Theoretische Informatik II

Einheit 5

Komplexitätstheorie

1. Konkrete Komplexitätsanalyse
2. Das $\mathcal{P}$–$\mathcal{NP}$ Problem
3. NP-vollständige Probleme
4. Grenzen überwinden

Komplexitätstheorie

Was kann mit vertretbarem Aufwand gelöst werden?

- **Berechenbarkeitsanalyse alleine reicht nicht**
  - Klärt nur die Grundsatzfrage: berechenbar/entscheidbar oder nicht
  - Für praktisch Lösbarkeit muß Berechnungsaufwand vertretbar sein

Komplexitätstheorie

Was kann mit vertretbarem Aufwand gelöst werden?

- **Berechenbarkeitsanalyse alleine reicht nicht**
  - Klärt nur die Grundsatzfrage: berechenbar/entscheidbar oder nicht
  - Für praktisch Lösbarkeit muß Berechnungsaufwand vertretbar sein
- **Analyse benötigter Ressourcen (Komplexität)**
  - Zeitbedarf des Algorithmus — Time
  - Speicherbedarf des Verfahrens (RAM, Harddisk) — Space
  - Netzzugriffe, Zugriff auf andere Medien, ...

Komplexitätstheorie

Was kann mit vertretbarem Aufwand gelöst werden?

- **Berechenbarkeitsanalyse alleine reicht nicht**
  - Klärt nur die Grundsatzfrage: berechenbar/entscheidbar oder nicht
  - Für praktisch Lösbarkeit muß Berechnungsaufwand vertretbar sein
- **Analyse benötigter Ressourcen (Komplexität)**
  - Zeitbedarf des Algorithmus — Time
  - Speicherbedarf des Verfahrens (RAM, Harddisk) — Space
  - Netzzugriffe, Zugriff auf andere Medien, ...
- **Die Meßgröße muß objektive sein**
  - Unabhängig von konkreter Hardware und Programmiersprache
    Optimierungsfähigkeiten des Compilers
    Auswahl der Testdaten

Komplexitätsmaße sollten abstrakt formuliert sein

Grössenordnung ist wichtiger als Detail

- **Genaue Betrachtungen sind unpraktikabel**
  - Mühsam bei nichttrivialen Algorithmen
  - Abhängig von Programmierdetails und Maschinenmodell

  Welches Maschinenmodell sollte der Standard sein?

Grössenordnung ist wichtiger als Detail

- **Genaue Betrachtungen sind unpraktikabel**
  – Mühsam bei nichttrivialen Algorithmen
  – Abhängig von Programmierdetails und Maschinenmodell

  Welches Maschinenmodell sollte der Standard sein?

- **Abschätzung der Komplexität ist sinnvoller**
  – Asymptotisches Verhalten für große Eingabedaten ist wichtig

Größenordnung ist wichtiger als Detail

- **Genaue Betrachtungen sind unpraktikabel**
  - Mühsam bei nichttrivialen Algorithmen
  - Abhängig von Programmierdetails und Maschinenmodell

  Welches Maschinenmodell sollte der Standard sein?

- **Abschätzung der Komplexität ist sinnvoller**
  - Asymptotisches Verhalten für große Eingabedaten ist wichtig

- **Verwende ein Einheitskostenmodell**
  - Vereinfachte (modellunabhängige) Zählung von Elementaroperationen

Grössenordnung ist wichtiger als Detail

- **Genaue Betrachtungen sind unpraktikabel**
  - Mühsam bei nichttrivialen Algorithmen
  - Abhängig von Programmierdetails und Maschinenmodell

  Welches Maschinenmodell sollte der Standard sein?

- **Abschätzung der Komplexität ist sinnvoller**
  - Asymptotisches Verhalten für große Eingabedaten ist wichtig

- **Verwende ein Einheitskostenmodell**
  - Vereinfachte (modellunabhängige) Zählung von Elementaroperationen

- **Ignoriere Konstanten**
  - Additive Konstanten und konstante Faktoren werden durch Hardwaresteigerungen ausgeglichen

Grössenordnung ist wichtiger als Detail

- **Genaue Betrachtungen sind unpraktikabel**
  - Mühsam bei nichttrivialen Algorithmen
  - Abhängig von Programmierdetails und Maschinenmodell

  Welches Maschinenmodell sollte der Standard sein?

- **Abschätzung der Komplexität ist sinnvoller**
  - Asymptotisches Verhalten für große Eingabedaten ist wichtig
- **Verwende ein Einheitskostenmodell**
  - Vereinfachte (modellunabhängige) Zählung von Elementaroperationen
- **Ignoriere Konstanten**
  - Additive Konstanten und konstante Faktoren werden durch Hardwaresteigerungen ausgeglichen

Analyse des wirklich relevanten Aufwands

Die Mathematik asymptotischer Vergleiche

- **$g \leq_a f$ (f wächst asymptotisch schneller als g)**

  Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

  – Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

Die Mathematik asymptotischer Vergleiche

- **$g \leq_a f$ (f wächst asymptotisch schneller als g)**

  Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

  – Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

- **(Größen-)Ordnung einer Funktion**

  – f als obere Schranke: $\mathcal{O}(f) = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c{>}0.\ g \leq_a c{*}f\}$

  – f als untere Schranke: $\Omega(f) = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c{>}0.\ c{*}f \leq_a g\}$

  – f als exakte Schranke: $\Theta(f) = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c, c{>}0.\ c{*}f \leq_a g \leq_a c'{*}f\}$

Die Mathematik asymptotischer Vergleiche

- **$g \leq_a f$ (f wächst asymptotisch schneller als g)**

  Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

  – Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

- **(Größen-)Ordnung einer Funktion**

  – f als obere Schranke: $\boldsymbol{\mathcal{O}(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \,|\, \exists c{>}0.\ g \leq_a c * f\}$

  – f als untere Schranke: $\boldsymbol{\Omega(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \,|\, \exists c{>}0.\ c * f \leq_a g\}$

  – f als exakte Schranke: $\boldsymbol{\Theta(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \,|\, \exists c, c{>}0.\ c * f \leq_a g \leq_a c' * f\}$

  Schreibweisen: $g = \mathcal{O}(f)$ statt $g \in \mathcal{O}(f)$, $\mathcal{O}(f) < \mathcal{O}(g)$ statt $\mathcal{O}(f) \subset \mathcal{O}(g)$

Die Mathematik asymptotischer Vergleiche

- **$g \leq_a f$ (f wächst asymptotisch schneller als g)**

  Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

  – Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

- **(Größen-)Ordnung einer Funktion**

  – f als obere Schranke: $\boldsymbol{\mathcal{O}(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \,|\, \exists c>0.\ g \leq_a c * f\}$

  – f als untere Schranke: $\boldsymbol{\Omega(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \,|\, \exists c>0.\ c * f \leq_a g\}$

  – f als exakte Schranke: $\boldsymbol{\Theta(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \,|\, \exists c, c>0.\ c * f \leq_a g \leq_a c' * f\}$

  Schreibweisen: $g = \mathcal{O}(f)$ statt $g \in \mathcal{O}(f)$, $\mathcal{O}(f) < \mathcal{O}(g)$ statt $\mathcal{O}(f) \subset \mathcal{O}(g)$

  $\mathcal{O}(1) \mathrel{\hat{=}} \mathcal{O}(\lambda n.1)$, $\mathcal{O}(n) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n)$, $\mathcal{O}(n^2) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n^2) \ldots$

Die Mathematik asymptotischer Vergleiche

- **$g \leq_a f$ (f wächst asymptotisch schneller als g)**

  Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

  – Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

- **(Größen-)Ordnung einer Funktion**

  – f als obere Schranke: $\mathcal{O}(f) = \{g{:}\mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c{>}0.\ g \leq_a c*f\}$

  – f als untere Schranke: $\Omega(f) = \{g{:}\mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c{>}0.\ c*f \leq_a g\}$

  – f als exakte Schranke: $\Theta(f) = \{g{:}\mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c, c{>}0.\ c*f \leq_a g \leq_a c'*f\}$

  Schreibweisen: $g{=}\mathcal{O}(f)$ statt $g \in \mathcal{O}(f)$, $\mathcal{O}(f) {<} \mathcal{O}(g)$ statt $\mathcal{O}(f) \subset \mathcal{O}(g)$

  $\mathcal{O}(1) \hat{=} \mathcal{O}(\lambda n.1)$, $\mathcal{O}(n) \hat{=} \mathcal{O}(\lambda n.n)$, $\mathcal{O}(n^2) \hat{=} \mathcal{O}(\lambda n.n^2)\ldots$

- **Beispiele für Ordnung konkreter Funktionen**

  - Konstante Funktion: $g_1(n){=}k$ für alle n

DIE MATHEMATIK ASYMPTOTISCHER VERGLEICHE

- **$g \leq_a f$ (f wächst asymptotisch schneller als g)**

  Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

  – Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

- **(Größen-)Ordnung einer Funktion**

  – f als obere Schranke: $\boldsymbol{\mathcal{O}(f)} = \{g{:}\mathbb{N} \rightarrow \mathbb{R}^+ \,|\, \exists c{>}0.\ g \leq_a c*f\}$

  – f als untere Schranke: $\boldsymbol{\Omega(f)} = \{g{:}\mathbb{N} \rightarrow \mathbb{R}^+ \,|\, \exists c{>}0.\ c*f \leq_a g\}$

  – f als exakte Schranke: $\boldsymbol{\Theta(f)} = \{g{:}\mathbb{N} \rightarrow \mathbb{R}^+ \,|\, \exists c, c{>}0.\ c*f \leq_a g \leq_a c'*f\}$

  Schreibweisen: $g{=}\mathcal{O}(f)$ statt $g \in \mathcal{O}(f)$, $\mathcal{O}(f){<}\mathcal{O}(g)$ statt $\mathcal{O}(f) {\subset} \mathcal{O}(g)$

  $\mathcal{O}(1) \mathrel{\hat{=}} \mathcal{O}(\lambda n.1)$, $\mathcal{O}(n) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n)$, $\mathcal{O}(n^2) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n^2)\ldots$

- **Beispiele für Ordnung konkreter Funktionen**

  - Konstante Funktion: $g_1(n){=}k$ für alle n $\qquad g_1 \in \mathcal{O}(1)$

Die Mathematik asymptotischer Vergleiche

- **$g\leq_a f$ (f wächst asymptotisch schneller als g)**

  Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

  – Es gibt ein $n_0\in\mathbb{N}$ mit $g(n)\leq f(n)$ für alle $n\geq n_0$

- **(Größen-)Ordnung einer Funktion**

  – f als obere Schranke: $\boldsymbol{\mathcal{O}(f)} = \{g:\mathbb{N}\to\mathbb{R}^+ \mid \exists c>0.\ g\leq_a c*f\}$

  – f als untere Schranke: $\boldsymbol{\Omega(f)} = \{g:\mathbb{N}\to\mathbb{R}^+ \mid \exists c>0.\ c*f\leq_a g\}$

  – f als exakte Schranke: $\boldsymbol{\Theta(f)} = \{g:\mathbb{N}\to\mathbb{R}^+ \mid \exists c,c>0.\ c*f\leq_a g\leq_a c'*f\}$

  Schreibweisen: $g=\mathcal{O}(f)$ statt $g\in\mathcal{O}(f)$, $\mathcal{O}(f)<\mathcal{O}(g)$ statt $\mathcal{O}(f)\subset\mathcal{O}(g)$

  $\mathcal{O}(1) \hat{=} \mathcal{O}(\lambda n.1)$, $\mathcal{O}(n) \hat{=} \mathcal{O}(\lambda n.n)$, $\mathcal{O}(n^2) \hat{=} \mathcal{O}(\lambda n.n^2)\ldots$

- **Beispiele für Ordnung konkreter Funktionen**

  - Konstante Funktion: $g_1(n)=k$ für alle n $\qquad g_1\in\mathcal{O}(1)$

  - Polynome: $g_2(n)=c_0+c_1*n+..+c_m*n^m$

Die Mathematik asymptotischer Vergleiche

- **$g \leq_a f$ (f wächst asymptotisch schneller als g)**

  Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

  – Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

- **(Größen-)Ordnung einer Funktion**

  – f als obere Schranke: $\boldsymbol{\mathcal{O}(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c{>}0.\ g \leq_a c * f\}$

  – f als untere Schranke: $\boldsymbol{\Omega(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c{>}0.\ c * f \leq_a g\}$

  – f als exakte Schranke: $\boldsymbol{\Theta(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c, c{>}0.\ c * f \leq_a g \leq_a c' * f\}$

  Schreibweisen: $g{=}\mathcal{O}(f)$ statt $g \in \mathcal{O}(f)$, $\mathcal{O}(f){<}\mathcal{O}(g)$ statt $\mathcal{O}(f) \subset \mathcal{O}(g)$

  $\mathcal{O}(1) \mathrel{\hat{=}} \mathcal{O}(\lambda n.1)$, $\mathcal{O}(n) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n)$, $\mathcal{O}(n^2) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n^2)\ldots$

- **Beispiele für Ordnung konkreter Funktionen**

  - Konstante Funktion: $g_1(n){=}k$ für alle n $\qquad g_1 \in \mathcal{O}(1)$

  - Polynome: $g_2(n){=}c_0 + c_1 {*} n + .. + c_m {*} n^m$ $\qquad g_2 \in \mathcal{O}(n^m)$

Die Mathematik asymptotischer Vergleiche

- **$g \leq_a f$ (f wächst asymptotisch schneller als g)**

  Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

  – Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

- **(Größen-)Ordnung einer Funktion**

  – f als obere Schranke: $\mathcal{O}(f) = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c{>}0.\ g \leq_a c{*}f\}$

  – f als untere Schranke: $\Omega(f) = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c{>}0.\ c{*}f \leq_a g\}$

  – f als exakte Schranke: $\Theta(f) = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c, c{>}0.\ c{*}f \leq_a g \leq_a c'{*}f\}$

  Schreibweisen: $g{=}\mathcal{O}(f)$ statt $g \in \mathcal{O}(f)$, $\mathcal{O}(f) {<} \mathcal{O}(g)$ statt $\mathcal{O}(f) \subset \mathcal{O}(g)$

  $\mathcal{O}(1) \mathrel{\hat{=}} \mathcal{O}(\lambda n.1)$, $\mathcal{O}(n) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n)$, $\mathcal{O}(n^2) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n^2) \ldots$

- **Beispiele für Ordnung konkreter Funktionen**

  - Konstante Funktion: $g_1(n){=}k$ für alle n $\qquad g_1 \in \mathcal{O}(1)$

  - Polynome: $g_2(n){=}c_0 + c_1{*}n + .. + c_m{*}n^m$ $\qquad g_2 \in \mathcal{O}(n^m)$

  - Logarithmenfunktionen: $g_3(n){=}log_b n$

DIE MATHEMATIK ASYMPTOTISCHER VERGLEICHE

- **$g \leq_a f$ (f wächst asymptotisch schneller als g)**

  Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

  – Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

- **(Größen-)Ordnung einer Funktion**

  – f als obere Schranke: $\boldsymbol{\mathcal{O}(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c>0.\ g \leq_a c*f\}$

  – f als untere Schranke: $\boldsymbol{\Omega(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c>0.\ c*f \leq_a g\}$

  – f als exakte Schranke: $\boldsymbol{\Theta(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c, c>0.\ c*f \leq_a g \leq_a c'*f\}$

  Schreibweisen: $g=\mathcal{O}(f)$ statt $g \in \mathcal{O}(f)$, $\mathcal{O}(f)<\mathcal{O}(g)$ statt $\mathcal{O}(f) \subset \mathcal{O}(g)$

  $\mathcal{O}(1) \mathrel{\hat{=}} \mathcal{O}(\lambda n.1)$, $\mathcal{O}(n) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n)$, $\mathcal{O}(n^2) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n^2)\ldots$

