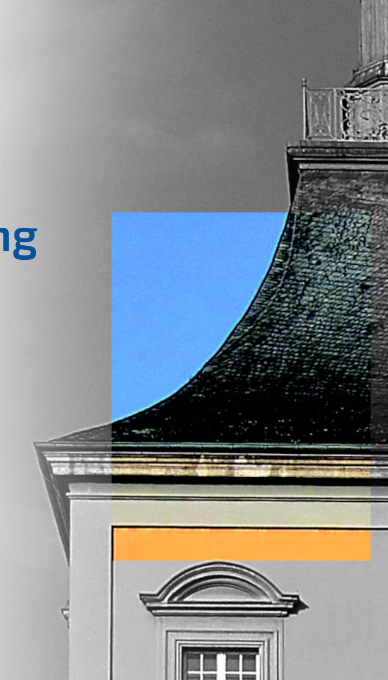
Algorithmen und Programmierung

Algorithmen I

Dr. Felix Jonathan Boes
boes@cs.uni-bonn.de
Institut für Informatik



KURZE WIEDERHOLUNG

Sortieren, Rekursion und Landausymbole

Sortierverfahren - Min Sort

Min Sort

Wir haben Varianten des Algorithmus Min Sort betrachten, bei denen sukzessive das aktuell kleinste Element eines Arrays gefunden wird, um anschließend nach vorn sortiert zu werden.

Dazu haben wir verstanden wie man Algorithmen notiert und nachweist, dass ein gegebenes Verfahren ein Algorithmus ist.

Wir haben die Beweistechnik der **Invarianten** kennen gelernt.

Sortieren, Rekursion und Landausymbole

Rekursion und Divide-And-Conquer-Verfahren

Rekursion und Divide-And-Conquer

Als **Rekursion** bezeichnet man die Tatsache, dass ein Objekte sich selbst als (verkleinerten) Bestandteil enthält oder dass ein Verfahren mithilfe seiner selbst definiert ist.

Es gibt Probleme die sich in selbstähnliche, kleinere Teilprobleme zerlegen lassen. Da die Teilprobleme ähnlich zum eigentlichen Problem sind, können wir diese immer weiter in kleinere, selbstähnliche Teilprobleme zerlegen. Diese Lösungsstrategie nennen wir **Divide-And-Conquer**. Algorithmen die sich diese Strategie zunutze machen, nennen wir **Divide-And-Conquer-Verfahren**.

Sortieren, Rekursion und Landausymbole

Sortierverfahren - Merge Sort

Merge Sort

Merge Sort ist ein Divide-And-Conquer-Sortierverfahren und besteht aus zwei Funktionen:

Die Funktion `mergesort` die das Problem rekursiv (conquer) in kleinere Teilprobleme zerlegt (divide) und die Funktion `merge` die zwei gegebene, sortierte Arrays in ein Ziel Array schreibt (combine).

Sortieren, Rekursion und Landausymbole

Landau Symbole

Offene Frage

Wie vergleicht man die Effizienz von Algorithmen?

Wie vergleicht man die Effizienz unabhängig von Implementierung und Hardware?

Wie vergleicht man die Effizienz bei wachsendem Input?

Motivation I

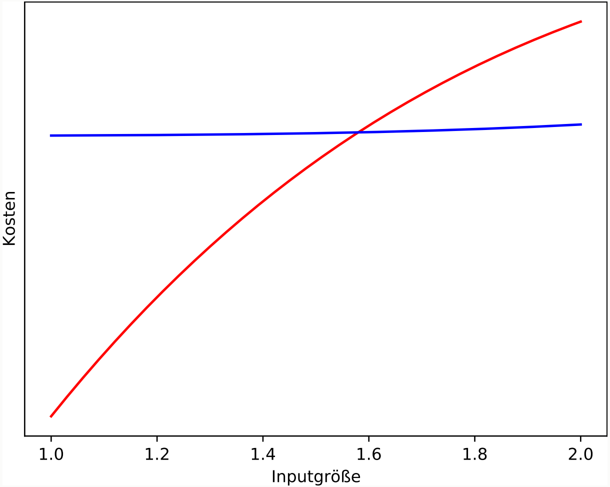
Wir wollen wissen wie „teuer“ zwei Algorithmen im Vergleich zueinander sind. Dabei wollen wir diesen Vergleich ziehen und dabei unabhängig sein von einer konkreten Implementierung die auf einer konkreten Hardware ausgeführt wird.

