:
 - Rechner kann 1.000 Operationen pro Sekunde ausführen (1.000 Hz)
 - Die folgende Tabelle zeigt jeweils die mögliche Problemgröße (Wert für n) bei vorgegebener Rechenzeit

Komplexität	1 Sekunde	1 Minute	1 Stunde	1 Tag	1 Woche
Linear n	1.000	60.000	3.600.000	86.400.000	604.800.000
Super-Linear $n \log_2 n$	140	4.895	204.094	3.943.234	24.631.427
Polynomiell n^2	31	244	1897	9.295	24.592
	n^3	10	39	153	442
Exponentiell 2^n	9	15	21	26	29

- Effekt der technischen Entwicklung ist
 - bei linearem Laufzeitverhalten: **ideal**
 - bei polynomiellem Laufzeitverhalten: **akzeptabel**
 - bei exponentiellem Laufzeitverhalten: **sehr schlecht**

Wesentliche Wachstumsfunktionen



Asymptotische Kostenmaße

(Landau-Symbole, O-Notation)

- Größenordnung der Komplexität in **Abhängigkeit der Eingabegröße**
 - Best Case, Worst Case, Average Case
- Seien $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$ zwei beliebige Funktionen, dann definieren wir
 - Groß-O-Notation (O-Notation, O-Kalkül), Obere Schranke:

g wächst höchstens wie f

$$O(f(n)) = \{ g(n) \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}: \forall n \geq n_0, g(n) \leq c \cdot f(n) \}$$

- Verwendung des Logarithmus im O-Kalkül:
 - Mit $\log(n)$ ist $\log_2(n)$ gemeint, d.h. der Logarithmus zur Basis 2
 - Zielbasis a ist unwesentlich, da:

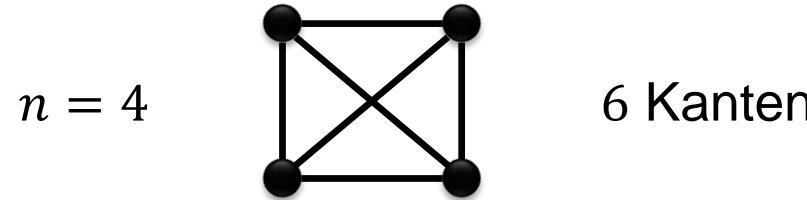
$$\log_b n = \frac{\log_a n}{\log_a b} = \frac{1}{\log_a b} \log_a n \quad (\log_a b \text{ ist konstant})$$

- Abstrakte Abschätzung zur Klassifizierung
 - Extrahiert dominante Terme (Optimierung von Konstanten oft ineffektiv)
 - Abstrahiert von Konstanten (Versteckte Konstanten können groß sein!)
 - Vernachlässigt „geringfügige“ Terme
- Rechenregeln

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \leq c \Rightarrow g(n) \in O(f(n))$$

- $\forall f: f(n) \in O(f(n))$
- $\forall f: \forall c \in \mathbb{R}^+: c f(n) \in O(f(n)), c + f(n) \in O(f(n))$
- $\forall g \in O(f(n)): \forall c \in \mathbb{R}^+: g(n) + g(n) \in O(f(n)), c g(n) \in O(f(n))$
- $\forall g_1 \in O(f_1(n)): \forall g_2 \in O(f_2(n)): g_1(n) + g_2(n) \in O(f_1(n) + f_2(n))$
- $\forall g_1 \in O(f_1(n)): \forall g_2 \in O(f_2(n)): g_1(n) + g_2(n) \in O(\max(f_1(n), f_2(n)))$
- $\forall g_1 \in O(f_1(n)): \forall g_2 \in O(f_2(n)): g_1(n) g_2(n) \in O(f_1(n) f_2(n))$

- Wie lautet die **minimale obere** Schranke für $3n^3 + 4n^2 + 5n + 42$?
 - $3n^3 + 4n^2 + 5n + 42 \in O(n^3)$, da:
 - Aus $3n^3 + 4n^2 + 5n + 42 \leq c \cdot n^3$ folgt:
$$3 + \frac{4}{n} + \frac{5}{n^2} + \frac{42}{n^3} \leq 3 + 4 + 5 + 42 = 54, \text{ d. h. } c \geq 54$$
- Anzahl der Kanten in einem vollständigen Graph mit n Knoten ist $O(n)$?
 - Falsch**, da:



- Anzahl Kanten, die ein Graph mit n Knoten hat, ist:

$$n - 1 + n - 2 + \dots + 0 = \sum_{i=0}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2 - n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \in O(n^2)$$

Erweiterungen der O-Notation

- Seien $f, g: \mathbb{N} \rightarrow \mathbb{R}^+$ zwei beliebige Funktionen, dann definieren wir
 - Groß-Omega-Notation, Untere Schranke:

g wächst mindestens wie f

$$\Omega(f(n)) = \{ g(n) \mid \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N}: \forall n \geq n_0, g(n) \geq c \cdot f(n) \}$$

- Groß-Theta-Notation, Exakte Schranke:

g wächst wie f

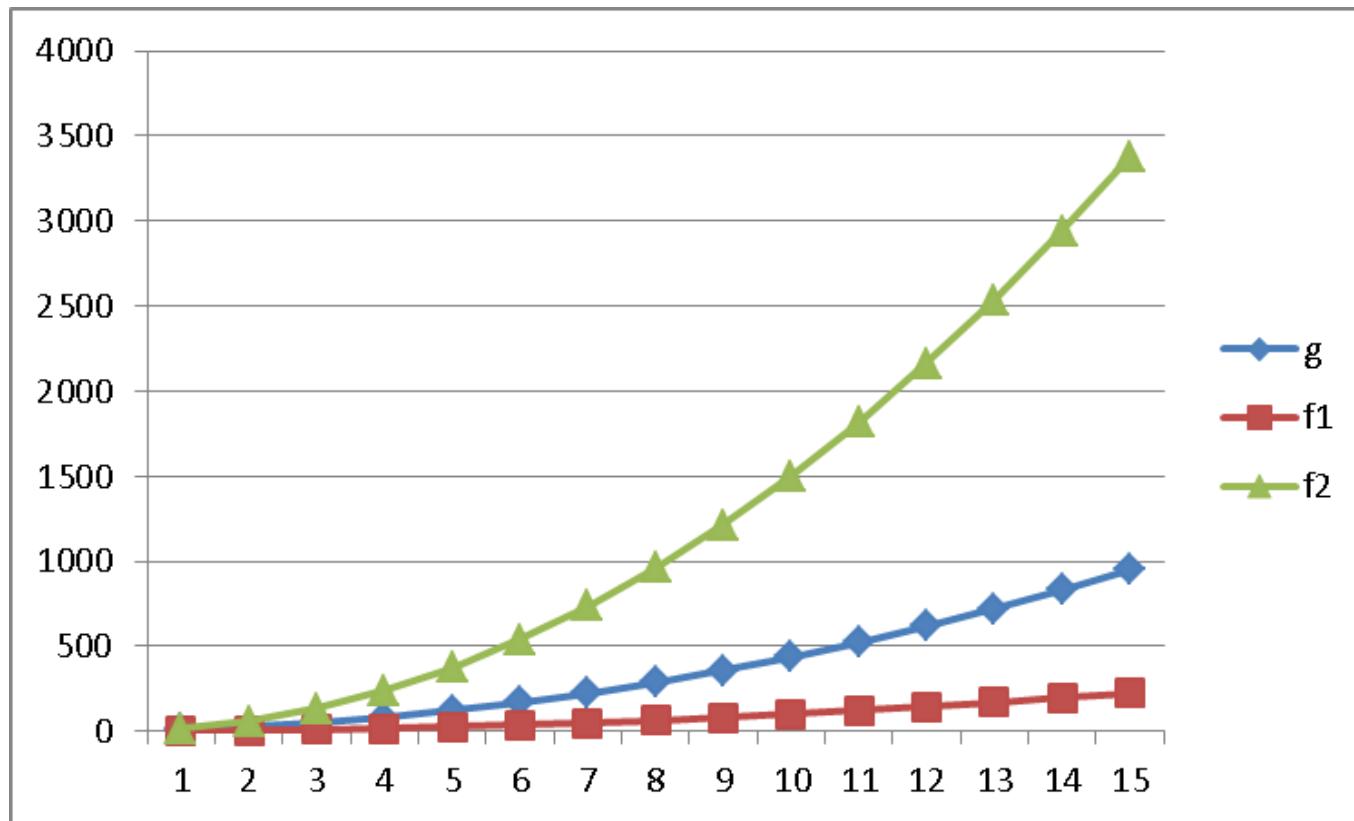
$$\Theta(f(n)) = \{ g(n) \mid g(n) \in O(f(n)) \text{ und } g(n) \in \Omega(f(n)) \}$$

- Seltener verwendet: o wie O, aber < statt \leq und ω wie Ω , aber > statt \geq
- Schreibweise:
 - Statt $g(n) \in O(f(n))$ wird auch $g(n) = O(f(n))$ (analog für Ω und Θ)
 - Vorsicht beim Umgang, da es sich nicht um ein echtes „=“ handelt !!!

Beispiel zur Θ -Notation

- Betrachte $g(n) = 4n^2 + 3n + 7$
- Es gilt $g(n) = \Theta(n^2)$, da für alle n :

$$f_1(n) := n^2 \leq g(n) \leq 15n^2 =: f_2(n)$$



Beispiel MaxTeilSum

- Eingabe: $a_0, \dots, a_{n-1} \in \mathbb{Z}$ (n ganze Zahlen)
- Ausgabe: Maximale Teilsumme, d.h.

$$s = \max_{0 \leq i \leq j \leq n-1} \sum_{k=i}^j a_k$$

- Anwendungen
 - Erkennung von grafischen Mustern
 - Analyse von Aktienkursen
(täglich neue Kurse: Ermittlung bester Ein-/Ausstiegszeitpunkt)
- Beispiel:
 - Eingabe: -13, 25, 34, 12, -3, 7, -87, 28, -77, 11
 - Ausgabe: 75 (ergibt sich aus $i = 1, j = 5$)

- Naiver Algorithmus: Durchlaufen aller Varianten

```
int MaxTeilsum1(int a[], int n) {
    int i, j, k, sum, max = int.MinValue;

    for (i=0; i<n; i++) {
        for (j=i; j<n; j++) {
            sum=0;
            for (k=i; k<=j; k++) sum += a[k];
            if (sum > max) max = sum;
        }
    }

    return max;
}
```

- Laufzeit: $T_1^{wc}(n) = O(n^3)$ (**kubisch**)

Laufzeitanalyse MaxTeilSum1

- Zu verarbeitende Eingabedaten: n ganze Zahlen
- Laufzeit in Abhängigkeit der Eingabe sei $T_1^{wc}(n)$, dann gilt:

$$\begin{aligned} T_1^{wc}(n) &= 1 + \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \left(\sum_{k=i}^j 1 + 2 \right) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 3) = \sum_{i=0}^{n-1} \sum_{k=0}^{n-1-i} (k + 3) \\ &= \sum_{i=0}^{n-1} \sum_{k=0}^i k + \sum_{i=0}^{n-1} \sum_{k=0}^i 3 = \sum_{i=0}^{n-1} \frac{i(i+1)}{2} + \sum_{i=0}^{n-1} 3(i+1) \\ &= \frac{1}{2} \left(\sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i \right) + 3 \sum_{i=0}^{n-1} i + n \\ &= \frac{1}{2} \left(\frac{(n-1)n(2(n-1)+1)}{6} + \frac{(n-1)n}{2} \right) + \frac{3(n-1)n}{2} = \dots = \Theta(n^3) \end{aligned}$$

- Verbesserung: Statt immer wieder vollständige Summe zu berechnen: Erweiterung um den aktuellen Summanden

```
int MaxTeilsum2(int a[], int n) {  
    int i, j, sum, max = int.MinValue;  
  
    for (i=0; i<n; i++) {  
        sum=0;  
        for (j=i; j<n; j++) {  
            sum += a[j];  
            if (sum > max) max = sum;  
        }  
    }  
  
    return max; }
```

- Laufzeit: $T_2^{wc}(n) = O(n^2)$ (quadratisch)

- Verbesserung: Berechne lokales und globales Maximum. Das globale Maximum ist das Maximum über alle lokalen Maxima. Das lokale Maximum ist an einer Position entweder der Wert oder der Wert plus Werte „von links“.

```
int MaxTeilsum3(int a[], int n) {  
    int i, s, max = int.MinValue, aktSum = 0;  
  
    for (i=0; i<n; i++) {  
        s = aktSum + a[i];  
        if (s > a[i]) aktSum = s;  
        else aktSum = a[i];  
        if (aktSum > max) max = aktSum;  
    }  
  
    return max; }
```

- Laufzeit: $T_3^{wc}(n) = O(n)$ (**linear**)

Beispiel MaxTeilSum3

- Folge

	i	aktSum	max
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11		0	-2147483648
<u>-13</u> , 25, 34, 12, -3, 7, -87, 28, -77, 11	0	-13	-13
-13, <u>25</u> , 34, 12, -3, 7, -87, 28, -77, 11	1	25	25
-13, 25, <u>34</u> , 12, -3, 7, -87, 28, -77, 11	2	59	59
-13, 25, 34, <u>12</u> , -3, 7, -87, 28, -77, 11	3	71	71
-13, 25, 34, 12, <u>-3</u> , 7, -87, 28, -77, 11	4	68	71
-13, 25, 34, 12, -3, <u>7</u> , -87, 28, -77, 11	5	75	75
-13, 25, 34, 12, -3, 7, <u>-87</u> , 28, -77, 11	6	-12	75
-13, 25, 34, 12, -3, 7, -87, <u>28</u> , -77, 11	7	28	75
-13, 25, 34, 12, -3, 7, -87, 28, <u>-77</u> , 11	8	-49	75
-13, 25, 34, 12, -3, 7, -87, 28, -77, <u>11</u>	9	11	75

MaxTeilSum3 (Korrektheit I)

- Behauptung:
 - Nach dem i .ten Schleifendurchlauf gilt (**Schleifeninvariante**):

$$\text{aktSum} = \max_{0 \leq l \leq i} \sum_{k=l}^i a_k, \quad \max = \max_{0 \leq l \leq m \leq i} \sum_{k=l}^m a_k$$

- IA: $i = 0$:
 - In der Schleife wird $s = \text{aktSum} + a[i] = 0 + a[0] = a[0]$
 - Da s nicht größer als $a[i] = a[0]$ wird $\text{aktSum} = a[i] = a[0]$
 - 1. Fall: $\text{aktSum} > \max$, dann wird $\max = \text{aktSum}$
 - 2. Fall: aktSum ist kleinster int-Wert,
dann hatte \max den Wert schon bei Initialisierung
- IV: Behauptung gilt für $i-1$

MaxTeilSum3 (Korrektheit II)

- IS: $i - 1 \rightarrow i$:

- Nach IV gilt nach dem $(i - 1)$.ten Schleifendurchlauf (Schleifeninvariante):

$$\text{aktSum} = \max_{0 \leq l \leq i-1} \sum_{k=l}^{i-1} a_k \quad \text{und} \quad \max = \max_{0 \leq l \leq m \leq i-1} \sum_{k=l}^m a_k$$

- Es wird $s = \text{aktSum} + a[i]$
- 1. Fall: $s > a[i]$, dann $\text{aktSum} = s$
- 2. Fall: sonst, dann $\text{aktSum} = a[i]$
- Falls $\text{aktSum} > \max$, dann $\max = \text{aktSum}$, d.h.

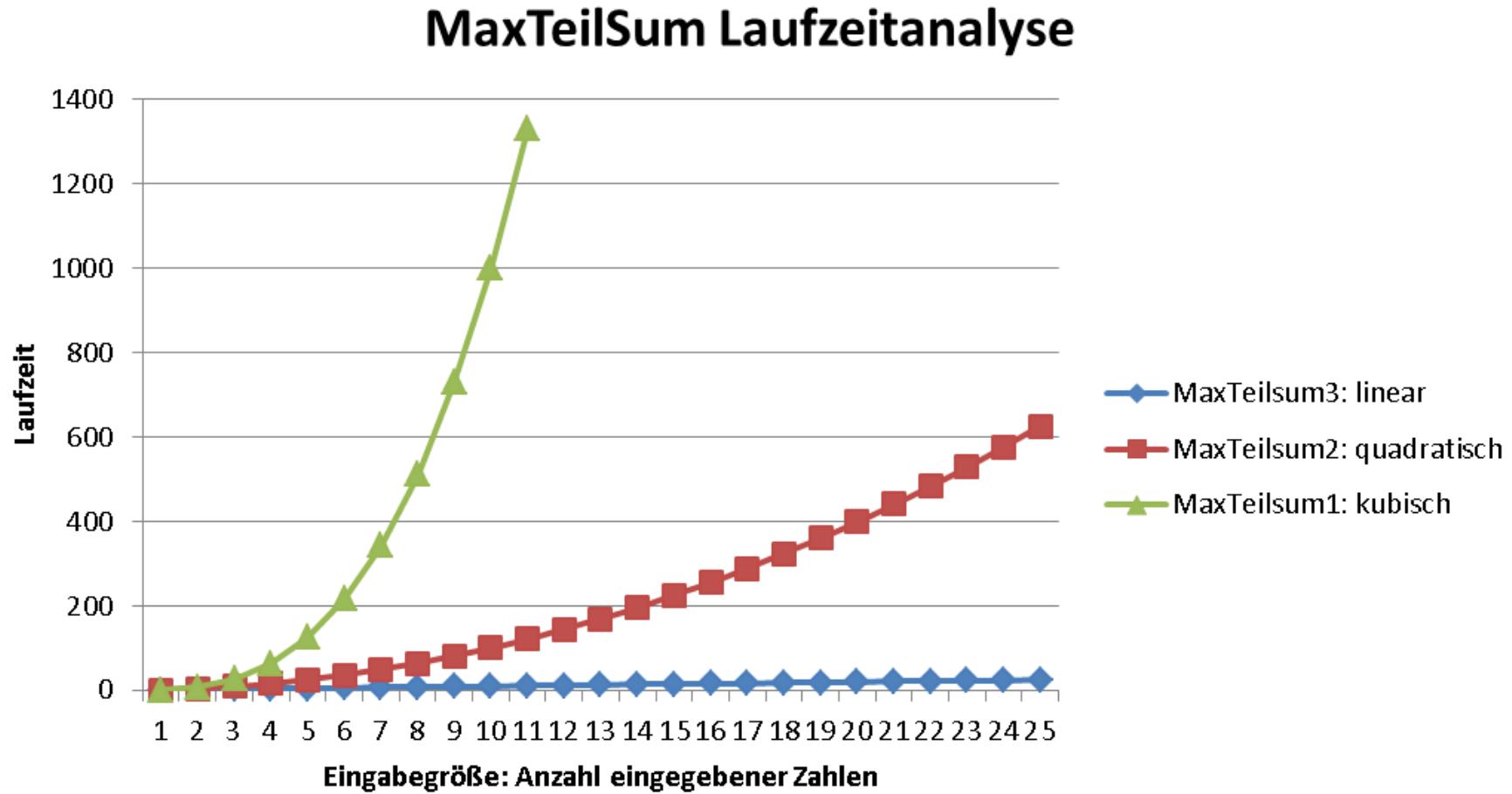
$$\max = \max \left\{ \max_{0 \leq l \leq m \leq i-1} \sum_{k=l}^m a_k, \text{aktSum} \right\} = \max_{0 \leq l \leq m \leq i} \sum_{k=l}^m a_k$$

- Nach dem letzten Schleifendurchlauf gilt: $i = n - 1$, d.h.

$$\max = \max_{0 \leq l \leq m \leq i} \sum_{k=l}^m a_k = \max_{0 \leq l \leq m \leq n-1} \sum_{k=l}^m a_k$$

Übersicht

Laufzeitverhalten MaxTeilSum



Komplexität von Algorithmen

- Ein Algorithmus A hat die **Komplexität $O(f(n))$** , wenn

$$T_A^{wc}(n) = O(f(n))$$

- Komplexität von nun an: Worst Case Laufzeitkomplexität
- Statt $T_A^{wc}(n)$ wird auch gerne einfach nur $T(n)$ verwendet
- Vorgehen
 - Bestimme $T(n)$ für den Algorithmus möglichst exakt
(Bestimmung genauer Konstanten ist oft schwierig/aufwendig)
 - Schätze $T(n)$ mit Hilfe der O-Notation ab
- Allgemein gilt:
 - Je kleiner die obere Schranke, desto besser
 - Je größer die untere Schranke, desto besser

Übersicht Komplexitätsklassen

Menge	Laufzeit	Beispiele
$O(1)$	Konstant	Abfrage eines Wertes an einer bestimmten Stelle in einem Feld
$O(\log n)$	Logarithmisch	Binäre Suche in einem Feld (Halbierungsprinzip)
$O(n)$	Linear	Suche eines Wertes in einem unsortierten Feld, Fibonacci iterativ
$O(n \log n)$	Super-Linear	Sortieren mit guten Algorithmen (z.B. Mergesort)
$O(n^k)$ Allgemein: Polynomiell	Quadratisch, Kubisch, ...	Sortieren mit einfachen Algorithmen (z.B. Bubblesort)
$O(2^n)$	Exponentiell	Rekursive Variante zur Berechnung der Fibonacci-Folge, SAT
$O(n!)$	Faktoriell	Problem des Handlungsreisenden (Traveling Salesman Problem, TSP)

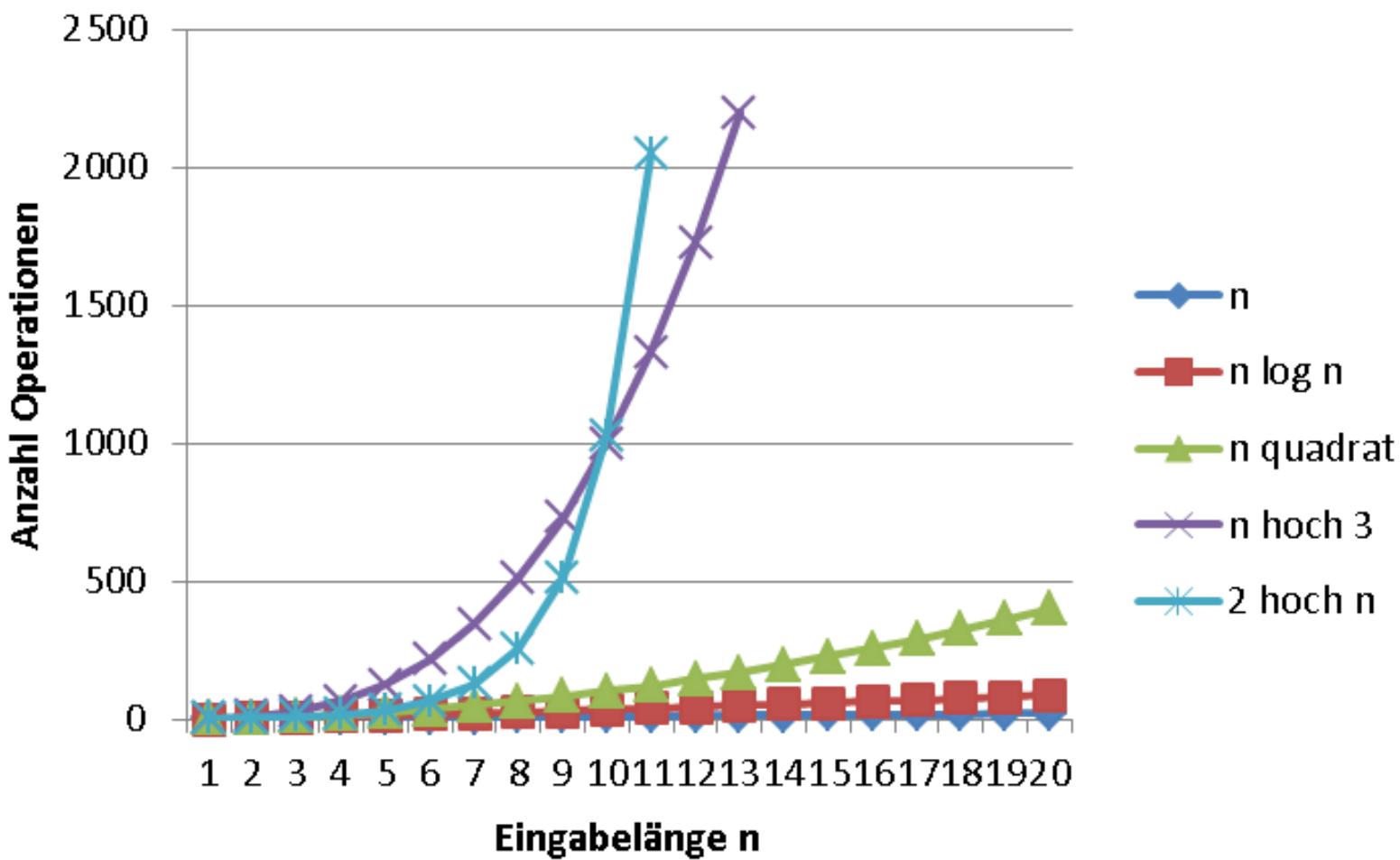
Algorithmen und Datenstrukturen

- Grundlagen (Komplexität) -

Prof. Dr. Klaus Volbert

Wintersemester 2018/19
Regensburg, 11. Oktober 2018

Wesentliche Wachstumsfunktionen



Beispiel MaxTeilSum

- Eingabe: $a_0, \dots, a_{n-1} \in \mathbb{Z}$ (n ganze Zahlen)
- Ausgabe: Maximale Teilsumme, d.h.

$$s = \max_{0 \leq i \leq j \leq n-1} \sum_{k=i}^j a_k$$

- Anwendungen
 - Erkennung von grafischen Mustern
 - Analyse von Aktienkursen
(täglich neue Kurse: Ermittlung bester Ein-/Ausstiegszeitpunkt)
- Beispiel:
 - Eingabe: -13, 25, 34, 12, -3, 7, -87, 28, -77, 11
 - Ausgabe: 75 (ergibt sich aus $i = 1, j = 5$)

- Naiver Algorithmus: Durchlaufen aller Varianten

```
int MaxTeilsum1(int a[], int n) {
    int i, j, k, sum, max = int.MinValue;

    for (i=0; i<n; i++) {
        for (j=i; j<n; j++) {
            sum=0;
            for (k=i; k<=j; k++) sum += a[k];
            if (sum > max) max = sum;
        }
    }

    return max;
}
```

- Laufzeit: $T_1^{wc}(n) = O(n^3)$ (**kubisch**)

Laufzeitanalyse MaxTeilSum1

- Zu verarbeitende Eingabedaten: n ganze Zahlen
- Laufzeit in Abhängigkeit der Eingabe sei $T_1^{wc}(n)$, dann gilt:

$$\begin{aligned} T_1^{wc}(n) &= 1 + \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \left(\sum_{k=i}^j 1 + 2 \right) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 3) = \sum_{i=0}^{n-1} \sum_{k=0}^{n-1-i} (k + 3) \\ &= \sum_{i=0}^{n-1} \sum_{k=0}^i k + \sum_{i=0}^{n-1} \sum_{k=0}^i 3 = \sum_{i=0}^{n-1} \frac{i(i+1)}{2} + \sum_{i=0}^{n-1} 3(i+1) \\ &= \frac{1}{2} \left(\sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i \right) + 3 \sum_{i=0}^{n-1} i + n \\ &= \frac{1}{2} \left(\frac{(n-1)n(2(n-1)+1)}{6} + \frac{(n-1)n}{2} \right) + \frac{3(n-1)n}{2} = \dots = \Theta(n^3) \end{aligned}$$

- Verbesserung: Statt immer wieder vollständige Summe zu berechnen: Erweiterung um den aktuellen Summanden

```
int MaxTeilsum2(int a[], int n) {  
    int i, j, sum, max = int.MinValue;  
  
    for (i=0; i<n; i++) {  
        sum=0;  
        for (j=i; j<n; j++) {  
            sum += a[j];  
            if (sum > max) max = sum;  
        }  
    }  
  
    return max; }
```

- Laufzeit: $T_2^{wc}(n) = O(n^2)$ (**quadratisch**)

- Verbesserung: Berechne lokales und globales Maximum. Das globale Maximum ist das Maximum über alle lokalen Maxima. Das lokale Maximum ist an einer Position entweder der Wert oder der Wert plus Werte „von links“.

```
int MaxTeilsum3(int a[], int n) {  
    int i, s, max = int.MinValue, aktSum = 0;  
  
    for (i=0; i<n; i++) {  
        s = aktSum + a[i];  
        if (s > a[i]) aktSum = s;  
        else aktSum = a[i];  
        if (aktSum > max) max = aktSum;  
    }  
  
    return max; }
```

- Laufzeit: $T_3^{wc}(n) = O(n)$ (**linear**)

Beispiel MaxTeilSum3

- Folge

	i	aktSum	max
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11		0	-2147483648
<u>-13</u> , 25, 34, 12, -3, 7, -87, 28, -77, 11	0	-13	-13
-13, <u>25</u> , 34, 12, -3, 7, -87, 28, -77, 11	1	25	25
-13, 25, <u>34</u> , 12, -3, 7, -87, 28, -77, 11	2	59	59
-13, 25, 34, <u>12</u> , -3, 7, -87, 28, -77, 11	3	71	71
-13, 25, 34, 12, <u>-3</u> , 7, -87, 28, -77, 11	4	68	71
-13, 25, 34, 12, -3, <u>7</u> , -87, 28, -77, 11	5	75	75
-13, 25, 34, 12, -3, 7, <u>-87</u> , 28, -77, 11	6	-12	75
-13, 25, 34, 12, -3, 7, -87, <u>28</u> , -77, 11	7	28	75
-13, 25, 34, 12, -3, 7, -87, 28, <u>-77</u> , 11	8	-49	75
-13, 25, 34, 12, -3, 7, -87, 28, -77, <u>11</u>	9	11	75