- **Beispiele für Ordnung konkreter Funktionen**

  - Konstante Funktion: $g_1(n)=k$ für alle n — $g_1 \in \mathcal{O}(1)$

  - Polynome: $g_2(n)=c_0 + c_1*n + .. + c_m*n^m$ — $g_2 \in \mathcal{O}(n^m)$

  - Logarithmenfunktionen: $g_3(n)=log_b n$ — $g_3 \in \mathcal{O}(log_2 n)$

Die Mathematik asymptotischer Vergleiche

- **$g \leq_a f$ (f wächst asymptotisch schneller als g)**

  Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

  – Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

- **(Größen-)Ordnung einer Funktion**

  – f als obere Schranke: $\boldsymbol{\mathcal{O}(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c{>}0.\ g \leq_a c * f\}$

  – f als untere Schranke: $\boldsymbol{\Omega(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c{>}0.\ c * f \leq_a g\}$

  – f als exakte Schranke: $\boldsymbol{\Theta(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c, c{>}0.\ c * f \leq_a g \leq_a c' * f\}$

  Schreibweisen: $g{=}\mathcal{O}(f)$ statt $g \in \mathcal{O}(f)$, $\mathcal{O}(f){<}\mathcal{O}(g)$ statt $\mathcal{O}(f) \subset \mathcal{O}(g)$

  $\mathcal{O}(1) \mathrel{\hat{=}} \mathcal{O}(\lambda n.1)$, $\mathcal{O}(n) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n)$, $\mathcal{O}(n^2) \mathrel{\hat{=}} \mathcal{O}(\lambda n.n^2)\ldots$

- **Beispiele für Ordnung konkreter Funktionen**

  - Konstante Funktion: $g_1(n){=}k$ für alle n — $g_1 \in \mathcal{O}(1)$
  - Polynome: $g_2(n){=}c_0 + c_1{*}n + .. + c_m{*}n^m$ — $g_2 \in \mathcal{O}(n^m)$
  - Logarithmenfunktionen: $g_3(n){=}log_b n$ — $g_3 \in \mathcal{O}(log_2 n)$
  - Fakultätsfunktion: $g_4(n){=}n!{=}1{*}2{*}..{*}n$

Die Mathematik asymptotischer Vergleiche

- **$g \leq_a f$ (f wächst asymptotisch schneller als g)**

  Ab einer bestimmten Stelle ist f immer mindestens so groß wie g

  – Es gibt ein $n_0 \in \mathbb{N}$ mit $g(n) \leq f(n)$ für alle $n \geq n_0$

- **(Größen-)Ordnung einer Funktion**

  – f als obere Schranke: $\boldsymbol{\mathcal{O}(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c{>}0.\ g \leq_a c*f\}$

  – f als untere Schranke: $\boldsymbol{\Omega(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c{>}0.\ c*f \leq_a g\}$

  – f als exakte Schranke: $\boldsymbol{\Theta(f)} = \{g{:}\mathbb{N} \to \mathbb{R}^+ \mid \exists c, c{>}0.\ c*f \leq_a g \leq_a c'*f\}$

  Schreibweisen: $g{=}\mathcal{O}(f)$ statt $g \in \mathcal{O}(f)$, $\mathcal{O}(f){<}\mathcal{O}(g)$ statt $\mathcal{O}(f) \subset \mathcal{O}(g)$

  $\mathcal{O}(1) \hat{=} \mathcal{O}(\lambda n.1)$, $\mathcal{O}(n) \hat{=} \mathcal{O}(\lambda n.n)$, $\mathcal{O}(n^2) \hat{=} \mathcal{O}(\lambda n.n^2)\ldots$

- **Beispiele für Ordnung konkreter Funktionen**

  - Konstante Funktion: $g_1(n){=}k$ für alle n $\qquad g_1 \in \mathcal{O}(1)$
  - Polynome: $g_2(n){=}c_0 + c_1{*}n + .. + c_m{*}n^m$ $\qquad g_2 \in \mathcal{O}(n^m)$
  - Logarithmenfunktionen: $g_3(n){=}log_b n$ $\qquad g_3 \in \mathcal{O}(log_2 n)$
  - Fakultätsfunktion: $g_4(n){=}n!{=}1{*}2{*}..{*}n$ $\qquad g_4 \in \mathcal{O}(n^n)$

Zeit- und Platzbedarf von Maschinen

- **Bestimme Aufwand relativ zur Eingabegröße**
  - $\boldsymbol{T_M(n)} = \max\{t_M(w) \mid |w|{=}n\}$ (worst-case)
  - $\boldsymbol{S_M(n)} = \max\{s_M(w) \mid |w|{=}n\}$ Einheit 4.2, Folie 7

Zeit- und Platzbedarf von Maschinen

- **Bestimme Aufwand relativ zur Eingabegröße**
  - $\boldsymbol{T_M(n)} = \max\{t_M(w) \mid |w|{=}n\}$ (worst-case)
  - $\boldsymbol{S_M(n)} = \max\{s_M(w) \mid |w|{=}n\}$ Einheit 4.2, Folie 7
- **Komplexität einer Maschine**
  - M hat **Zeitkomplexität $\mathcal{O}(f)$**, falls $T_M \in \mathcal{O}(f)$
  - M hat **Platzkomplexität $\mathcal{O}(f)$**, falls $S_M \in \mathcal{O}(f)$

Zeit- und Platzbedarf von Maschinen

- **Bestimmme Aufwand relativ zur Eingabegröße**
  - $\boldsymbol{T_M(n)} = \max\{t_M(w) \mid |w|{=}n\}$ (worst-case)
  - $\boldsymbol{S_M(n)} = \max\{s_M(w) \mid |w|{=}n\}$ Einheit 4.2, Folie 7
- **Komplexität einer Maschine**
  - M hat **Zeitkomplexität $\mathcal{O}(f)$**, falls $T_M \in \mathcal{O}(f)$
  - M hat **Platzkomplexität $\mathcal{O}(f)$**, falls $S_M \in \mathcal{O}(f)$
- **Wichtige Komplexitätklassen**
  - **polynomielle** (Zeit-)Komplexität: $t_M \in \mathcal{O}(n^k)$ für ein $k \in \mathbb{N}$
  - **exponentielle** (Zeit-)Komplexität: $t_M \in \mathcal{O}(2^{n^k})$ für ein $k \in \mathbb{N}$

Zeit- und Platzbedarf von Maschinen

- **Bestimme Aufwand relativ zur Eingabegröße**
  - $\boldsymbol{T_M(n)} = \max\{t_M(w) \mid |w|{=}n\}$ (worst-case)
  - $\boldsymbol{S_M(n)} = \max\{s_M(w) \mid |w|{=}n\}$ Einheit 4.2, Folie 7
- **Komplexität einer Maschine**
  - M hat **Zeitkomplexität $\mathcal{O}(f)$**, falls $T_M \in \mathcal{O}(f)$
  - M hat **Platzkomplexität $\mathcal{O}(f)$**, falls $S_M \in \mathcal{O}(f)$
- **Wichtige Komplexitätklassen**
  - **polynomielle** (Zeit-)Komplexität: $t_M \in \mathcal{O}(n^k)$ für ein $k \in \mathbb{N}$
  - **exponentielle** (Zeit-)Komplexität: $t_M \in \mathcal{O}(2^{n^k})$ für ein $k \in \mathbb{N}$

Weitere Klassen: konstant: $\mathcal{O}(1)$, logarithmisch: $\mathcal{O}(log_2 n)$
linear: $\mathcal{O}(n)$, quadratisch: $\mathcal{O}(n^2)$, kubisch: $\mathcal{O}(n^3)$,
superexponentiell: $\mathcal{O}(2^{2^{n^k}})$ für ein $k \in \mathbb{N}$

Zeit- und Platzbedarf von Maschinen

- **Bestimme Aufwand relativ zur Eingabegröße**
  - $\boldsymbol{T_M(n)} = \max\{t_M(w) \mid |w|{=}n\}$ (worst-case)
  - $\boldsymbol{S_M(n)} = \max\{s_M(w) \mid |w|{=}n\}$ Einheit 4.2, Folie 7
- **Komplexität einer Maschine**
  - M hat **Zeitkomplexität $\mathcal{O}(f)$**, falls $T_M \in \mathcal{O}(f)$
  - M hat **Platzkomplexität $\mathcal{O}(f)$**, falls $S_M \in \mathcal{O}(f)$
- **Wichtige Komplexitätklassen**
  - **polynomielle** (Zeit-)Komplexität: $t_M \in \mathcal{O}(n^k)$ für ein $k \in \mathbb{N}$
  - **exponentielle** (Zeit-)Komplexität: $t_M \in \mathcal{O}(2^{n^k})$ für ein $k \in \mathbb{N}$

  Weitere Klassen: konstant: $\mathcal{O}(1)$, logarithmisch: $\mathcal{O}(log_2 n)$
  linear: $\mathcal{O}(n)$, quadratisch: $\mathcal{O}(n^2)$, kubisch: $\mathcal{O}(n^3)$,
  superexponentiell: $\mathcal{O}(2^{2^{n^k}})$ für ein $k \in \mathbb{N}$

Maße für andere Berechnungsmodelle analog

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	…	1000	1.000.000
Wachstum									
$\log_2 n$	1ns								
n									
n^2									
n^3									
2^n									
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	...	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns							
n									
n^2									
n^3									
2^n									
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	...	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns					
n									
n^2									
n^3									
2^n									
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	…	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	
n									
n^2									
n^3									
2^n									
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	…	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n									
n^2									
n^3									
2^n									
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	…	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns								
n^2									
n^3									
2^n									
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n / Wachstum	10	20	30	40	50	60	...	1000	1.000.000
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns			
n^2									
n^3									
2^n									
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	...	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2									
n^3									
2^n									
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n / Wachstum	10	20	30	40	50	60	...	1000	1.000.000
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns								
n^3									
2^n									
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	…	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs			
n^3									
2^n									
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	…	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3									
2^n									
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n / Wachstum	10	20	30	40	50	60	...	1000	1.000.000
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n									
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n / Wachstum	10	20	30	40	50	60	...	1000	1.000.000
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns								
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n / Wachstum	10	20	30	40	50	60	…	1000	1.000.000
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs							
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	...	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms						
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n / Wachstum	10	20	30	40	50	60	…	1000	1.000.000
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s					
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	…	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h				
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	…	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n									

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n / Wachstum	10	20	30	40	50	60	…	1000	1.000.000
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs								

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	…	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs	1.1s							

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n / Wachstum	10	20	30	40	50	60	...	1000	1.000.000
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs	1.1s	17.3h						

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n / Wachstum	10	20	30	40	50	60	...	1000	1.000.000
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs	1.1s	17.3h	116y					

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	...	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs	1.1s	17.3h	116y	2.500.000.000y				

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	…	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs	1.1s	17.3h	116y	2.500.000.000y				

Wieviel mehr kann man in der gleichen Zeit berechnen, wenn Computer um den Faktor 1000 schneller werden?

	$\log_2 n$	n	n^2	n^3	2^n	3^n
Problemsteigerung						

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	...	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs	1.1s	17.3h	116y	2.500.000.000y				

Wieviel mehr kann man in der gleichen Zeit berechnen, wenn Computer um den Faktor 1000 schneller werden?

	$\log_2 n$	n	n^2	n^3	2^n	3^n
Problemsteigerung	10^{300}-fach					

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	...	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs	1.1s	17.3h	116y	2.500.000.000y				

Wieviel mehr kann man in der gleichen Zeit berechnen, wenn Computer um den Faktor 1000 schneller werden?

	$\log_2 n$	n	n^2	n^3	2^n	3^n
Problemsteigerung	10^{300}-fach	1000-fach				

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor

Größe n / Wachstum	10	20	30	40	50	60	...	1000	1.000.000
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs	1.1s	17.3h	116y	2.500.000.000y				

Wieviel mehr kann man in der gleichen Zeit berechnen, wenn Computer um den Faktor 1000 schneller werden?

	$\log_2$n	n	n^2	n^3	2^n	3^n
Problemsteigerung	10^{300}-fach	1000-fach	31-fach			

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n / Wachstum	10	20	30	40	50	60	...	1000	1.000.000
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs	1.1s	17.3h	116y	2.500.000.000y				

Wieviel mehr kann man in der gleichen Zeit berechnen, wenn Computer um den Faktor 1000 schneller werden?

	$\log_2 n$	n	n^2	n^3	2^n	3^n
Problemsteigerung	10^{300}-fach	1000-fach	31-fach	10-fach		

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	...	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs	1.1s	17.3h	116y	2.500.000.000y				

Wieviel mehr kann man in der gleichen Zeit berechnen, wenn Computer um den Faktor 1000 schneller werden?

	$\log_2$n	n	n^2	n^3	2^n	3^n
Problemsteigerung	10^{300}-fach	1000-fach	31-fach	10-fach	plus 10	

Rechenzeit: Wo liegt die Grenze des Handhabbaren?

Rechenzeiten auf 3.3 Ghz Prozessor									
Größe n	10	20	30	40	50	60	...	1000	1.000.000
Wachstum									
$\log_2 n$	1ns	2ns		3ns				10ns	100ns
n	3ns	6ns	9ns	12ns	15ns	18ns		300ns	300μs
n^2	30ns	120ns	270ns	480ns	750ns	1.1μs		300μs	300s
n^3	300ns	2.4μs	8.1μs	19.2μs	37.5μs	64μs		300ms	9.5y
2^n	300ns	300μs	300ms	300s	83.3h	9.5y			
3^n	17.8μs	1.1s	17.3h	116y	2.500.000.000y				

Wieviel mehr kann man in der gleichen Zeit berechnen, wenn Computer um den Faktor 1000 schneller werden?

	$\log_2 n$	n	n^2	n^3	2^n	3^n
Problemsteigerung	10^{300}-fach	1000-fach	31-fach	10-fach	plus 10	plus 6

Auswirkung asymptotischer Analyse

- **Polynomielle Lösbarkeit ist entscheidend**
  – Exponentieller Aufwand ist für die Praxis unakzeptabel

Auswirkung asymptotischer Analyse

- **Polynomielle Lösbarkeit ist entscheidend**
  – Exponentieller Aufwand ist für die Praxis unakzeptabel
  – Unterschiede innerhalb polynomieller Komplexität sind tolerierbar aber durchaus relevant für konkrete Implementierungen

Auswirkung asymptotischer Analyse

- **Polynomielle Lösbarkeit ist entscheidend**
  – Exponentieller Aufwand ist für die Praxis unakzeptabel
  – Unterschiede innerhalb polynomieller Komplexität sind tolerierbar aber durchaus relevant für konkrete Implementierungen
- **Bessere Hardware ist selten eine gute Lösung**
  – Wenn Algorithmen schlecht sind, nützt die beste Hardware wenig

Auswirkung asymptotischer Analyse

- **Polynomielle Lösbarkeit ist entscheidend**
  - Exponentieller Aufwand ist für die Praxis unakzeptabel
  - Unterschiede innerhalb polynomieller Komplexität sind tolerierbar aber durchaus relevant für konkrete Implementierungen
- **Bessere Hardware ist selten eine gute Lösung**
  - Wenn Algorithmen schlecht sind, nützt die beste Hardware wenig
  - Es lohnt sich, in die Verbesserung von Algorithmen zu investieren

Auswirkung asymptotischer Analyse

- **Polynomielle Lösbarkeit ist entscheidend**
  – Exponentieller Aufwand ist für die Praxis unakzeptabel
  – Unterschiede innerhalb polynomieller Komplexität sind tolerierbar aber durchaus relevant für konkrete Implementierungen
- **Bessere Hardware ist selten eine gute Lösung**
  – Wenn Algorithmen schlecht sind, nützt die beste Hardware wenig
  – Es lohnt sich, in die Verbesserung von Algorithmen zu investieren
- **Es gibt noch ungeklärte Fragen**
  – Macht Parallelismus / Nichtdeterminismus Probleme handhabbar?
    · Effizienzsteigerung von exponentieller auf polynomielle Zeit?