Um klarer zu benennen was es heißt „teuer“ zu sein, müssen wir eine Ressource auswählen und den Verbrauch dieser Ressource bestimmen. In diesem Kontext sind typische Ressourcen:

- Die Anzahl von elementaren Anweisungen
- Die Anzahl von getätigten Vergleichen bei Sortierverfahren
- Die maximale Menge von benötigtem Speicher während das Verfahrens ausgeführt wird.

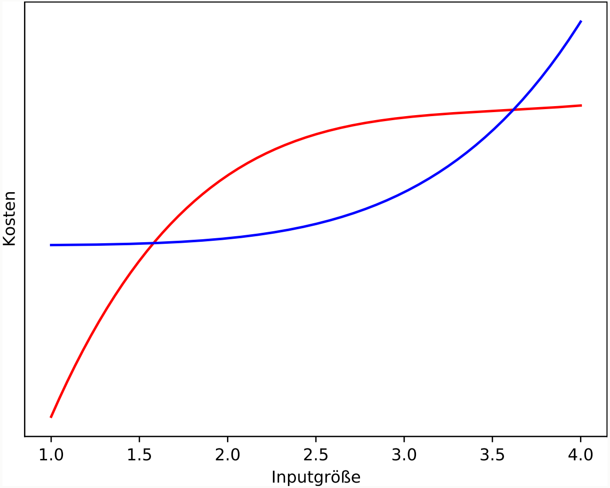
Motivation II

Wir betrachten die Kosten von zwei Verfahren. Welches Verfahren ist teurer?



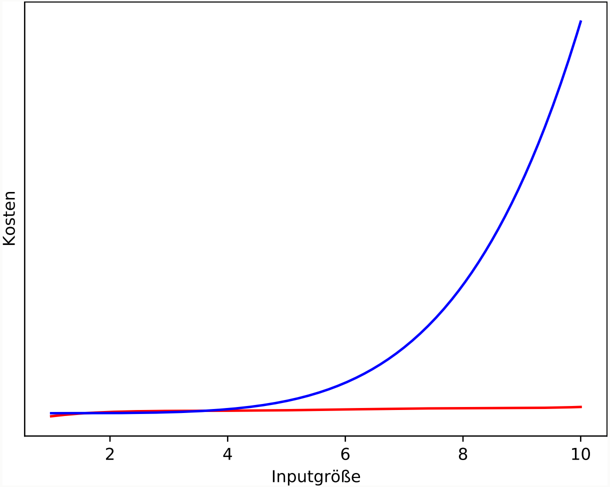
Motivation II

Wir betrachten die Kosten von zwei Verfahren. Welches Verfahren ist teurer?



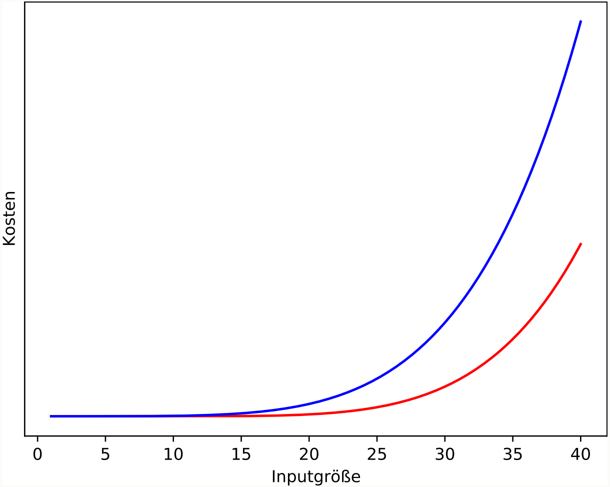
Motivation II

Wir betrachten die Kosten von zwei Verfahren. Welches Verfahren ist teurer?