MaxTeilSum3 (Korrektheit I)

- Behauptung:
 - Nach dem i .ten Schleifendurchlauf gilt ([Schleifeninvariante](#)):

$$\text{aktSum} = \max_{0 \leq l \leq i} \sum_{k=l}^i a_k, \quad \max = \max_{0 \leq l \leq m \leq i} \sum_{k=l}^m a_k$$

- IA: $i = 0$:
 - In der Schleife wird $s = \text{aktSum} + a[i] = 0 + a[0] = a[0]$
 - Da s nicht größer als $a[i] = a[0]$ wird $\text{aktSum} = a[i] = a[0]$
 - 1. Fall: $\text{aktSum} > \max$, dann wird $\max = \text{aktSum}$
 - 2. Fall: aktSum ist kleinster int-Wert,
dann hatte \max den Wert schon bei Initialisierung
- IV: Behauptung gilt für $i-1$

MaxTeilSum3 (Korrektheit II)

- IS: $i - 1 \rightarrow i$:

- Nach IV gilt nach dem $(i - 1)$.ten Schleifendurchlauf (Schleifeninvariante):

$$\text{aktSum} = \max_{0 \leq l \leq i-1} \sum_{k=l}^{i-1} a_k \quad \text{und} \quad \max = \max_{0 \leq l \leq m \leq i-1} \sum_{k=l}^m a_k$$

- Es wird $s = \text{aktSum} + a[i]$
- 1. Fall: $s > a[i]$, dann $\text{aktSum} = s$
- 2. Fall: sonst, dann $\text{aktSum} = a[i]$
- Falls $\text{aktSum} > \max$, dann $\max = \text{aktSum}$, d.h.

$$\text{aktSum} = \max_{0 \leq l \leq i} \sum_{k=l}^i a_k$$

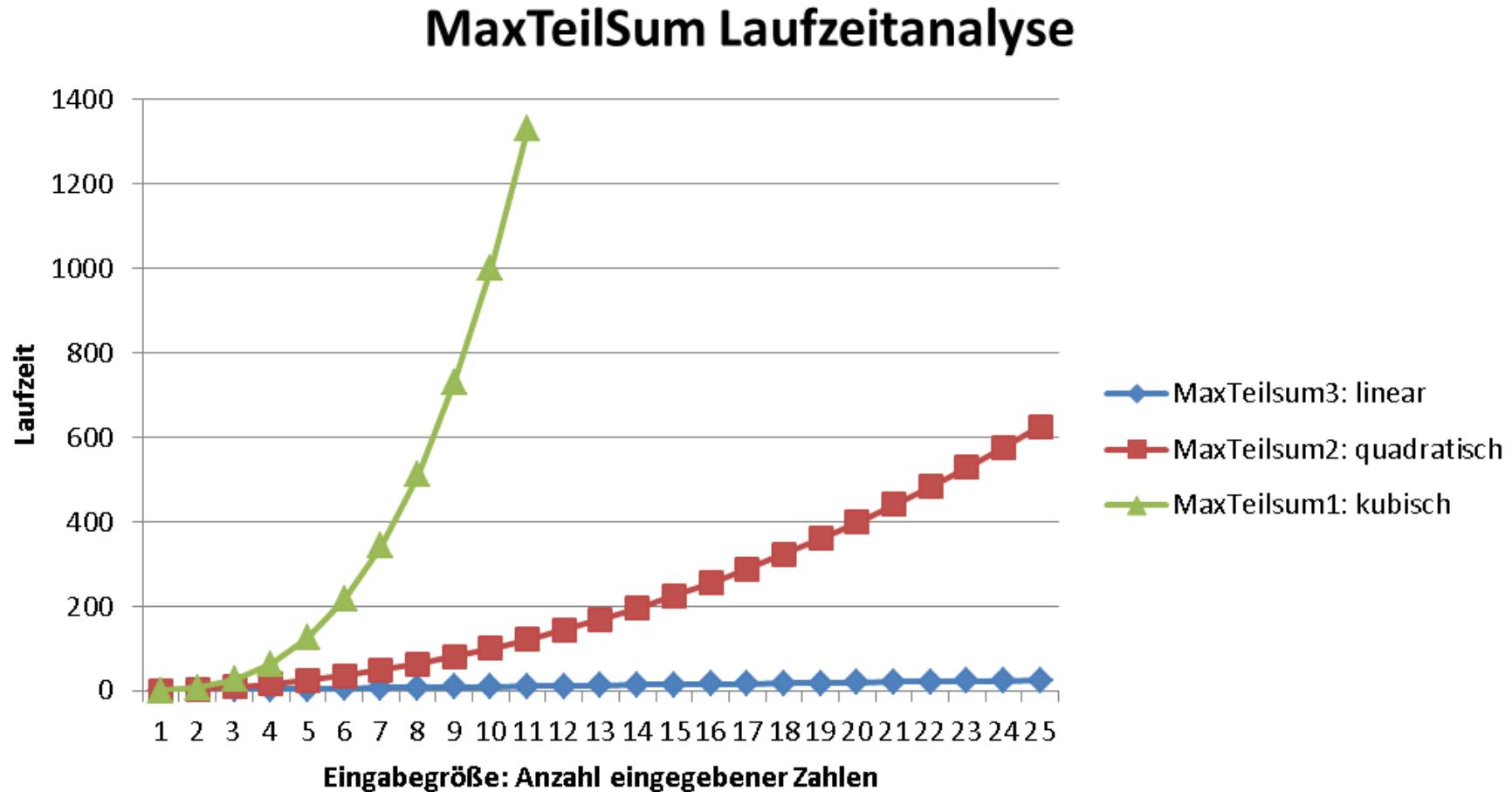
$$\max = \max \left\{ \max_{0 \leq l \leq m \leq i-1} \sum_{k=l}^m a_k, \text{aktSum} \right\} = \max_{0 \leq l \leq m \leq i} \sum_{k=l}^m a_k$$

- Nach dem letzten Schleifendurchlauf gilt: $i = n - 1$, d.h.

$$\max = \max_{0 \leq l \leq m \leq i} \sum_{k=l}^m a_k = \max_{0 \leq l \leq m \leq n-1} \sum_{k=l}^m a_k$$

Übersicht

Laufzeitverhalten MaxTeilSum



Komplexität von Algorithmen

- Ein Algorithmus A hat die Komplexität $O(f(n))$, wenn

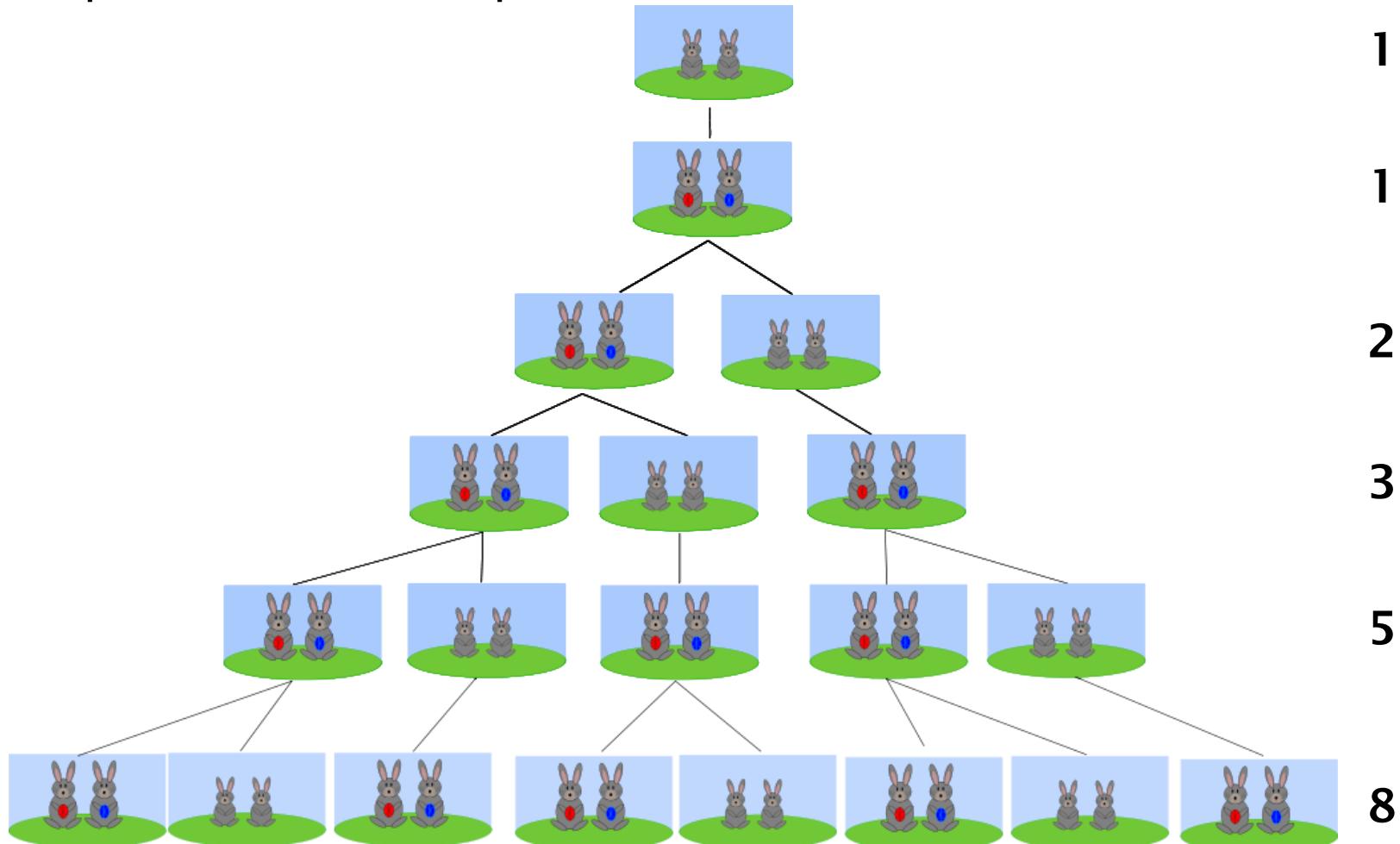
$$T_A^{wc}(n) = O(f(n))$$

- Komplexität von nun an: Worst Case Laufzeitkomplexität
- Statt $T_A^{wc}(n)$ wird auch gerne einfach nur $T(n)$ verwendet
- Vorgehen
 - Bestimme $T(n)$ für den Algorithmus möglichst exakt
(Bestimmung genauer Konstanten ist oft schwierig/aufwendig)
 - Schätze $T(n)$ mit Hilfe der O-Notation ab
- Allgemein gilt:
 - Je kleiner die obere Schranke, desto besser
 - Je größer die untere Schranke, desto besser

Übersicht Komplexitätsklassen

Menge	Laufzeit	Beispiele
$O(1)$	Konstant	Abfrage eines Wertes an einer bestimmten Stelle in einem Feld
$O(\log n)$	Logarithmisch	Binäre Suche in einem Feld (Halbierungsprinzip)
$O(n)$	Linear	Suche eines Wertes in einem unsortierten Feld, Fibonacci iterativ
$O(n \log n)$	Super-Linear	Sortieren mit guten Algorithmen (z.B. Mergesort)
$O(n^k)$ Allgemein: Polynomiell	Quadratisch, Kubisch, ...	Sortieren mit einfachen Algorithmen (z.B. Bubblesort)
$O(2^n)$	Exponentiell	Rekursive Variante zur Berechnung der Fibonacci-Folge, SAT
$O(n!)$	Faktoriell	Problem des Handlungsreisenden (Traveling Salesperson Problem, TSP)

- Beispiel: Kaninchen-Population



Iterative vs. Rekursive Algorithmen I

- Beispiel: Fibonacci-Zahlen (Kaninchen-Population)

$$- \text{fib}(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ \text{fib}(n - 1) + \text{fib}(n - 2), & \text{sonst} \end{cases} \quad (\text{fib}(n) = 1, 1, 2, 3, 5, 8, 13, \dots)$$

- Iterative Implementierung

```
int fib(int n) {
    int i=2, result=1, f1=1, f2=1;
    while (i<n)
    {
        i=i+1;
        result=f1+f2;
        f1 = f2;
        f2 = result;
    }
    return result; }
```

Laufzeit: $T(n) = \Theta(n)$

Speicher: $S(n) = \Theta(1)$

Iterative vs. Rekursive Algorithmen II

- Rekursive Implementierung

```
int fib_rek(int n) {  
    if (n<3) return 1;  
    else return fib_rek(n-1) + fib_rek(n-2);  
}
```

- Laufzeitanalyse (**Substitutionsmethode**)

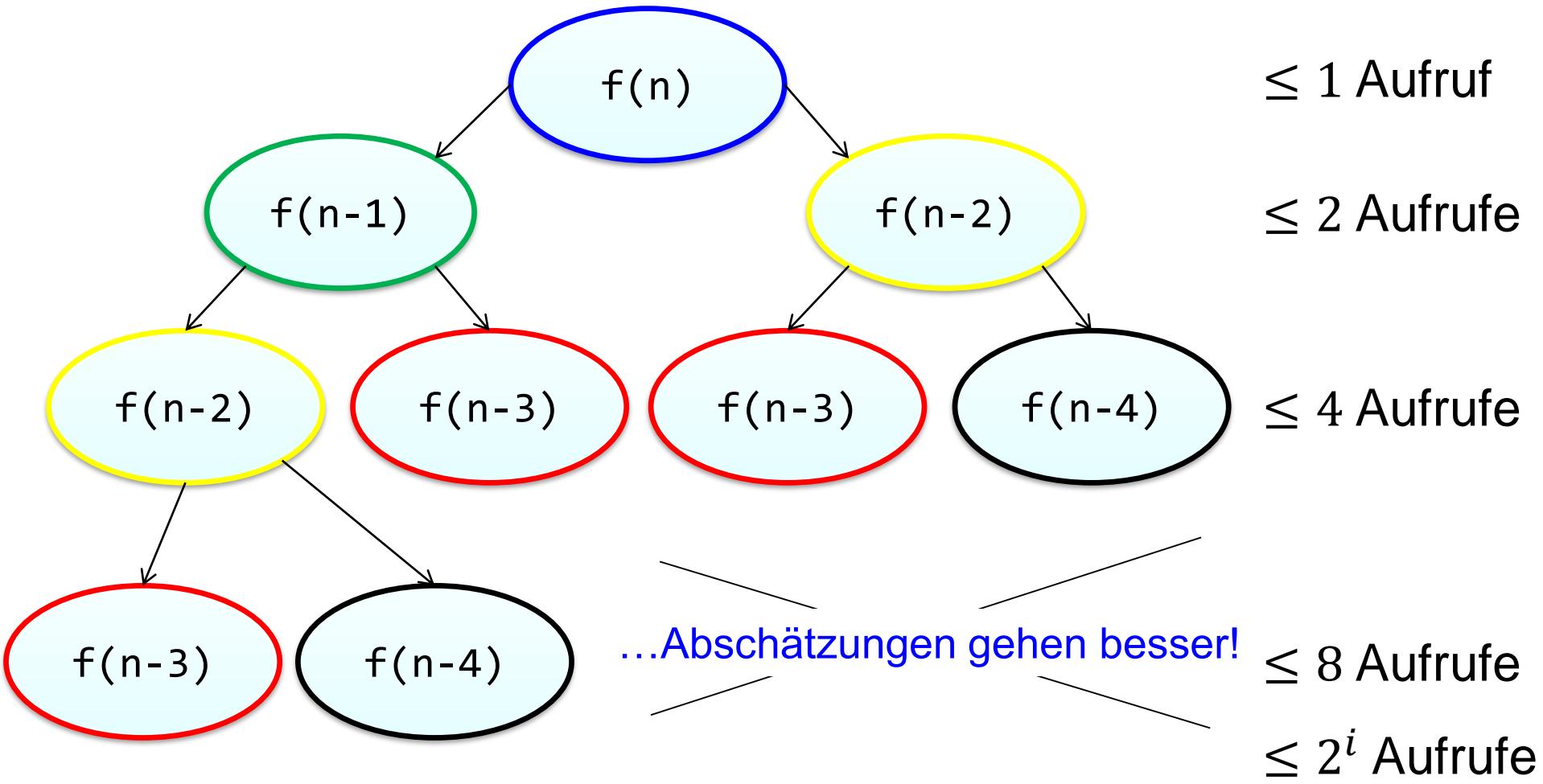
$$T(1) = 1, T(2) = 1, T(n) = T(n - 1) + T(n - 2)$$

- Behauptung: $T(n) \leq 2^n = O(2^n)$

- IA: $T(1) = 1 \leq 2^1 = O(2^n), T(2) = 1 \leq 2^2 = O(2^n)$
- IV: Die Behauptung gilt für $n-1$ und $n-2$
- IS: $T(n) = T(n - 1) + T(n - 2) \stackrel{IV}{\leq} 2^{n-1} + 2^{n-2} < 2 \cdot 2^{n-1} = O(2^n)$

- Wie kommt man auf die Behauptung?

Rekursionsbaum



Maximal 2^n Aufrufe mit konstantem Aufwand, Speicher: $O(2^n)$

- Komplexität rekursive Implementierung Fibonacci
 - Man kann zeigen (Übungsaufgabe), dass gilt:

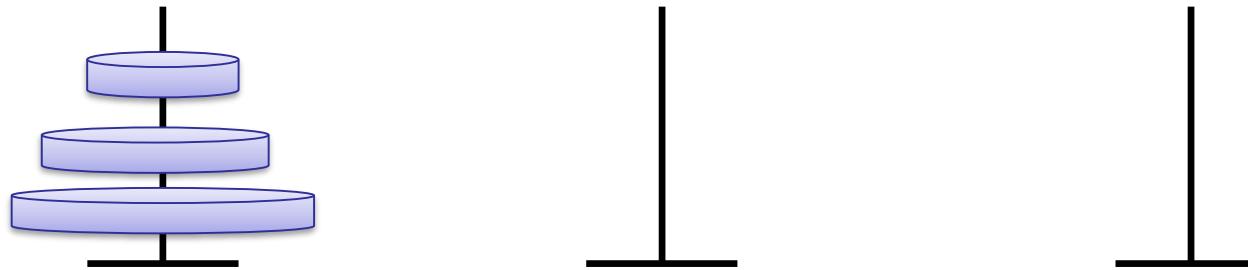
$$T(n) = \Theta\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$$



- Rekursion ist elegant, sollte aber vermieden werden
- Iterative Varianten sind meist effizienter, aber komplizierter
- Funktionale Sprachen (z.B. Haskell, ML, SML) versuchen Probleme der Rekursion in imperativen Sprachen zu lösen
- Jede rekursive Implementierung kann in eine iterative Implementierung überführt werden
- Aus didaktischen Gründen wird weiter Rekursion verwendet, auch wenn sie nicht unbedingt sinnvoll ist

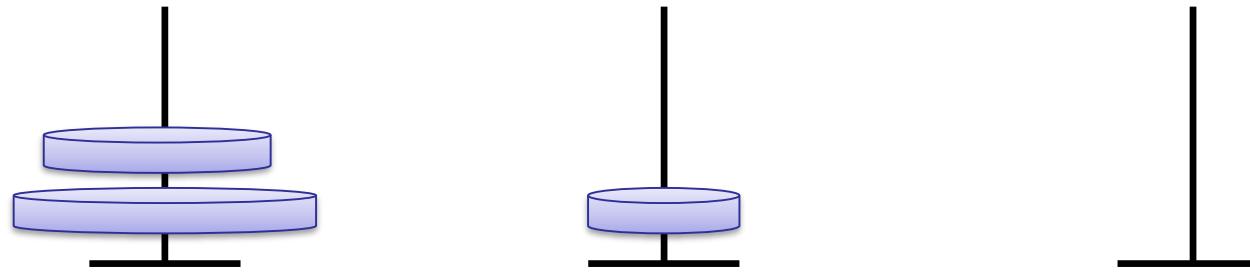
Beispiel: Türme von Hanoi

- Eingabe:
 - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
 - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
 - Es darf immer nur der oberste Ring von einem Stab versetzt werden
 - Ein Ring darf nicht auf einen kleineren Ring abgelegt werden



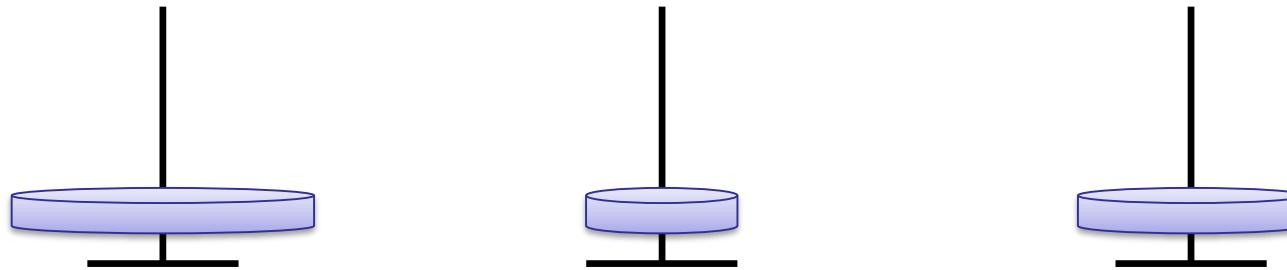
Beispiel: Türme von Hanoi

- Eingabe:
 - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
 - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
 - Es darf immer nur der oberste Ring von einem Stab versetzt werden
 - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



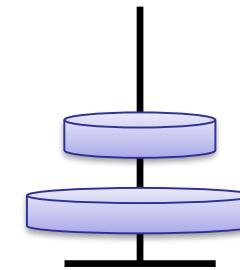
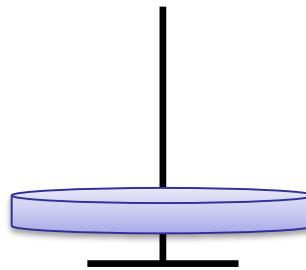
Beispiel: Türme von Hanoi

- Eingabe:
 - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
 - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
 - Es darf immer nur der oberste Ring von einem Stab versetzt werden
 - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



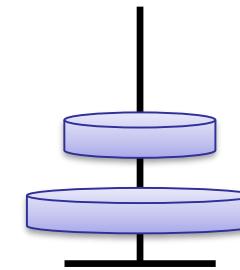
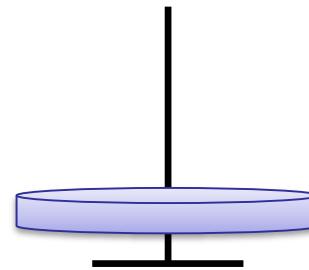
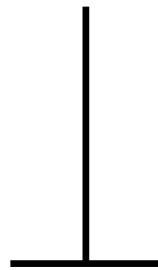
Beispiel: Türme von Hanoi

- Eingabe:
 - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
 - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
 - Es darf immer nur der oberste Ring von einem Stab versetzt werden
 - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



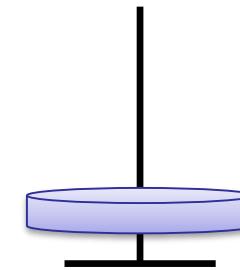
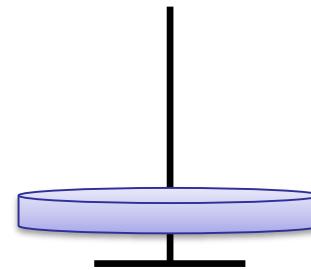
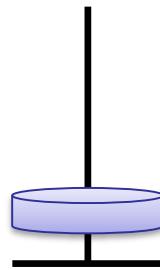
Beispiel: Türme von Hanoi

- Eingabe:
 - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
 - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
 - Es darf immer nur der oberste Ring von einem Stab versetzt werden
 - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



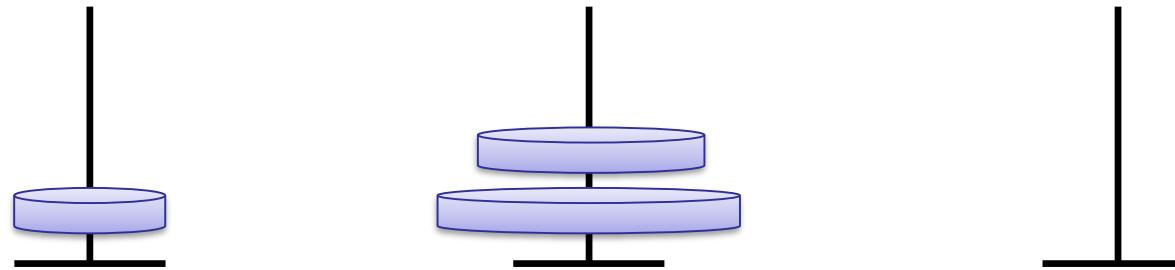
Beispiel: Türme von Hanoi

- Eingabe:
 - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
 - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
 - Es darf immer nur der oberste Ring von einem Stab versetzt werden
 - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



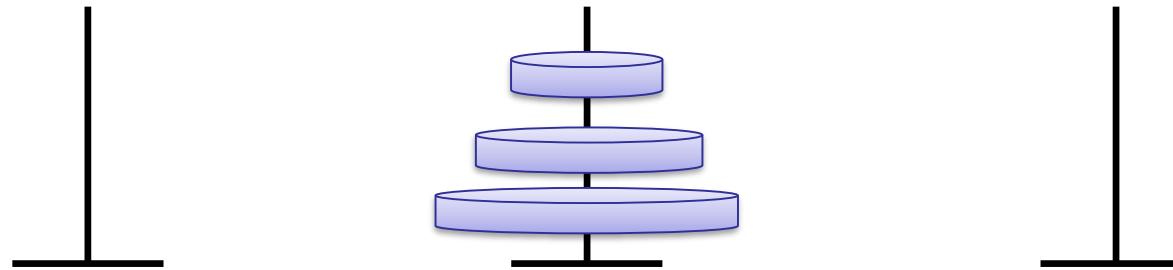
Beispiel: Türme von Hanoi

- Eingabe:
 - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
 - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
 - Es darf immer nur der oberste Ring von einem Stab versetzt werden
 - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



Beispiel: Türme von Hanoi

- Eingabe:
 - Drei Stäbe; 1 Stab enthält n Ringe abgestuften Durchmessers; ein kleinerer Ring darf nur auf einem größerem Ring liegen
- Ausgabe:
 - Versetzen der n Ringe von dem einen Stab auf einen anderen
- Regeln:
 - Es darf immer nur der oberste Ring von einem Stab versetzt werden
 - Ein Ring darf nicht auf einem kleineren Ring abgelegt werden



Türme von Hanoi (Rekursiver Algorithmus)

```
void hanoi(int n, char stab1[], char stab2[], char stab3[])
{
    if (n==1)
        // Bewege den obersten Ring von Stab 1 auf Stab 2
    else
    {
        hanoi(n-1,stab1,stab3,stab2);
        // Bewege den obersten Ring von Stab 1 auf Stab 2
        hanoi(n-1,stab3,stab2,stab1);
    }
}
```

- Korrektheit (Beh.: n Ringe werden korrekt von Stab 1 auf Stab 2 bewegt)
 - IA: $n = 1$: Der oberste Ring wird von Stab 1 auf Stab 2 bewegt
 - IV: Die Behauptung gilt für $n - 1$
 - IS: $n - 1 \rightarrow n$: Nach IV werden $n - 1$ Ringe von Stab 1 auf Stab 3 bewegt (Stab 2 dient als Hilfe). Der n .te Ring wird von Stab 1 auf Stab 2 bewegt. Zuletzt werden nach IV $n - 1$ Ringe von Stab 3 auf Stab 2 bewegt (Stab 1 dient als Hilfe). D.h. n Ringe wurden von Stab 1 auf Stab 2 bewegt

- Beobachtung **Rekursionsgleichungen**
 - $T(1) = 1, T(n) = 2T(n - 1) + 1$
- Iterationsmethode
 - $$\begin{aligned} T(n) &= 2T(n - 1) + 1 \\ &= 2(2T(n - 2) + 1) + 1 = 2^2T(n - 2) + (1 + 2) \\ &= 2(2(2T(n - 3) + 1) + 1) + 1 = 2^3T(n - 3) + (1 + 2 + 4) \\ &\dots \\ &= 2^i T(n - i) + \sum_{k=0}^{i-1} 2^k \end{aligned}$$
 - Rekursionsbasis wird erreicht, wenn: $n - i = 1$, d.h. setze $i = n - 1$
 - $$\begin{aligned} T(n) &= 2^{n-1}T(n - (n - 1)) + \sum_{k=0}^{(n-1)-1} 2^k \\ &= 2^{n-1} + \frac{2^{(n-2)+1}-1}{2-1} = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1 = \Theta(2^n) \end{aligned}$$
- Beweis durch vollständige Induktion (Übung)

Beispiel MaxTeilSum

- Eingabe: $a_0, \dots, a_{n-1} \in \mathbb{Z}$ (n ganze Zahlen)
- Ausgabe: Maximale Teilsumme, d.h.

$$s = \max_{0 \leq i \leq j \leq n-1} \sum_{k=i}^j a_k$$

- Anwendungen
 - Erkennung von grafischen Mustern
 - Analyse von Aktienkursen
(täglich neue Kurse: Ermittlung bester Ein-/Ausstiegszeitpunkt)
- Beispiel:
 - Eingabe: -13, 25, 34, 12, -3, 7, -87, 28, -77, 11
 - Ausgabe: 75 (ergibt sich aus $i = 1, j = 5$)