Auswirkung asymptotischer Analyse

- **Polynomielle Lösbarkeit ist entscheidend**
  - Exponentieller Aufwand ist für die Praxis unakzeptabel
  - Unterschiede innerhalb polynomieller Komplexität sind tolerierbar aber durchaus relevant für konkrete Implementierungen
- **Bessere Hardware ist selten eine gute Lösung**
  - Wenn Algorithmen schlecht sind, nützt die beste Hardware wenig
  - Es lohnt sich, in die Verbesserung von Algorithmen zu investieren
- **Es gibt noch ungeklärte Fragen**
  - Macht Parallelismus / Nichtdeterminismus Probleme handhabbar?
    - Effizienzsteigerung von exponentieller auf polynomielle Zeit?
  - Zusammenhang zwischen Platzbedarf und Laufzeitverhalten?
    - Bisher nur grobe Abschätzungen bekannt

Fragestellungen der Komplexitätstheorie

- **Komplexität konkreter Verfahren**
  - Maximaler Verbrauch im Einzelfall (worst case)

    Wichtig bei sicherheitskritischen Anwendungen

Fragestellungen der Komplexitätstheorie

- **Komplexität konkreter Verfahren**
  - Maximaler Verbrauch im Einzelfall (worst case)

    Wichtig bei sicherheitskritischen Anwendungen
  - Durchschnittlicher Bedarf im Langzeitverhalten (average case)

    Verlangt mathematisch schwierige statistische Analyse

Fragestellungen der Komplexitätstheorie

- **Komplexität konkreter Verfahren**
  - Maximaler Verbrauch im Einzelfall (worst case)
    Wichtig bei sicherheitskritischen Anwendungen
  - Durchschnittlicher Bedarf im Langzeitverhalten (average case)
    Verlangt mathematisch schwierige statistische Analyse
- **Analyse von Problemen**
  - Wie effizient ist die bestmögliche Lösung? (untere Schranken)

Fragestellungen der Komplexitätstheorie

- **Komplexität konkreter Verfahren**
  - Maximaler Verbrauch im Einzelfall (worst case)
    Wichtig bei sicherheitskritischen Anwendungen
  - Durchschnittlicher Bedarf im Langzeitverhalten (average case)
    Verlangt mathematisch schwierige statistische Analyse
- **Analyse von Problemen**
  - Wie effizient ist die bestmögliche Lösung? (untere Schranken)
- **Komplexitätsklassen**
  - Welche Probleme haben (in etwa) den gleichen Schwierigkeitsgrad?
  - Problemreduktion: effiziente Lösungen wiederverwenden

Fragestellungen der Komplexitätstheorie

- **Komplexität konkreter Verfahren**
  - Maximaler Verbrauch im Einzelfall (worst case)
    Wichtig bei sicherheitskritischen Anwendungen
  - Durchschnittlicher Bedarf im Langzeitverhalten (average case)
    Verlangt mathematisch schwierige statistische Analyse
- **Analyse von Problemen**
  - Wie effizient ist die bestmögliche Lösung? (untere Schranken)
- **Komplexitätsklassen**
  - Welche Probleme haben (in etwa) den gleichen Schwierigkeitsgrad?
  - Problemreduktion: effiziente Lösungen wiederverwenden
- **Welche Probleme sind handhabbar?**
  - Welche Fragestellungen sind (nicht) polynomiell lösbar
  - Welche Verbesserung können unkonventionelle Ansätze erreichen?
    (Nichtdeterministische, approximierende, probabilistische Verfahren)

Theoretische Informatik II

Einheit 5.1

Konkrete Komplexitätsanalyse

1. Komplexität spezifischer Algorithmen
2. Komplexität von Problemstellungen

Obere Schranken für die Laufzeit von Verfahren

Obere Schranken für die Laufzeit von Verfahren

- **Analyse auf Ebene abstrakter Algorithmen**
  – Asymptotische Komplexität hängt nicht von Programmiersprache ab

Abschätzung der Komplexität von Algorithmen

Obere Schranken für die Laufzeit von Verfahren

- **Analyse auf Ebene abstrakter Algorithmen**
  - Asymptotische Komplexität hängt nicht von Programmiersprache ab
  - Konstanter Expansionsfaktor bei Übersetzung in Maschinensprache
    (Simulation durch Turingmaschinen würde zu polynomieller Expansion führen)

Abschätzung der Komplexität von Algorithmen

Obere Schranken für die Laufzeit von Verfahren

- **Analyse auf Ebene abstrakter Algorithmen**
  - Asymptotische Komplexität hängt nicht von Programmiersprache ab
  - Konstanter Expansionsfaktor bei Übersetzung in Maschinensprache
    (Simulation durch Turingmaschinen würde zu polynomieller Expansion führen)

- **Elementaroperationen gelten als ein Schritt**
  - `+`, `-`, `*`,`/`,... Einzelschritte, wenn Zahlengröße beschränkt (z.B. 64-bit)

Abschätzung der Komplexität von Algorithmen

Obere Schranken für die Laufzeit von Verfahren

- **Analyse auf Ebene abstrakter Algorithmen**
  - Asymptotische Komplexität hängt nicht von Programmiersprache ab
  - Konstanter Expansionsfaktor bei Übersetzung in Maschinensprache
    (Simulation durch Turingmaschinen würde zu polynomieller Expansion führen)

- **Elementaroperationen gelten als ein Schritt**
  - `+`, `-`, `*`,`/`,... Einzelschritte, wenn Zahlengröße beschränkt (z.B. 64-bit)
  - Höherer Aufwand bei beliebig großen Zahlen

Abschätzung der Komplexität von Algorithmen

Obere Schranken für die Laufzeit von Verfahren

- **Analyse auf Ebene abstrakter Algorithmen**
  - Asymptotische Komplexität hängt nicht von Programmiersprache ab
  - Konstanter Expansionsfaktor bei Übersetzung in Maschinensprache
    (Simulation durch Turingmaschinen würde zu polynomieller Expansion führen)
- **Elementaroperationen gelten als ein Schritt**
  - `+`, `-`, `*`,`/`,... Einzelschritte, wenn Zahlengröße beschränkt (z.B. 64-bit)
  - Höherer Aufwand bei beliebig großen Zahlen
- **Fokus auf sequentielle Algorithmen**

Abschätzung der Komplexität von Algorithmen

Obere Schranken für die Laufzeit von Verfahren

- **Analyse auf Ebene abstrakter Algorithmen**
  - Asymptotische Komplexität hängt nicht von Programmiersprache ab
  - Konstanter Expansionsfaktor bei Übersetzung in Maschinensprache
    (Simulation durch Turingmaschinen würde zu polynomieller Expansion führen)
- **Elementaroperationen gelten als ein Schritt**
  - `+`, `-`, `*`,`/`,... Einzelschritte, wenn Zahlengröße beschränkt (z.B. 64-bit)
  - Höherer Aufwand bei beliebig großen Zahlen
- **Fokus auf sequentielle Algorithmen**
  - Parallele/nichtdeterministische Maschinen haben evtl. bessere Laufzeit

Sequentielle Suche: kommt x in L vor?

- **Durchsuche Liste L von links nach rechts**

```
function search_seq(x,L) ≡
   found := false;
   for i = 1 to length(L) do
      if L[i]=x then found:=true
   od;
   return found;
```

Sequentielle Suche: kommt x in L vor?

- **Durchsuche Liste L von links nach rechts**

```
function search_seq(x,L) ≡
   found := false;
   for i = 1 to length(L) do
      if L[i]=x then found:=true
   od;
   return found;
```

Verfahren ist anwendbar auf beliebige Listen

Sequentielle Suche: kommt x in L vor?

- **Durchsuche Liste L von links nach rechts**

```
function search_seq(x,L) ≡
   found := false;
   for i = 1 to length(L) do
      if L[i]=x then found:=true
   od;
   return found;
```

  Verfahren ist anwendbar auf beliebige Listen

- **Laufzeitanalyse**
  - Eine Operation für Initialisierung `found:=false`

Sequentielle Suche: kommt x in L vor?

- **Durchsuche Liste L von links nach rechts**

  ```
  function search_seq(x,L) ≡
     found := false;
     for i = 1 to length(L) do
        if L[i]=x then found:=true
     od;
     return found;
  ```

  Verfahren ist anwendbar auf beliebige Listen

- **Laufzeitanalyse**
  - Eine Operation für Initialisierung `found:=false`
  - Je 2 Operationen pro Element von L in der `for`-Schleife

Sequentielle Suche: kommt x in L vor?

- **Durchsuche Liste L von links nach rechts**

  function $\mathtt{search}_{seq}$(x,L) $\equiv$
```
   found := false;
   for i = 1 to length(L) do
      if L[i]=x then found:=true
   od;
   return found;
```
  Verfahren ist anwendbar auf beliebige Listen

- **Laufzeitanalyse**
  - Eine Operation für Initialisierung `found:=false`
  - Je 2 Operationen pro Element von L in der `for`-Schleife
  - Eine Operation für Ausgabe des Ergebnisses

Sequentielle Suche: kommt x in L vor?

- **Durchsuche Liste L von links nach rechts**

```
function search_seq(x,L) ≡
  found := false;
  for i = 1 to length(L) do
    if L[i]=x then found:=true
  od;
  return found;
```

  Verfahren ist anwendbar auf beliebige Listen

- **Laufzeitanalyse**
  - Eine Operation für Initialisierung `found:=false`
  - Je 2 Operationen pro Element von L in der `for`-Schleife
  - Eine Operation für Ausgabe des Ergebnisses
  - Insgesamt $2n+2$ Schritte, wenn n die Größe der Liste L ist

Sequentielle Suche: kommt x in L vor?

- **Durchsuche Liste L von links nach rechts**

```
function search_seq(x,L) ≡
   found := false;
   for i = 1 to length(L) do
      if L[i]=x then found:=true
   od;
   return found;
```

  Verfahren ist anwendbar auf beliebige Listen

- **Laufzeitanalyse**
  - Eine Operation für Initialisierung `found:=false`
  - Je 2 Operationen pro Element von L in der `for`-Schleife
  - Eine Operation für Ausgabe des Ergebnisses
  - Insgesamt $2n+2$ Schritte, wenn n die Größe der Liste L ist

Sequentielle Suche ist in $\mathcal{O}(n)$

Binäre Suche

Nur anwendbar, wenn Liste L geordnet ist

Binäre Suche

Nur anwendbar, wenn Liste *L* geordnet ist

- **Teste mittleres Element; dann rechts oder links**

function search$_{bin}$(x,L) $\equiv$
  let function search$_b$(x,L,left,right) $\equiv$
    if left>right then return false
      else
        mid := (left+right) div 2;
        if x<L[mid] then search$_b$(x,L,left,mid-1)
          elseif x>L[mid] then search$_b$(x,L,mid+1,right)
          else return true
        fi;
  return search$_b$(x,L,1,length(L))

Binäre Suche

Nur anwendbar, wenn Liste *L* geordnet ist

- **Teste mittleres Element; dann rechts oder links**

```
function search_bin(x,L) ≡
   let function search_b(x,L,left,right) ≡
      if left>right then return false
         else
            mid := (left+right) div 2;
            if x<L[mid] then search_b(x,L,left,mid-1)
               elseif x>L[mid] then search_b(x,L,mid+1,right)
               else return true
            fi;
   return search_b(x,L,1,length(L))
```

- **Eine grobe Laufzeitanalyse reicht aus**
  – Konstante Anzahl von Operationen pro Aufruf von `search`$_b$

Binäre Suche

Nur anwendbar, wenn Liste L geordnet ist

- **Teste mittleres Element; dann rechts oder links**

```
function search_bin(x,L) ≡
   let function search_b(x,L,left,right) ≡
      if left>right then return false
         else
            mid := (left+right) div 2;
            if x<L[mid] then search_b(x,L,left,mid-1)
               elseif x>L[mid] then search_b(x,L,mid+1,right)
               else return true
            fi;
   return search_b(x,L,1,length(L))
```

- **Eine grobe Laufzeitanalyse reicht aus**
  - Konstante Anzahl von Operationen pro Aufruf von $\texttt{search}_b$
  - **Wie oft wird $\texttt{search}_b$ aufgerufen?**

Binäre Suche – Analyse

```
function search_bin(x,L) ≡
   let function search_b(x,L,left,right) ≡
      if left>right then return false
         else
            mid := (left+right) div 2;
            if x<L[mid] then search_b(x,L,left,mid-1)
               elseif x>L[mid] then search_b(x,L,mid+1,right)
               else return true
            fi;
   return search_b(x,L,1,length(L))
```

Binäre Suche – Analyse

```
function search_bin(x,L) ≡
   let function search_b(x,L,left,right) ≡
      if left>right then return false
         else
            mid := (left+right) div 2;
            if x<L[mid] then search_b(x,L,left,mid-1)
               elseif x>L[mid] then search_b(x,L,mid+1,right)
               else return true
            fi;
   return search_b(x,L,1,length(L))
```

Abstand von `left` und `right` halbiert sich pro Aufruf (mit Abrundung)

Anzahl von Operationen pro Aufruf von $\texttt{search}_b$ ist eine Konstante k

Binäre Suche – Analyse

```
function search_bin(x,L) ≡
   let function search_b(x,L,left,right) ≡
      if left>right then return false
         else
            mid := (left+right) div 2;
            if x<L[mid] then search_b(x,L,left,mid-1)
               elseif x>L[mid] then search_b(x,L,mid+1,right)
               else return true
            fi;
   return search_b(x,L,1,length(L))
```

Abstand von **left** und **right** halbiert sich pro Aufruf (mit Abrundung)

Anzahl von Operationen pro Aufruf von **search**$_b$ ist eine Konstante k

Abstand zu Beginn ist $n-1$ (n ist die Größe der Liste L)

search$_b$ terminiert bei Erfolg oder wenn Abstand Null ist

Binäre Suche – Analyse

```
function search_bin(x,L) ≡
   let function search_b(x,L,left,right) ≡
      if left>right then return false
         else
            mid := (left+right) div 2;
            if x<L[mid] then search_b(x,L,left,mid-1)
               elseif x>L[mid] then search_b(x,L,mid+1,right)
               else return true
            fi;
   return search_b(x,L,1,length(L))
```

Abstand von **left** und **right** halbiert sich pro Aufruf (mit Abrundung)

Anzahl von Operationen pro Aufruf von **search**$_b$ ist eine Konstante k

Abstand zu Beginn ist $n{-}1$ (n ist die Größe der Liste L)

search$_b$ terminiert bei Erfolg oder wenn Abstand Null ist

Lösung der Gleichung $time(n) = k{+}time(\lfloor n/2 \rfloor)$ ist $time(n) = k * \log_2 n$

Binäre Suche – Analyse

```
function search_bin(x,L) ≡
   let function search_b(x,L,left,right) ≡
      if left>right then return false
         else
            mid := (left+right) div 2;
            if x<L[mid] then search_b(x,L,left,mid-1)
               elseif x>L[mid] then search_b(x,L,mid+1,right)
               else return true
            fi;
   return search_b(x,L,1,length(L))
```

Abstand von **left** und **right** halbiert sich pro Aufruf (mit Abrundung)

Anzahl von Operationen pro Aufruf von **search**$_b$ ist eine Konstante k

Abstand zu Beginn ist $n-1$ (n ist die Größe der Liste L)

search$_b$ terminiert bei Erfolg oder wenn Abstand Null ist

Lösung der Gleichung $time(n) = k+time(\lfloor n/2 \rfloor)$ ist $time(n) = k * \log_2 n$

$\Downarrow$

Binäre Suche ist in $\mathcal{O}(\log_2 n)$

Sortierverfahren

- **Ordne Elemente in aufsteigender Reihenfolge**
  - Geordnete Listen unterstützen effizienten Zugriff auf Elemente
  - Eine der häufigsten Operationen in der Programmierung

Sortierverfahren

- **Ordne Elemente in aufsteigender Reihenfolge**
  - Geordnete Listen unterstützen effizienten Zugriff auf Elemente
  - Eine der häufigsten Operationen in der Programmierung
- **Viele Verfahren bekannt** http://www.sortieralgorithmen.de