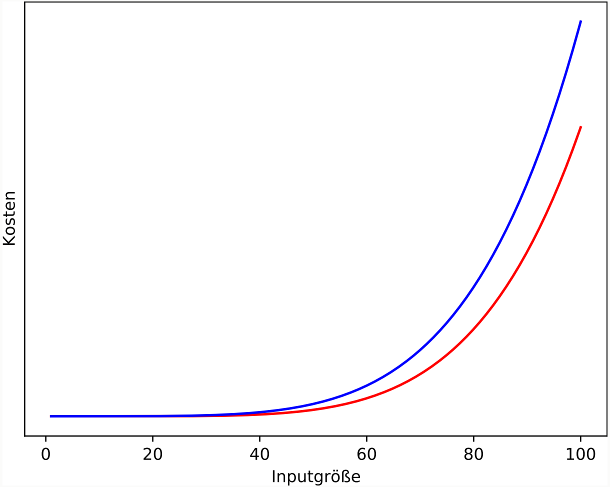
Motivation II

Wir betrachten die Kosten von zwei Verfahren. Welches Verfahren ist teurer?



Motivation II

Wir betrachten die Kosten von zwei Verfahren. Welches Verfahren ist teurer?



Motivation III

Um zu entscheiden welches Verfahren teurer ist wollen wir nur eine Entscheidung treffen. Eine Entscheidung pro Inputgröße zu treffen macht den Vergleich zu kompliziert.

Wir fragen uns also wie sich der Preis über die Zeit entwickelt. Dazu betrachten wir (Kosten)funktionen $f : \mathbb{N} \to \mathbb{R}$ die von einer **Inputgröße** n abhängen und einen (Kosten)wert $f(n)$ produziert.

Asymptotisch gleichen Funktionen I

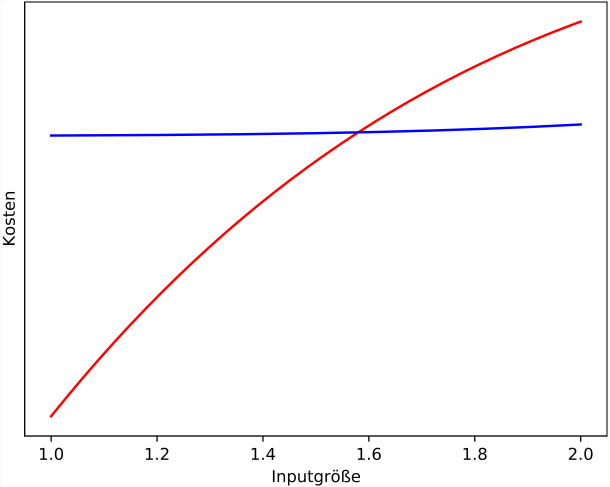
Wir konzentrieren uns auf das „allgemeine Wachstumsverhalten“ von zwei (Kosten)funktionen. Genauer sind wir daran interessiert, ob sich das allgemeine Wachstumsverhalten von zwei (Kosten)funktionen stärker als um einen konstanten, multiplikativen Faktor unterscheidet.

Wenn sich das allgemeine Wachstumsverhalten von zwei (Kosten)funktionen nur um einen konstanten, multiplikativen Faktor unterscheidet nennen wir sie **asymptotisch gleich**.