MaxTeilSum4 (Divide-&-Conquer)

```
int MaxTeilsum4(int a[], int f, int l) { int n = l - f + 1;  
    if (n == 1) return a[f];  
    else {
```

Triviallösung

Divide: Teile a in zwei Teile

```
        int newn = (n % 2 == 0 ? n / 2 : n / 2 + 1);
```



```
        int MaxBorderSum1=a[f+newn-1], i=f+newn-2, currVal=MaxBorderSum1;  
        while (i>=f) { currVal+=a[i];  
            if (currVal>MaxBorderSum1) MaxBorderSum1=currVal;  
            i--; }
```

```
        int MaxBorderSum2=a[f+newn], i=f+newn+1, currVal=MaxBorderSum2;  
        while (i<=l) { currVal+=a[i];  
            if (currVal>MaxBorderSum2) MaxBorderSum2=currVal;  
            i++; }
```

Conquer: Berechne die Teillösungen

```
    return max (MaxTeilsum4(a,f,f+newn-1),  
               max (MaxTeilsum4(a,f+newn,l), MaxBorderSum1+  
                     MaxBorderSum2)); } }
```

Merge: Füge die Einzelergebnisse zusammen

- Laufzeit: $T(n) = \Theta(n \log n)$

Beispiel MaxTeilSum4 I

• Folge

	MBS1	MBS2	Σ	MT1	MT2	Max
-13 25 34 12 -3 7 -87 28 -77 11	68	7	75	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11	59	12	71	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11	25	34	59	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11	-13	25	12	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11						-13
-13 25 34 12 -3 7 -87 28 -77 11						25
-13 25 34 12 -3 7 -87 28 -77 11	-13	25	12	-13	25	25
-13 25 34 12 -3 7 -87 28 -77 11						34
-13 25 34 12 -3 7 -87 28 -77 11	25	34	59	25	34	59
-13 25 34 12 -3 7 -87 28 -77 11	12	-3	9	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11						12

Beispiel MaxTeilSum4 II

• Folge	MBS1	MBS2	Σ	MT1	MT2	Max
-13 25 34 12 <u>-3</u> 7 -87 28 -77 11						-3
-13 25 34 <u>12</u> <u>-3</u> 7 -87 28 -77 11 12	-3	9	12	-3	12	
<u>-13</u> 25 34 <u>12</u> <u>-3</u> 7 -87 28 -77 11 59	12	71	59	12	71	
-13 25 34 12 -3 <u>7</u> -87 <u>28</u> <u>-77</u> 11 28	-66	-38	?	?	?	
-13 25 34 12 -3 <u>7</u> -87 <u>28</u> -77 11 -80	28	-52	?	?	?	
-13 25 34 12 -3 <u>7</u> <u>-87</u> 28 -77 11 7	-87	-80	?	?	?	
-13 25 34 12 -3 <u>7</u> -87 28 -77 11						7
-13 25 34 12 -3 7 <u>-87</u> 28 -77 11						-87
-13 25 34 12 -3 <u>7</u> <u>-87</u> 28 -77 11 7	-87	-80	7	-87	7	
-13 25 34 12 -3 7 -87 <u>28</u> -77 11						28
-13 25 34 12 -3 7 -87 <u>28</u> -77 11 -80	28	-52	7	28	28	

Beispiel MaxTeilSum4 III

- Folge

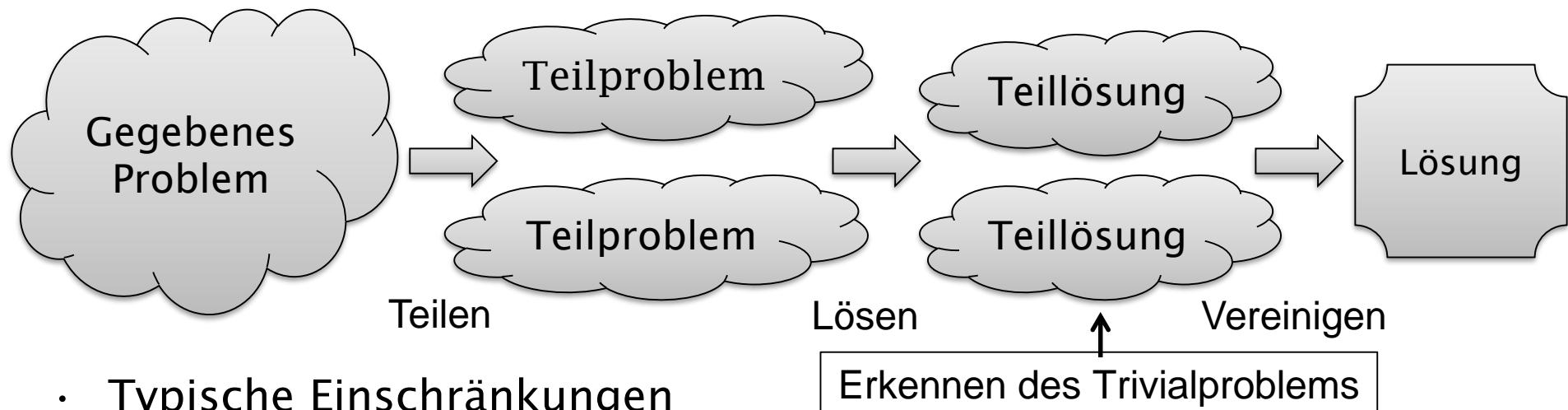
								MBS1	MBS2	Σ	MT1	MT2	Max
-13	25	34	12	-3	7	-87	28	<u>-77</u> 11	-77	11	-66	?	?
-13	25	34	12	-3	7	-87	28	<u>-77</u> 11					-77
-13	25	34	12	-3	7	-87	28	-77 <u>11</u>					11
-13	25	34	12	-3	7	-87	28	<u>-77</u> 11	-77	11	-66	-77	11
-13	25	34	12	-3	<u>7</u> -87	28	<u>-77</u> 11	28	-66	-38	28	11	28
-13	25	34	12	-3	<u>7</u>	-87	28	<u>-77</u> 11	68	7	75	71	28
													75

- ...und jetzt mit:

- Eingabe: -5, 13, -32, 7, -3, 17, 23, 12, -35, 19
- Ausgabe: ?

Entwurfsprinzip: Divide & Conquer

- Schema: Teilen und Herrschen



- Typische Einschränkungen
 - Teilprobleme müssen unabhängig voneinander lösbar sein
 - Gesamtlösung muss aus Teillösungen entstehen können (Vereinigung)
- Teilung bis zum Erreichen des Trivialproblems (oft rekursiv)
- Beispiele
 - Quicksort, Schnelle Fouriertransformation (FFT)

- Korrektheitsbeweise häufig mit vollständiger Induktion
 - Identifikation einer Bedingung, die nach allen Schleifendurchläufen gilt (Schleifeninvariante, Analyse: vor, während, nach)
 - Bedingung für das Verlassen der Schleife zusammen mit der Schleifeninvariante liefern das gewünschte Ergebnis
- Komplexitätsabschätzung durch Aufstellen von
 - Komplexitätsgleichungen ($T(n) = \dots$)
 - Rekursionsgleichungen mit Rekursionsbasis ($T(n) = \dots, T(a) = b$)
- Lösen von Rekursionsgleichungen durch
 - Substitutionsmethode (Lösung raten, Korrektheit beweisen)
 - Iterationsmethode (sukzessives Einsetzen liefert Abschätzung)
 - Master-Methode (folgt)

Algorithmen und Datenstrukturen

- Grundlagen (Komplexität) -

Prof. Dr. Klaus Volbert

Wintersemester 2018/19
Regensburg, 15. Oktober 2018

Beispiel MaxTeilSum

- Eingabe: $a_0, \dots, a_{n-1} \in \mathbb{Z}$ (n ganze Zahlen)
- Ausgabe: Maximale Teilsumme, d.h.

$$s = \max_{0 \leq i \leq j \leq n-1} \sum_{k=i}^j a_k$$

- Anwendungen
 - Erkennung von grafischen Mustern
 - Analyse von Aktienkursen
(täglich neue Kurse: Ermittlung bester Ein-/Ausstiegszeitpunkt)
- Beispiel:
 - Eingabe: -13, 25, 34, 12, -3, 7, -87, 28, -77, 11
 - Ausgabe: 75 (ergibt sich aus $i = 1, j = 5$)

MaxTeilSum4 (Divide-&-Conquer)

```
int MaxTeilsum4(int a[], int f, int l) { int n = l - f + 1;  
    if (n == 1) return a[f];  
    else {
```

Triviallösung

Divide: Teile a in zwei Teile

```
        int newn = (n % 2 == 0 ? n / 2 : n / 2 + 1);
```



```
        int MaxBorderSum1=a[f+newn-1], i=f+newn-2, currVal=MaxBorderSum1;  
        while (i>=f) { currVal+=a[i];  
            if (currVal>MaxBorderSum1) MaxBorderSum1=currVal;  
            i--; }
```

```
        int MaxBorderSum2=a[f+newn], i=f+newn+1, currVal=MaxBorderSum2;  
        while (i<=l) { currVal+=a[i];  
            if (currVal>MaxBorderSum2) MaxBorderSum2=currVal;  
            i++; }
```

Conquer: Berechne die Teillösungen

```
    return max (MaxTeilsum4(a,f,f+newn-1),  
               max (MaxTeilsum4(a,f+newn,l), MaxBorderSum1+  
                     MaxBorderSum2)); } }
```

Merge: Füge die Einzelergebnisse zusammen

- Laufzeit: $T(n) = \Theta(n \log n)$

Beispiel MaxTeilSum4 I

• Folge	MBS1	MBS2	Σ	MT1	MT2	Max
-13 25 34 12 -3 7 -87 28 -77 11	68	7	75	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11	59	12	71	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11	25	34	59	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11	-13	25	12	?	?	?
-13 25 34 12 -3 7 -87 28 -77 11						-13
-13 <u>25</u> 34 12 -3 7 -87 28 -77 11						25
-13 25 34 12 -3 7 -87 28 -77 11	-13	25	12	-13	25	25
-13 25 <u>34</u> 12 -3 7 -87 28 -77 11						34
-13 25 34 12 -3 7 -87 28 -77 11	25	34	59	25	34	59
-13 25 34 <u>12 -3</u> 7 -87 28 -77 11	12	-3	9	?	?	?
-13 25 34 <u>12</u> -3 7 -87 28 -77 11						12

Beispiel MaxTeilSum4 II

• Folge	MBS1	MBS2	Σ	MT1	MT2	Max
-13 25 34 12 <u>-3</u> 7 -87 28 -77 11						-3
-13 25 34 <u>12</u> <u>-3</u> 7 -87 28 -77 11 12	-3	9	12	-3	12	
<u>-13</u> 25 34 <u>12</u> <u>-3</u> 7 -87 28 -77 11 59	12	71	59	12	71	
-13 25 34 12 -3 <u>7</u> -87 <u>28</u> <u>-77</u> 11 28	-66	-38	?	?	?	
-13 25 34 12 -3 <u>7</u> -87 <u>28</u> -77 11 -80	28	-52	?	?	?	
-13 25 34 12 -3 <u>7</u> <u>-87</u> 28 -77 11 7	-87	-80	?	?	?	
-13 25 34 12 -3 <u>7</u> -87 28 -77 11						7
-13 25 34 12 -3 7 <u>-87</u> 28 -77 11						-87
-13 25 34 12 -3 <u>7</u> <u>-87</u> 28 -77 11 7	-87	-80	7	-87	7	
-13 25 34 12 -3 7 -87 <u>28</u> -77 11						28
-13 25 34 12 -3 7 -87 <u>28</u> -77 11 -80	28	-52	7	28	28	

Beispiel MaxTeilSum4 III

- Folge

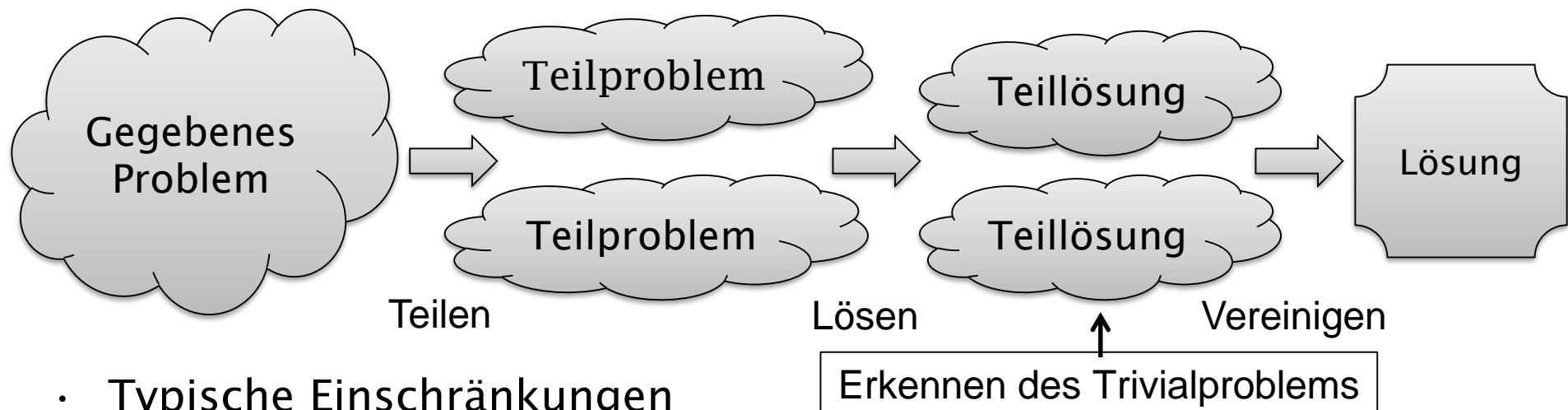
								MBS1	MBS2	Σ	MT1	MT2	Max
-13	25	34	12	-3	7	-87	28	<u>-77</u> 11	-77	11	-66	?	?
-13	25	34	12	-3	7	-87	28	<u>-77</u> 11					-77
-13	25	34	12	-3	7	-87	28	-77 <u>11</u>					11
-13	25	34	12	-3	7	-87	28	<u>-77</u> 11	-77	11	-66	-77	11
-13	25	34	12	-3	<u>7</u> -87	28	<u>-77</u> 11	28	-66	-38	28	11	28
-13	25	34	12	-3	<u>7</u>	-87	28	<u>-77</u> 11	68	7	75	71	28
													75

- ...und jetzt mit:

- Eingabe: -5, 13, -32, 7, -3, 17, 23, 12, -35, 19
- Ausgabe: ?

Entwurfsprinzip: Divide & Conquer

- Schema: Teilen und Herrschen



- Typische Einschränkungen
 - Teilprobleme müssen unabhängig voneinander lösbar sein
 - Gesamtlösung muss aus Teillösungen entstehen können (Vereinigung)
- Teilung bis zum Erreichen des Trivialproblems (oft rekursiv)
- Beispiele
 - Quicksort, Schnelle Fouriertransformation (FFT)

- Korrektheitsbeweise häufig mit vollständiger Induktion
 - Identifikation einer Bedingung, die nach allen Schleifendurchläufen gilt (Schleifeninvariante, Analyse: vor, während, nach)
 - Bedingung für das Verlassen der Schleife zusammen mit der Schleifeninvariante liefern das gewünschte Ergebnis
- Komplexitätsabschätzung durch Aufstellen von
 - Komplexitätsgleichungen ($T(n) = \dots$)
 - Rekursionsgleichungen mit Rekursionsbasis ($T(n) = \dots, T(a) = b$)
- Lösen von Rekursionsgleichungen durch
 - Substitutionsmethode (Lösung raten, Korrektheit beweisen)
 - Iterationsmethode (sukzessives Einsetzen liefert Abschätzung)
 - Master-Methode (jetzt)

- Rekursionsgleichungen haben oft die Form
 - $T(1) = 1, T(n) = aT\left(\frac{n}{b}\right) + f(n)$ mit $a \geq 1, b > 1, f: \mathbb{N}_0 \rightarrow \mathbb{N}_0$
- Das **Master-Theorem** gibt an, wie man solche Gleichungen lösen kann:
 - 1. Fall: Falls $f(n) = O(n^{\log_b a - \varepsilon})$ für ein $\varepsilon > 0$, dann: $T(n) = \Theta(n^{\log_b a})$
Anmerkung: $f(n)$ wächst schwächer als $aT\left(\frac{n}{b}\right)$
 - 2. Fall: Falls $f(n) = \Theta(n^{\log_b a})$, dann: $T(n) = \Theta(n^{\log_b a} \log n)$
Anmerkung: $f(n)$ und $aT\left(\frac{n}{b}\right)$ wachsen gleich, dazu kommt: log-Faktor
 - 3. Fall: Falls $f(n) = \Omega(n^{\log_b a + \varepsilon})$ für ein $\varepsilon > 0$ und falls $af\left(\frac{n}{b}\right) \leq cf(n)$ für ein $c < 1$ und alle $n \geq n_0$, dann: $T(n) = \Theta(f(n))$
Anmerkung: f wächst stärker als $aT\left(\frac{n}{b}\right)$
- Bemerkungen:
 - Aussagen gelten auch für $\dots T\left(\left\lceil \frac{n}{b} \right\rceil\right) \dots$ und $\dots T\left(\left\lfloor \frac{n}{b} \right\rfloor\right) \dots$
 - Beweise können in Cormen, Leiserson, Rivest, Stein: *Introduction to Algorithms*, 3rd Ed., MIT Press, 2009 nachgelesen werden (Kapitel 4)

Beispiele zur Master-Methode I

- $T(n) = 2T\left(\frac{n}{2}\right) + n$ (Rekursionsgleichung MaxTeilSum4)
 - $a = 2, b = 2, f(n) = n$, es gilt $\log_b a = \log_2 2 = 1$
 - D.h. $f(n) = \Theta(n^{\log_b a}) = \Theta(n)$ und nach Fall 2:

$$T(n) = \Theta(n \log n) \quad \text{👍}$$

- $T(n) = 9T\left(\frac{n}{3}\right) + n$
 - $a = 9, b = 3, f(n) = n$, es gilt $\log_b a = \log_3 9 = 2$
 - D.h. $f(n) = O(n^{\log_b a - \varepsilon})$ für ein $\varepsilon > 0$ und nach Fall 1:

$$T(n) = \Theta(n^2) \quad \text{👍}$$

Beispiele zur Master-Methode II

- $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$
 - $a = 2, b = 2, f(n) = n \log n$, es gilt $\log_b a = \log_2 2 = 1$
 - Zusätzlich gilt: $f(n) = \Omega(n^{\log_b a + \varepsilon})$ für ein $\varepsilon > 0$, da $n \log n \geq cn^{1+\varepsilon}$
 - Was zunächst für Fall 3 spricht, aber gilt $af\left(\frac{n}{b}\right) \leq cf(n)$ für ein $c < 1$?
 - Aus $2\frac{n}{2} \log \frac{n}{2} \leq cn \log n$ folgt $1 - \frac{1}{\log n} \leq c$. D.h. aber $c < 1$ existiert nicht
 - Daher ist Fall 3 nicht anwendbar (alle anderen Fälle auch nicht) 
- Beobachtung:
 - Zwischen Fall 2 und Fall 3 existiert eine Lücke
 - Man kann erweitern:
 Wenn $f(n) = \Theta(n^{\log_b a} (\log n)^k)$ für $k \geq 0$, dann $T(n) = \Theta(n^{\log_b a} (\log n)^{k+1})$

Beispiele zur Master-Methode III

- Lösen Sie folgende Rekursionsgleichungen mit der Master-Methode:

$$1. \quad T(n) = 8T\left(\frac{n}{3}\right) + n^2$$

$$2. \quad T(n) = 9T\left(\frac{n}{3}\right) + n^2$$

$$3. \quad T(n) = 10T\left(\frac{n}{3}\right) + n^2$$

Beispiele zur Master-Methode III

- Lösen Sie folgende Rekursionsgleichungen mit der Master-Methode:

1. $T(n) = 8T\left(\frac{n}{3}\right) + n^2$

- $a = 8, b = 3, f(n) = n^2 = \Omega(n^{\log_3 8 + \varepsilon})$ für $\varepsilon > 0$ (also evtl. Fall 3)
- Prüfe: $8 \cdot f\left(\frac{n}{3}\right) = 8 \cdot \frac{n^2}{9} \leq \frac{8}{9} \cdot n^2 = c \cdot f(n)$ mit $c = \frac{8}{9} < 1$ für alle $n \geq n_0 = 1$
- Also nach Fall 3: $T(n) = \Theta(f(n)) = \Theta(n^2)$

2. $T(n) = 9T\left(\frac{n}{3}\right) + n^2$

- $a = 9, b = 3, f(n) = n^2 = \Theta(n^{\log_3 9}) = \Theta(n^{\log_3 3^2}) = \Theta(n^2)$
- Also nach Fall 2: $T(n) = \Theta(n^{\log_3 3^2} \cdot \log(n)) = \Theta(n^2 \cdot \log(n))$

3. $T(n) = 10T\left(\frac{n}{3}\right) + n^2$

- $a = 10, b = 3, f(n) = n^2 = O(n^{\log_3 10 - \varepsilon})$ für $\varepsilon > 0$
- Also nach Fall 1: $T(n) = \Theta(n^{\log_3 10}) = \Theta(n^{2,0959})$

- Einführung und Organisation
- Grundlagen
 - Begriffe
 - Algorithmus, Datentyp, Datenstruktur, Datenstruktur Stapel
 - Korrektheit und Komplexität
 - Totale Korrektheit, Iterationen, Rekursionen
 - RAM, Church'sche These, O-Notation (O , Ω , Θ)
 - Rekursionsgleichungen
 - Iterationsmethode, Substitutionsmethode, Master-Methode
 - Entwurfsmethode Divide & Conquer
- Sortieralgorithmen

Algorithmen und Datenstrukturen

- Sortieralgorithmen -

Prof. Dr. Klaus Volbert

Wintersemester 2018/19
Regensburg, 18. Oktober 2018

- Einführung und Organisation
- Grundlagen
 - Begriffe
 - Algorithmus, Datentyp, Datenstruktur, Datenstruktur Stapel
 - Korrektheit und Komplexität
 - Totale Korrektheit, Iterationen, Rekursionen
 - RAM, Church'sche These, O-Notation (O , Ω , Θ)
 - Rekursionsgleichungen
 - Iterationsmethode, Substitutionsmethode, Master-Methode
 - Entwurfsmethode Divide & Conquer
- Sortieralgorithmen

- Standardproblem: Sortieren
 - Schnelles Finden von Informationen durch vorheriges Sortieren
 - Anfang der 70er: ca. 25 % der weltweit verbrauchten Rechenzeit wird fürs Sortieren verwendet (heute nicht mehr so)
 - Heutige Anwendungen: DNA-Analyse, Internet, Computergrafik, ...
- Sortierproblem
 - Eingabe: a_0, \dots, a_{n-1} (n Zahlen einer total geordneten Menge, $n \in \mathbb{N}$)
 - Ausgabe: $a_{\pi(0)}, \dots, a_{\pi(n-1)}$ mit $a_{\pi(0)} \leq \dots \leq a_{\pi(n-1)}$ und π ist Permutation, d.h. $\pi: \{0, \dots, n-1\} \rightarrow \{0, \dots, n-1\}$
- Beispiel
 - Eingabe: 31, 41, 59, 26, 41, 58
 - Ausgabe: 26, 31, 41, 41, 58, 59

i	0	1	2	3	4	5
$\pi(i)$	1	2	5	0	3	4

Überblick Sortieralgorithmen

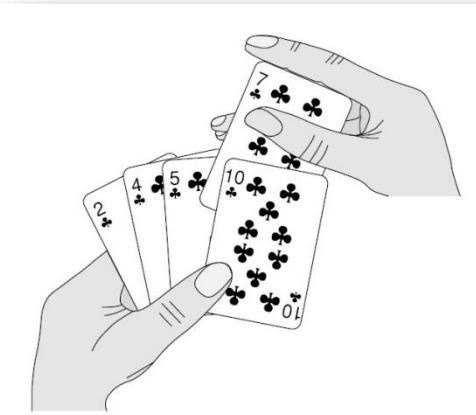
- Nichts ist besser verstanden als das Sortieren...
- Einfache Sortieralgorithmen
 - Sortieren durch Einfügen (Insertion Sort)
 - Sortieren durch Vertauschen (Bubble Sort)
 - Sortieren durch Auswählen (Selection Sort)
- Fortgeschrittene Sortieralgorithmen
 - Sortieren durch Divide & Conquer (Quicksort)
 - Sortieren durch Mischen (Merge Sort)
 - Sortieren durch Bäume (Heap Sort)
- Spezielle Sortieralgorithmen
 - Sortieren durch Zählen (Count Sort)
 - Sortieren durch Abbilden (Map Sort)
- Untere Schranke für vergleichsbasierte Sortieralgorithmen

Sortieren durch Einfügen (Insertion Sort)

```
void InsertionSort(int a[], int n)
{
    int i, j, key;

    for (j = 1; j < n; j++)
    {
        key = a[j];
        i = j-1;
        while (i >= 0 && a[i] > key)
        {
            a[i+1] = a[i];
            i = i-1;
        }

        a[i+1] = key;
    }
}
```



Beispiel Insertion Sort

34	45	12	34	23	18	38	17	43	51
<u>34</u>	<u>45</u>	12	34	23	18	38	17	43	51
<u>12</u>	<u>34</u>	<u>45</u>	34	23	18	38	17	43	51
<u>12</u>	<u>34</u>	<u>34</u>	<u>45</u>	23	18	38	17	43	51
<u>12</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>45</u>	18	38	17	43	51
<u>12</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>45</u>	38	17	43	51
<u>12</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	<u>45</u>	17	43	51
<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	<u>45</u>	43	51
<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	<u>43</u>	<u>45</u>	51
<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	<u>43</u>	<u>45</u>	<u>51</u>

- Behauptung:
 - Nach dem j .ten Schleifendurchlauf gilt (Schleifeninvariante):

$$a_0 \leq \dots \leq a_j$$

- IA: $j = 1$:
 - In der Schleife wird $\text{key} = a[j] = a[1]$ und $i = j - 1 = 0$
 - 1. Fall: $a[i] \leq \text{key}$, dann wird die while-Schleife nicht betreten, $a[j]$ bleibt $a[j]$ und es gilt: $a[0] = a[i] \leq a[j] = a[1]$
 - 2. Fall: $a[i] > \text{key}$, dann wird die while-Schleife betreten, und es werden die Inhalte von $a[0]$ und $a[1]$ getauscht.
Es gilt: $a[0] = a[j] \leq a[i] = a[1]$
- IV: Behauptung gilt für $j-1$

Korrektheit InsertionSort II

- IS: $j - 1 \rightarrow j$:
 - Nach IV gilt nach dem $(j - 1)$.ten Schleifendurchlauf (Schleifeninvariante):
$$a_0 \leq \dots \leq a_{j-1}$$
 - Im j .ten Schleifendurchlauf wird $\text{key} = a[j]$ (Sicherung aktueller Wert) und $i = j - 1$ (Index letztes Element in der bereits sortierten Folge)
 - In der while-Schleife wird nun mittels Iteration über die bereits sortierte Folge die Position gesucht, an der das neue Element eingeordnet werden muss
 - Je Iteration wird der Laufindex um 1 verringert und nicht ungültig
 - Da der aktuelle Wert gesichert ist, ist eine Tauschlücke entstanden, die genutzt wird, um die Werte bei Iteration jeweils um eine Position nach rechts zu versetzen (solange bis die Zielstelle gefunden wurde)
 - Zuletzt wird die Zielposition mit dem gesicherten Wert gefüllt, so dass:

$$a_0 \leq \dots \leq \text{key} \leq \dots \leq a_{j-1}, \text{ d.h. } a_0 \leq \dots \leq a_j \quad (\text{Schleifenende bei } j = n - 1)$$

- Best Case

$$T(n) = \sum_{i=1}^{n-1} 4 = \Theta(n)$$

- Average Case

$$T(n) = \sum_{i=1}^{n-1} \frac{i}{2} = \frac{(n-1)n}{4} = \Theta(n^2)$$

- Worst Case

$$T(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = \Theta(n^2)$$

Sortieren durch Vertauschen (Bubble Sort)

```
void BubbleSort(int a[], int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = n-2; j >=i; j--)
        {
            if (a[j] > a[j+1])
            {
                int h = a[j];
                a[j] = a[j+1];
                a[j+1] = h;
            }
        }
    }
}
```

Vertauschung:
Minimum wandert wie
eine Blase nach vorne

Anmerkung: Man kann auch die Maxima „wandern“ lassen

- Laufzeit: $T(n) = \Theta(n^2)$ (gilt für Best Case, Average Case und Worst Case!)