Sortierverfahren

- **Ordne Elemente in aufsteigender Reihenfolge**
  - Geordnete Listen unterstützen effizienten Zugriff auf Elemente
  - Eine der häufigsten Operationen in der Programmierung
- **Viele Verfahren bekannt** http://www.sortieralgorithmen.de
  - **Insertion Sort**: Einfügen des Listenanfangs in geordnete Teilliste
  - **Selection Sort**: Auswahl des jeweils kleinsten Elements als Listenanfang

Sortierverfahren

- **Ordne Elemente in aufsteigender Reihenfolge**
  - Geordnete Listen unterstützen effizienten Zugriff auf Elemente
  - Eine der häufigsten Operationen in der Programmierung
- **Viele Verfahren bekannt** http://www.sortieralgorithmen.de
  - **Insertion Sort**: Einfügen des Listenanfangs in geordnete Teilliste
  - **Selection Sort**: Auswahl des jeweils kleinsten Elements als Listenanfang
  - **Bubblesort**: Austauschen benachbarter Elemente

Sortierverfahren

- **Ordne Elemente in aufsteigender Reihenfolge**
  - Geordnete Listen unterstützen effizienten Zugriff auf Elemente
  - Eine der häufigsten Operationen in der Programmierung
- **Viele Verfahren bekannt** http://www.sortieralgorithmen.de
  - **Insertion Sort**: Einfügen des Listenanfangs in geordnete Teilliste
  - **Selection Sort**: Auswahl des jeweils kleinsten Elements als Listenanfang
  - **Bubblesort**: Austauschen benachbarter Elemente
  - **Quicksort**: Aufteilung nach Größe, Sortieren der entstehenden Teillisten

Sortierverfahren

- **Ordne Elemente in aufsteigender Reihenfolge**
  – Geordnete Listen unterstützen effizienten Zugriff auf Elemente
  – Eine der häufigsten Operationen in der Programmierung
- **Viele Verfahren bekannt** http://www.sortieralgorithmen.de
  – **Insertion Sort**: Einfügen des Listenanfangs in geordnete Teilliste
  – **Selection Sort**: Auswahl des jeweils kleinsten Elements als Listenanfang
  – **Bubblesort**: Austauschen benachbarter Elemente
  – **Quicksort**: Aufteilung nach Größe, Sortieren der entstehenden Teillisten
  – **Mergesort**: Aufteilen in Teillisten, Sortieren und Mischen der Teillisten
  – **Mergesort (II)**: Identifizieren und Mischen geordneter Teillisten

Sortierverfahren

- **Ordne Elemente in aufsteigender Reihenfolge**
  - Geordnete Listen unterstützen effizienten Zugriff auf Elemente
  - Eine der häufigsten Operationen in der Programmierung
- **Viele Verfahren bekannt** http://www.sortieralgorithmen.de
  - **Insertion Sort**: Einfügen des Listenanfangs in geordnete Teilliste
  - **Selection Sort**: Auswahl des jeweils kleinsten Elements als Listenanfang
  - **Bubblesort**: Austauschen benachbarter Elemente
  - **Quicksort**: Aufteilung nach Größe, Sortieren der entstehenden Teillisten
  - **Mergesort**: Aufteilen in Teillisten, Sortieren und Mischen der Teillisten
  - **Mergesort (II)**: Identifizieren und Mischen geordneter Teillisten

‘Bestes’ Verfahren hängt von Problemgröße ab

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

9	7	8	2	1	5	6

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

9	7	8	2	1	5	6

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	9	8	2	1	5	6

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	8	9	2	1	5	6

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	8	2	9	1	5	6

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	8	2	1	9	5	6

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	8	2	1	5	9	6

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	8	2	1	5	6	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	8	2	1	5	6	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	8	2	1	5	6	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	2	8	1	5	6	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	2	1	8	5	6	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	2	1	5	8	6	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	2	1	5	6	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

7	2	1	5	6	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

2	7	1	5	6	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

2	1	7	5	6	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

2	1	5	7	6	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

2	1	5	6	7	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

2	1	5	6	7	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

1	2	5	6	7	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

1	2	5	6	7	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

1	2	5	6	7	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

1	2	5	6	7	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

1	2	5	6	7	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
  for upper = length(L)-1 downto 1 do
    for j = 1 to upper do
      if L[j]>L[j+1] then
        aux := L[j];
        L[j] := L[j+1];
        L[j+1] := aux
      fi
    od
  od
```

- **Beispiel einer Sortierung mit Bubblesort**

1	2	5	6	7	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

1	2	5	6	7	8	9

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort

Fortlaufender Vergleich benachbarter Elemente
Austausch bei falscher Reihenfolge

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Beispiel einer Sortierung mit Bubblesort**

1	2	5	6	7	8	9	$\surd$

Elemente steigen wie Blasen auf, bis sie auf größere treffen

Bubblesort - Laufzeitanalyse

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

Bubblesort - Laufzeitanalyse

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Feste Anzahl von Operationen im Schleifenrumpf**
  - Vergleich benachbarter Elemente
  - ggf. Austauch unter Verwendung einer Hilfsvariablen

Bubblesort - Laufzeitanalyse

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Feste Anzahl von Operationen im Schleifenrumpf**
  - Vergleich benachbarter Elemente
  - ggf. Austauch unter Verwendung einer Hilfsvariablen
- **Anzahl Schleifen abhängig von Listengröße n**

Bubblesort - Laufzeitanalyse

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Feste Anzahl von Operationen im Schleifenrumpf**
  - Vergleich benachbarter Elemente
  - ggf. Austauch unter Verwendung einer Hilfsvariablen
- **Anzahl Schleifen abhängig von Listengröße n**
  - Innere Schleife wird jeweils genau `upper`-mal durchlaufen

Bubblesort - Laufzeitanalyse

```
function bubblesort(L) ≡
    for upper = length(L)-1 downto 1 do
        for j = 1 to upper do
            if L[j]>L[j+1] then
                aux := L[j];
                L[j] := L[j+1];
                L[j+1] := aux
            fi
        od
    od
```

- **Feste Anzahl von Operationen im Schleifenrumpf**
  - Vergleich benachbarter Elemente
  - ggf. Austausch unter Verwendung einer Hilfsvariablen
- **Anzahl Schleifen abhängig von Listengröße n**
  - Innere Schleife wird jeweils genau `upper`-mal durchlaufen
  - Insgesamt $n\text{-}1 + n\text{-}2 + ... + 2 + 1 = n*(n\text{-}1)/2$ Durchläufe

Bubblesort - Laufzeitanalyse