Asymptotisch gleichen Funktionen I

Wir konzentrieren uns auf das „allgemeine Wachstumsverhalten“ von zwei (Kosten)funk-
tionen. Genauer s … ne Wachstumsver-
halten von zwei (K … en, multiplikativen
Faktor unterscheid …

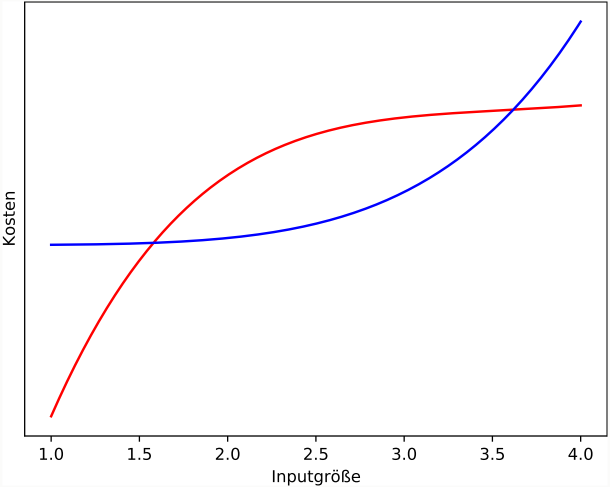
Wenn sich das allg … unktionen nur um
einen konstanten, … r sie **asymptotisch**
gleich.



Asymptotisch gleichen Funktionen I

Wir konzentrieren uns auf das „allgemeine Wachstumsverhalten“ von zwei (Kosten)funk-
tionen. Genauer s … ne Wachstumsver-
halten von zwei (K … en, multiplikativen
Faktor unterscheid …

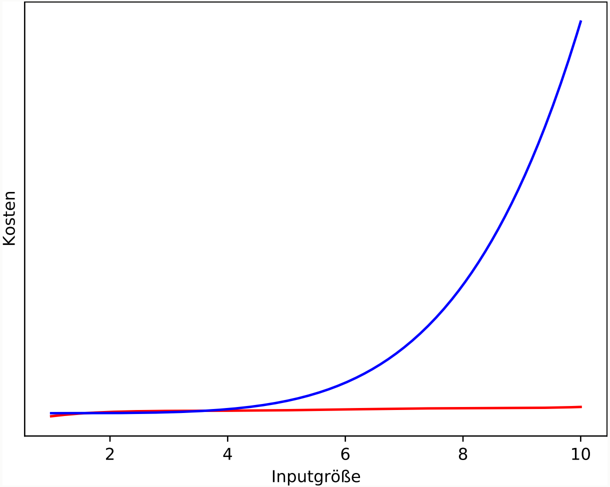
Wenn sich das allg … unktionen nur um
einen konstanten, … r sie **asymptotisch**
gleich.



Asymptotisch gleichen Funktionen I

Wir konzentrieren uns auf das „allgemeine Wachstumsverhalten“ von zwei (Kosten)funktionen. Genauer s ne Wachstumsverhalten von zwei (K en, multiplikativen Faktor unterscheid

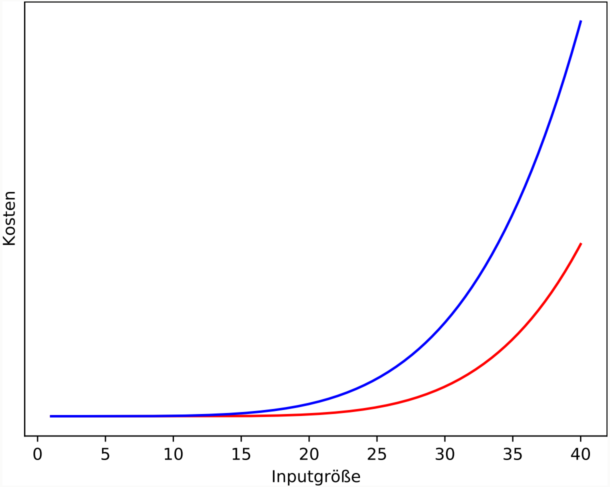
Wenn sich das allg Funktionen nur um einen konstanten, r sie **asymptotisch gleich**.