Beispiel Bubble Sort I

34	45	12	34	23	18	38	17	43	7
34	45	12	34	23	18	38	17	7	43
34	45	12	34	23	18	38	7	17	43
34	45	12	34	23	18	7	38	17	43
34	45	12	34	23	7	18	38	17	43
34	45	12	34	7	23	18	38	17	43
34	45	12	7	34	23	18	38	17	43
34	45	7	12	34	23	18	38	17	43
34	7	45	12	34	23	18	38	17	43
<u>7</u>	34	45	12	34	23	18	38	17	43
<u>7</u>	34	45	12	34	23	18	38	17	43
<u>7</u>	34	45	12	34	23	18	17	38	43

Beispiel Bubble Sort II

7	34	45	12	34	23	17	18	38	43
7	34	45	12	34	17	23	18	38	43
7	34	45	12	17	34	23	18	38	43
7	34	45	12	17	34	23	18	38	43
7	34	12	45	17	34	23	18	38	43
7	<u>12</u>	34	45	17	34	23	18	38	43
7	<u>12</u>	34	45	17	34	23	18	38	43
7	<u>12</u>	34	45	17	34	23	18	38	43
7	<u>12</u>	34	45	17	34	18	23	38	43
7	<u>12</u>	34	45	17	18	34	23	38	43
7	<u>12</u>	34	45	17	18	34	23	38	43
7	<u>12</u>	34	17	45	18	34	23	38	43

Beispiel Bubble Sort III

7	12	17	34	45	18	34	23	38	43
7	12	17	34	45	18	34	23	38	43
7	12	17	34	45	18	34	23	38	43
7	12	17	34	45	18	23	34	38	43
7	12	17	34	45	18	23	34	38	43
7	12	17	34	18	45	23	34	38	43
7	12	17	18	34	45	23	34	38	43
7	12	17	18	34	45	23	34	38	43
7	12	17	18	34	45	23	34	38	43
7	12	17	18	34	23	45	34	38	43
7	12	17	18	23	34	45	34	38	43

Beispiel Bubble Sort IV

7	12	17	18	<u>23</u>	34	45	34	38	43
7	12	17	18	<u>23</u>	34	45	34	38	43
7	12	17	18	<u>23</u>	34	34	45	38	43
7	12	17	18	<u>23</u>	34	34	45	38	43
7	12	17	18	<u>23</u>	<u>34</u>	34	45	38	43
7	12	17	18	<u>23</u>	<u>34</u>	34	38	45	43
7	12	17	18	<u>23</u>	<u>34</u>	<u>34</u>	38	45	43
7	12	17	18	<u>23</u>	<u>34</u>	<u>34</u>	38	43	45
7	12	17	18	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	43	45
7	12	17	18	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	<u>43</u>	45

Sortieren durch Auswählen (Selection Sort)

```
void SelectionSort(int a[],int n)
{
    int i, j, min;
    for (i = 0; i < n; i++)
    {
        min = i;
        for (j = i; j < n; j++)
        {
            if (a[j] < a[min]) min = j;
        }
        int h = a[i];
        a[i] = a[min];
        a[min] = h;
    }
}
```

Minimum wird ermittelt und an die richtige Stelle gesetzt

- Laufzeit: $T(n) = \Theta(n^2)$ (gilt für Best Case, Average Case und Worst Case!)

Beispiel Selection Sort

34	45	12	34	23	18	38	17	43	7
7	45	12	34	23	18	38	17	43	34
7	<u>12</u>	45	34	23	18	38	17	43	34
7	<u>12</u>	<u>17</u>	34	23	18	38	45	43	34
7	<u>12</u>	<u>17</u>	<u>18</u>	23	34	38	45	43	34
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	34	38	45	43	34
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	38	45	43	34
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	43	45
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	<u>43</u>	<u>45</u>

Anmerkungen einfache Sortieralgorithmen

- Komponentenweiser Aufbau
 - Lösungsfindung erfolgt dadurch, dass die Initiallösung schrittweise vergrößert wird (induktive Fortsetzung)
 - Gleichzeitig wird das Restproblem in jedem Schritt um eins verkleinert
- In jedem Schritt sind $\Theta(n)$ viele Vergleichs- bzw. Vertauschoperationen notwendig, daher **quadratischer Aufwand**
- Einfache Sortieralgorithmen sind nur für kurze Folgen geeignet
- Sortieren durch Einfügen ist ideal, wenn eine sehr gut vorsortierte Folge vorliegt
 - Anwendung z.B. Aktualisierung von geometrischen Datenstrukturen

Quicksort (C.A.R. Hoare, 1962)

```
void PreparePartition(int a[], int f, int l, int &p)
{
    // Pivot-Element
    int pivot = a[f]; p = f-1;
    for (int i = f; i <= l; i++)
    {
        if (a[i] <= pivot)
        {
            p++; swap(a[i], a[p]);
        }
    }
    // Pivot an die
    // richtige Stelle
    swap(a[f], a[p]);
}
```

- Divide & Conquer
- Gruppierung nach
Pivot- bzw. Split-Element

```
void swap(int &a, int &b)
{
    int h=b;
    b=a;
    a=h;
}
```

```
void Quicksort(int a[], int f, int l)
{
    int part;
    if (f<l) {
        PreparePartition(a, f, l, part);
        Quicksort(a, f, part-1);
        Quicksort(a, part+1, l);
    }
}
```

Beispiel Quicksort I

34	45	12	34	23	18	38	17	43	7
34	45	12	34	23	18	38	17	43	7
34	12	45	34	23	18	38	17	43	7
34	12	34	45	23	18	38	17	43	7
34	12	34	23	45	18	38	17	43	7
34	12	34	23	18	45	38	17	43	7
34	12	34	23	18	45	38	17	43	7
34	12	34	23	18	17	38	45	43	7
34	12	34	23	18	17	38	45	43	7
34	12	34	23	18	17	7	45	43	38

Beispiel Quicksort II

7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
<u>7</u>	12	34	23	18	17	<u>34</u>	45	43	38
<u>7</u>	12	34	23	18	17	<u>34</u>	45	43	38
<u>7</u>	12	34	23	18	17	<u>34</u>	45	43	38

Beispiel Quicksort III

7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	17	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	17	23	18	<u>34</u>	<u>34</u>	45	43	38

Beispiel Quicksort IV

7	<u>12</u>	17	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	17	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	38	43	<u>45</u>

Beispiel Quicksort V

7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	38	43	<u>45</u>
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	38	43	<u>45</u>
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	<u>43</u>	<u>45</u>

Beispiel Quicksort (kompakt)

34	45	12	34	23	18	38	17	43	7
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	17	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	23	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	34	<u>34</u>	38	43	<u>45</u>
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	43	<u>45</u>
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	<u>43</u>	<u>45</u>

- Anzahl der Schritte hängt von der **Wahl des Pivot-Elements** ab
- Best Case (Halbierung mit jedem Rekursionsaufruf)

$$T(n) = 2T\left(\frac{n}{2}\right) + n = \Theta(n \log n)$$

- Worst Case (Je Rekursionsaufruf wird nur 1 Element bearbeitet)

$$T(n) = T(n - 1) + T(0) + n = \Theta(n^2)$$

- Average Case

$$T_i(n) = T(i - 1) + T(n - i) + n$$

(Anzahl der Schritte bei Trennwert $i \in \{1, \dots, n\}$)

- Annahme: Alle Positionen sind gleichwahrscheinlich, dann gilt:

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

- Es gilt:

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

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

$$= \frac{1}{n} (T(0) + T(n-1) + T(1) + T(n-2) + \dots + T(n-1) + T(0)) + \frac{n^2}{n}$$

$$= \frac{2}{n} \sum_{i=0}^{n-1} T(i) + n$$

- Weiter gilt:

$$n \cdot T(n) = 2 \sum_{i=0}^{n-1} T(i) + n^2$$

- Für $n - 1$ gilt also:

$$(n - 1) \cdot T(n - 1) = 2 \sum_{i=0}^{n-2} T(i) + (n - 1)^2$$

- Differenz der unteren Gleichung von der oberen liefert:

$$n \cdot T(n) - (n - 1) \cdot T(n - 1) = 2 \sum_{i=0}^{n-1} T(i) + n^2 - \left(2 \sum_{i=0}^{n-2} T(i) + (n - 1)^2 \right)$$

- Umformung ergibt:

$$2 \sum_{i=0}^{n-1} T(i) + n^2 - \left(2 \sum_{i=0}^{n-2} T(i) + (n - 1)^2 \right) = 2T(n - 1) + 2n - 1$$

- Insgesamt folgt:

$$n \cdot T(n) - (n - 1) \cdot T(n - 1) = 2T(n - 1) + 2n - 1$$

- D.h.:

$$n \cdot T(n) - (n + 1) \cdot T(n - 1) = 2n - 1$$

- Teilung durch $n(n + 1)$ liefert:

$$\frac{T(n)}{n + 1} = \frac{T(n - 1)}{n} + \frac{2n - 1}{n(n + 1)}$$

- Substitution $\hat{T}(n) = \frac{T(n)}{n+1}$ liefert:

$$\hat{T}(n) = \hat{T}(n - 1) + \frac{2n - 1}{n(n + 1)} = \hat{T}(n - 2) + \frac{2(n - 1) - 1}{(n - 1)((n - 1) + 1)} + \frac{2n - 1}{n(n + 1)}$$

$$\dots = \hat{T}(1) + \sum_{i=2}^n \frac{2i - 1}{i(i + 1)} = \hat{T}(1) + \sum_{i=2}^n \frac{3}{i + 1} - \sum_{i=2}^n \frac{1}{i}$$

- Erinnerung **Harmonische Reihe**:

$$\sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = \Theta(\log n)$$

- Damit folgt:

$$\hat{T}(n) = \hat{T}(1) + \sum_{i=2}^n \frac{3}{i+1} - \sum_{i=2}^n \frac{1}{i} = \hat{T}(1) + \left(\frac{3}{3} - \frac{1}{2} \right) + \left(\frac{3}{4} - \frac{1}{3} \right) + \left(\frac{3}{5} - \frac{1}{4} \right) + \dots$$

$$= \hat{T}(1) - \frac{1}{2} + 2 \sum_{i=3}^n \frac{1}{i} + \frac{3}{n+1} = \Theta(\log n)$$

- Rücksubstitution $T(n) = \hat{T}(n) \cdot (n + 1)$ liefert für die Laufzeit von Quicksort:

$$T(n) = \Theta(n \log n)$$

Algorithmen und Datenstrukturen

- Sortieralgorithmen -

Prof. Dr. Klaus Volbert

Wintersemester 2018/19
Regensburg, 22. Oktober 2018

Überblick Sortieralgorithmen

- Nichts ist besser verstanden als das Sortieren...
- Einfache Sortieralgorithmen
 - Sortieren durch Einfügen (Insertion Sort) ✓
 - Sortieren durch Vertauschen (Bubble Sort) ✓
 - Sortieren durch Auswählen (Selection Sort) ✓
- Fortgeschrittene Sortieralgorithmen
 - Sortieren durch Divide & Conquer (Quicksort) (✓)
 - Sortieren durch Mischen (Merge Sort)
 - Sortieren durch Bäume (Heap Sort)
- Spezielle Sortieralgorithmen
 - Sortieren durch Zählen (Count Sort)
 - Sortieren durch Abbilden (Map Sort)
- Untere Schranke für vergleichsbasierte Sortieralgorithmen

Quicksort (C.A.R. Hoare, 1962)

```
void PreparePartition(int a[], int f, int l, int &p)
{
    // Pivot-Element
    int pivot = a[f]; p = f-1;
    for (int i = f; i <= l; i++)
    {
        if (a[i] <= pivot)
        {
            p++; swap(a[i], a[p]);
        }
    }
    // Pivot an die
    // richtige Stelle
    swap(a[f], a[p]);
}
```

- Divide & Conquer
- Gruppierung nach
Pivot- bzw. Split-Element

```
void swap(int &a, int &b)
{
    int h=b;
    b=a;
    a=h;
}
```

```
void Quicksort(int a[], int f, int l)
{
    int part;
    if (f<l) {
        PreparePartition(a, f, l, part);
        Quicksort(a, f, part-1);
        Quicksort(a, part+1, l);
    }
}
```

Beispiel Quicksort I

34	45	12	34	23	18	38	17	43	7
34	45	12	34	23	18	38	17	43	7
34	12	45	34	23	18	38	17	43	7
34	12	34	45	23	18	38	17	43	7
34	12	34	23	45	18	38	17	43	7
34	12	34	23	18	45	38	17	43	7
34	12	34	23	18	45	38	17	43	7
34	12	34	23	18	17	38	45	43	7
34	12	34	23	18	17	38	45	43	7
34	12	34	23	18	17	7	45	43	38

Beispiel Quicksort II

7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
<u>7</u>	12	34	23	18	17	<u>34</u>	45	43	38
<u>7</u>	12	34	23	18	17	<u>34</u>	45	43	38
<u>7</u>	12	34	23	18	17	<u>34</u>	45	43	38

Beispiel Quicksort III

7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	17	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	17	23	18	<u>34</u>	<u>34</u>	45	43	38

Beispiel Quicksort IV

7	<u>12</u>	17	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	17	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	38	43	<u>45</u>

Beispiel Quicksort V

7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	38	43	<u>45</u>
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	38	43	<u>45</u>
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	<u>43</u>	<u>45</u>

Beispiel Quicksort (kompakt)

34	45	12	34	23	18	38	17	43	7
7	12	34	23	18	17	<u>34</u>	45	43	38
7	12	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	34	23	18	17	<u>34</u>	45	43	38
7	<u>12</u>	17	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	23	18	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	45	43	38
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	38	43	<u>45</u>
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	43	<u>45</u>
7	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	<u>43</u>	<u>45</u>

- Anzahl der Schritte hängt von der **Wahl des Pivot-Elements** ab
- Best Case (Halbierung mit jedem Rekursionsaufruf)

$$T(n) = 2T\left(\frac{n}{2}\right) + n = \Theta(n \log n)$$

- Worst Case (Je Rekursionsaufruf wird nur 1 Element bearbeitet)

$$T(n) = T(n - 1) + T(0) + n = \Theta(n^2)$$

- Average Case

$$T_i(n) = T(i - 1) + T(n - i) + n$$

(Anzahl der Schritte bei Trennwert $i \in \{1, \dots, n\}$)

- Annahme: Alle Positionen sind gleichwahrscheinlich, dann gilt:

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

- Es gilt:

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

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

$$= \frac{1}{n} (T(0) + T(n-1) + T(1) + T(n-2) + \dots + T(n-1) + T(0)) + \frac{n^2}{n}$$

$$= \frac{2}{n} \sum_{i=0}^{n-1} T(i) + n$$

- Weiter gilt:

$$n \cdot T(n) = 2 \sum_{i=0}^{n-1} T(i) + n^2$$

- Für $n - 1$ gilt also:

$$(n - 1) \cdot T(n - 1) = 2 \sum_{i=0}^{n-2} T(i) + (n - 1)^2$$

- Differenz der unteren Gleichung von der oberen liefert:

$$n \cdot T(n) - (n - 1) \cdot T(n - 1) = 2 \sum_{i=0}^{n-1} T(i) + n^2 - \left(2 \sum_{i=0}^{n-2} T(i) + (n - 1)^2 \right)$$

- Umformung ergibt:

$$2 \sum_{i=0}^{n-1} T(i) + n^2 - \left(2 \sum_{i=0}^{n-2} T(i) + (n - 1)^2 \right) = 2T(n - 1) + 2n - 1$$

- Insgesamt folgt:

$$n \cdot T(n) - (n - 1) \cdot T(n - 1) = 2T(n - 1) + 2n - 1$$

- D.h.:

$$n \cdot T(n) - (n + 1) \cdot T(n - 1) = 2n - 1$$

- Teilung durch $n(n + 1)$ liefert:

$$\frac{T(n)}{n + 1} = \frac{T(n - 1)}{n} + \frac{2n - 1}{n(n + 1)}$$

- Substitution $\hat{T}(n) = \frac{T(n)}{n+1}$ liefert:

$$\hat{T}(n) = \hat{T}(n - 1) + \frac{2n - 1}{n(n + 1)} = \hat{T}(n - 2) + \frac{2(n - 1) - 1}{(n - 1)((n - 1) + 1)} + \frac{2n - 1}{n(n + 1)}$$

$$\dots = \hat{T}(1) + \sum_{i=2}^n \frac{2i - 1}{i(i + 1)} = \hat{T}(1) + \sum_{i=2}^n \frac{3}{i + 1} - \sum_{i=2}^n \frac{1}{i}$$

- Erinnerung **Harmonische Reihe**:

$$\sum_{k=1}^n \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = \Theta(\log n)$$

- Damit folgt:

$$\hat{T}(n) = \hat{T}(1) + \sum_{i=2}^n \frac{3}{i+1} - \sum_{i=2}^n \frac{1}{i} = \hat{T}(1) + \left(\frac{3}{3} - \frac{1}{2} \right) + \left(\frac{3}{4} - \frac{1}{3} \right) + \left(\frac{3}{5} - \frac{1}{4} \right) + \dots$$

$$= \hat{T}(1) - \frac{1}{2} + 2 \sum_{i=3}^n \frac{1}{i} + \frac{3}{n+1} = \Theta(\log n)$$

- Rücksubstitution $T(n) = \hat{T}(n) \cdot (n + 1)$ liefert für die Laufzeit von Quicksort:

$$T(n) = \Theta(n \log n)$$

Anmerkungen Quicksort

- Quicksort gilt in der Praxis als
 - Schnell, berühmt und breit einsetzbar (Laufzeit ideal)
 - In vielen Programmiersprachen ist die Sortierung von Objekten über Standard-Bibliotheken realisiert und verfügbar
 - Bei Verwendung kann/muss eine individuelle Vergleichsfunktion implementiert werden (siehe z.B. Comparable in Java oder C#)
- Varianten für die Wahl des Pivot-Elements:
 - Erstes, letztes, i .tes Element des Feldes
 - Median aus erstes, letztes, i .tes Element des Feldes
 - Ein zufälliges Element des Feldes (Zufallszahl)
- Die letzten beiden Varianten sind in der Praxis auch gut bei fast sortierten Folgen

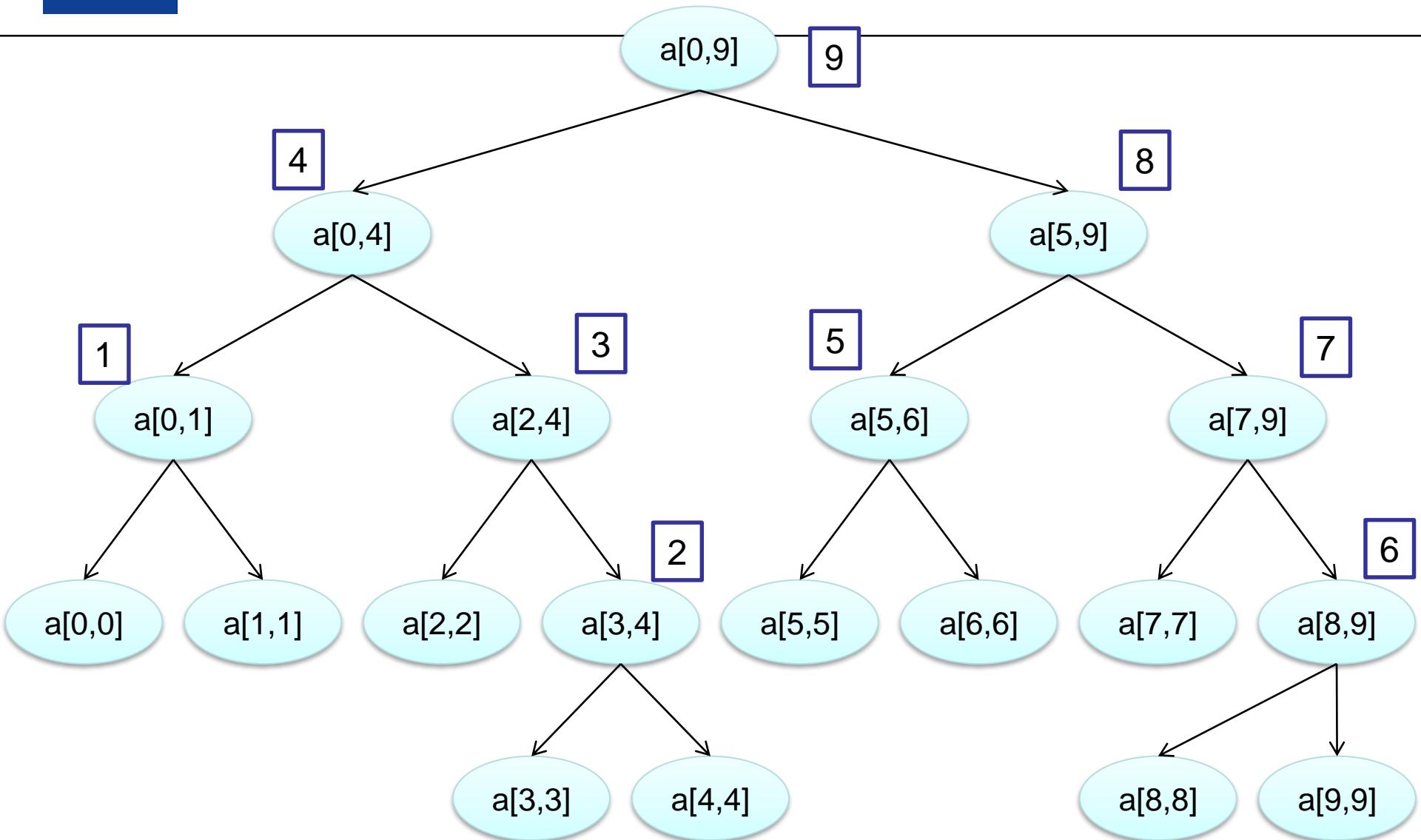
Sortieren durch Mischen (Merge Sort)

```
void Merge(int a[], int f, int l, int m) {  
    int i, n = l - f + 1;  
    int a1f = f, a1l = m-1;  
    int a2f = m, a2l = l;  
    int *anew = new int[n];  
  
    for (i = 0; i < n; i++)  
    { if (a1f <= a1l) {  
        if (a2f <= a2l)  
        { if (a[a1f] <= a[a2f]) anew[i]=a[a1f++];  
          else anew[i]=a[a2f++]; }  
        else anew[i]=a[a1f++]; }  
    else anew[i]=a[a2f++]; }  
  
    for (i=0; i<n; i++) a[f+i]=anew[i];  
  
    delete [] anew; }
```

```
void MergeSort(int a[], int f, int l)  
{ if (f<l) {  
    int m = (f+l+1)/2;  
    MergeSort(a, f, m-1);  
    MergeSort(a, m, l);  
    Merge(a, f, l, m); }  
}
```

Zwei sortierte Teilstufen werden unter Verwendung eines neuen Feldes zum Umspeichern zu einer sortierten Folge „gemischt“

Rekursionsbaum über die Aufrufe



Beispiel Merge Sort I

34	45	12	34	23		18	38	17	43	7	Teilen
34	45		12	34	23	18	38	17	43	7	Teilen
<u>34</u>	<u>45</u>		12	34	23	18	38	17	43	7	Teilen
<u>34</u>	<u>45</u>		12	34	23	18	38	17	43	7	Mischen
34	45	<u>12</u>	<u>34</u>	<u>23</u>		18	38	17	43	7	Teilen
34	45	12	<u>34</u>	<u>23</u>		18	38	17	43	7	Teilen
34	45	12	<u>23</u>	<u>34</u>		18	38	17	43	7	Mischen
34	45	<u>12</u>	<u>23</u>	<u>34</u>		18	38	17	43	7	Mischen
<u>12</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>45</u>		18	38	17	43	7	Mischen
12	23	34	34	45		<u>18</u>	<u>38</u>		17	43	7

Beispiel Merge Sort II

12	23	34	34	45	<u>18</u> <u>38</u>	17	43	7	Teilen	
12	23	34	34	45	<u>18</u> <u>38</u>	17	43	7	Mischen	
12	23	34	34	45	18 38	<u>17</u> <u>43</u> 7			Teilen	
12	23	34	34	45	18 38	17	<u>43</u> <u>7</u>		Teilen	
12	23	34	34	45	18 38	17	<u>7</u> <u>43</u>		Mischen	
12	23	34	34	45	18 38	<u>7</u> <u>17</u> <u>43</u>			Mischen	
12	23	34	34	45	<u>7</u> <u>17</u> <u>18</u> <u>38</u> <u>43</u>				Mischen	
<u>7</u>	<u>12</u>	<u>17</u>	<u>18</u>	<u>23</u>	<u>34</u>	<u>34</u>	<u>38</u>	<u>43</u>	<u>45</u>	Mischen

Laufzeit Merge Sort

- Laufzeit (Best Case, Average Case , Worst Case):

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

- Lösen der Rekursionsgleichung durch
 - Iterationsmethode, oder
 - Master-Methode
- Liefert:

$$T(n) = \Theta(n \log n)$$

- Wir sagen:
 - In jeder Rekursionsstufe werden alle Zahlen gemischt: $\Theta(n)$
 - Halbierung hat $\Theta(\log n)$ Rekursionsstufen, also folgt $\Theta(n \log n)$

Anmerkungen Merge Sort

- Merge Sort geht nach dem **Divide-&-Conquer**-Entwurfsprinzip vor
- Merge Sort eignet sich ideal zum **externen Sortieren**, wenn die Daten nicht in den Hauptspeicher passen
 - Teilstücke können in Dateien vorliegen
 - Folge wird in einer weiteren Datei gemischt
 - Zeitaufwand ist auch hier linear
- Time-Space-Tradeoff (Mehr Platz kann Laufzeit verbessern):
 - Mischen mit Umspeichern (*ex situ*) hat **lineare Komplexität**
 - Mischen ohne Umspeichern (*in situ*) hat **quadratische Komplexität**
- Merge Sort ist stabil bzgl. der Laufzeit (gleiche Laufzeit in allen Fällen)
- Viele Varianten/Implementierungsarten (z. B. Natural Merge Sort)

Stabilität von Sortieralgorithmen

- Ein Sortieralgorithmus ist **stabil**, wenn die relative Ordnung der Elemente mit gleichen Schlüsseln durch den Sortierprozess nicht verändert wird
- Beispiel: Sortierung nach Anfangsbuchstaben (1-26)
 - Eingabe: Markus, Sven, Walter, Michael, Franz
 13, 19, 23, 13, 6
 - Ausgabe 1: Franz, Markus, Michael, Sven, Walter
 6, 13, 13, 19, 23
 - Ausgabe 2: Franz, Michael, Markus, Sven, Walter
 6, 13, 13, 19, 23
- Welche der Ausgaben kommt von einem stabilen Sortieralgorithmus?
 - Ausgabe 1: Markus steht weiterhin vor Michael
- Welche Sortieralgorithmen sind stabil?
 - Einfache: Insertion Sort, Bubble Sort
 - Fortgeschrittene: Merge Sort

Algorithmen und Datenstrukturen

- Sortieralgorithmen -

Prof. Dr. Klaus Volbert

Wintersemester 2018/19
Regensburg, 25. Oktober 2018

Überblick Sortieralgorithmen

- Nichts ist besser verstanden als das Sortieren...
- Einfache Sortieralgorithmen
 - Sortieren durch Einfügen (Insertion Sort) ✓
 - Sortieren durch Vertauschen (Bubble Sort) ✓
 - Sortieren durch Auswählen (Selection Sort) ✓
- Fortgeschrittene Sortieralgorithmen
 - Sortieren durch Divide & Conquer (Quicksort) ✓
 - Sortieren durch Mischen (Merge Sort) ✓
 - Sortieren durch Bäume (Heap Sort)
- Spezielle Sortieralgorithmen
 - Sortieren durch Zählen (Count Sort)
 - Sortieren durch Abbilden (Map Sort)
- Untere Schranke für vergleichsbasierte Sortieralgorithmen

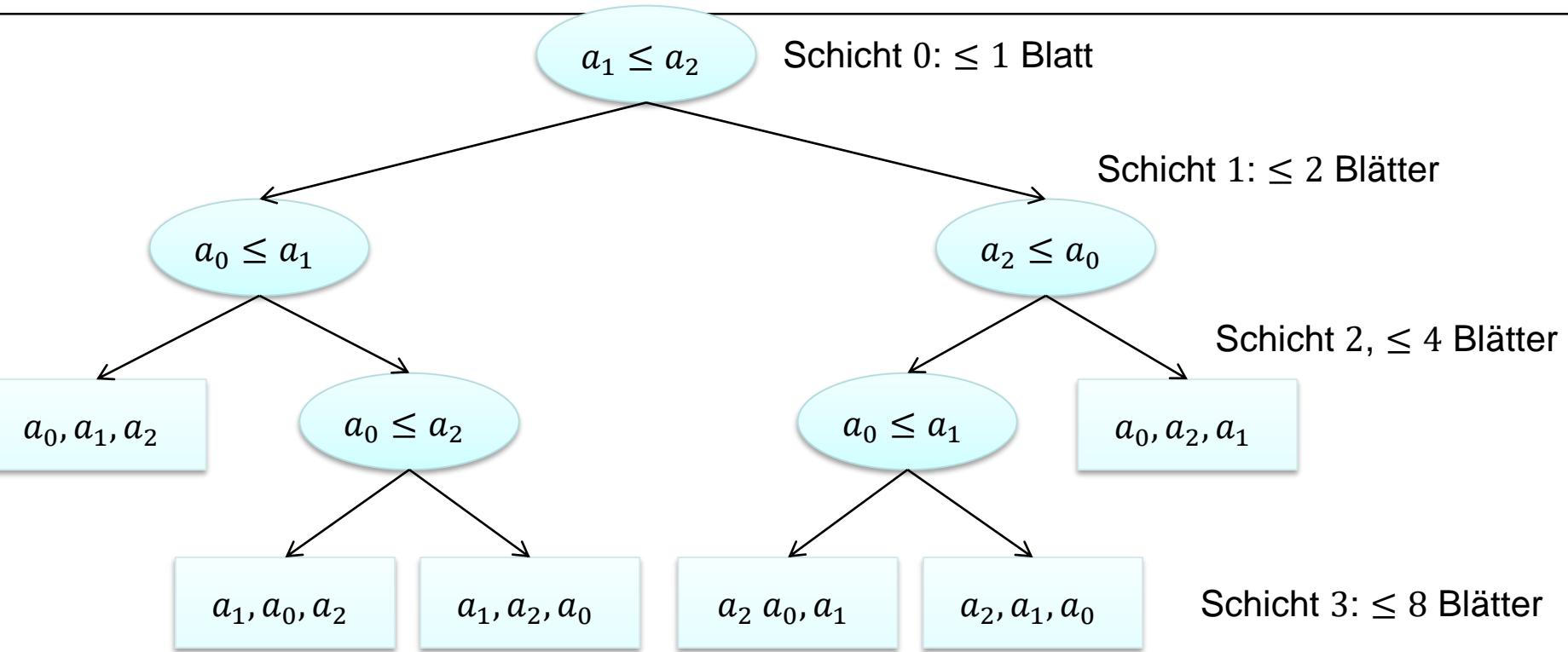
Stabilität von Sortieralgorithmen

- Ein Sortieralgorithmus ist **stabil**, wenn die relative Ordnung der Elemente mit gleichen Schlüsseln durch den Sortierprozess nicht verändert wird
- Beispiel: Sortierung nach Anfangsbuchstaben (1-26)
 - Eingabe: Markus, Sven, Walter, Michael, Franz
 13, 19, 23, 13, 6
 - Ausgabe 1: Franz, Markus, Michael, Sven, Walter
 6, 13, 13, 19, 23
 - Ausgabe 2: Franz, Michael, Markus, Sven, Walter
 6, 13, 13, 19, 23
- Welche der Ausgaben kommt von einem stabilen Sortieralgorithmus?
 - Ausgabe 1: Markus steht weiterhin vor Michael
- Welche Sortieralgorithmen sind stabil?
 - Einfache: Insertion Sort, Bubble Sort
 - Fortgeschrittene: Merge Sort

Untere Schranke für vergleichsbasiertes Sortieren I

- Bisher betrachtete Sortieralgorithmen
 - Kein a-priori-Wissen über die zu sortierenden Objekte
 - Basis: Paarweiser Größenvergleich (Vergleichsorientierte Verfahren)
 - D.h. Reihenfolge von $a, b \in \mathbb{N}$ wird durch $a < b?$ entschieden
- Eine untere Schranke für die Laufzeit von vergleichsorientierten Sortieralgorithmen liefert ein **Entscheidungs- oder Vergleichsbau**
- Ein Entscheidungs- oder Vergleichsbau für a_0, \dots, a_{n-1} zu sortierenden Werten ist ein **binärer Baum**, an dessen innere Knoten Vergleiche der Form $a_i \leq a_j?$ stehen (Vergleich zweier Elemente). Über den linken Ast geht es weiter, wenn $a_i \leq a_j$, ansonsten über den rechten Ast
- Eine konkrete Sortierung ergibt sich aus einem **Pfad** von der **Wurzel** bis zu einem **Blatt**
- Ein Entscheidungs- oder Vergleichsbau heißt **Sortierbaum**, falls alle Eingaben mit dem gleichen Pfad auch die gleiche Permutation haben

Ein Sortierbaum für $n = 3$



- Beobachtungen:
 - Ein Sortierbaum hat mindestens $n!$ Blätter (alle Permutationen müssen vertreten sein)
 - Die Höhe des Baumes ist eine untere Schranke für jeden vergleichsorientierten Sortieralgorithmus, wie hoch muss der Baum sein?