```
function bubblesort(L) ≡
   for upper = length(L)-1 downto 1 do
      for j = 1 to upper do
         if L[j]>L[j+1] then
            aux := L[j];
            L[j] := L[j+1];
            L[j+1] := aux
         fi
      od
   od
```

- **Feste Anzahl von Operationen im Schleifenrumpf**
  - Vergleich benachbarter Elemente
  - ggf. Austauch unter Verwendung einer Hilfsvariablen
- **Anzahl Schleifen abhängig von Listengröße n**
  - Innere Schleife wird jeweils genau `upper`-mal durchlaufen
  - Insgesamt n-1 + n-2 + ... + 2 + 1 = n*(n-1)/2 Durchläufe

$$\Downarrow$$

Bubblesort ist in $\mathcal{O}(n^2)$

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

SORTIEREN SCHNELLER ALS $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

Sortieren schneller als $\mathcal{O}(n^2)$

- Identifiziere **Läufe**, d.h. geordnete Teilfolgen

9	7	8	2	1	5	6

Sortieren schneller als $\mathcal{O}(n^2)$

- Identifiziere Läufe, d.h. geordnete Teilfolgen

9	7	8	2	1	5	6

- Verschmelze Läufe zu neuen Läufen

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7						

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8					

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9				

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1			

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2		

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Länge der Läufe wächst – Anzahl halbiert sich

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Länge der Läufe wächst – Anzahl halbiert sich

- **Wiederhole bis Folge geordnet**

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Länge der Läufe wächst – Anzahl halbiert sich

- **Wiederhole bis Folge geordnet**

7	8	9	1	2	5	6

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Länge der Läufe wächst – Anzahl halbiert sich

- **Wiederhole bis Folge geordnet**

7	8	9	1	2	5	6
1						

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Länge der Läufe wächst – Anzahl halbiert sich

- **Wiederhole bis Folge geordnet**

7	8	9	1	2	5	6
1	2					

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Länge der Läufe wächst – Anzahl halbiert sich

- **Wiederhole bis Folge geordnet**

7	8	9	1	2	5	6
1	2	5				

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Länge der Läufe wächst – Anzahl halbiert sich

- **Wiederhole bis Folge geordnet**

7	8	9	1	2	5	6
1	2	5	6			

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Länge der Läufe wächst – Anzahl halbiert sich

- **Wiederhole bis Folge geordnet**

7	8	9	1	2	5	6
1	2	5	6	7		

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Länge der Läufe wächst – Anzahl halbiert sich

- **Wiederhole bis Folge geordnet**

7	8	9	1	2	5	6
1	2	5	6	7	8	

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Länge der Läufe wächst – Anzahl halbiert sich

- **Wiederhole bis Folge geordnet**

7	8	9	1	2	5	6
1	2	5	6	7	8	9

Sortieren schneller als $\mathcal{O}(n^2)$

- **Identifiziere Läufe, d.h. geordnete Teilfolgen**

9	7	8	2	1	5	6

- **Verschmelze Läufe zu neuen Läufen**

9	7	8	2	1	5	6
7	8	9	1	2	5	6

  – Länge der Läufe wächst – Anzahl halbiert sich

- **Wiederhole bis Folge geordnet**

7	8	9	1	2	5	6
1	2	5	6	7	8	9

  – Liste ist eine einzige (komplett) geordnete Teilfolge $\surd$

Analyse des Verfahrens

Abstrakte Skizze reicht für Laufzeitanalyse

Analyse des Verfahrens

Abstrakte Skizze reicht für Laufzeitanalyse

- **Verschmelzen ist in $\mathcal{O}(n)$**
  – Folge wird jeweils komplett durchlaufen

Analyse des Verfahrens

Abstrakte Skizze reicht für Laufzeitanalyse

- **Verschmelzen ist in $\mathcal{O}(n)$**
  - Folge wird jeweils komplett durchlaufen
- **Verschmelzen halbiert Anzahl der Läufe**
  - Je zwei Läufe werden zu einem gemischt

Analyse des Verfahrens

Abstrakte Skizze reicht für Laufzeitanalyse

- **Verschmelzen ist in $\mathcal{O}(n)$**
  – Folge wird jeweils komplett durchlaufen
- **Verschmelzen halbiert Anzahl der Läufe**
  – Je zwei Läufe werden zu einem gemischt
- **Man braucht maximal $\log_2 n$ Verschmelzungen**
  – Danach ist nur ein einziger Lauf übrig, d.h. die Liste ist sortiert

Analyse des Verfahrens

Abstrakte Skizze reicht für Laufzeitanalyse

- **Verschmelzen ist in $\mathcal{O}(n)$**
  - Folge wird jeweils komplett durchlaufen
- **Verschmelzen halbiert Anzahl der Läufe**
  - Je zwei Läufe werden zu einem gemischt
- **Man braucht maximal $\log_2 n$ Verschmelzungen**
  - Danach ist nur ein einziger Lauf übrig, d.h. die Liste ist sortiert

$$\Downarrow$$

Sortieren durch Verschmelzen ist in $\mathcal{O}(n * \log_2 n)$

Graphen und Bäume (Handout)

- **Fundamentale Datenstruktur vieler Anwendungen**

- **Fundamentale Datenstruktur vieler Anwendungen**
  - Ordnungstruktur für effiziente Verwaltung großer Datenmengen
  - Beschreibung der Topographie von Netzwerken

⋮

- **Fundamentale Datenstruktur vieler Anwendungen**
  – Ordnungstruktur für effiziente Verwaltung großer Datenmengen
  – Beschreibung der Topographie von Netzwerken
  ⋮
- **Graphen haben Knoten und Kanten ($G = (V, E)$)**

- **Fundamentale Datenstruktur vieler Anwendungen**
  - Ordnungstruktur für effiziente Verwaltung großer Datenmengen
  - Beschreibung der Topographie von Netzwerken

  ⋮

- **Graphen haben Knoten und Kanten ($G = (V, E)$)**
  - Eine Kante e zwischen zwei Knoten $v{\neq}v'$ kann gerichtet ($e = (v, v')$) oder ungerichtet ($e = \{v, v'\}$) sein

- **Fundamentale Datenstruktur vieler Anwendungen**
  - Ordnungstruktur für effiziente Verwaltung großer Datenmengen
  - Beschreibung der Topographie von Netzwerken

    $\vdots$

- **Graphen haben Knoten und Kanten ($G = (V, E)$)**
  - Eine Kante e zwischen zwei Knoten $v \neq v'$ kann gerichtet ($e = (v, v')$) oder ungerichtet ($e = \{v, v'\}$) sein
  - Beschreibbar als Liste $v_1, \ldots, v_n$, $\{v_{i_1}, v'_{i_1}\}, \ldots, \{v_{i_m}, v'_{i_m}\}$

- **Fundamentale Datenstruktur vieler Anwendungen**
  - Ordnungstruktur für effiziente Verwaltung großer Datenmengen
  - Beschreibung der Topographie von Netzwerken

  ⋮

- **Graphen haben Knoten und Kanten ($G = (V, E)$)**
  - Eine Kante e zwischen zwei Knoten $v \neq v'$ kann gerichtet ($e = (v, v')$) oder ungerichtet ($e = \{v, v'\}$) sein
  - Beschreibbar als Liste $v_1, ..., v_n,\ \{v_{i_1}, v'_{i_1}\}, ..., \{v_{i_m}, v'_{i_m}\}$
  - In gewichteten Graphen ist jede Kante mit einer Zahl markiert

Graphen und Bäume (Handout)

- **Fundamentale Datenstruktur vieler Anwendungen**
  - Ordnungstruktur für effiziente Verwaltung großer Datenmengen
  - Beschreibung der Topographie von Netzwerken

    ⋮

- **Graphen haben Knoten und Kanten ($G = (V, E)$)**
  - Eine Kante e zwischen zwei Knoten $v \neq v'$ kann gerichtet ($e = (v, v')$) oder ungerichtet ($e = \{v, v'\}$) sein
  - Beschreibbar als Liste $v_1, ..., v_n,\ \{v_{i_1}, v'_{i_1}\}, ..., \{v_{i_m}, v'_{i_m}\}$
  - In gewichteten Graphen ist jede Kante mit einer Zahl markiert

- **Bäume sind zyklenfreie ungerichtete Graphen**

Graphen und Bäume (Handout)

- **Fundamentale Datenstruktur vieler Anwendungen**
  - Ordnungstruktur für effiziente Verwaltung großer Datenmengen
  - Beschreibung der Topographie von Netzwerken

  ⋮

- **Graphen haben Knoten und Kanten ($G = (V, E)$)**
  - Eine Kante e zwischen zwei Knoten $v \neq v'$ kann gerichtet ($e = (v, v')$) oder ungerichtet ($e = \{v, v'\}$) sein
  - Beschreibbar als Liste $v_1, ..., v_n,\ \{v_{i_1}, v'_{i_1}\}, ..., \{v_{i_m}, v'_{i_m}\}$
  - In gewichteten Graphen ist jede Kante mit einer Zahl markiert

- **Bäume sind zyklenfreie ungerichtete Graphen**
  - Ein Baum spannt einen Graphen auf, wenn jeder Knoten von G von der Wurzel des Baums aus erreichbar ist

Graphen und Bäume (Handout)

- **Fundamentale Datenstruktur vieler Anwendungen**
  - Ordnungstruktur für effiziente Verwaltung großer Datenmengen
  - Beschreibung der Topographie von Netzwerken

    ⋮

- **Graphen haben Knoten und Kanten ($G = (V, E)$)**
  - Eine Kante e zwischen zwei Knoten $v \neq v'$ kann gerichtet ($e = (v, v')$) oder ungerichtet ($e = \{v, v'\}$) sein
  - Beschreibbar als Liste $v_1, \ldots, v_n,\ \{v_{i_1}, v'_{i_1}\}, \ldots, \{v_{i_m}, v'_{i_m}\}$
  - In gewichteten Graphen ist jede Kante mit einer Zahl markiert

- **Bäume sind zyklenfreie ungerichtete Graphen**
  - Ein Baum spannt einen Graphen auf, wenn jeder Knoten von G von der Wurzel des Baums aus erreichbar ist
  - Ein MWST ist ein aufspannender Baum mit minimalem Gewicht

Der Kruskal Algorithmus

Bestimme einen MWST in einem Graphen G

Der Kruskal Algorithmus

Bestimme einen MWST in einem Graphen G

- **Erzeuge Zusammenhangskomponenten in G**

Der Kruskal Algorithmus

Bestimme einen MWST in einem Graphen G

- **Erzeuge Zusammenhangskomponenten in G**
  - Initialwert ist $\{v\}$ für jeden Knoten $v \in V$ $\qquad (Z := \{\{v\} \mid v \in V\})$

Der Kruskal Algorithmus

Bestimme einen MWST in einem Graphen G

- **Erzeuge Zusammenhangskomponenten in G**
  - Initialwert ist $\{v\}$ für jeden Knoten $v \in V$ $\qquad (Z := \{\{v\} \mid v \in V\})$
  - Betrachte eine neue Kante $e \in E$ mit geringstem Gewicht

Der Kruskal Algorithmus

Bestimme einen MWST in einem Graphen G

- **Erzeuge Zusammenhangskomponenten in G**
  - Initialwert ist $\{v\}$ für jeden Knoten $v \in V$ $\quad (Z := \{\{v\} \mid v \in V\})$
  - Betrachte eine neue Kante $e \in E$ mit geringstem Gewicht