Asymptotisch gleichen Funktionen I

Wir konzentrieren uns auf das „allgemeine Wachstumsverhalten“ von zwei (Kosten)funk-
tionen. Genauer s … ne Wachstumsver-
halten von zwei (K … en, multiplikativen
Faktor unterscheid …

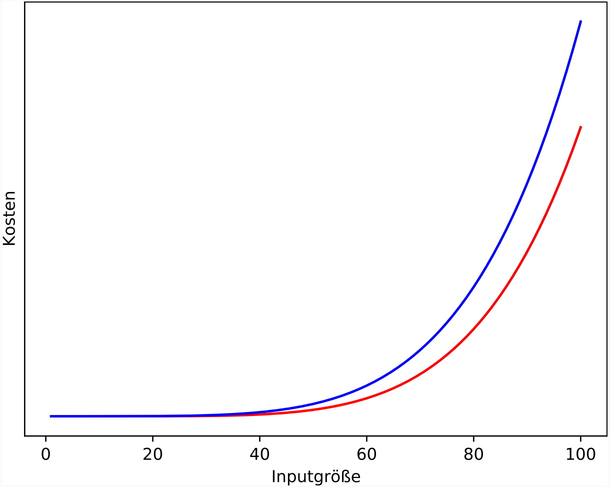
Wenn sich das allg … funktionen nur um
einen konstanten, … r sie **asymptotisch
gleich**.



Asymptotisch gleichen Funktionen I

Wir konzentrieren uns auf das „allgemeine Wachstumsverhalten“ von zwei (Kosten)funk-
tionen. Genauer s … ne Wachstumsver-
halten von zwei (K … en, multiplikativen
Faktor unterscheid …

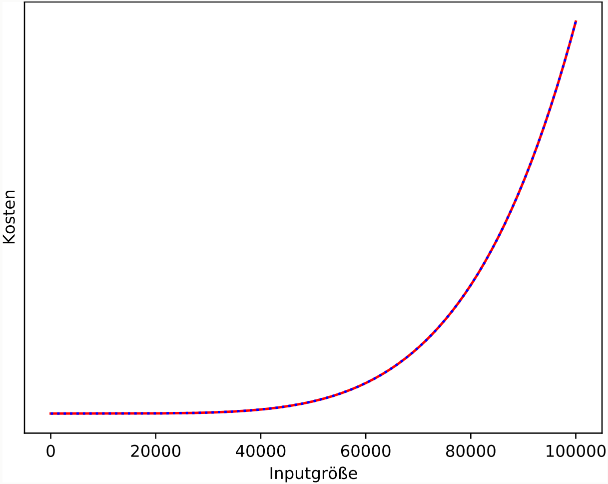
Wenn sich das allg … funktionen nur um
einen konstanten, … r sie **asymptotisch
gleich**.



Asymptotisch gleichen Funktionen I

Wir konzentrieren uns auf das „allgemeine Wachstumsverhalten“ von zwei (Kosten)funk-
tionen. Genauer s ne Wachstumsver-
halten von zwei (K en, multiplikativen
Faktor unterscheid

Wenn sich das allg funktionen nur um
einen konstanten, r sie **asymptotisch gleich**.



Asymptotisch gleichen Funktionen II

Wenn wir eine (Kosten)funktion g fixieren, dann ist die Menge der zu g asymptotisch gleichen Funktionen

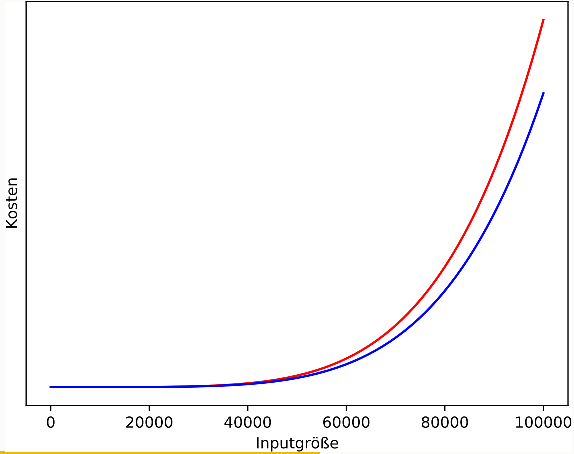
$$\Theta(g) = \{f : \mathbb{N} \to \mathbb{R} \mid f \text{ und } g \text{ sind asymptotisch gleich}\}$$