Schicht i : $\leq 2^i$ Blätter

Untere Schranke für vergleichsbasiertes Sortieren II

- Damit alle Permutationen abgedeckt werden, muss gelten:

$$n! \leq 2^i$$

- Dann sind mindestens i Schritte notwendig (vergleichsbasiertes Sortieren)
- Umformen mittels Stirling-Formel ergibt (Übung):

$$i \geq \log(n!) = \Omega(n \log n)$$

- Satz: Für die Laufzeit von **vergleichsbasierten Sortieralgorithmen** folgt:

$$T(n) = \Omega(n \log n)$$

- Folgerungen
 - Die Laufzeit von Quicksort im Average Case ist asymptotisch optimal
 - Die Laufzeit von Merge Sort ist in allen Fällen asymptotisch optimal

Übersicht Sortieralgorithmen

Algorithmus	Best Case	Average Case	Worst Case	Stabil
Insertion Sort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	ja
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	ja
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	nein
Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	nein
Merge Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	ja
Heap Sort
...

- Untere Schranke für vergleichsbasierte Sortieralgorithmen

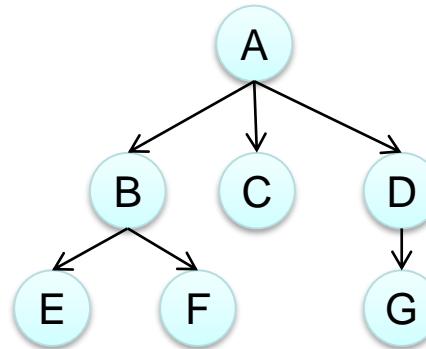
$$T(n) = \Omega(n \log n)$$

Datenstruktur: Baum (Tree)

Deutsch	Englisch	Bemerkung	Beispiel
Wurzel	root	Kein VG, allgemein: Knoten	A
Nachfolger (NF)	successor	Allgemein: Knoten	B ist NF von A
Vorgänger (VG)	predecessor	Allgemein: Knoten	D ist VG von G
Grad eines Knotens	degree	Anzahl der NF eines Knotens	D: Grad 1
Grad eines Baums	degree	Max. Grad im Baum	Baum: Grad 3
Blatt	leaf	Kein NF	C, E, F, G
Innerer Knoten	inner node	Alle NF besetzt	A
Randknoten	boundary node	Nicht alle NF besetzt	B, D
Pfad	path	Knotenfolge (Start, ..., Ziel)	A-F: A, B, F
Pfadlänge	path length	Anzahl der Kanten des Pfades	A-F: 2
Tiefe/Höhe eines Knotens	depth/heigth	Pfadlänge zur Wurzel/zu einem Blatt	G: Tiefe 2/Höhe 0
Tiefe/Höhe eines Baums	depth/heigth	Maximale Tiefe/Höhe	2

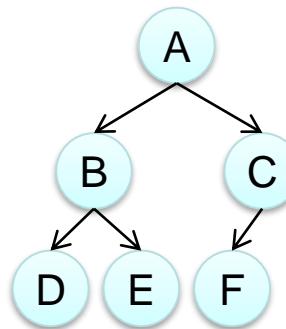
Beispiel: Baum (Tree)

- Beispiel:



- Eigenschaften:
 - Ein Baum mit n Knoten hat genau $n - 1$ Kanten
 - Eine **Schicht** besteht aus allen Knoten gleicher Tiefe (Schicht 0 bis Schicht h)
 - Sind alle Schichten voll, so ist der Baum **vollständig**

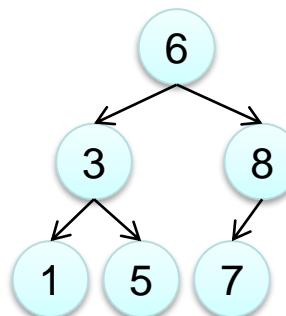
- Ein **Binärbaum** ist ein Baum mit zwei Nachfolgern
- Ein Binärbaum heißt **linksvoll** bzw. **rechtsvoll**, wenn der Baum bis auf die unterste Schicht vollbesetzt ist und die unterste Schicht entweder von links oder von rechts her ohne Lücken besetzt ist
- Beispiel:



Speicherung als Feld:

A	B	C	D	E	F
---	---	---	---	---	---

- Ein **binärer Suchbaum** ist ein linksvoller Binärbaum mit der Eigenschaft:
 - \forall Knoten gilt: Werte links < eigener Wert < Werte rechts
- Beispiel:

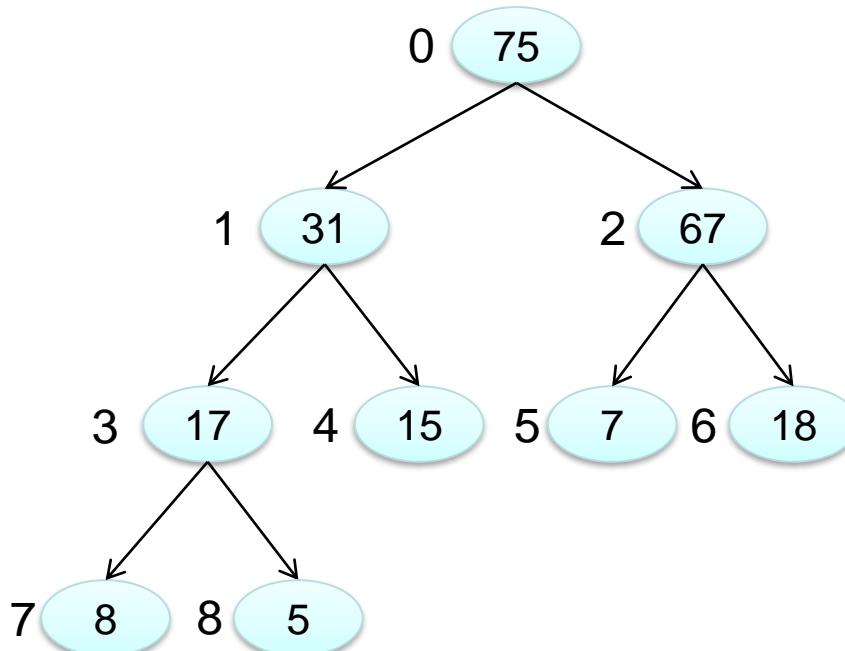


Speicherung als Feld:

6	3	8	1	5	7
---	---	---	---	---	---

Heap (Vorbereitung für Heap Sort)

- Ein **Heap** (Haufen) ist ein linksvoller Binärbaum mit der Eigenschaft:
 - \forall Knoten gilt: Eigener Wert \leq Werte der Nachfolger (Minimum), oder
 - \forall Knoten gilt: Eigener Wert \geq Werte der Nachfolger (Maximum)
- Beispiel (Max-Heap):



Speicherung als Feld a :

75	31	67	17	15	7	18	8	5
----	----	----	----	----	---	----	---	---

Zugriff:

- Wurzel: $a[0]$
- Aktuelle Position: i
- Vorgänger: $\left\lfloor \frac{i-1}{2} \right\rfloor$
- Linker NF: $a[2i + 1]$
- Rechter NF: $a[2i + 2]$

Heap-Eigenschaft: $\forall i: a[i] \geq a[2i + 1]$ und $a[i] \geq a[2i + 2]$

Sortieren mit Heaps (Heap Sort)

```
void Heapify(int a[], int f, int l, int root)
{
    int largest, left=f+(root-f)*2+1, right=f+(root-f)*2+2;
    if (left<=l && a[left]>a[root]) largest=left;
    else largest=root;
    if (right<=l && a[right]>a[largest]) largest=right;
    if (largest!=root) {
        swap(a[root],a[largest]);
        Heapify(a,f,l,largest);
    }
}
```

Bestimme max{root, left, right}
und korrigiere ggf. den Heap

```
void HeapSort(int a[], int f, int l)
{
    BuildHeap(a,f,l);
    for (int i=l; i>f; i--)
    {
        swap(a[f],a[i]);
        Heapify(a,f,i-1,f);
    }
}
```

Baue aus a[] einen Heap

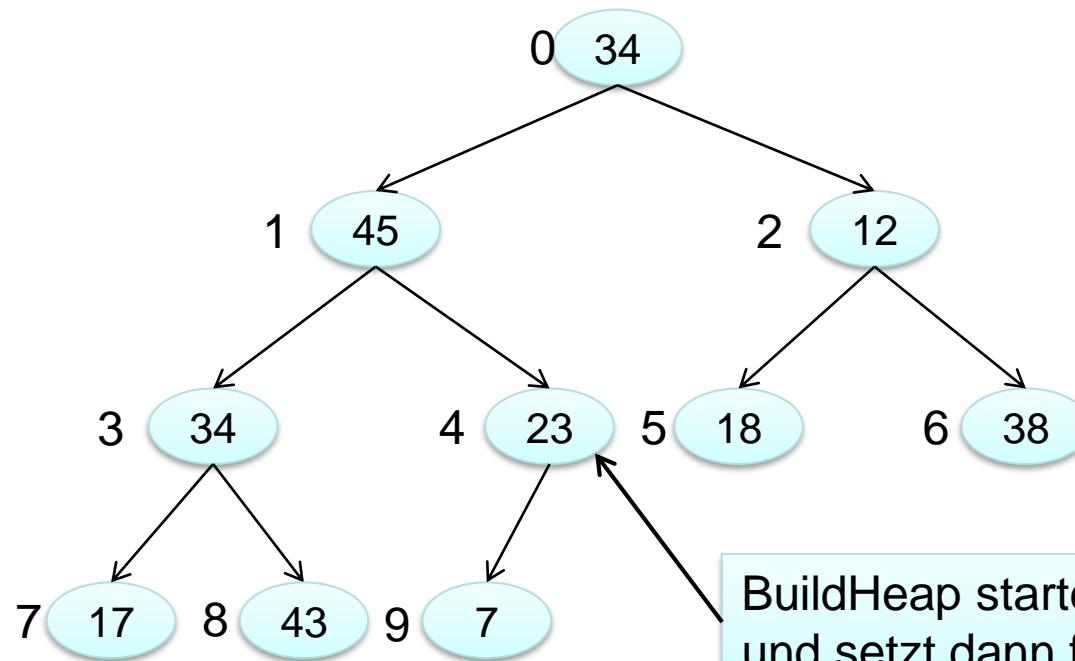
```
void BuildHeap(int a[], int f, int l)
{
    int n=l-f+1;
    for (int i=f+(n-2)/2; i>=f; i--)
        Heapify(a,f,l,i);
}
```

Extrahiere jeweils das Maximum aus dem Heap und baue so eine sortierte Folge

- Eingabe

34 45 12 34 23 18 38 17 43 7

- Darstellung als Heap



BuildHeap startet mit Knoten 4 und setzt dann fort mit 3,2,1,0

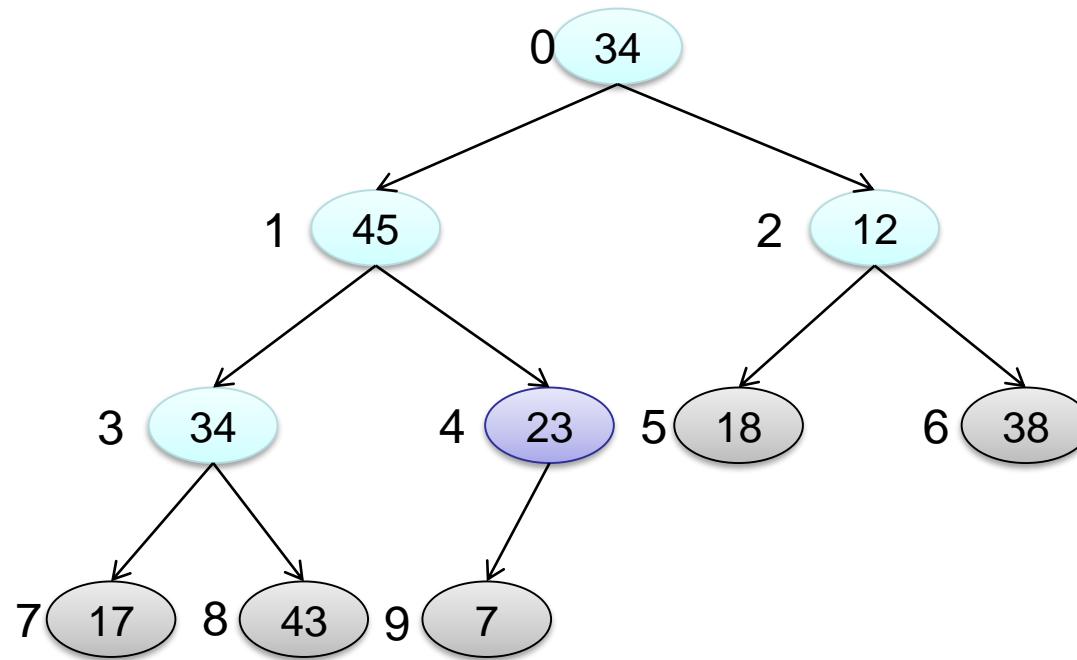
- Beobachtung: Blätter erfüllen die Heap-Eigenschaft

Beispiel Heap Sort

- Aktuelles Feld

34 45 12 34 23 18 38 17 43 7

- Darstellung als Heap



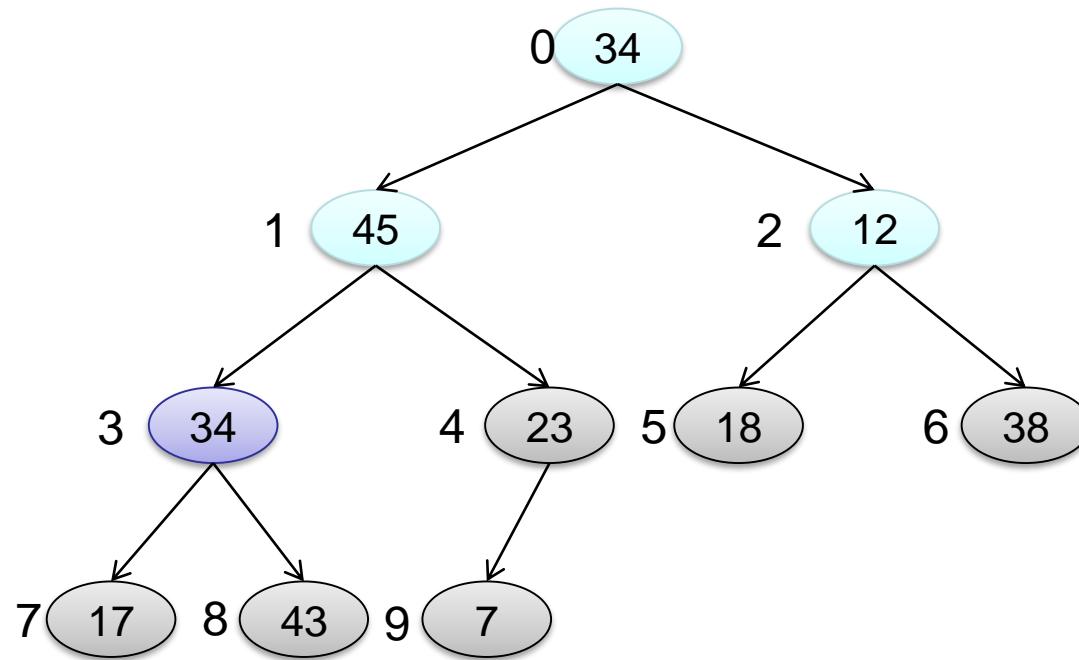
- Bearbeitung Knoten 4: Größtes Element ist die Wurzel

Beispiel Heap Sort

- Aktuelles Feld

34 45 12 34 23 18 38 17 43 7

- Darstellung als Heap



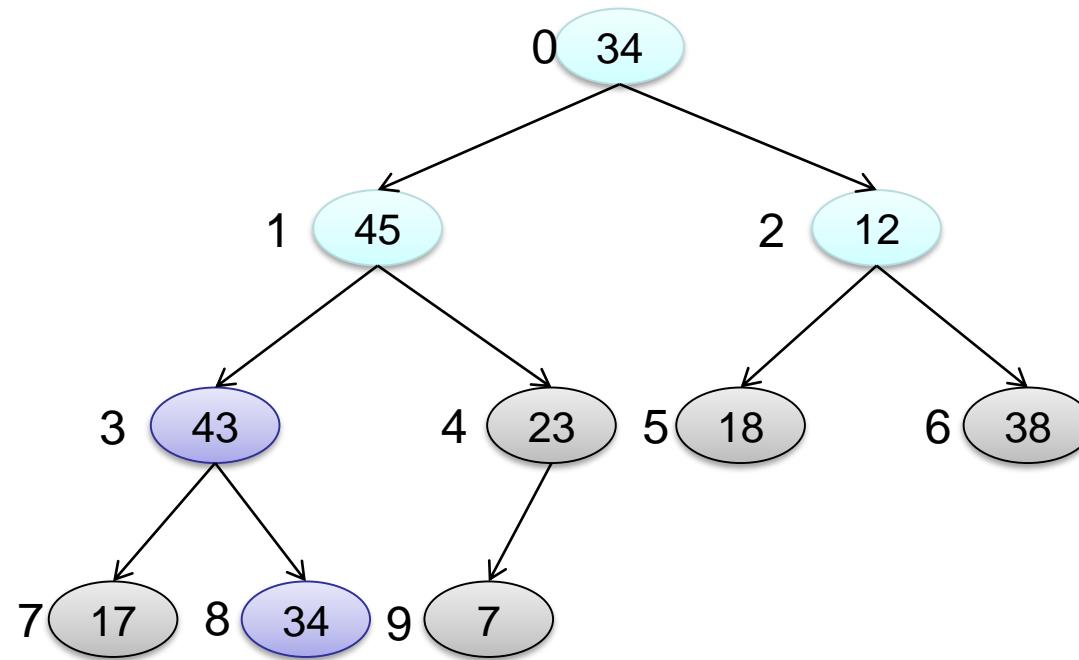
- Bearbeitung Knoten 3: Größtes Element ist der rechte Nachfolger

Beispiel Heap Sort

- Aktuelles Feld

34 45 12 43 23 18 38 17 34 7

- Darstellung als Heap



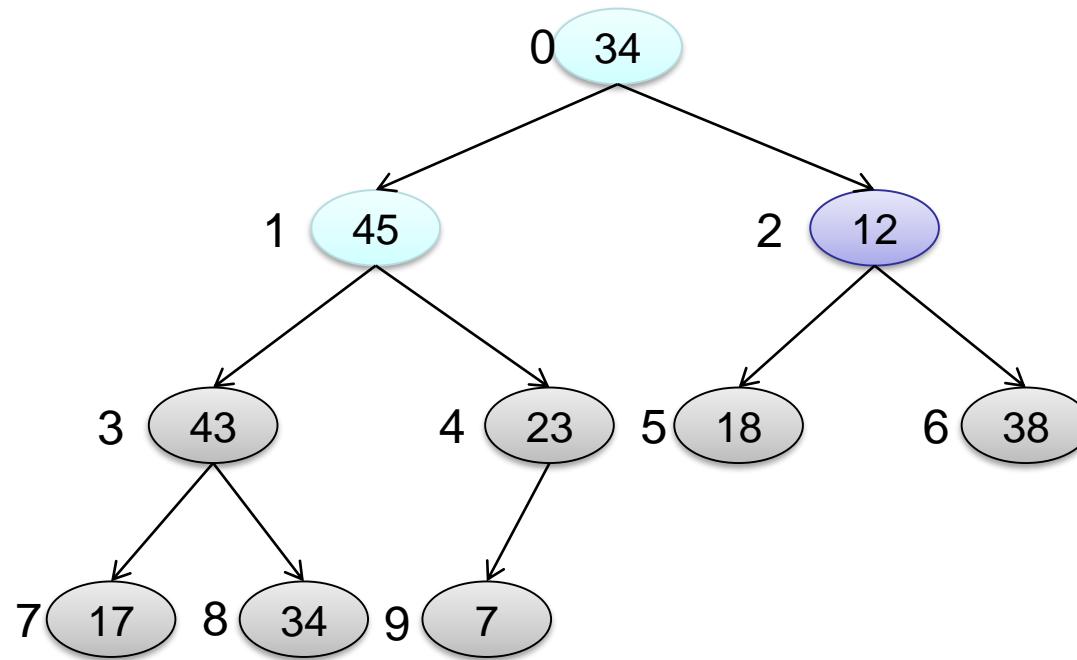
- Bearbeitung Knoten 8: Größtes Element ist die Wurzel

Beispiel Heap Sort

- Aktuelles Feld

34 45 12 43 23 18 38 17 34 7

- Darstellung als Heap



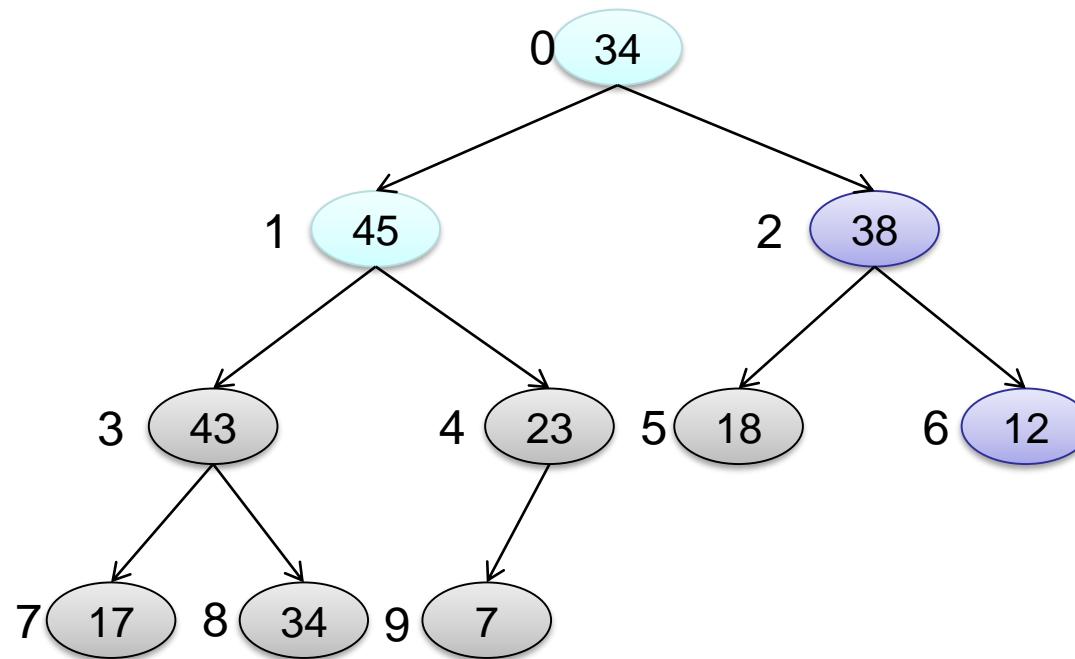
- Bearbeitung Knoten 2: Größtes Element ist der rechte Nachfolger

Beispiel Heap Sort

- Aktuelles Feld

34 45 38 43 23 18 12 17 34 7

- Darstellung als Heap



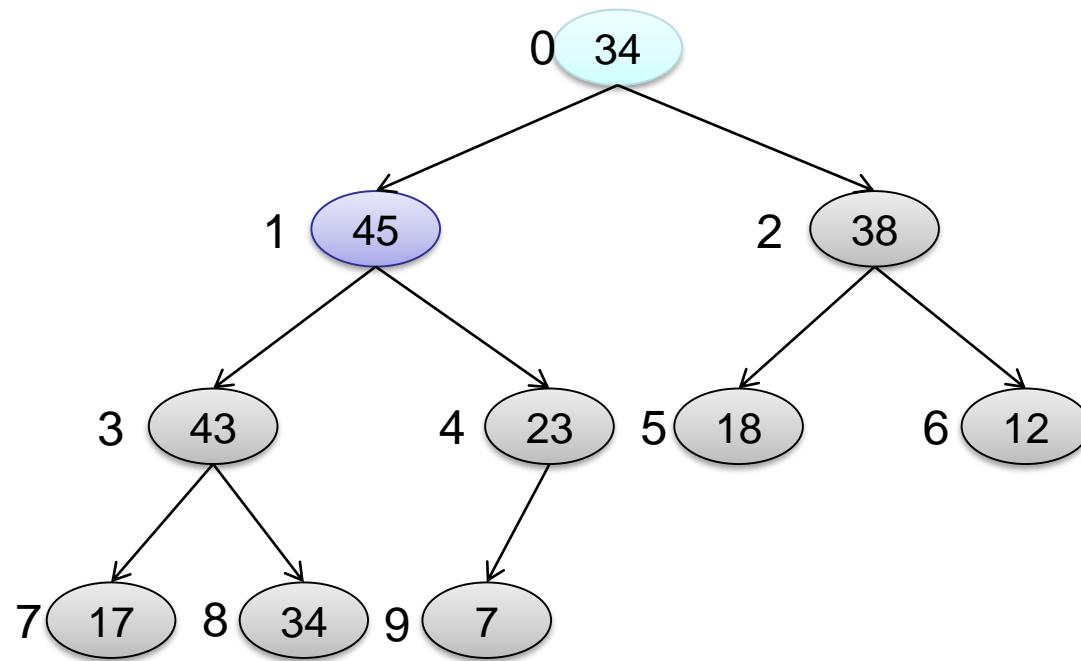
- Bearbeitung Knoten 6: Größtes Element ist die Wurzel