    Falls e Knoten aus verschiedenen Zusammenhangskomponenten verbindet, füge e dem MWST hinzu und vereinige die beiden Komponenten

Der Kruskal Algorithmus

Bestimme einen MWST in einem Graphen G

- **Erzeuge Zusammenhangskomponenten in G**
  - Initialwert ist $\{v\}$ für jeden Knoten $v \in V$ $\quad (Z := \{\{v\} \mid v \in V\})$
  - Betrachte eine neue Kante $e \in E$ mit geringstem Gewicht

    Falls e Knoten aus verschiedenen Zusammenhangskomponenten verbindet, füge e dem MWST hinzu und vereinige die beiden Komponenten
  - Wiederhole dies, bis alle Knoten in einer Komponente sind
    oder alle Kanten betrachtet wurden

Der Kruskal Algorithmus

Bestimme einen MWST in einem Graphen G

- **Erzeuge Zusammenhangskomponenten in G**
  - Initialwert ist $\{v\}$ für jeden Knoten $v \in V$ $\quad (Z := \{\{v\} \mid v \in V\})$
  - Betrachte eine neue Kante $e \in E$ mit geringstem Gewicht

    Falls e Knoten aus verschiedenen Zusammenhangskomponenten verbindet, füge e dem MWST hinzu und vereinige die beiden Komponenten
  - Wiederhole dies, bis alle Knoten in einer Komponente sind oder alle Kanten betrachtet wurden

- **Implementierbar mit Laufzeit $\mathcal{O}(|V| + |E| \log |E|)$**
  - Liste der Kanten muß zuerst nach Gewicht sortiert werden
  - Zusammenhangskomponenten müssen mit Pointern repräsentiert werden

Der Kruskal Algorithmus

Bestimme einen MWST in einem Graphen G

- **Erzeuge Zusammenhangskomponenten in G**
  - Initialwert ist $\{v\}$ für jeden Knoten $v \in V$ $\quad (Z := \{\{v\} \mid v \in V\})$
  - Betrachte eine neue Kante $e \in E$ mit geringstem Gewicht

    Falls e Knoten aus verschiedenen Zusammenhangskomponenten verbindet, füge e dem MWST hinzu und vereinige die beiden Komponenten
  - Wiederhole dies, bis alle Knoten in einer Komponente sind oder alle Kanten betrachtet wurden

- **Implementierbar mit Laufzeit $\mathcal{O}(|V| + |E| \log |E|)$**
  - Liste der Kanten muß zuerst nach Gewicht sortiert werden
  - Zusammenhangskomponenten müssen mit Pointern repräsentiert werden
  - Turingmaschine würde Laufzeit $\mathcal{O}((|V|+|E|)^4)$ benötigen $\quad$ HMU §10.1.2

Das Problem des Handlungsreisenden

Gegeben n Städte, eine Kostentabelle von Kosten c_{ij} um von Stadt i nach Stadt j zu reisen und eine Kostenbeschränkung B. Gibt es eine Rundreise durch alle n Städte, deren Kosten unter dem Limit B liegt?

Das Problem des Handlungsreisenden

Gegeben n Städte, eine Kostentabelle von Kosten c_{ij} um von Stadt i nach Stadt j zu reisen und eine Kostenbeschränkung B. Gibt es eine Rundreise durch alle n Städte, deren Kosten unter dem Limit B liegt?

- **Formulierung als Graphenproblem**
  - Ein Hamiltonscher Kreis im Graphen $G = (V, E)$ ist ein Kreis, der nur aus Kanten aus E besteht und jeden Knoten genau einmal berührt.

Das Problem des Handlungsreisenden

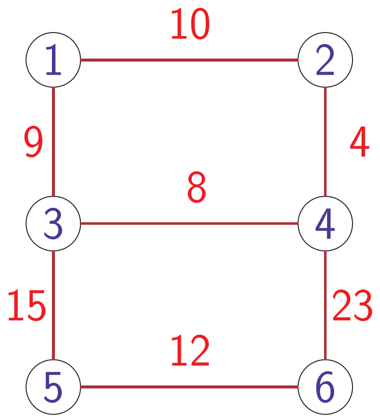
Gegeben n Städte, eine Kostentabelle von Kosten c_{ij} um von Stadt i nach Stadt j zu reisen und eine Kostenbeschränkung B. Gibt es eine Rundreise durch alle n Städte, deren Kosten unter dem Limit B liegt?

- **Formulierung als Graphenproblem**
  - Ein Hamiltonscher Kreis im Graphen $G = (V, E)$ ist ein Kreis, der nur aus Kanten aus E besteht und jeden Knoten genau einmal berührt.
  - **TSP**: Finde einen Hamiltonschen Kreis mit maximalen Kosten B

Das Problem des Handlungsreisenden

Gegeben n Städte, eine Kostentabelle von Kosten c_{ij} um von Stadt i nach Stadt j zu reisen und eine Kostenbeschränkung B. Gibt es eine Rundreise durch alle n Städte, deren Kosten unter dem Limit B liegt?

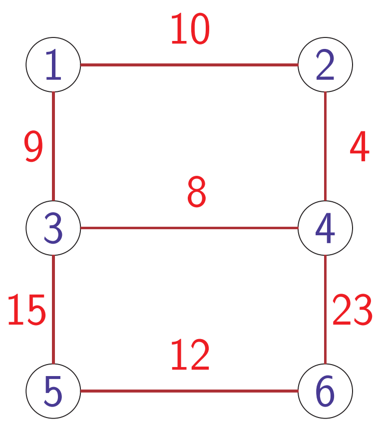
- **Formulierung als Graphenproblem**
  - Ein Hamiltonscher Kreis im Graphen $G = (V, E)$ ist ein Kreis, der nur aus Kanten aus E besteht und jeden Knoten genau einmal berührt.
  - **TSP**: Finde einen Hamiltonschen Kreis mit maximalen Kosten B



Das Problem des Handlungsreisenden

Gegeben n Städte, eine Kostentabelle von Kosten c_{ij} um von Stadt i nach Stadt j zu reisen und eine Kostenbeschränkung B. Gibt es eine Rundreise durch alle n Städte, deren Kosten unter dem Limit B liegt?

- **Formulierung als Graphenproblem**
  - Ein Hamiltonscher Kreis im Graphen $G = (V, E)$ ist ein Kreis, der nur aus Kanten aus E besteht und jeden Knoten genau einmal berührt.
  - **TSP**: Finde einen Hamiltonschen Kreis mit maximalen Kosten B



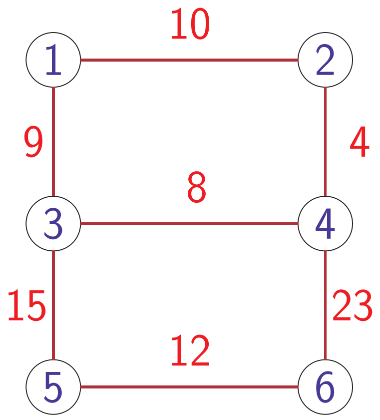
Nur eine Rundreise: [1,3,5,6,4,2]

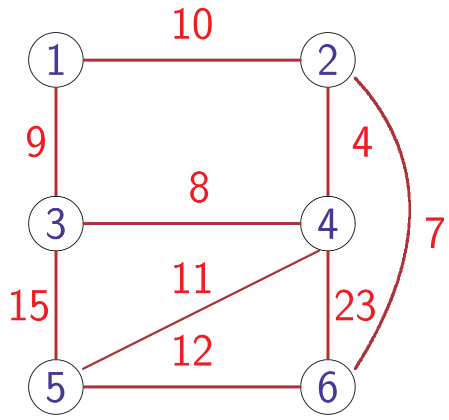
Kosten: 73

Das Problem des Handlungsreisenden

Gegeben n Städte, eine Kostentabelle von Kosten c_{ij} um von Stadt i nach Stadt j zu reisen und eine Kostenbeschränkung B. Gibt es eine Rundreise durch alle n Städte, deren Kosten unter dem Limit B liegt?

- **Formulierung als Graphenproblem**
  - Ein Hamiltonscher Kreis im Graphen $G = (V, E)$ ist ein Kreis, der nur aus Kanten aus E besteht und jeden Knoten genau einmal berührt.
  - **TSP**: Finde einen Hamiltonschen Kreis mit maximalen Kosten B





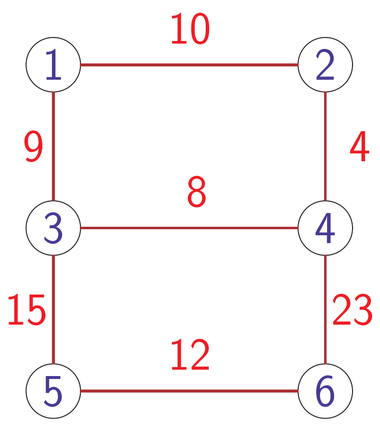
Nur eine Rundreise: [1,3,5,6,4,2]

Kosten: 73

Das Problem des Handlungsreisenden

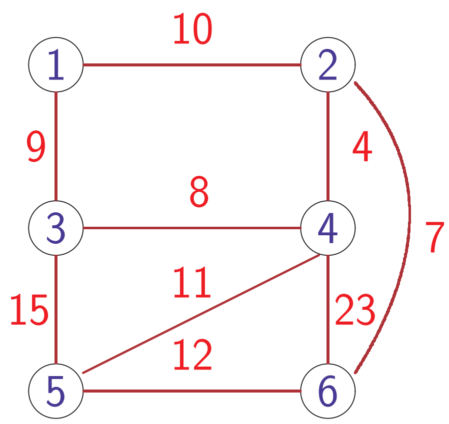
Gegeben n Städte, eine Kostentabelle von Kosten c_{ij} um von Stadt i nach Stadt j zu reisen und eine Kostenbeschränkung B. Gibt es eine Rundreise durch alle n Städte, deren Kosten unter dem Limit B liegt?

- **Formulierung als Graphenproblem**
  - Ein Hamiltonscher Kreis im Graphen $G = (V, E)$ ist ein Kreis, der nur aus Kanten aus E besteht und jeden Knoten genau einmal berührt.
  - **TSP**: Finde einen Hamiltonschen Kreis mit maximalen Kosten B



Nur eine Rundreise: [1,3,5,6,4,2]

Kosten: 73



Billigere Rundreise: [1,2,6,5,4,3]

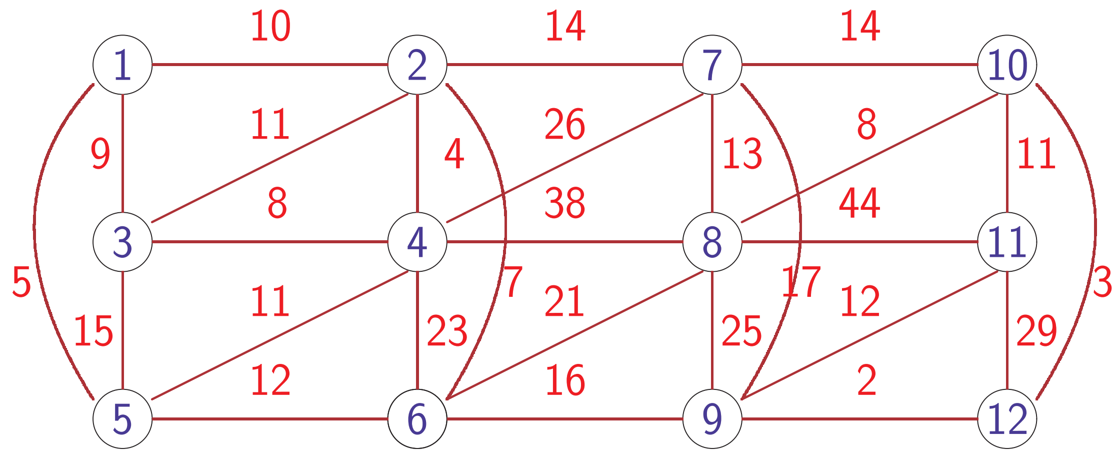
Kosten: 57

Das Problem des Handlungsreisenden

- **Graphen können sehr komplex sein**

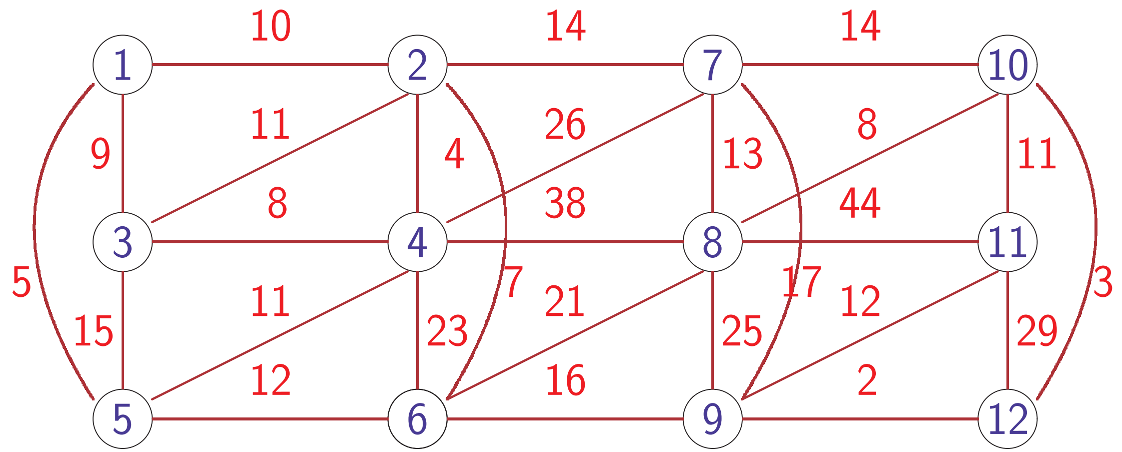
Das Problem des Handlungsreisenden

- **Graphen können sehr komplex sein**



Das Problem des Handlungsreisenden

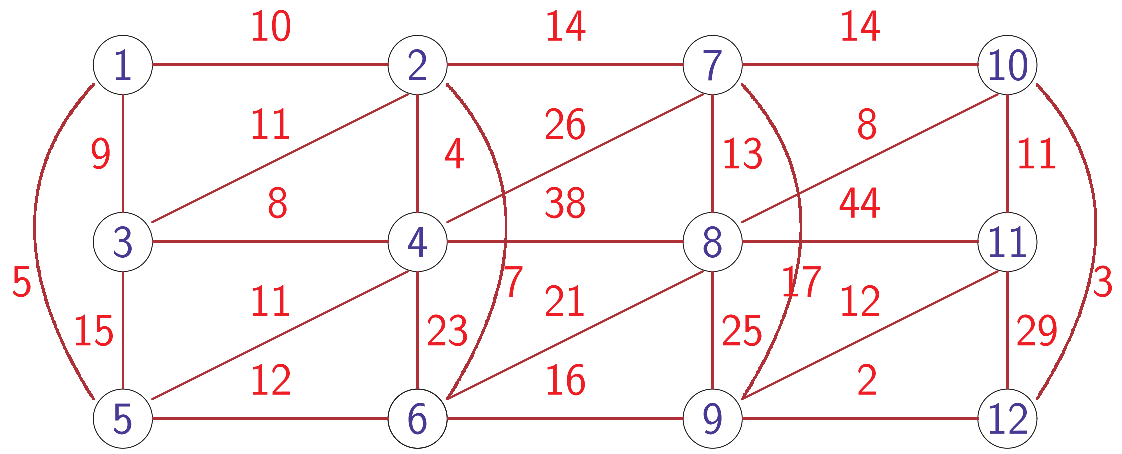
- **Graphen können sehr komplex sein**



- **Keine effiziente allgemeine Lösung bekannt**
  – Bester Ansatz ist "Generate & Test"

Das Problem des Handlungsreisenden

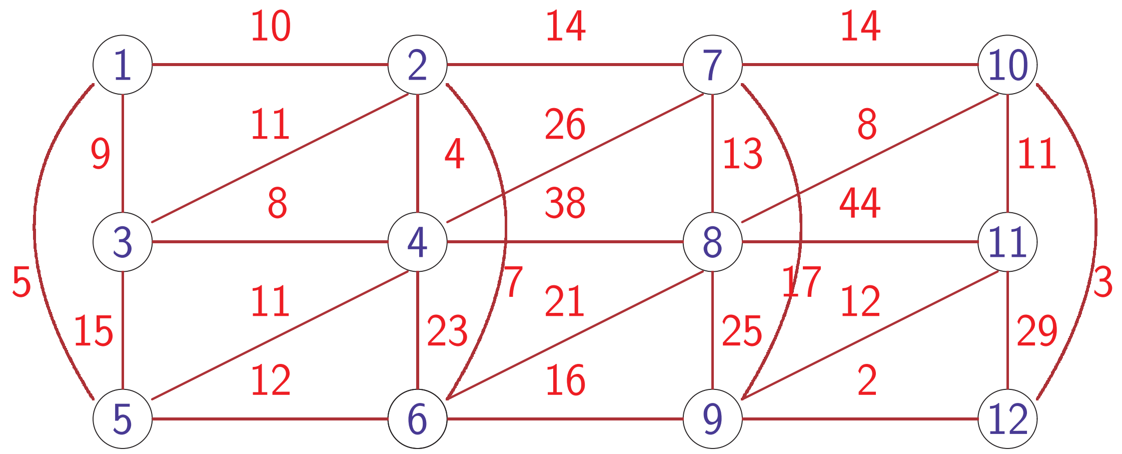
- **Graphen können sehr komplex sein**



- **Keine effiziente allgemeine Lösung bekannt**
  - Bester Ansatz ist “Generate & Test”
  - Test ist polynomiell, aber es gibt exponentiell viele Möglichkeiten

Das Problem des Handlungsreisenden

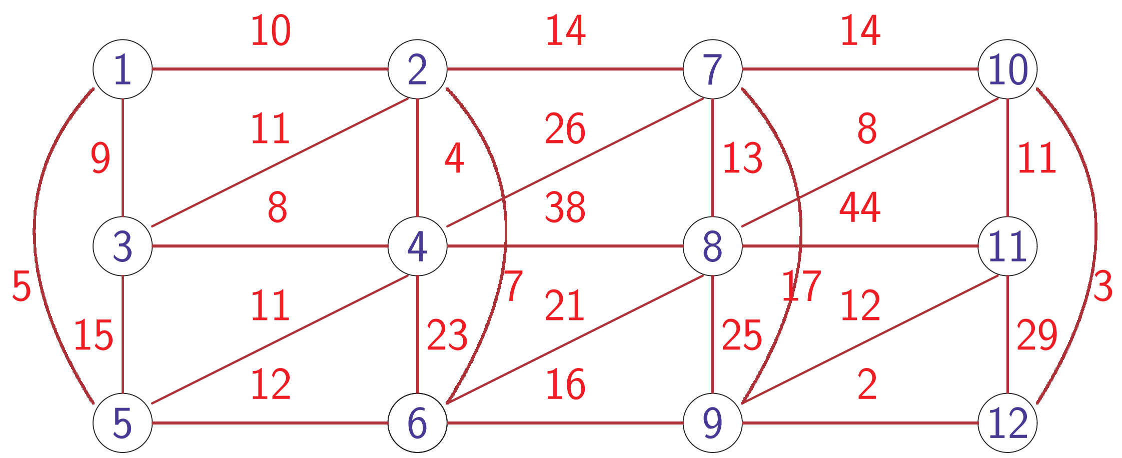
- **Graphen können sehr komplex sein**



- **Keine effiziente allgemeine Lösung bekannt**
  – Bester Ansatz ist "Generate & Test"
  – Test ist polynomiell, aber es gibt exponentiell viele Möglichkeiten
- **Approximative Lösungen möglich**
  – Rundreise mit Kosten 50% über Optimum polynomiell bestimmbar

Das Problem des Handlungsreisenden

- **Graphen können sehr komplex sein**



- **Keine effiziente allgemeine Lösung bekannt**
  - Bester Ansatz ist “Generate & Test”
  - Test ist polynomiell, aber es gibt exponentiell viele Möglichkeiten
- **Approximative Lösungen möglich**
  - Rundreise mit Kosten 50% über Optimum polynomiell bestimmbar
  - Benötigt Rahmenbedingung $c_{ij} \leq c_{ik} + c_{kj}$ (Dreiecksungleichung)

Untere Schranken für Komplexität von Lösungen

Komplexität von Problemen

Untere Schranken für Komplexität von Lösungen

- **Lösungen eines Problems sind unterschiedlich gut**
  - Suchen: Lineare Suche $\mathcal{O}(n)$ — Binärsuche $\mathcal{O}(\log_2 n)$
  - Sortieren: Bubblesort $\mathcal{O}(n^2)$ — Mergesort $\mathcal{O}(n * \log_2 n)$

Komplexität von Problemen

Untere Schranken für Komplexität von Lösungen

- **Lösungen eines Problems sind unterschiedlich gut**
  – Suchen: Lineare Suche $\mathcal{O}(n)$ — Binärsuche $\mathcal{O}(\log_2 n)$
  – Sortieren: Bubblesort $\mathcal{O}(n^2)$ — Mergesort $\mathcal{O}(n{*}\log_2 n)$
- **Wie effizient kann ein Problem gelöst werden?**
  – Gibt es eine Mindestkomplexität für eine optimale Lösung?
  – Wann ist eine Lösung gut genug?

Untere Schranken für Komplexität von Lösungen

- **Lösungen eines Problems sind unterschiedlich gut**
  - Suchen: Lineare Suche $\mathcal{O}(n)$ — Binärsuche $\mathcal{O}(\log_2 n)$
  - Sortieren: Bubblesort $\mathcal{O}(n^2)$ — Mergesort $\mathcal{O}(n * \log_2 n)$
- **Wie effizient kann ein Problem gelöst werden?**
  - Gibt es eine Mindestkomplexität für eine optimale Lösung?
  - Wann ist eine Lösung gut genug?
- **Antwort könnte von Art der Frage abhängen**
  - Entscheidungsproblem: *Gibt es überhaupt eine Lösung der Aufgabe?*
  - Optimierungsproblem: *Was ist die bestmögliche Lösung?*
  - Berechnungsproblem: *Bestimme eine konkrete Lösung*

Komplexität von Problemen

Untere Schranken für Komplexität von Lösungen

- **Lösungen eines Problems sind unterschiedlich gut**
  - Suchen: Lineare Suche $\mathcal{O}(n)$ — Binärsuche $\mathcal{O}(\log_2 n)$
  - Sortieren: Bubblesort $\mathcal{O}(n^2)$ — Mergesort $\mathcal{O}(n * \log_2 n)$
- **Wie effizient kann ein Problem gelöst werden?**
  - Gibt es eine Mindestkomplexität für eine optimale Lösung?
  - Wann ist eine Lösung gut genug?
- **Antwort könnte von Art der Frage abhängen**
  - Entscheidungsproblem: *Gibt es überhaupt eine Lösung der Aufgabe?*
  - Optimierungsproblem: *Was ist die bestmögliche Lösung?*