Beispiel Heap Sort

- Aktuelles Feld

34 45 38 43 23 18 12 17 34 7

- Darstellung als Heap



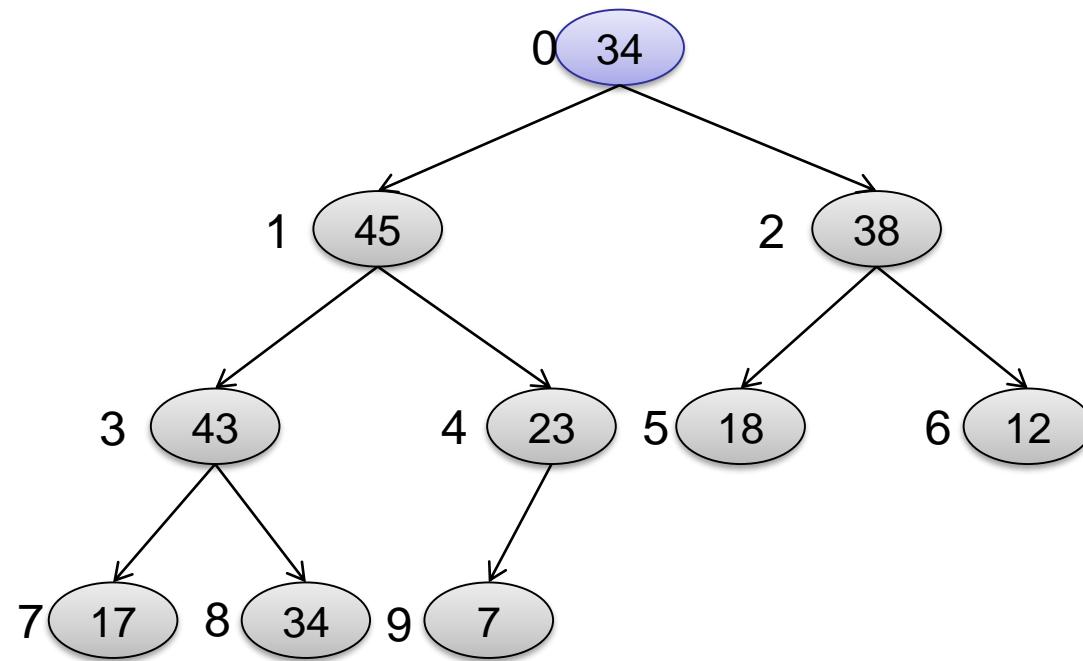
- Bearbeitung Knoten 1: Größtes Element ist die Wurzel

Beispiel Heap Sort

- Aktuelles Feld

34 45 38 43 23 18 12 17 34 7

- Darstellung als Heap



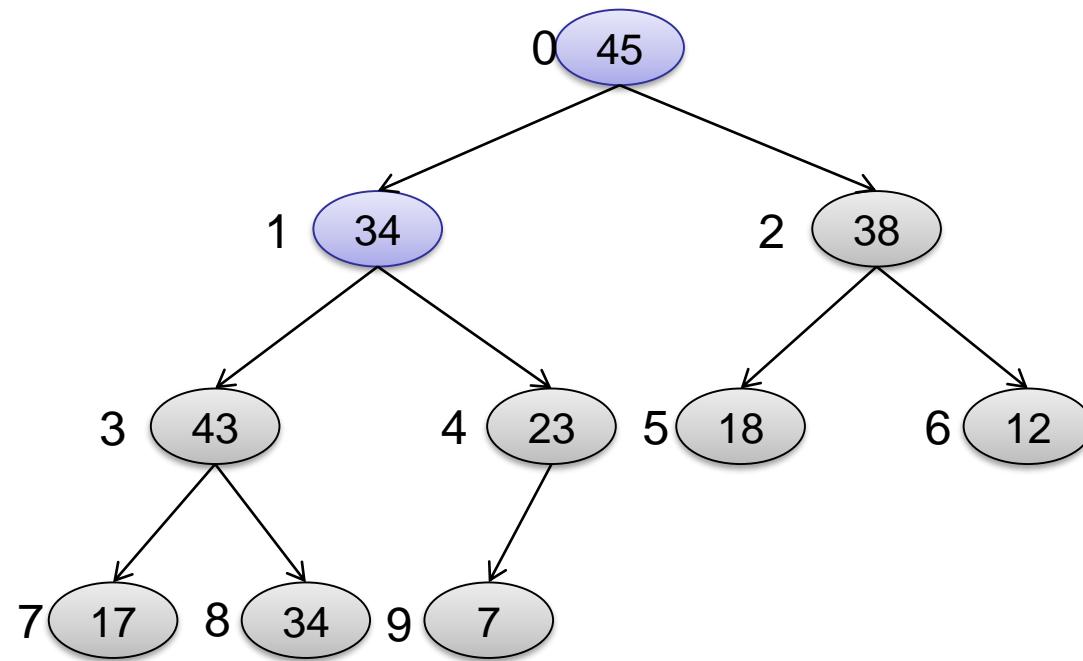
- Bearbeitung Knoten 0: Größtes Element ist der linke Nachfolger

Beispiel Heap Sort

- Aktuelles Feld

45 34 38 43 23 18 12 17 34 7

- Darstellung als Heap



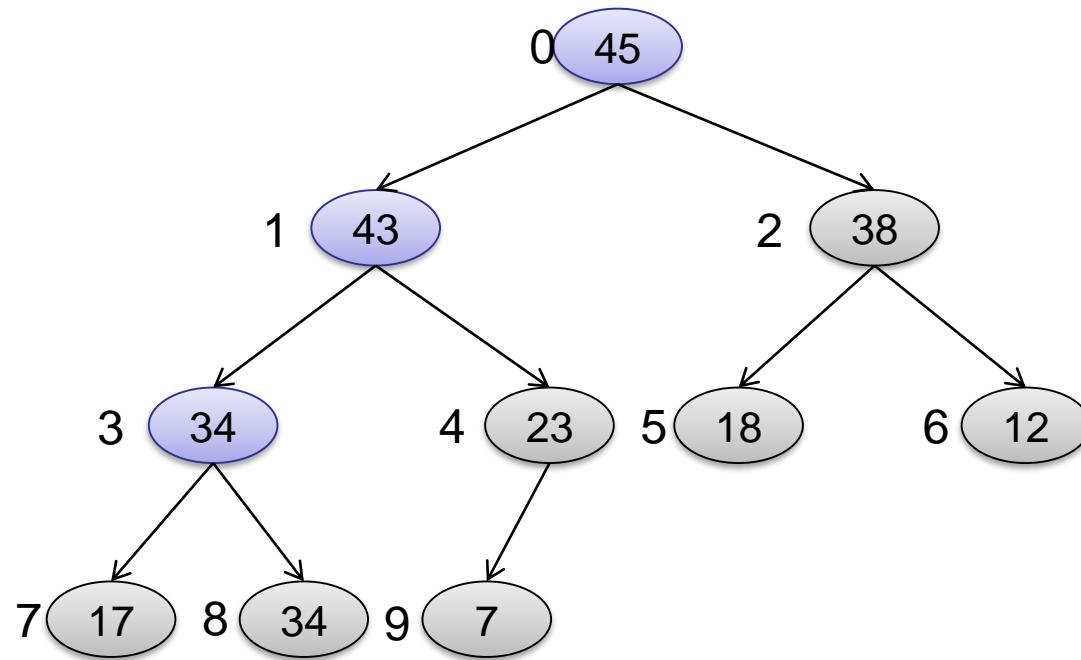
- Bearbeitung Knoten 1: Größtes Element ist der linke Nachfolger

Beispiel Heap Sort

- Aktuelles Feld

45 43 38 34 23 18 12 17 34 7

- Darstellung als Heap



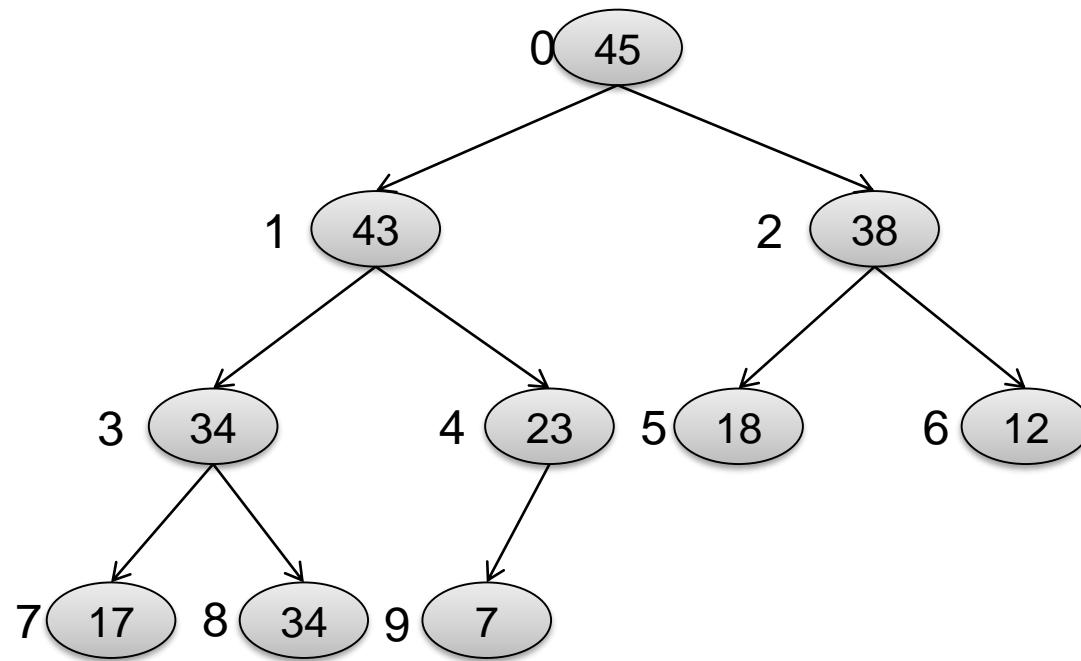
- Bearbeitung Knoten 3: Größtes Element ist die Wurzel

Beispiel Heap Sort

- Aktuelles Feld

45 43 38 34 23 18 12 17 34 7

- Darstellung als Heap

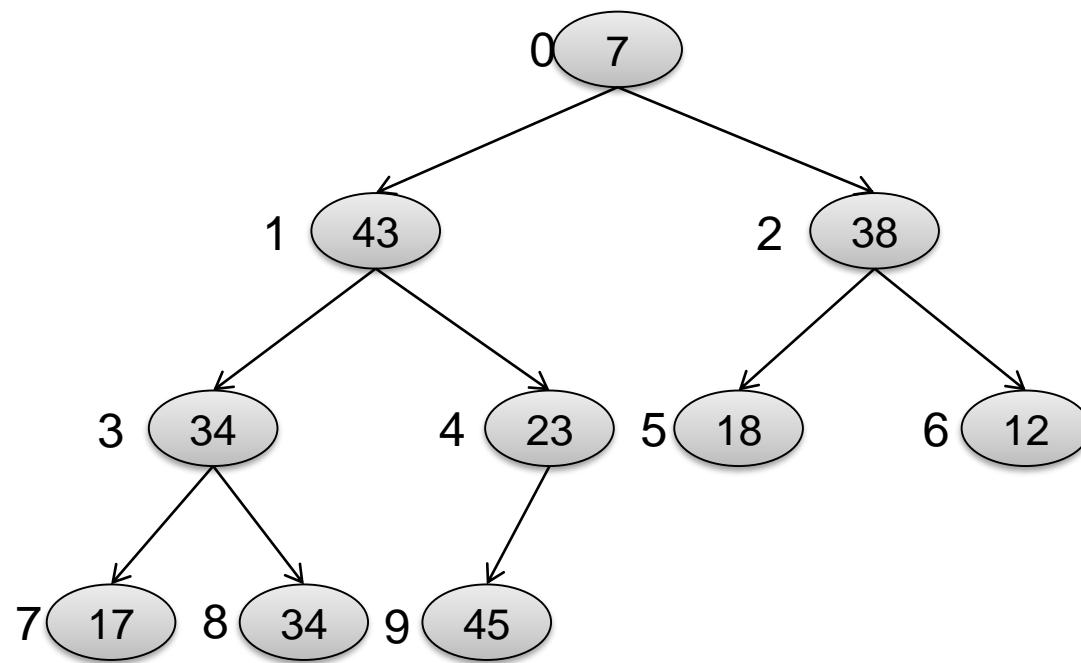


- Durchführung von BuildHeap ist beendet!

- Aktuelles Feld

7 43 38 34 23 18 12 17 34 45

- Darstellung als Heap



- Erstes und aktuell letztes Element werden vertauscht

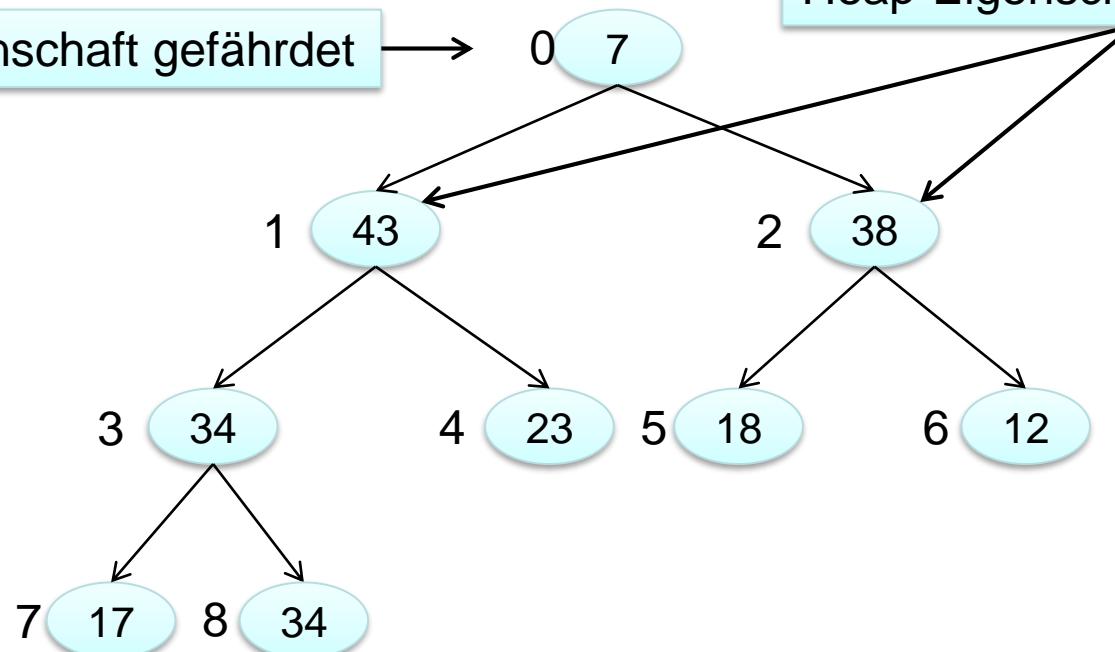
- Aktuelles Feld

7 43 38 34 23 18 12 17 34 45

- Darstellung als Heap

Heap-Eigenschaft gefährdet

Heap-Eigenschaft immer erfüllt



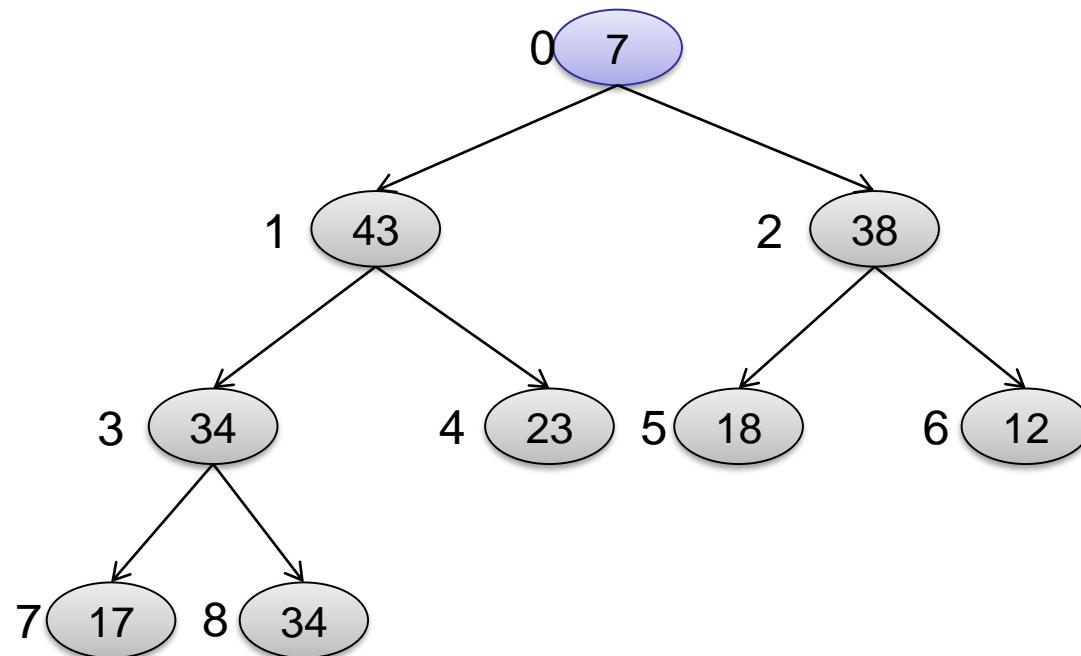
- Maximum ist an richtiger Stelle, Fortsetzung ohne das letzte Element

Beispiel Heap Sort

- Aktuelles Feld

7 43 38 34 23 18 12 17 34 45

- Darstellung als Heap

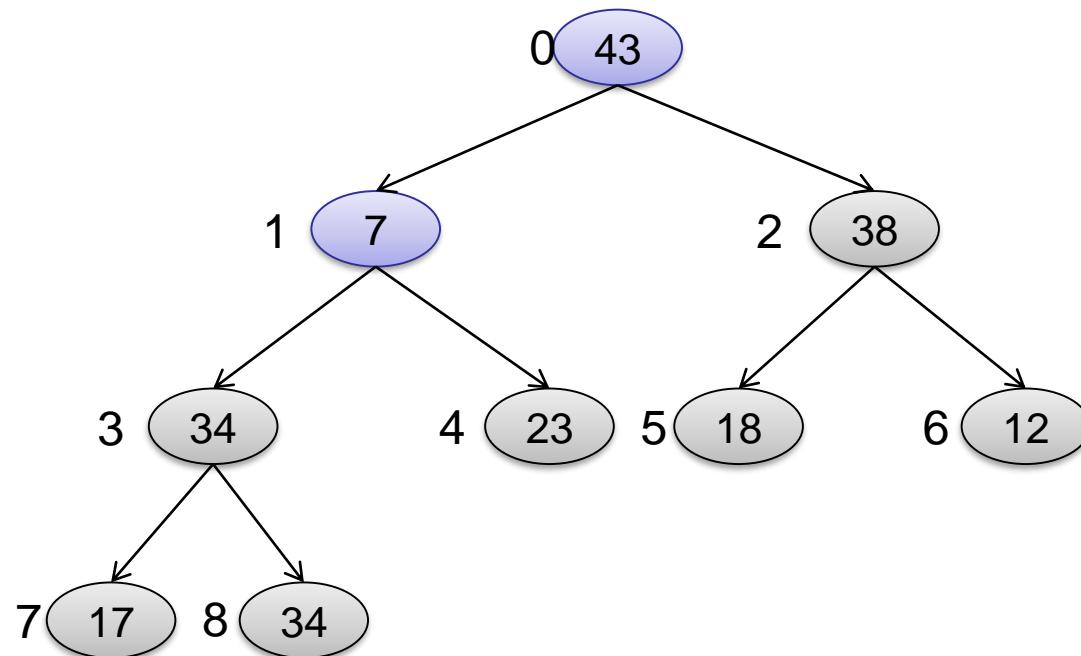


- Bearbeitung Knoten 0: Größtes Element ist der linke Nachfolger

- Aktuelles Feld

43 7 38 34 23 18 12 17 34 45

- Darstellung als Heap



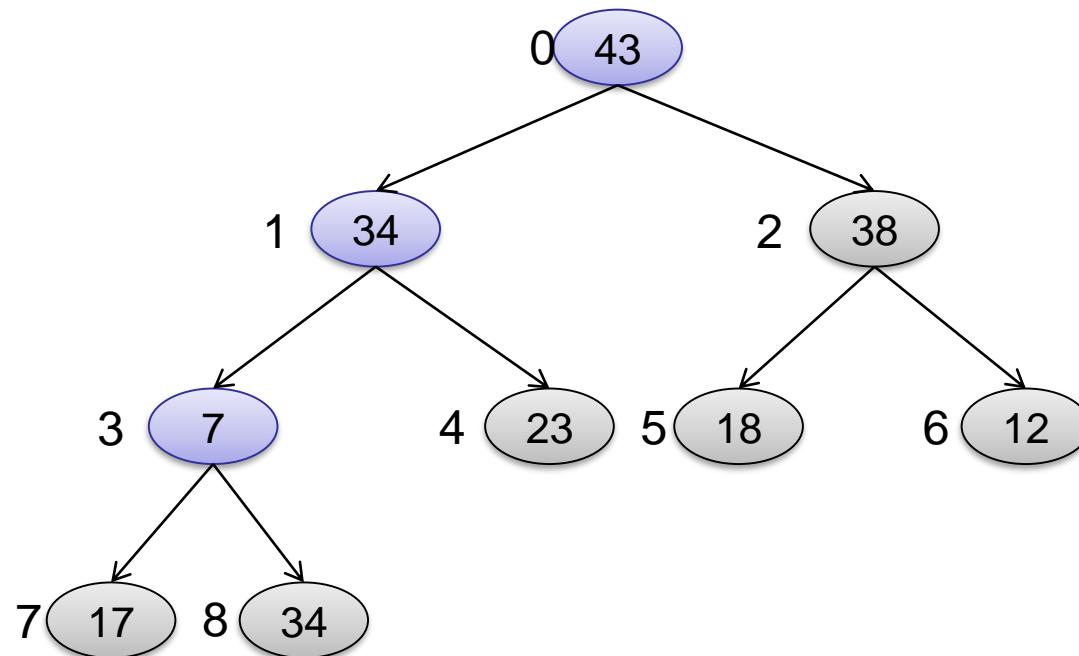
- Bearbeitung Knoten 1: Größtes Element ist der linke Nachfolger

Beispiel Heap Sort

- Aktuelles Feld

43 34 38 7 23 18 12 17 34 45

- Darstellung als Heap



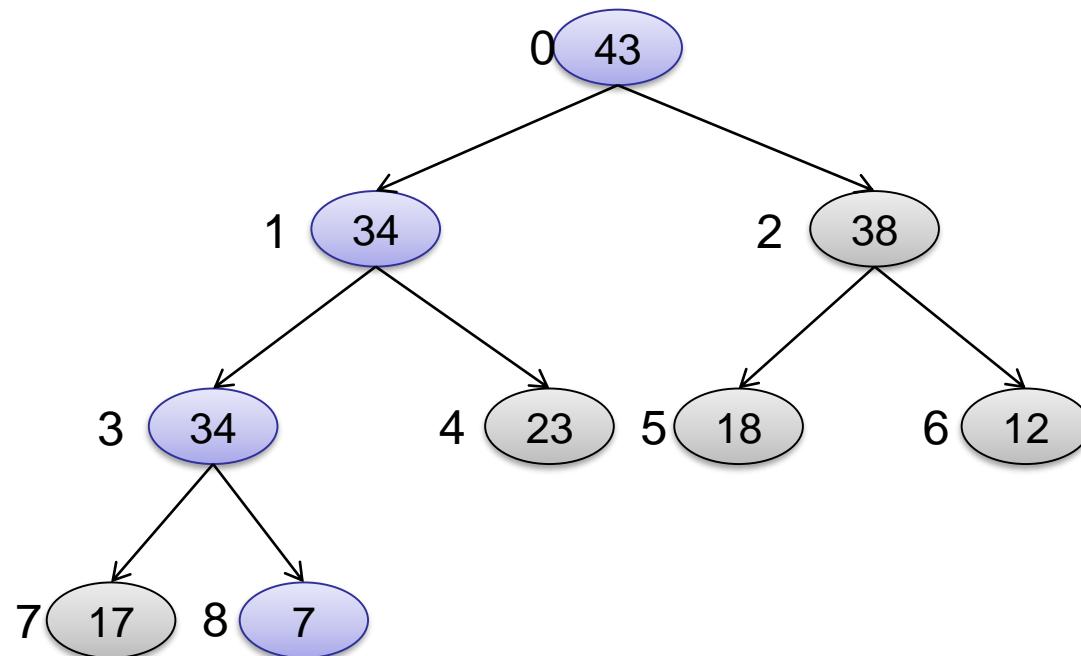
- Bearbeitung Knoten 3: Größtes Element ist der rechte Nachfolger

Beispiel Heap Sort

- Aktuelles Feld

43 34 38 34 23 18 12 17 7 45

- Darstellung als Heap



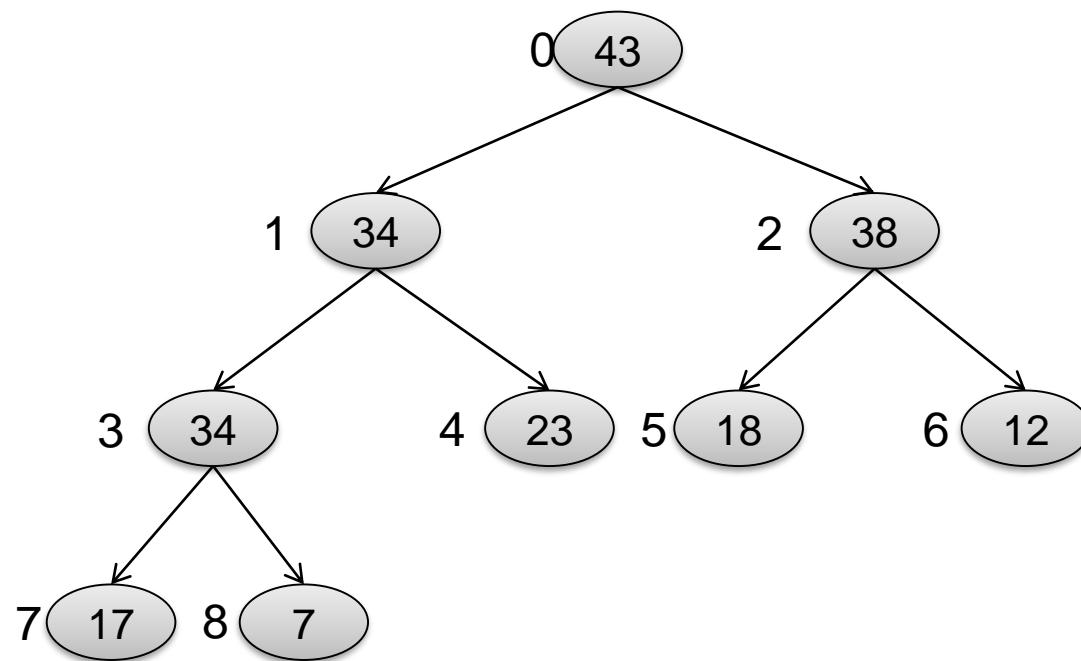
- Bearbeitung Knoten 7: Größtes Element ist die Wurzel

Beispiel Heap Sort

- Aktuelles Feld

43 34 38 34 23 18 12 17 7 45

- Darstellung als Heap



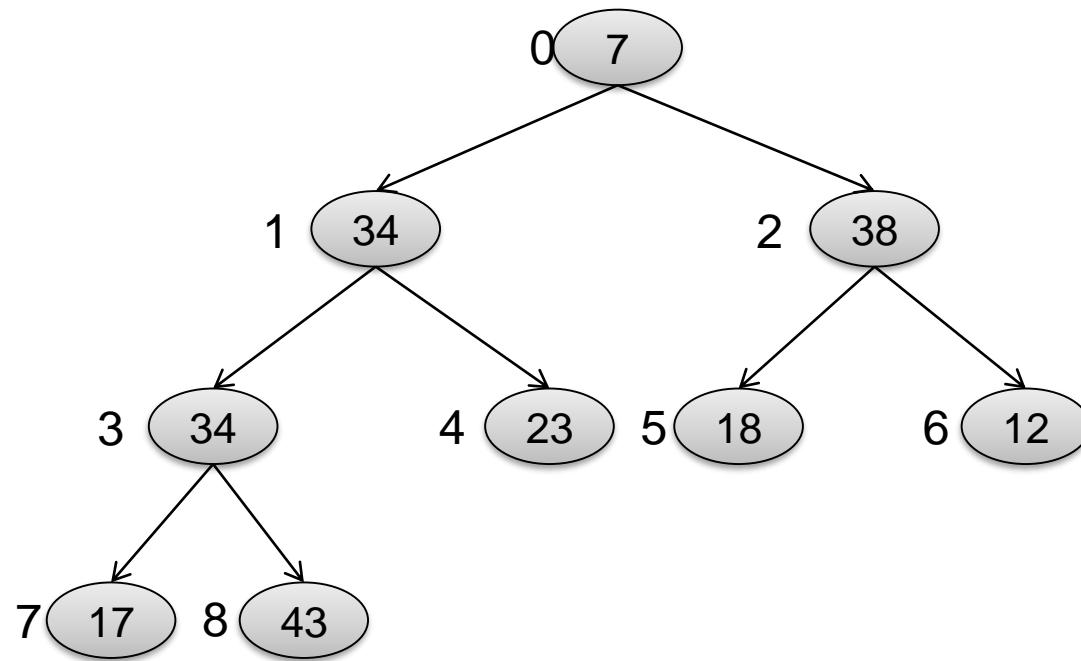
- Nächster Wert ist identifiziert

Beispiel Heap Sort

- Aktuelles Feld

7 34 38 34 23 18 12 17 43 45

- Darstellung als Heap



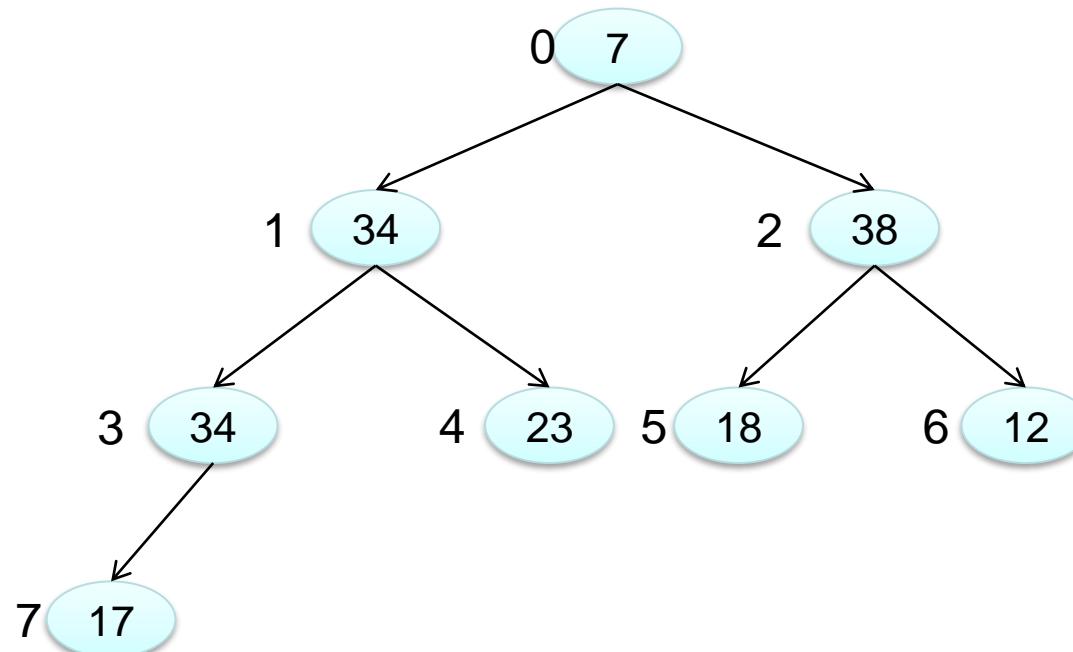
- Erstes und aktuell letztes Element werden vertauscht

Beispiel Heap Sort

- Aktuelles Feld

7 34 38 34 23 18 12 17 43 45

- Darstellung als Heap



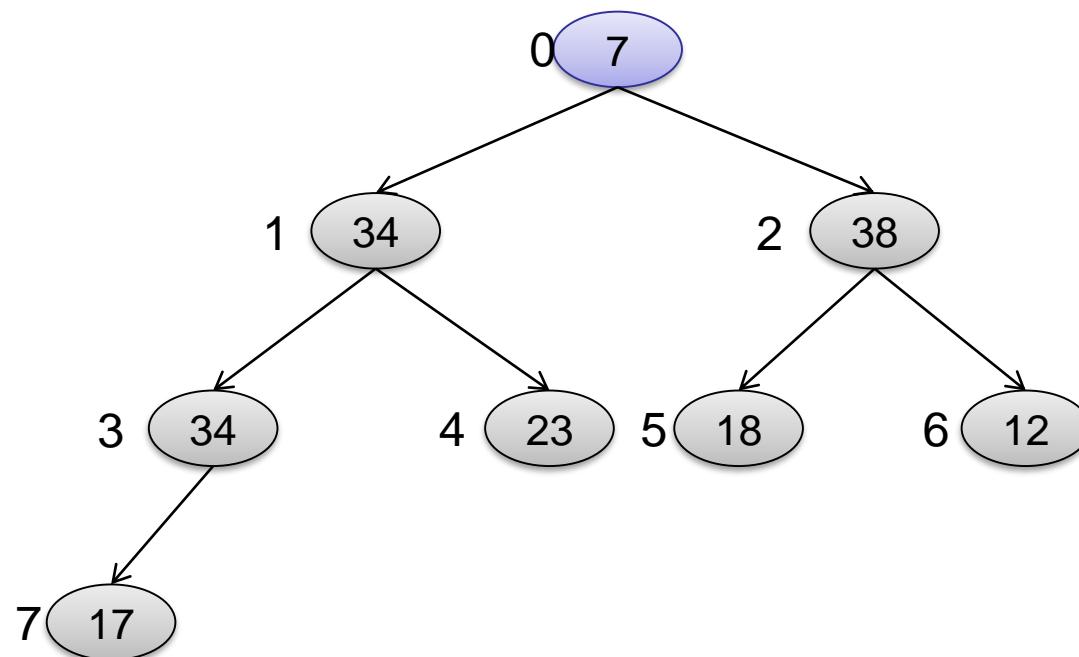
- Maximum ist an richtiger Stelle, Fortsetzung ohne das letzte Element

Beispiel Heap Sort

- Aktuelles Feld

7 34 38 34 23 18 12 17 43 45

- Darstellung als Heap



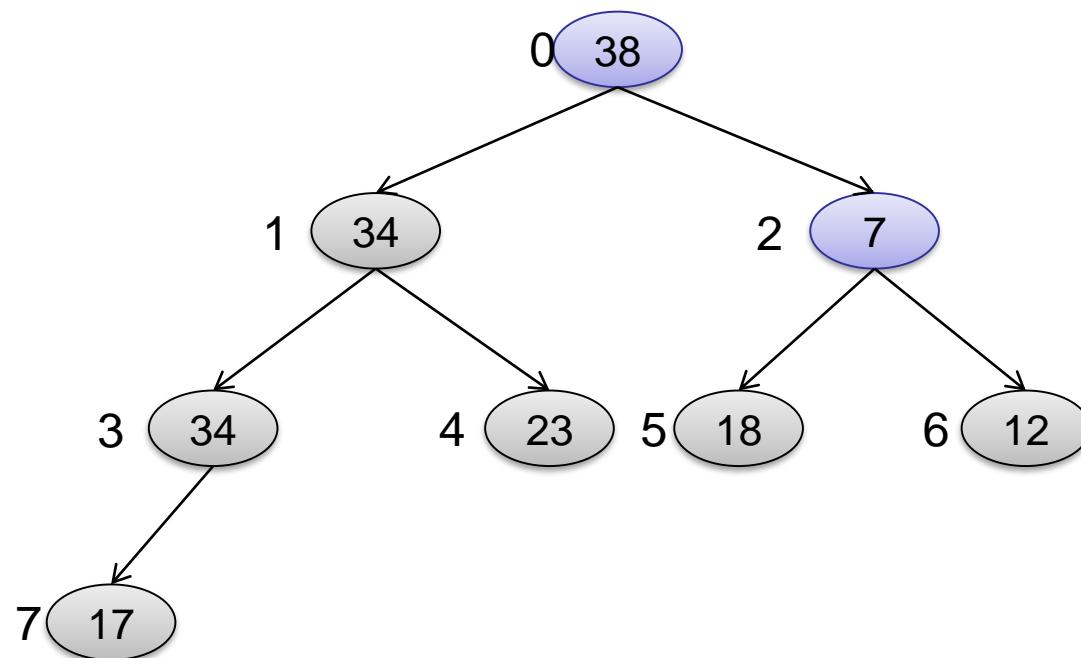
- Bearbeitung Knoten 0: Größtes Element ist der rechte Nachfolger

Beispiel Heap Sort

- Aktuelles Feld

38 34 7 34 23 18 12 17 43 45

- Darstellung als Heap



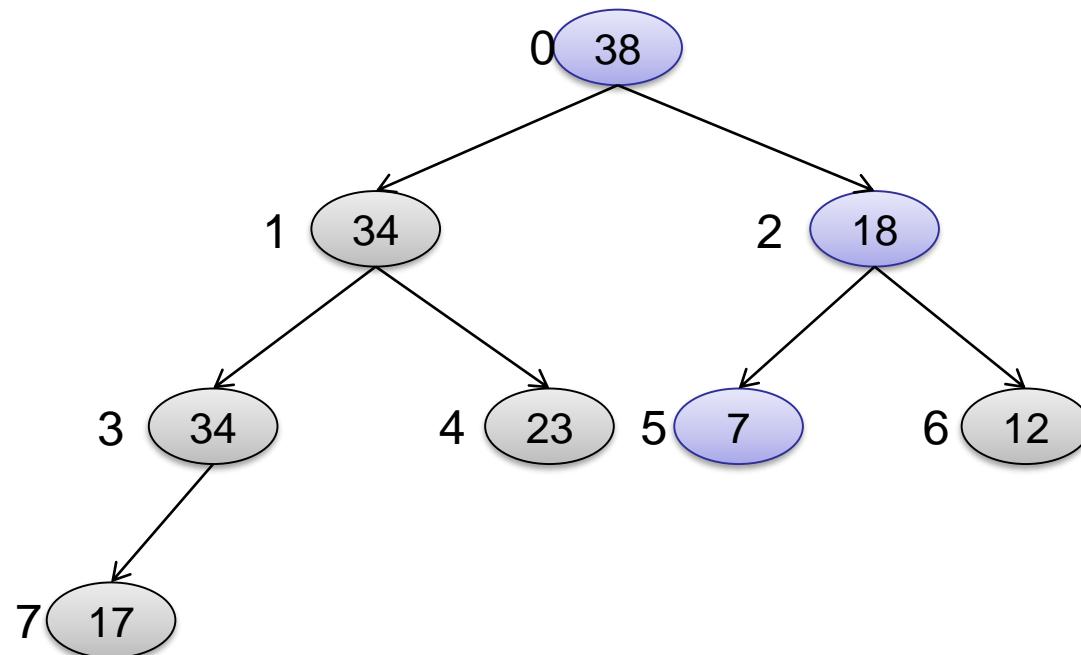
- Bearbeitung Knoten 2: Größtes Element ist der linke Nachfolger

Beispiel Heap Sort

- Aktuelles Feld

38 34 18 34 23 7 12 17 43 45

- Darstellung als Heap



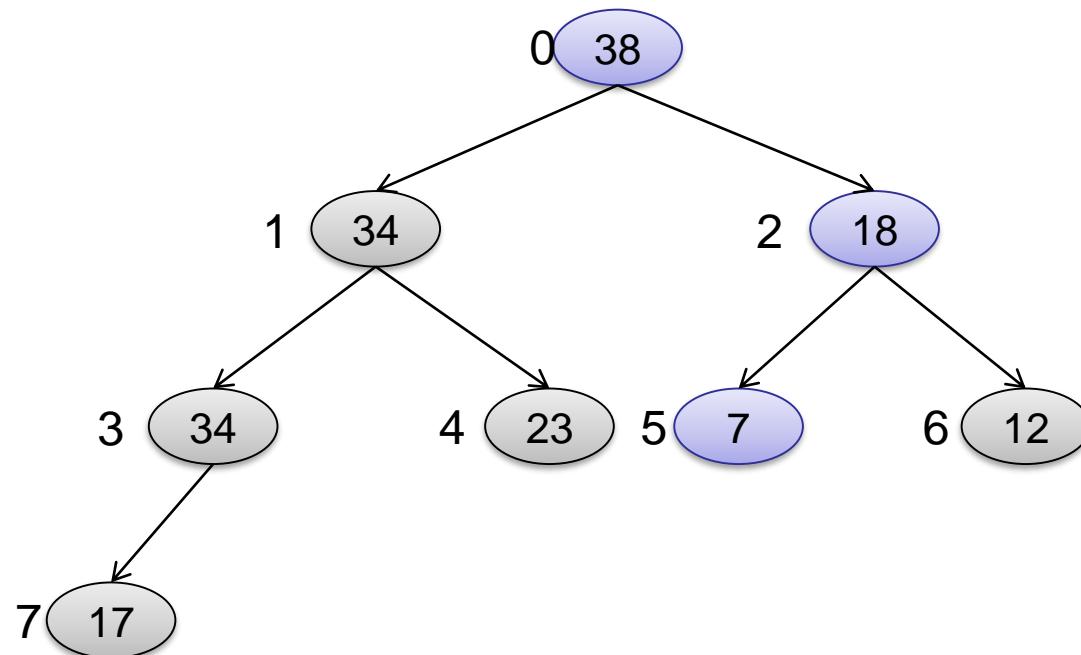
- Bearbeitung Knoten 2: Größtes Element ist der linke Nachfolger

Beispiel Heap Sort

- Aktuelles Feld

38 34 18 34 23 7 12 17 43 45

- Darstellung als Heap



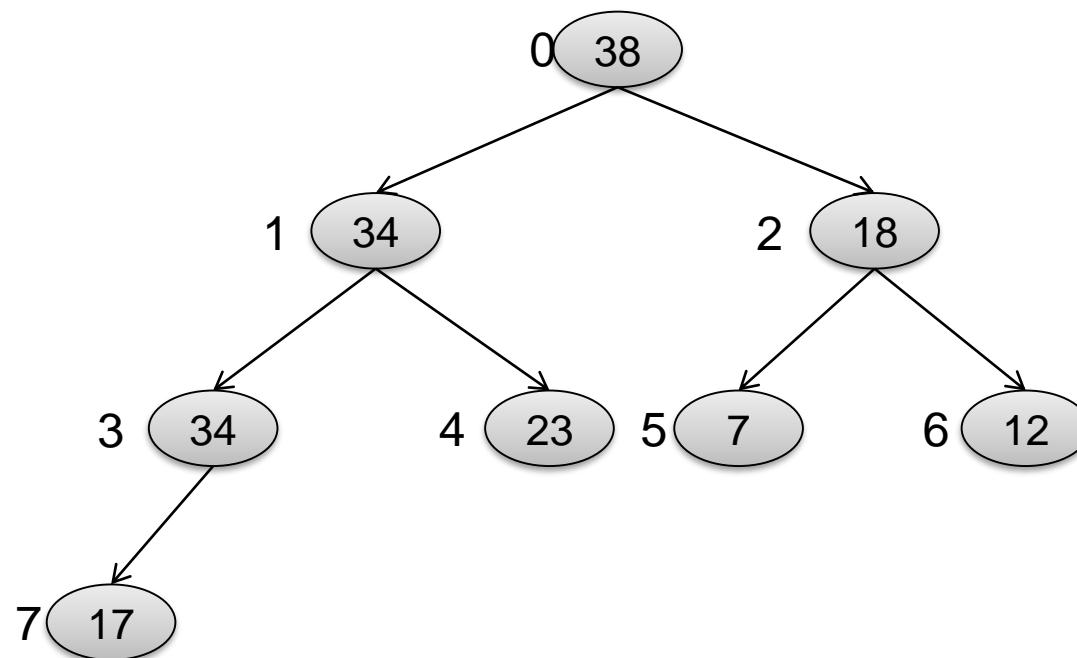
- Bearbeitung Knoten 5: Größtes Element ist die Wurzel

Beispiel Heap Sort

- Aktuelles Feld

38 34 18 34 23 7 12 17 43 45

- Darstellung als Heap



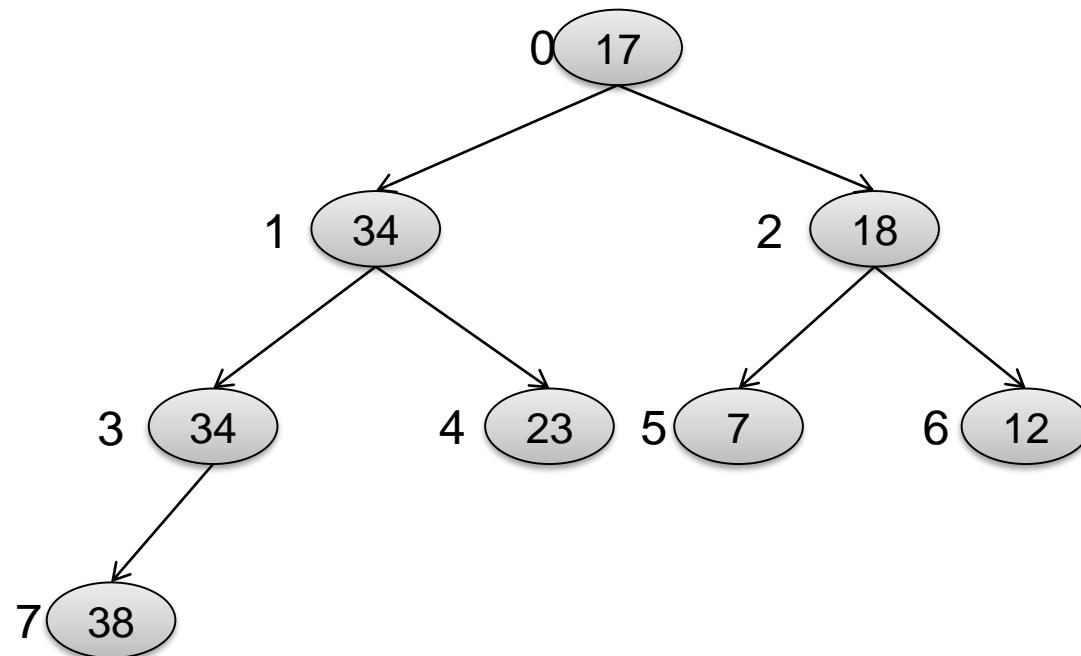
- Nächster Wert ist identifiziert

Beispiel Heap Sort

- Aktuelles Feld

17 34 18 34 23 7 12 38 43 45

- Darstellung als Heap



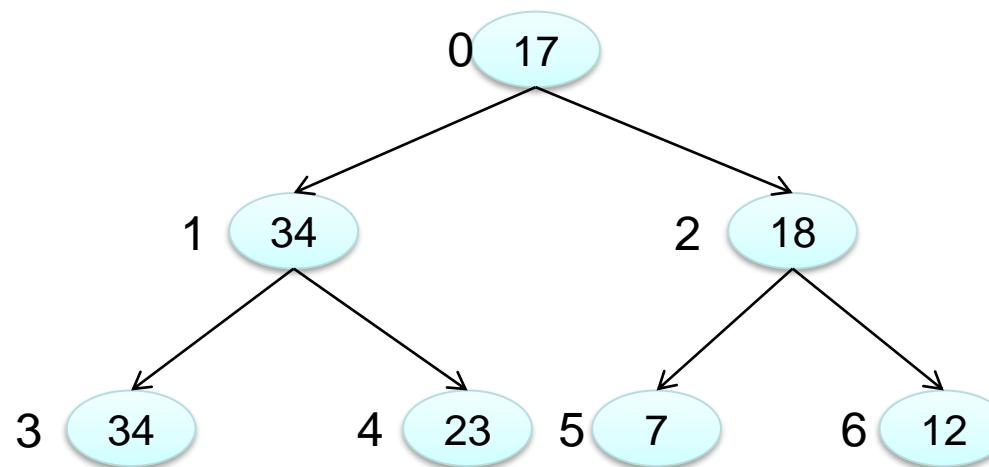
- Erstes und aktuell letztes Element werden vertauscht

Beispiel Heap Sort

- Aktuelles Feld

17 34 18 34 23 7 12 38 43 45

- Darstellung als Heap



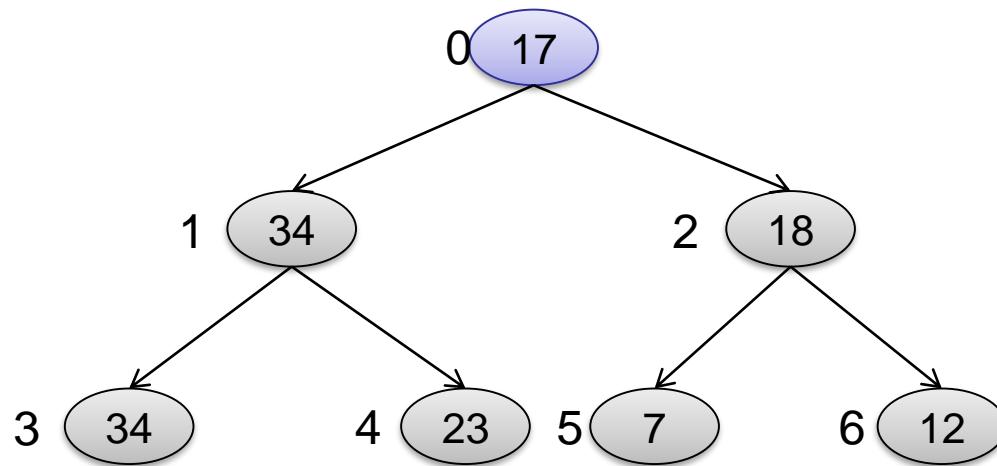
- Maximum ist an richtiger Stelle, Fortsetzung ohne das letzte Element

Beispiel Heap Sort

- Aktuelles Feld

17 34 18 34 23 7 12 38 43 45

- Darstellung als Heap



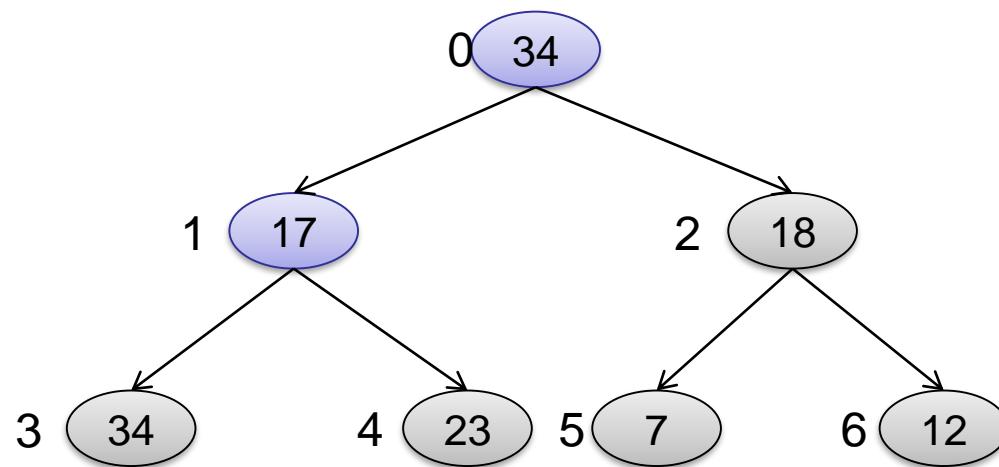
- Bearbeitung Knoten 0: Größtes Element ist der linke Nachfolger

Beispiel Heap Sort

- Aktuelles Feld

34 17 18 34 23 7 12 38 43 45

- Darstellung als Heap



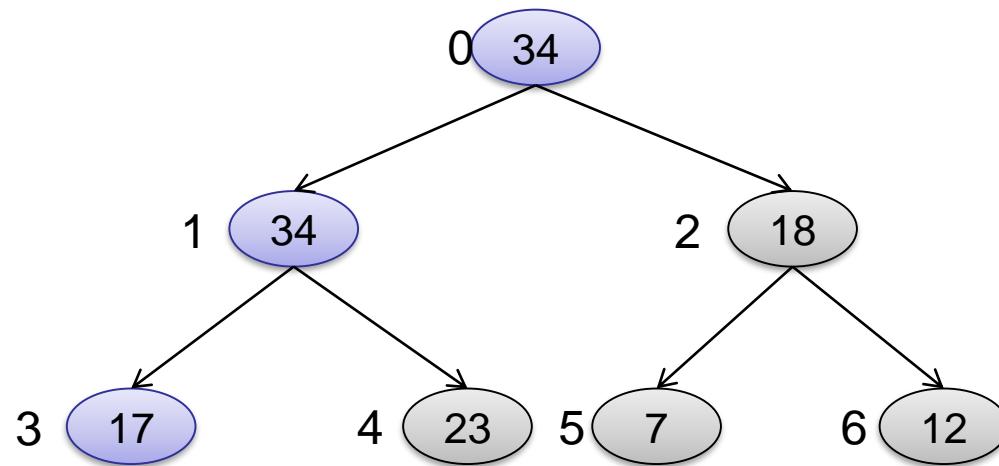
- Bearbeitung Knoten 1: Größtes Element ist der linke Nachfolger

Beispiel Heap Sort

- Aktuelles Feld

34 34 18 17 23 7 12 38 43 45

- Darstellung als Heap



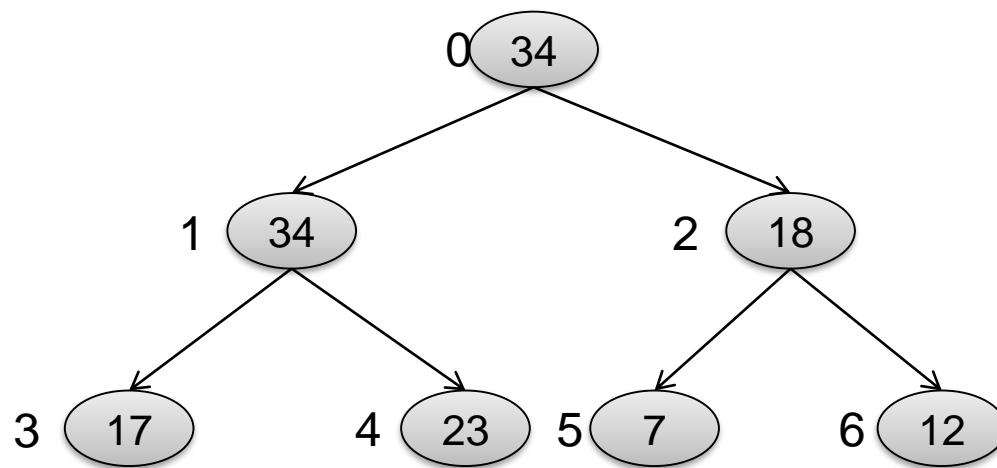
- Bearbeitung Knoten 3: Größtes Element ist die Wurzel

Beispiel Heap Sort

- Aktuelles Feld

34 34 18 17 23 7 12 38 43 45

- Darstellung als Heap

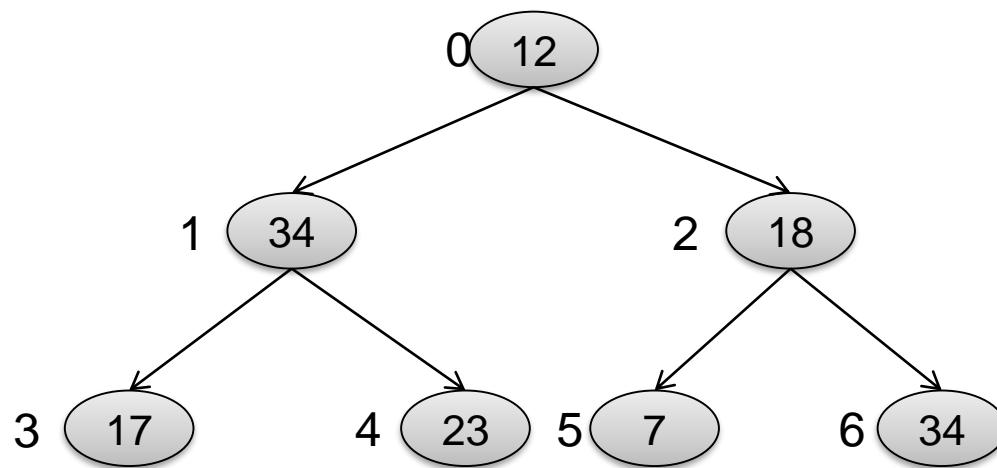


- Nächster Wert ist identifiziert

- Aktuelles Feld

12 34 18 17 23 7 34 38 43 45

- Darstellung als Heap



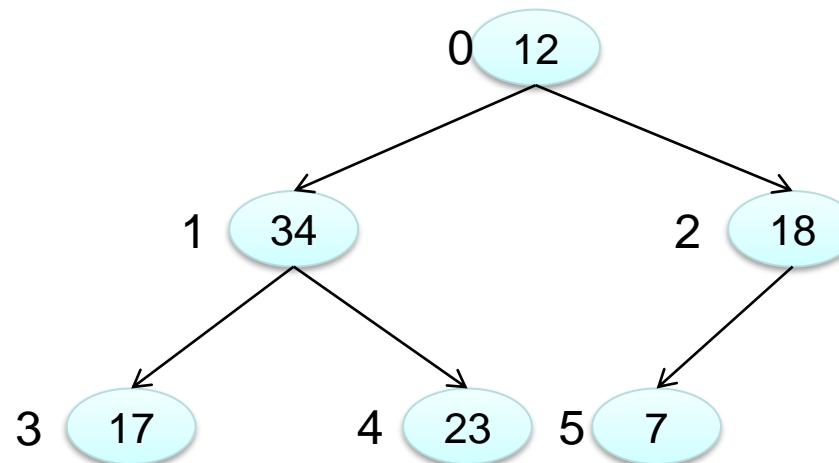
- Erstes und aktuell letztes Element werden vertauscht

Beispiel Heap Sort

- Aktuelles Feld

12 34 18 17 23 7 34 38 43 45

- Darstellung als Heap



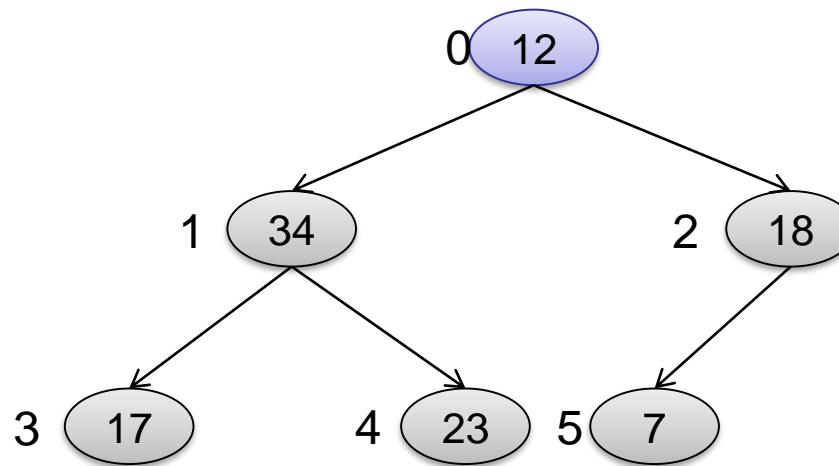
- Maximum ist an richtiger Stelle, Fortsetzung ohne das letzte Element