  - Berechnungsproblem: *Bestimme eine konkrete Lösung*
- **Nachweis ist im Normalfall aufwendig**
  - Man muß über alle möglichen Algorithmen argumentieren

Komplexität von Sortierverfahren

Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

Komplexität von Sortierverfahren

Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

- **Sortierverfahren müssen Elemente vergleichen**
  – Sonst kann die Anordnung der Elemente nicht garantiert werden

Komplexität von Sortierverfahren

Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

- **Sortierverfahren müssen Elemente vergleichen**
  - Sonst kann die Anordnung der Elemente nicht garantiert werden
  - Wieviel Vergleiche werden benötigt um $a_1, .., a_n$ zu ordnen?
  - Bestimme Anzahl der Vergleiche für den ungünstigsten Fall

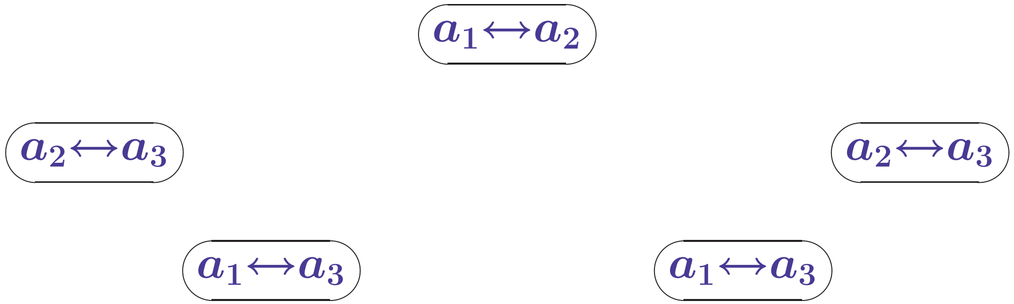
Komplexität von Sortierverfahren

Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

- **Sortierverfahren müssen Elemente vergleichen**
  - Sonst kann die Anordnung der Elemente nicht garantiert werden
  - Wieviel Vergleiche werden benötigt um $a_1, .., a_n$ zu ordnen?
  - Bestimme Anzahl der Vergleiche für den ungünstigsten Fall
- **Betrachte Entscheidungsbaum von Algorithmen**

Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

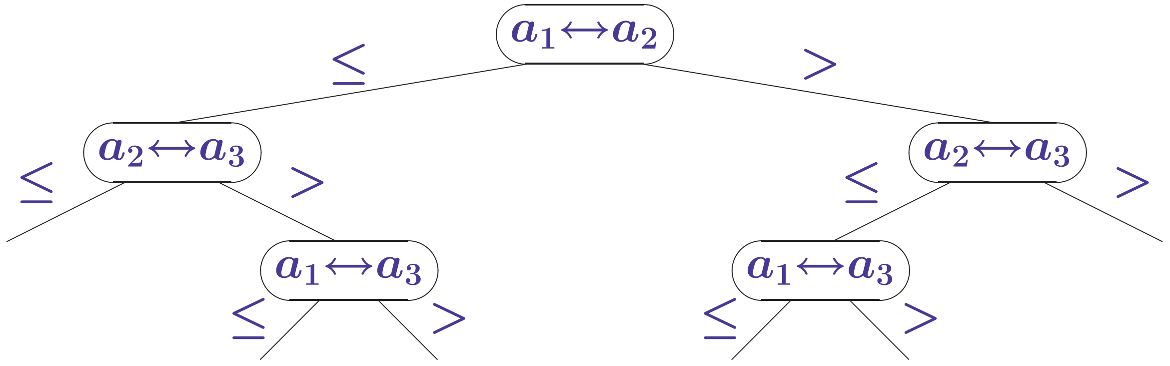
- **Sortierverfahren müssen Elemente vergleichen**
  - Sonst kann die Anordnung der Elemente nicht garantiert werden
  - Wieviel Vergleiche werden benötigt um $a_1, .., a_n$ zu ordnen?
  - Bestimme Anzahl der Vergleiche für den ungünstigsten Fall
- **Betrachte Entscheidungsbaum von Algorithmen**



  - Innere Knoten entsprechen den durchgeführten Vergleichen

Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

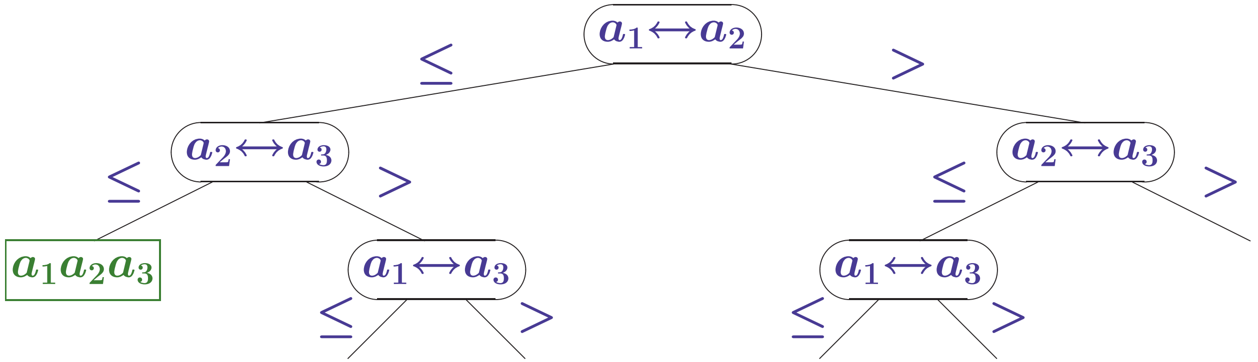
- **Sortierverfahren müssen Elemente vergleichen**
  - Sonst kann die Anordnung der Elemente nicht garantiert werden
  - Wieviel Vergleiche werden benötigt um $a_1, .., a_n$ zu ordnen?
  - Bestimme Anzahl der Vergleiche für den ungünstigsten Fall
- **Betrachte Entscheidungsbaum von Algorithmen**



  - Innere Knoten entsprechen den durchgeführten Vergleichen
  - Kanten markiert mit Vergleichergebnis ($\leq$,$>$)

Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

- **Sortierverfahren müssen Elemente vergleichen**
  - Sonst kann die Anordnung der Elemente nicht garantiert werden
  - Wieviel Vergleiche werden benötigt um $a_1, .., a_n$ zu ordnen?
  - Bestimme Anzahl der Vergleiche für den ungünstigsten Fall
- **Betrachte Entscheidungsbaum von Algorithmen**

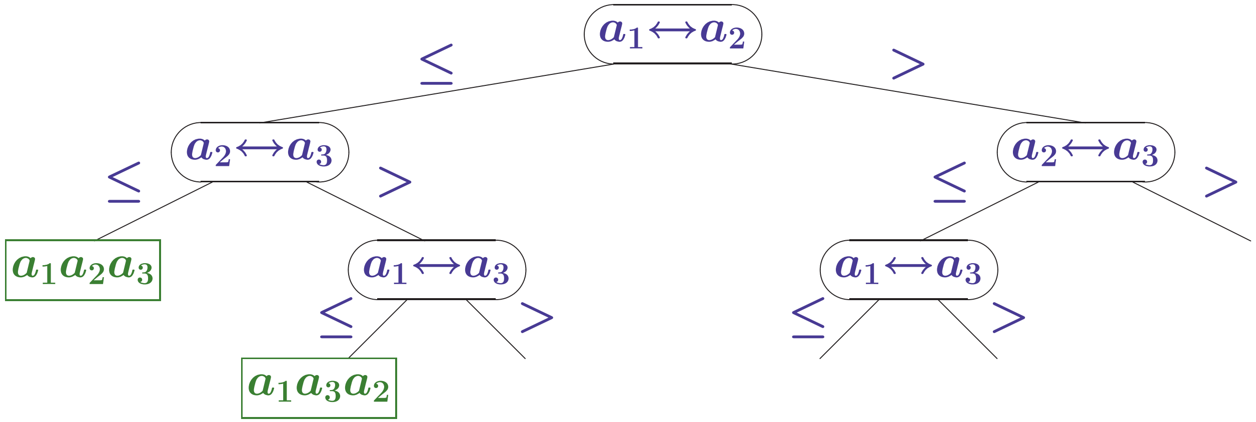


  - Innere Knoten entsprechen den durchgeführten Vergleichen
  - Kanten markiert mit Vergleichergebnis ($\leq$,$>$)
  - Blätter sind resultierende Anordnung der Elemente

Komplexität von Sortierverfahren

Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

- **Sortierverfahren müssen Elemente vergleichen**
  - Sonst kann die Anordnung der Elemente nicht garantiert werden
  - Wieviel Vergleiche werden benötigt um $a_1, .., a_n$ zu ordnen?
  - Bestimme Anzahl der Vergleiche für den ungünstigsten Fall
- **Betrachte Entscheidungsbaum von Algorithmen**

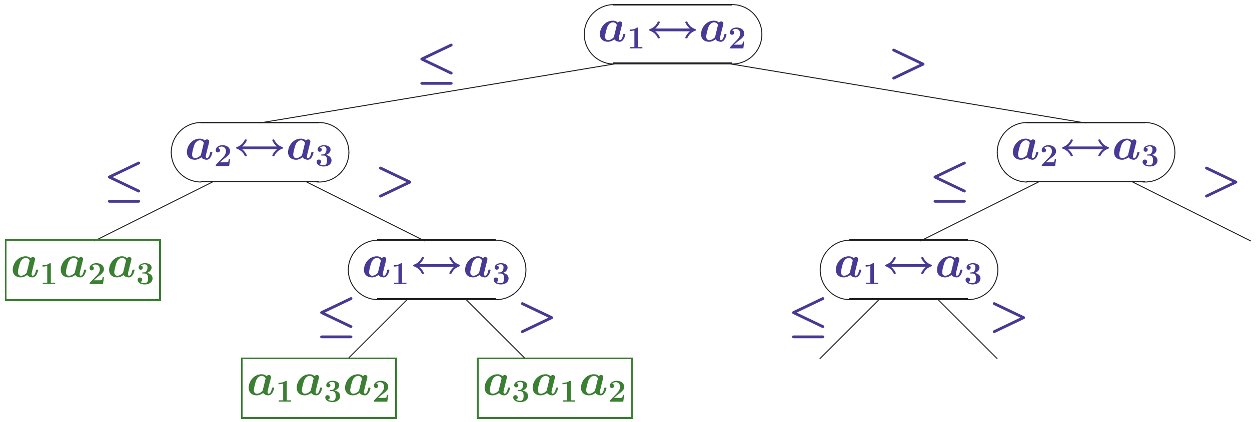


  - Innere Knoten entsprechen den durchgeführten Vergleichen
  - Kanten markiert mit Vergleichergebnis ($\leq$,$>$)
  - Blätter sind resultierende Anordnung der Elemente

Komplexität von Sortierverfahren

Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

- **Sortierverfahren müssen Elemente vergleichen**
  - Sonst kann die Anordnung der Elemente nicht garantiert werden
  - Wieviel Vergleiche werden benötigt um $a_1, .., a_n$ zu ordnen?
  - Bestimme Anzahl der Vergleiche für den ungünstigsten Fall
- **Betrachte Entscheidungsbaum von Algorithmen**

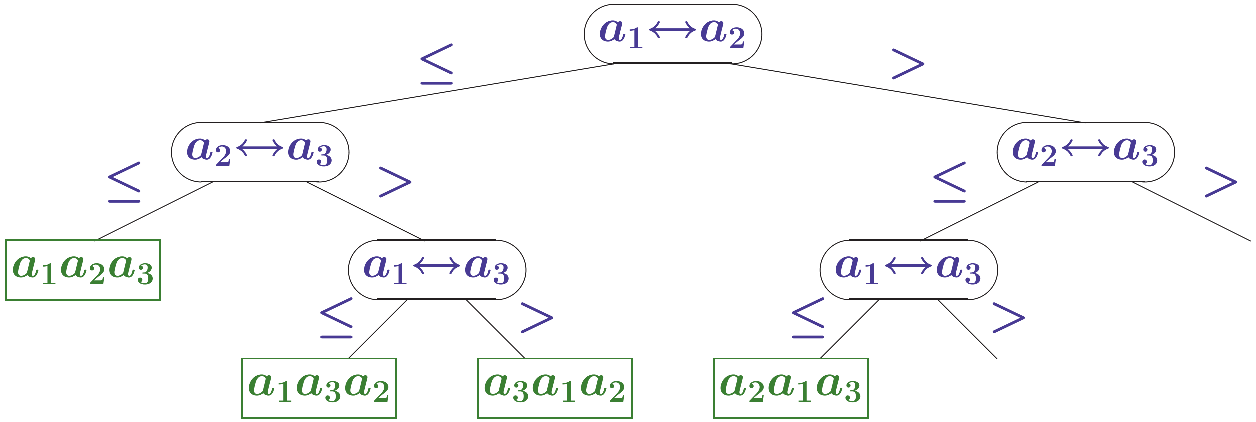


  - Innere Knoten entsprechen den durchgeführten Vergleichen
  - Kanten markiert mit Vergleichergebnis ($\leq$,$>$)
  - Blätter sind resultierende Anordnung der Elemente

Komplexität von Sortierverfahren

Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

- **Sortierverfahren müssen Elemente vergleichen**
  - Sonst kann die Anordnung der Elemente nicht garantiert werden
  - Wieviel Vergleiche werden benötigt um $a_1, .., a_n$ zu ordnen?
  - Bestimme Anzahl der Vergleiche für den ungünstigsten Fall
- **Betrachte Entscheidungsbaum von Algorithmen**

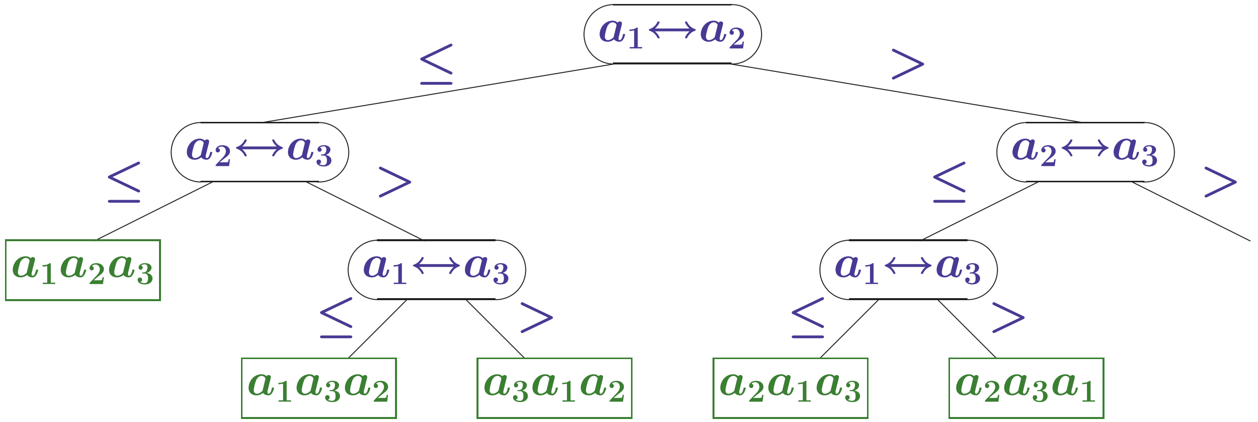


  - Innere Knoten entsprechen den durchgeführten Vergleichen
  - Kanten markiert mit Vergleichergebnis ($\leq$,$>$)
  - Blätter sind resultierende Anordnung der Elemente

Komplexität von Sortierverfahren

Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

- **Sortierverfahren müssen Elemente vergleichen**
  - Sonst kann die Anordnung der Elemente nicht garantiert werden
  - Wieviel Vergleiche werden benötigt um $a_1, .., a_n$ zu ordnen?
  - Bestimme Anzahl der Vergleiche für den ungünstigsten Fall
- **Betrachte Entscheidungsbaum von Algorithmen**

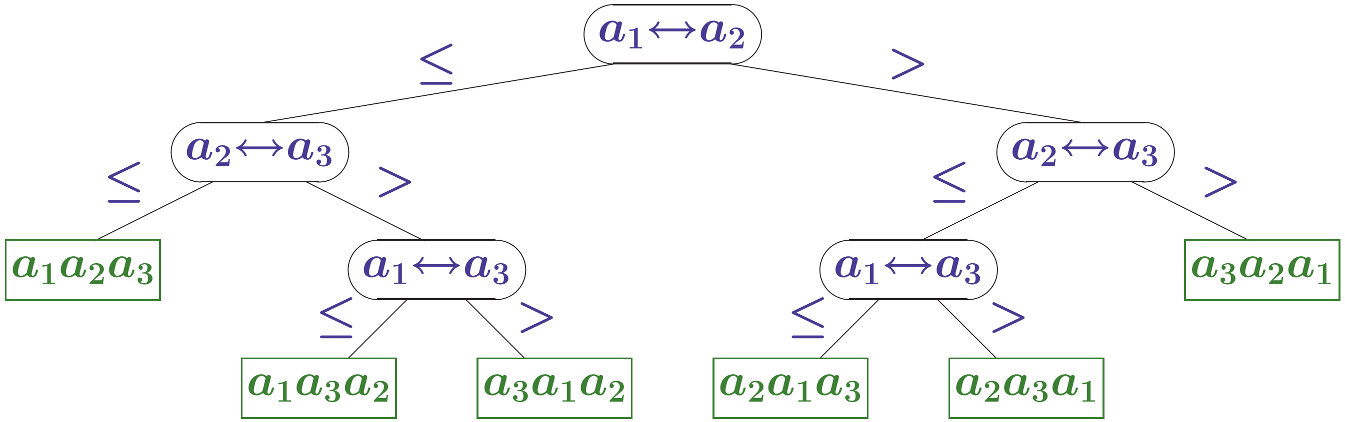


  - Innere Knoten entsprechen den durchgeführten Vergleichen
  - Kanten markiert mit Vergleichergebnis ($\leq$,$>$)
  - Blätter sind resultierende Anordnung der Elemente

Komplexität von Sortierverfahren

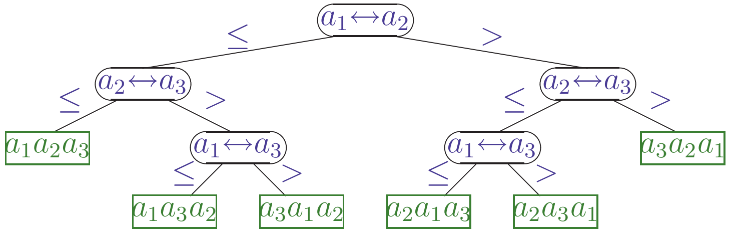
Geht es schneller als $\mathcal{O}(n * \log_2 n)$?

- **Sortierverfahren müssen Elemente vergleichen**
  - Sonst kann die Anordnung der Elemente nicht garantiert werden
  - Wieviel Vergleiche werden benötigt um $a_1, .., a_n$ zu ordnen?
  - Bestimme Anzahl der Vergleiche für den ungünstigsten Fall
- **Betrachte Entscheidungsbaum von Algorithmen**



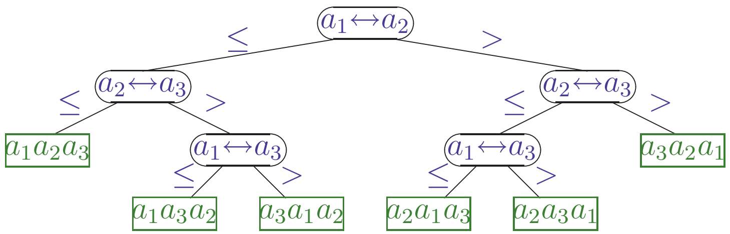
  - Innere Knoten entsprechen den durchgeführten Vergleichen
  - Kanten markiert mit Vergleichergebnis ($\leq$,$>$)
  - Blätter sind resultierende Anordnung der Elemente

Komplexität von Sortierverfahren (II)



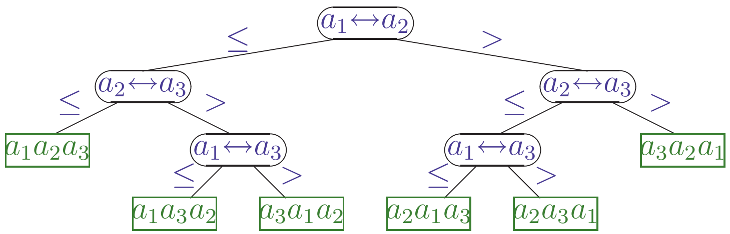
- **Algorithmen entsprechen Entscheidungsbäumen**
  – Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum

Komplexität von Sortierverfahren (II)



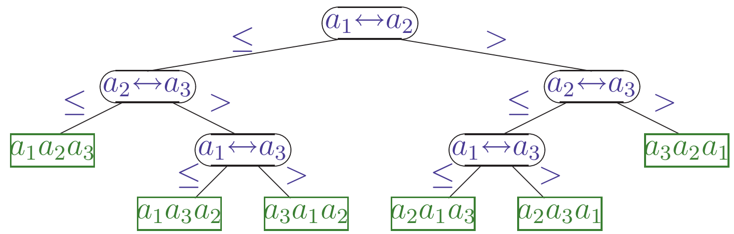
- **Algorithmen entsprechen Entscheidungsbäumen**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes

Komplexität von Sortierverfahren (II)



- **Algorithmen entsprechen Entscheidungsbäumen**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
  - Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes

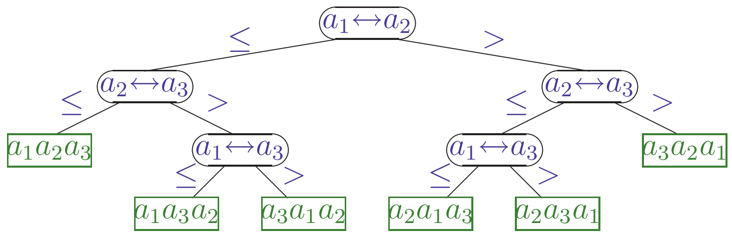
Komplexität von Sortierverfahren (II)



- **Algorithmen entsprechen Entscheidungsbäumen**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
  - Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes

  Komplexität von Sortieren $\equiv$ minimale Tiefe von Entscheidungsbäumen

Komplexität von Sortierverfahren (II)

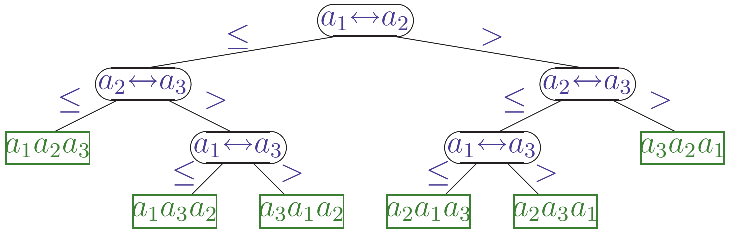


- **Algorithmen entsprechen Entscheidungsbäumen**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
  - Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes

  Komplexität von Sortieren $\equiv$ minimale Tiefe von Entscheidungsbäumen

- **Wie tief ist ein Entscheidungsbaum?**

Komplexität von Sortierverfahren (II)

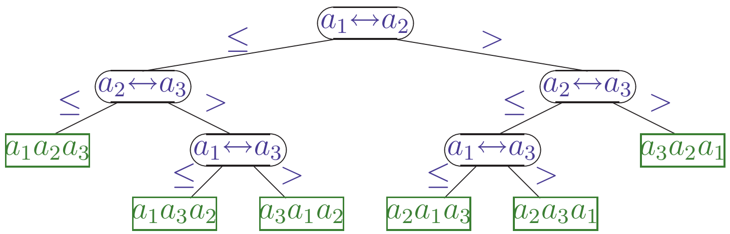


- **Algorithmen entsprechen Entscheidungsbäumen**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
  - Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes

  Komplexität von Sortieren $\equiv$ minimale Tiefe von Entscheidungsbäumen

- **Wie tief ist ein Entscheidungsbaum?**
  - Jeder Entscheidungsbaum für hat $a_1, .., a_n$ hat $n!$ Blätter

Komplexität von Sortierverfahren (II)

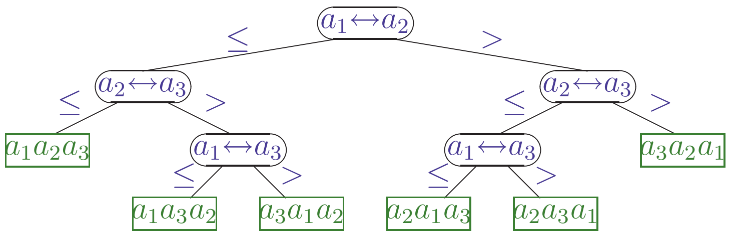


- **Algorithmen entsprechen Entscheidungsbäumen**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
  - Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes

  Komplexität von Sortieren $\equiv$ minimale Tiefe von Entscheidungsbäumen

- **Wie tief ist ein Entscheidungsbaum?**
  - Jeder Entscheidungsbaum für hat $a_1, .., a_n$ hat $n!$ Blätter
  - Ein binärer Baum der Tiefe k hat maximal 2^k Blätter

Komplexität von Sortierverfahren (II)

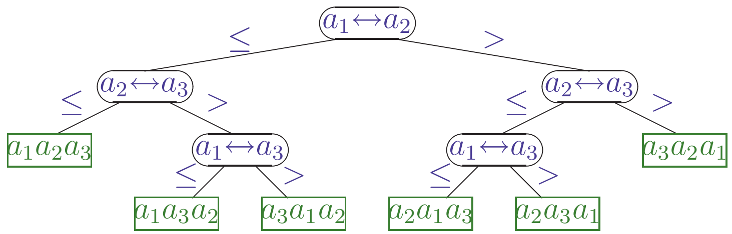


- **Algorithmen entsprechen Entscheidungsbäumen**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
  - Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes

  Komplexität von Sortieren $\equiv$ minimale Tiefe von Entscheidungsbäumen

- **Wie tief ist ein Entscheidungsbaum?**
  - Jeder Entscheidungsbaum für hat $a_1, .., a_n$ hat $n!$ Blätter
  - Ein binärer Baum der Tiefe k hat maximal 2^k Blätter
  - Jeder Entscheidungsbaum hat mindestens Tiefe $\log_2 n!$

Komplexität von Sortierverfahren (II)

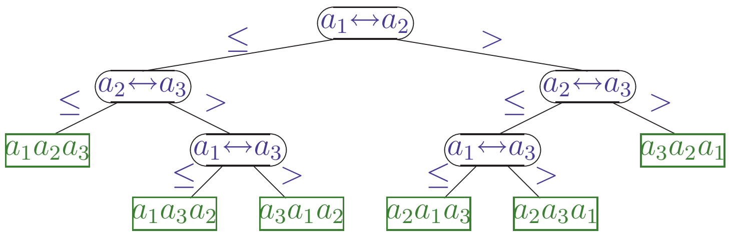


- **Algorithmen entsprechen Entscheidungsbäumen**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
  - Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes

  Komplexität von Sortieren $\equiv$ minimale Tiefe von Entscheidungsbäumen

- **Wie tief ist ein Entscheidungsbaum?**
  - Jeder Entscheidungsbaum für hat $a_1, .., a_n$ hat $n!$ Blätter
  - Ein binärer Baum der Tiefe k hat maximal 2^k Blätter
  - Jeder Entscheidungsbaum hat mindestens Tiefe $\log_2 n!$
  - $\log_2 n! = \log_2(\Pi_{i=1}^n i)$

Komplexität von Sortierverfahren (II)

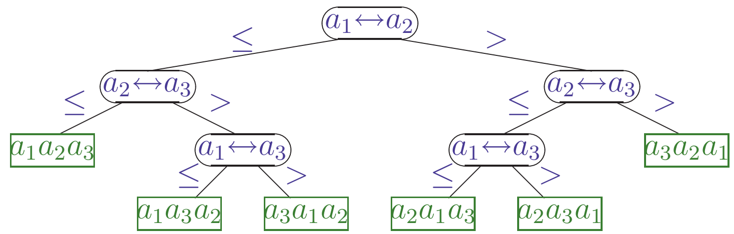


- **Algorithmen entsprechen Entscheidungsbäumen**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
  - Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes

  Komplexität von Sortieren $\equiv$ minimale Tiefe von Entscheidungsbäumen

- **Wie tief ist ein Entscheidungsbaum?**
  - Jeder Entscheidungsbaum für hat $a_1, .., a_n$ hat $n!$ Blätter
  - Ein binärer Baum der Tiefe k hat maximal 2^k Blätter
  - Jeder Entscheidungsbaum hat mindestens Tiefe $\log_2 n!$
  - $\log_2 n! = \log_2(\Pi_{i=1}^n i) = \Sigma_{i=1}^n \log_2 i$

Komplexität von Sortierverfahren (II)

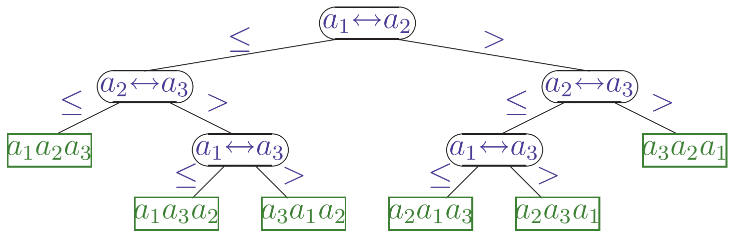


- **Algorithmen entsprechen Entscheidungsbäumen**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
  - Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes

  Komplexität von Sortieren $\equiv$ minimale Tiefe von Entscheidungsbäumen

- **Wie tief ist ein Entscheidungsbaum?**
  - Jeder Entscheidungsbaum für hat $a_1, .., a_n$ hat $n!$ Blätter
  - Ein binärer Baum der Tiefe k hat maximal 2^k Blätter
  - Jeder Entscheidungsbaum hat mindestens Tiefe $\log_2 n!$
  - $\log_2 n! = \log_2(\Pi_{i=1}^n i) = \Sigma_{i=1}^n \log_2 i \geq \Sigma_{i=n/2}^n \log_2(n/2)$

Komplexität von Sortierverfahren (II)

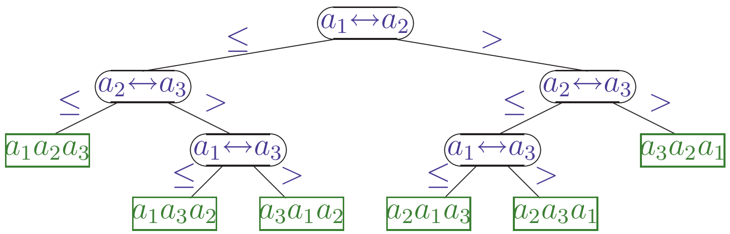


- **Algorithmen entsprechen Entscheidungsbäumen**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
  - Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes

  Komplexität von Sortieren $\equiv$ minimale Tiefe von Entscheidungsbäumen

- **Wie tief ist ein Entscheidungsbaum?**
  - Jeder Entscheidungsbaum für hat $a_1, .., a_n$ hat $n!$ Blätter
  - Ein binärer Baum der Tiefe k hat maximal 2^k Blätter
  - Jeder Entscheidungsbaum hat mindestens Tiefe $\log_2 n!$
  - $\log_2 n! = \log_2(\Pi_{i=1}^{n} i) = \Sigma_{i=1}^{n} \log_2 i \geq \Sigma_{i=n/2}^{n} \log_2(n/2) = n/2*(\log_2 n - 1)$

Komplexität von Sortierverfahren (II)



- **Algorithmen entsprechen Entscheidungsbäumen**
  - Abarbeitung für konkrete Eingaben entspricht einem Ast im Baum
  - Konkrete Laufzeit des Algorithmus entspricht Länge des Astes
  - Komplexität des Algorithmus entspricht Tiefe des Entscheidungsbaumes

  Komplexität von Sortieren $\equiv$ minimale Tiefe von Entscheidungsbäumen

- **Wie tief ist ein Entscheidungsbaum?**
  - Jeder Entscheidungsbaum für hat $a_1, .., a_n$ hat $n!$ Blätter
  - Ein binärer Baum der Tiefe k hat maximal 2^k Blätter
  - Jeder Entscheidungsbaum hat mindestens Tiefe $\log_2 n!$
  - $\log_2 n! = \log_2(\Pi_{i=1}^n i) = \Sigma_{i=1}^n \log_2 i \geq \Sigma_{i=n/2}^n \log_2(n/2) = n/2*(\log_2 n - 1)$

Sortieren ist in $\Omega(n * \log_2 n)$

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen**

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  - Einstellige Addition von rechts nach links mit Übertrag

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  - Einstellige Addition von rechts nach links mit Übertrag
- **Multiplikation n-stelliger Zahlen**

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  - Einstellige Addition von rechts nach links mit Übertrag
- **Multiplikation n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Jede Stelle muß mit jeder Stelle multipliziert werden

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  - Einstellige Addition von rechts nach links mit Übertrag
- **Multiplikation n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Jede Stelle muß mit jeder Stelle multipliziert werden
- **Division n-stelliger Zahlen**

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  - Einstellige Addition von rechts nach links mit Übertrag
- **Multiplikation n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Jede Stelle muß mit jeder Stelle multipliziert werden
- **Division n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Schriftliche Division bestimmt Ergebnis von links nach rechts

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  - Einstellige Addition von rechts nach links mit Übertrag
- **Multiplikation n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Jede Stelle muß mit jeder Stelle multipliziert werden
- **Division n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Schriftliche Division bestimmt Ergebnis von links nach rechts
- **Matrixmultiplikation $n \times n$-Matrizen**

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  - Einstellige Addition von rechts nach links mit Übertrag
- **Multiplikation n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Jede Stelle muß mit jeder Stelle multipliziert werden
- **Division n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Schriftliche Division bestimmt Ergebnis von links nach rechts
- **Matrixmultiplikation $n \times n$-Matrizen** $\mathcal{O}(n^3)$

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  - Einstellige Addition von rechts nach links mit Übertrag
- **Multiplikation n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Jede Stelle muß mit jeder Stelle multipliziert werden
- **Division n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Schriftliche Division bestimmt Ergebnis von links nach rechts
- **Matrixmultiplikation $n{\times}n$-Matrizen** $\mathcal{O}(n^3)$
- **Berechnung von $n!$**

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  - Einstellige Addition von rechts nach links mit Übertrag
- **Multiplikation n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Jede Stelle muß mit jeder Stelle multipliziert werden
- **Division n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Schriftliche Division bestimmt Ergebnis von links nach rechts
- **Matrixmultiplikation $n{\times}n$-Matrizen** $\mathcal{O}(n^3)$
- **Berechnung von $n!$** $\mathcal{O}(n^2 * (\log_2 n)^2)$
  - Obergrenze: n-fache Multiplikation von n und $n!$: $n * \log_2 n * \log_2(n^n)$
  - Untergrenze: $n/2$-fach $n/2*(n/2)!$: $n/2 * \log_2(n/2) * n/4*(\log_2 n - 2)$

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  - Einstellige Addition von rechts nach links mit Übertrag
- **Multiplikation n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Jede Stelle muß mit jeder Stelle multipliziert werden
- **Division n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Schriftliche Division bestimmt Ergebnis von links nach rechts
- **Matrixmultiplikation $n\times n$-Matrizen** $\mathcal{O}(n^3)$
- **Berechnung von $n!$** $\mathcal{O}(n^2 * (\log_2 n)^2)$
  - Obergrenze: n-fache Multiplikation von n und $n!$: $n * \log_2 n * \log_2(n^n)$
  - Untergrenze: $n/2$-fach $n/2*(n/2)!$: $n/2 * \log_2(n/2) * n/4*(\log_2 n - 2)$
- **Primzahltest bei n-stelliger Zahlen**

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  – Einstellige Addition von rechts nach links mit Übertrag
- **Multiplikation n-stelliger Zahlen** $\mathcal{O}(n^2)$
  – Jede Stelle muß mit jeder Stelle multipliziert werden
- **Division n-stelliger Zahlen** $\mathcal{O}(n^2)$
  – Schriftliche Division bestimmt Ergebnis von links nach rechts
- **Matrixmultiplikation $n{\times}n$-Matrizen** $\mathcal{O}(n^3)$
- **Berechnung von $n!$** $\mathcal{O}(n^2 * (\log_2 n)^2)$
  – Obergrenze: n-fache Multiplikation von n und $n!$: $n * \log_2 n * \log_2(n^n)$
  – Untergrenze: $n/2$-fach $n/2*(n/2)!$: $n/2 * \log_2(n/2) * n/4*(\log_2 n - 2)$
- **Primzahltest bei n-stelliger Zahlen** $\mathcal{O}(n^{12})$
  – AKS Algorithmus auf Basis tiefer mathematischer Einsichten (2002)

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  - Einstellige Addition von rechts nach links mit Übertrag
- **Multiplikation n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Jede Stelle muß mit jeder Stelle multipliziert werden
- **Division n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Schriftliche Division bestimmt Ergebnis von links nach rechts
- **Matrixmultiplikation $n{\times}n$-Matrizen** $\mathcal{O}(n^3)$
- **Berechnung von $n!$** $\mathcal{O}(n^2 * (\log_2 n)^2)$
  - Obergrenze: n-fache Multiplikation von n und $n!$: $n * \log_2 n * \log_2(n^n)$
  - Untergrenze: $n/2$-fach $n/2{*}(n/2)!$: $n/2 * \log_2(n/2) * n/4{*}(\log_2 n - 2)$
- **Primzahltest bei n-stelliger Zahlen** $\mathcal{O}(n^{12})$
  - AKS Algorithmus auf Basis tiefer mathematischer Einsichten (2002)
  - Alle früheren Verfahren waren exponentiell
  - Alle bekannten Faktorisierungsverfahren sind exponentiell

Komplexität anderer Problemstellungen

- **Addition n-stelliger Zahlen** $\mathcal{O}(n)$
  - Einstellige Addition von rechts nach links mit Übertrag
- **Multiplikation n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Jede Stelle muß mit jeder Stelle multipliziert werden
- **Division n-stelliger Zahlen** $\mathcal{O}(n^2)$
  - Schriftliche Division bestimmt Ergebnis von links nach rechts
- **Matrixmultiplikation $n\times n$-Matrizen** $\mathcal{O}(n^3)$
- **Berechnung von $n!$** $\mathcal{O}(n^2 * (\log_2 n)^2)$
  - Obergrenze: n-fache Multiplikation von n und $n!$: $n * \log_2 n * \log_2(n^n)$
  - Untergrenze: $n/2$-fach $n/2*(n/2)!$: $n/2 * \log_2(n/2) * n/4*(\log_2 n - 2)$
- **Primzahltest bei n-stelliger Zahlen** $\mathcal{O}(n^{12})$
  - AKS Algorithmus auf Basis tiefer mathematischer Einsichten (2002)
  - Alle früheren Verfahren waren exponentiell
  - Alle bekannten Faktorisierungsverfahren sind exponentiell
  - Ergebnis gut für offene kryptographische Systeme (wähle $n > 200$)