Beispiel Heap Sort

- Aktuelles Feld

12 34 18 17 23 7 34 38 43 45

- Darstellung als Heap

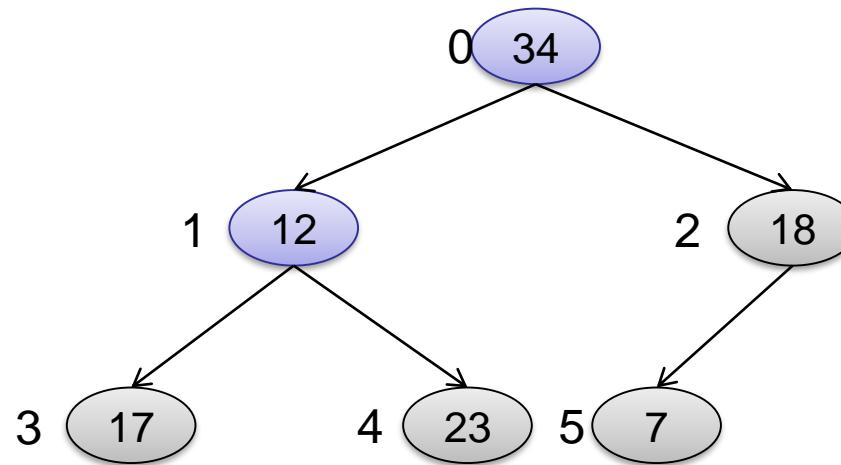


- Bearbeitung Knoten 0: Größtes Element ist der linke Nachfolger

- Aktuelles Feld

34 12 18 17 23 7 34 38 43 45

- Darstellung als Heap

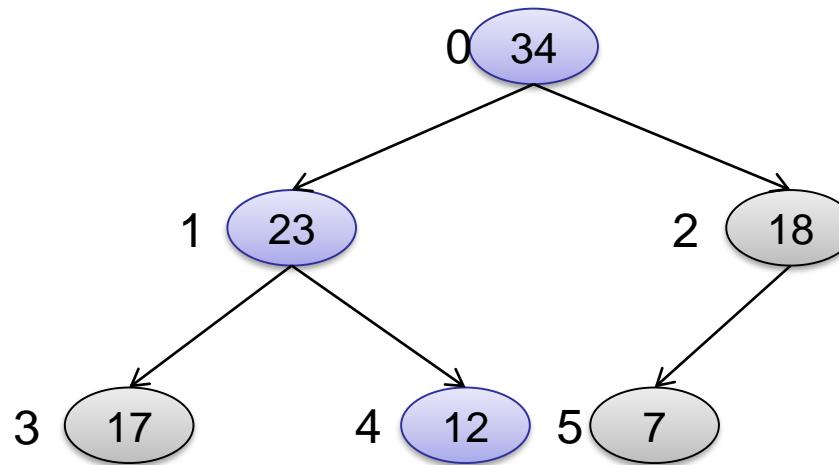


- Bearbeitung Knoten 1: Größtes Element ist der rechte Nachfolger

- Aktuelles Feld

34 23 18 17 12 7 34 38 43 45

- Darstellung als Heap

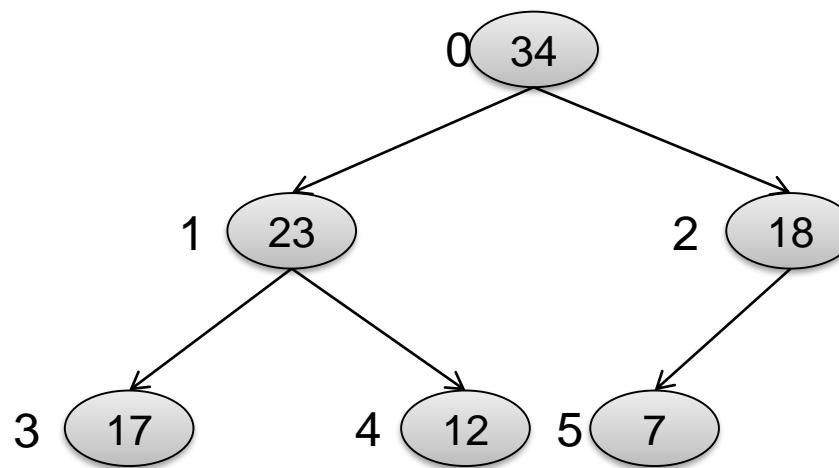


- Bearbeitung Knoten 4: Größtes Element ist die Wurzel

- Aktuelles Feld

34 23 18 17 12 7 34 38 43 45

- Darstellung als Heap

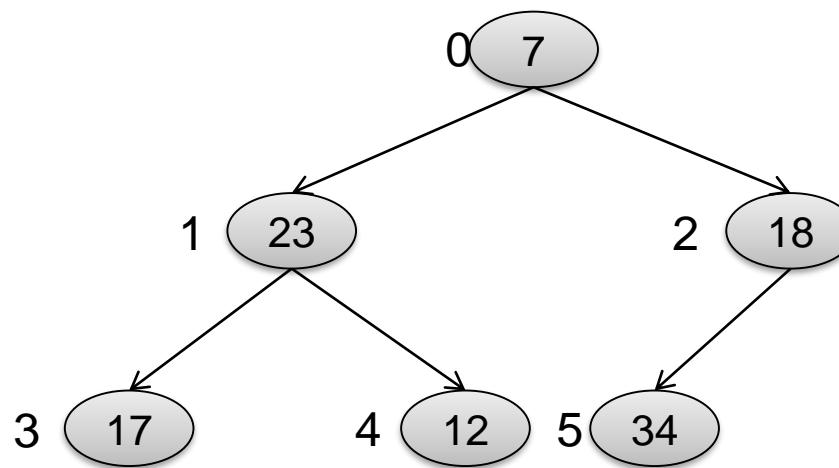


- Nächster Wert ist identifiziert

- Aktuelles Feld

7 23 18 17 12 34 34 38 43 45

- Darstellung als Heap



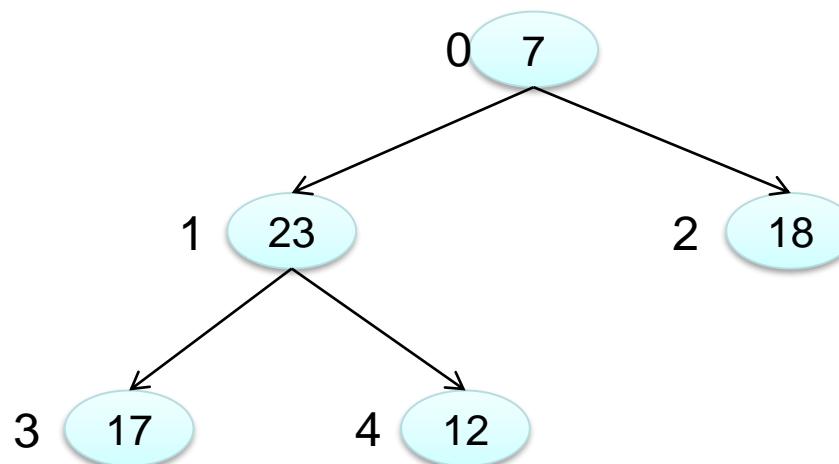
- Erstes und aktuell letztes Element werden vertauscht

Beispiel Heap Sort

- Aktuelles Feld

7 23 18 17 12 34 34 38 43 45

- Darstellung als Heap

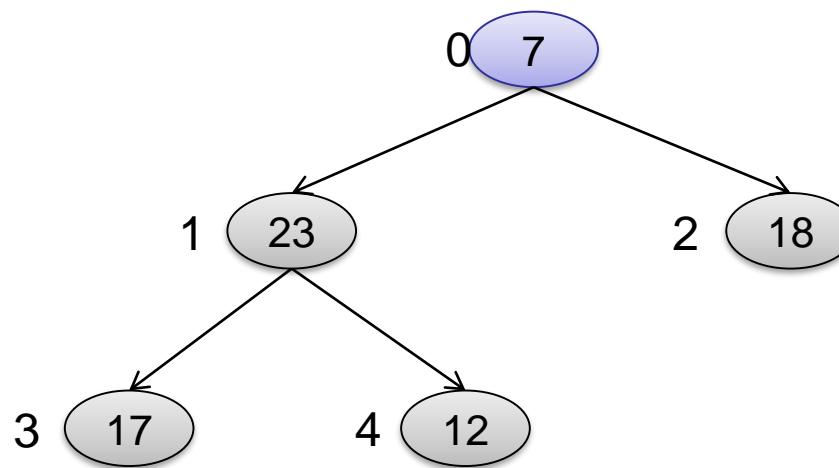


- Maximum ist an richtiger Stelle, Fortsetzung ohne das letzte Element

- Aktuelles Feld

7 23 18 17 12 34 34 38 43 45

- Darstellung als Heap

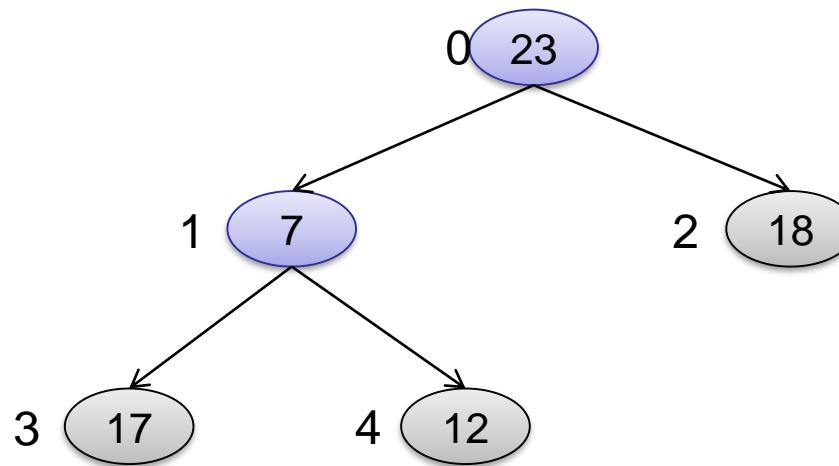


- Bearbeitung Knoten 0: Größtes Element ist der linke Nachfolger

- Aktuelles Feld

23 7 18 17 12 34 34 38 43 45

- Darstellung als Heap



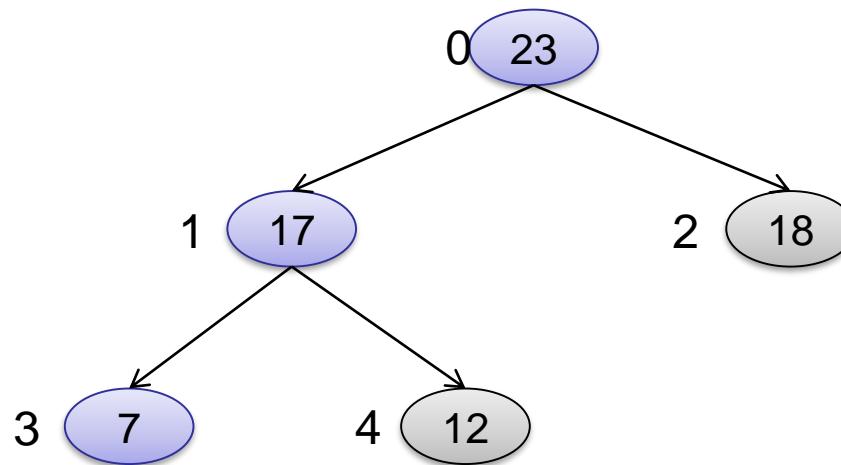
- Bearbeitung Knoten 1: Größtes Element ist der linke Nachfolger

Beispiel Heap Sort

- Aktuelles Feld

23 17 18 7 12 34 34 38 43 45

- Darstellung als Heap

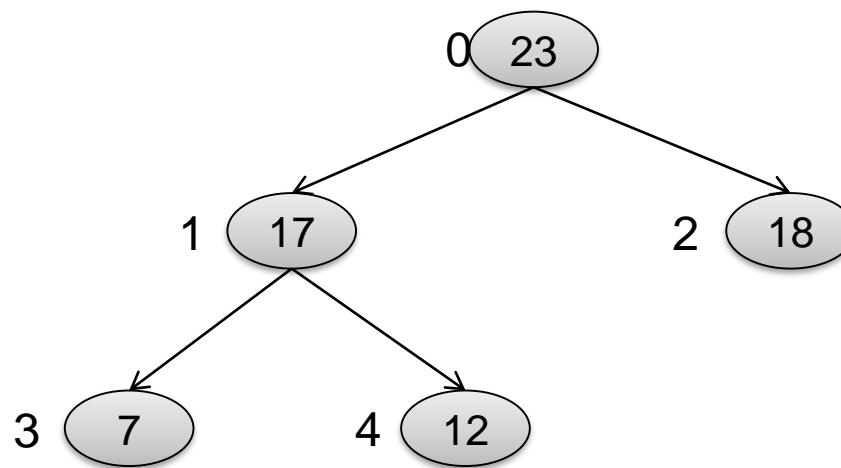


- Bearbeitung Knoten 3: Größtes Element ist die Wurzel

- Aktuelles Feld

23 17 18 7 12 34 34 38 43 45

- Darstellung als Heap

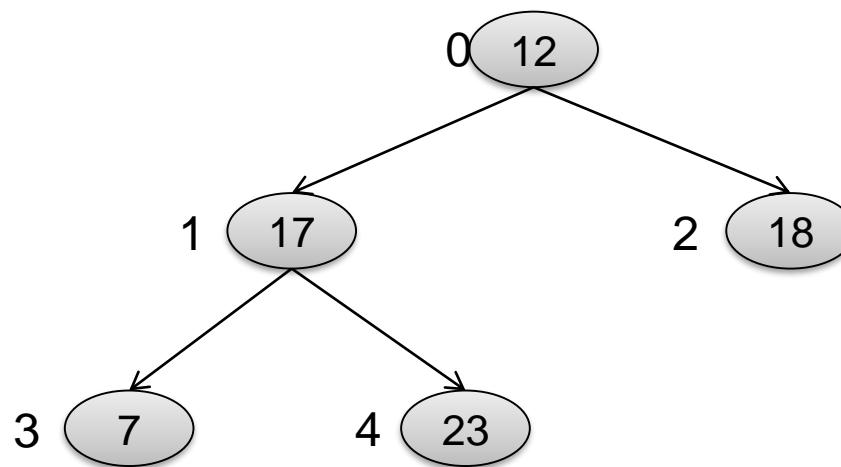


- Nächster Wert ist identifiziert

- Aktuelles Feld

12 17 18 7 23 34 34 38 43 45

- Darstellung als Heap



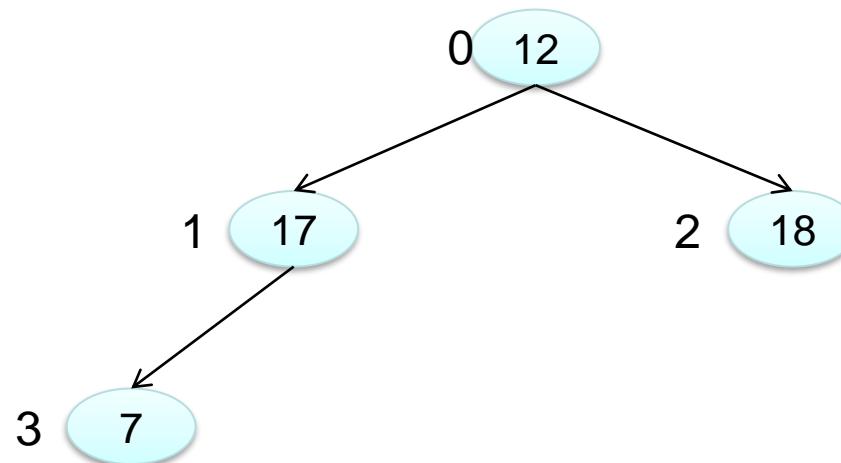
- Erstes und aktuell letztes Element werden vertauscht

Beispiel Heap Sort

- Aktuelles Feld

12 17 18 7 23 34 34 38 43 45

- Darstellung als Heap

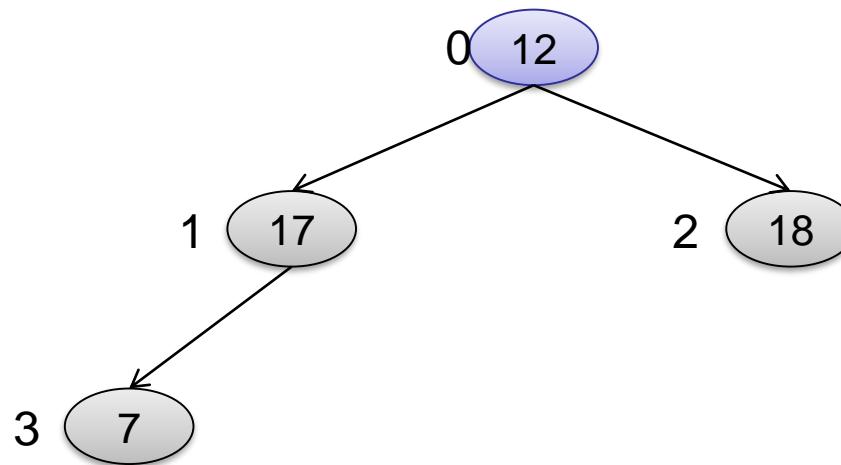


- Maximum ist an richtiger Stelle, Fortsetzung ohne das letzte Element

- Aktuelles Feld

12 17 18 7 23 34 34 38 43 45

- Darstellung als Heap

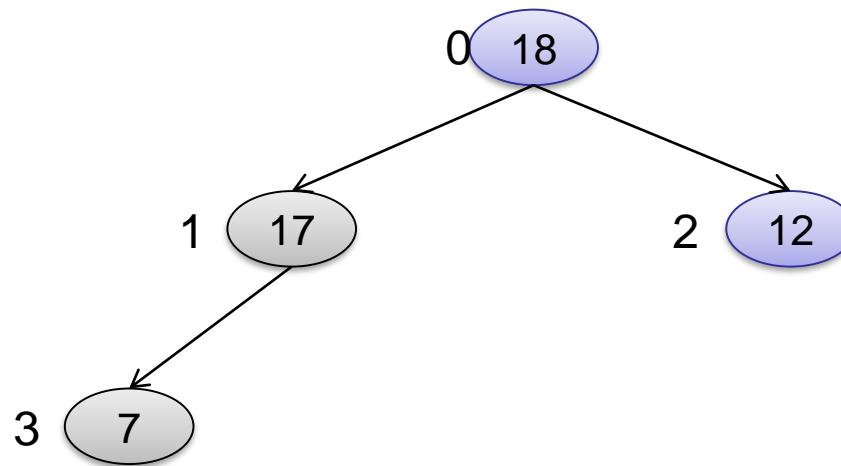


- Bearbeitung Knoten 0: Größtes Element ist der rechte Nachfolger

- Aktuelles Feld

18 17 12 7 23 34 34 38 43 45

- Darstellung als Heap

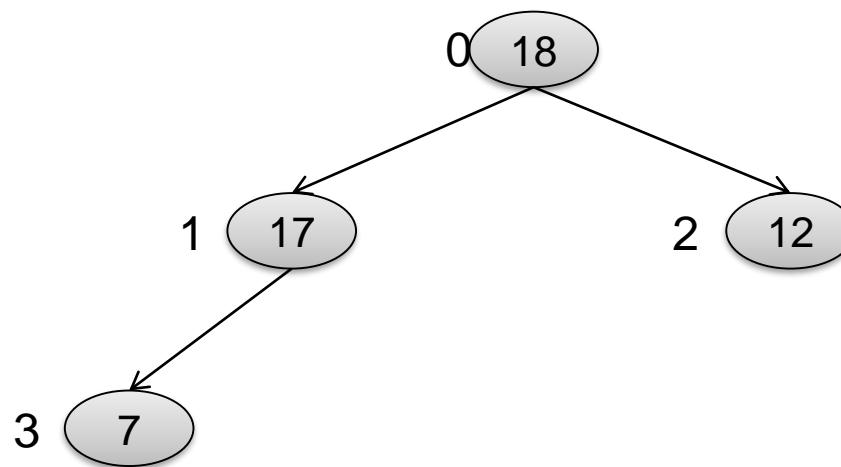


- Bearbeitung Knoten 2: Größtes Element ist die Wurzel

- Aktuelles Feld

18 17 12 7 23 34 34 38 43 45

- Darstellung als Heap

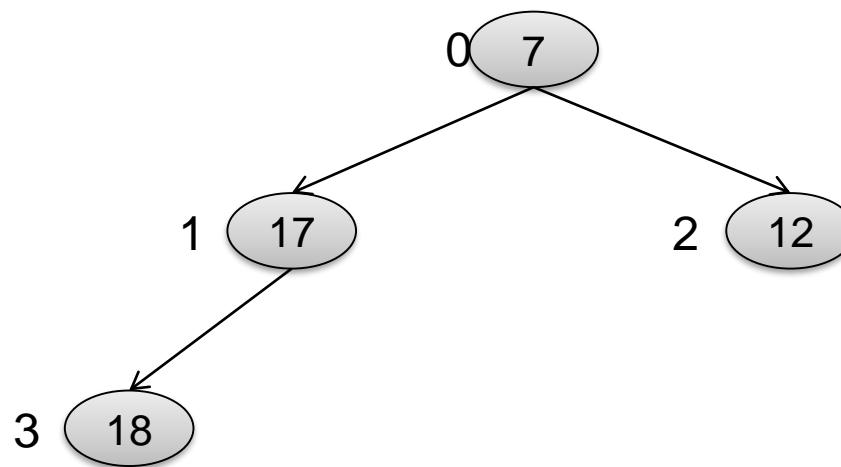


- Nächster Wert ist identifiziert

- Aktuelles Feld

7 17 12 18 23 34 34 38 43 45

- Darstellung als Heap



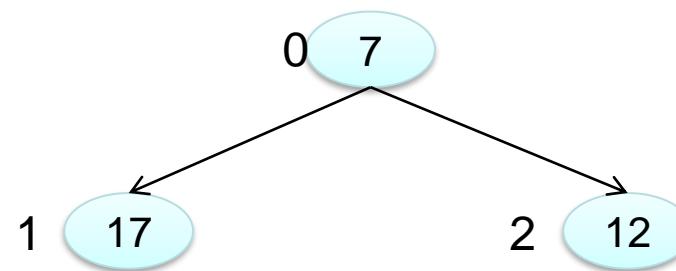
- Erstes und aktuell letztes Element werden vertauscht

Beispiel Heap Sort

- Aktuelles Feld

7 17 12 18 23 34 34 38 43 45

- Darstellung als Heap



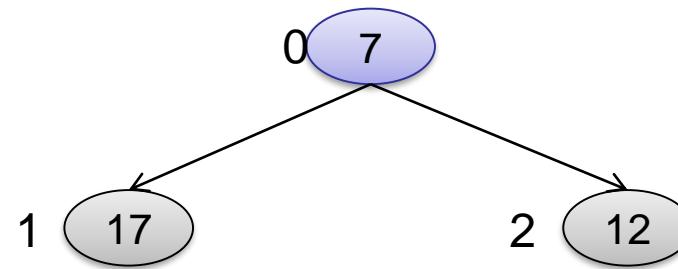
- Maximum ist an richtiger Stelle, Fortsetzung ohne das letzte Element

Beispiel Heap Sort

- Aktuelles Feld

7 17 12 18 23 34 34 38 43 45

- Darstellung als Heap

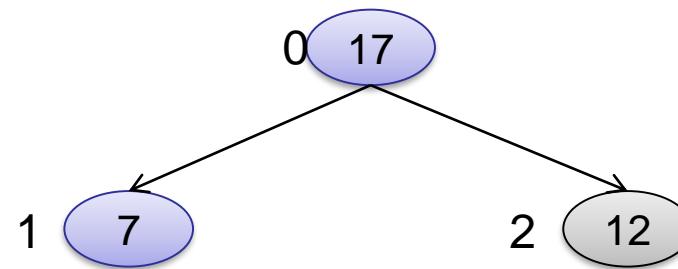


- Bearbeitung Knoten 0: Größtes Element ist der linke Nachfolger

- Aktuelles Feld

17 7 12 18 23 34 34 38 43 45

- Darstellung als Heap



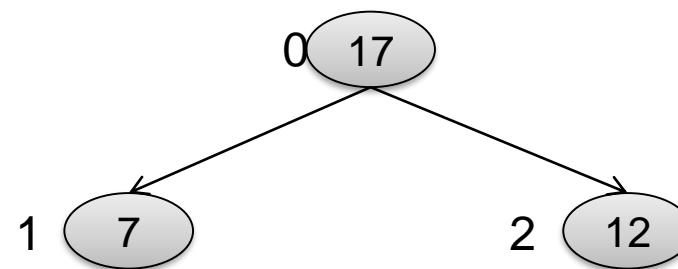
- Bearbeitung Knoten 1: Größtes Element ist die Wurzel

Beispiel Heap Sort

- Aktuelles Feld

17 7 12 18 23 34 34 38 43 45

- Darstellung als Heap

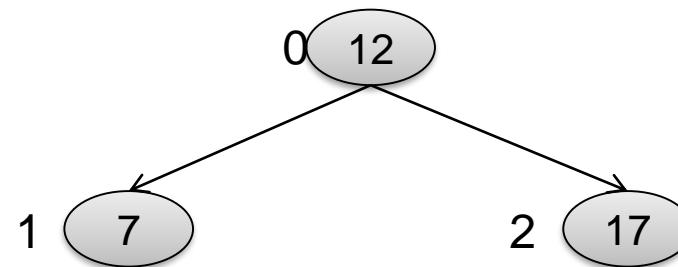


- Nächster Wert ist identifiziert

- Aktuelles Feld

12 7 17 18 23 34 34 38 43 45

- Darstellung als Heap



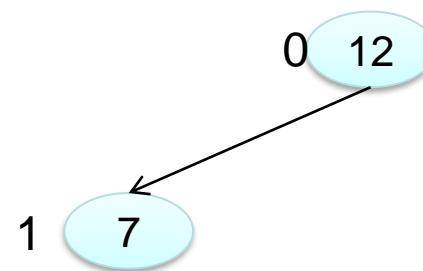
- Erstes und aktuell letztes Element werden vertauscht

Beispiel Heap Sort

- Aktuelles Feld

12 7 17 18 23 34 34 38 43 45

- Darstellung als Heap

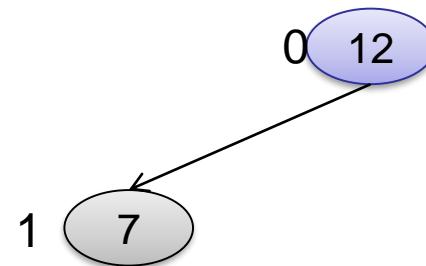


- Maximum ist an richtiger Stelle, Fortsetzung ohne das letzte Element

- Aktuelles Feld

12 7 17 18 23 34 34 38 43 45

- Darstellung als Heap

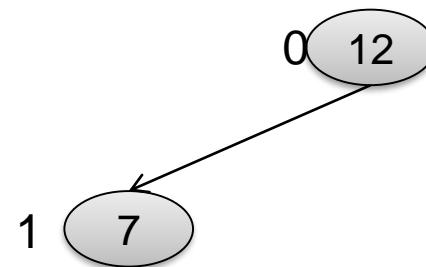


- Bearbeitung Knoten 0: Größtes Element ist die Wurzel

- Aktuelles Feld

12 7 17 18 23 34 34 38 43 45

- Darstellung als Heap

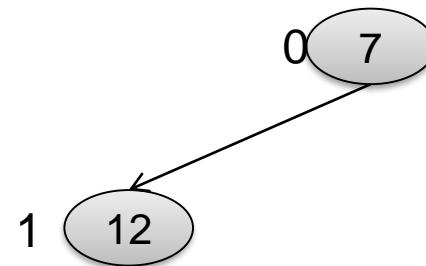


- Nächster Wert ist identifiziert

- Aktuelles Feld

7 12 17 18 23 34 34 38 43 45

- Darstellung als Heap



- Erstes und aktuell letztes Element werden vertauscht

Beispiel Heap Sort

- Aktuelles Feld

7 12 17 18 23 34 34 38 43 45

- Darstellung als Heap



- Maximum ist an richtiger Stelle, Fortsetzung ohne das letzte Element

Beispiel Heap Sort

- Aktuelles Feld

7 12 17 18 23 34 34 38 43 45

- Darstellung als Heap



- Bearbeitung Knoten 0: Größtes Element ist die Wurzel

Beispiel Heap Sort

- Aktuelles Feld

7 12 17 18 23 34 34 38 43 45

- Darstellung als Heap



- Nächster Wert ist identifiziert

- Aktuelles Feld

7 12 17 18 23 34 34 38 43 45

- Darstellung als Heap



- Erstes und aktuell letztes Element werden vertauscht

Beispiel Heap Sort

- Aktuelles Feld

7	12	17	18	23	34	34	38	43	45
---	----	----	----	----	----	----	----	----	----

- Darstellung als Heap

Kein weiteres Element verfügbar!

- Maximum ist an richtiger Stelle, HeapSort ist beendet

- Welche der folgenden Felder stellen einen Heap dar:
 - $a[] = \{19, 5, 7, 1, 2, 6, 0\}$
 - $b[] = \{12, 5, 7, 1, 2, 6, 0\}$
 - $c[] = \{15, 5, 7, 1, 2, 6, 8\}$
 - $d[] = \{23, 5, 7, 1, 2, 7, 2\}$
- Sortieren Sie das Feld $e[] = \{2, 4, 9, 18, 21, 37\}$ mittels HeapSort!