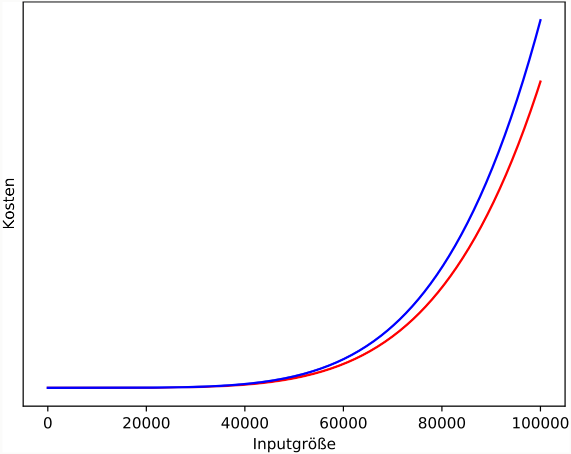
Formalisiert man diese Idee, dann sind zwei Funktionen asymptotisch gleich, wenn es einen Zeitpunkt n_0 und positive Konstanten c_1, c_2 gibt sodass für alle Zeitpunkte $n > n_0$ gilt:

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Asymptotisch gleichen Funktionen III

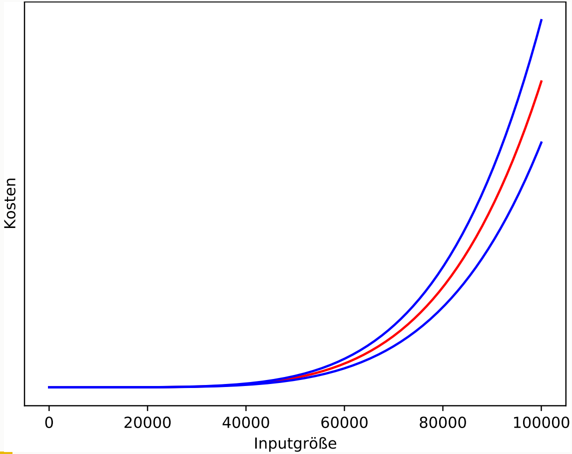
Die Ungleichung $0 \leq c_1 \cdot g(n) \leq f(n)$ sperrt f von unten ein (nach Zeitpunkt n_0). Die Ungleichung $0 \leq f(n) \leq c_2 \cdot g(n)$ sperrt f von oben ein (nach Zeitpunkt n_0).





Asymptotisch gleichen Funktionen III

Die Ungleichung $0 \leq c_1 \cdot g(n) \leq f(n)$ sperrt f von unten ein (nach Zeitpunkt n_0). Die Ungleichung $0 \leq f(n) \leq c_2 \cdot g(n)$ sperrt f von oben ein (nach Zeitpunkt n_0).



Asymptotisch obere Schranken I

Wenn Sie untersuchen wollen wie teuer ein Verfahren im schlechtesten Fall ist dann geben Sie eine **asymptotisch obere Schranke** g an. Diese Schranke ist im besten Fall sehr klein (in dem Sinne dass jede kleinere, asymptotisch obere Schranke g' schon asymptotisch gleich ist zu g).

Asymptotisch obere Schranken II

Wenn wir eine (Kosten)funktion g fixieren, dann ist die Menge der Funktionen für die g eine asymptotisch obere Schranke ist

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} \mid g \text{ ist asymptotisch obere Schrank von } f\}$$

Formalisiert man diese Idee, dann ist g eine asymptotisch obere Schranke von f, wenn es einen Zeitpunkt n_0 und eine positive Konstanten c gibt sodass für alle Zeitpunkte $n > n_0$ gilt:

$$0 \leq f(n) \leq c \cdot g(n)$$

Asymptotisch untere Schranken I

Wenn Sie untersuchen wollen wie teuer ein Verfahren mindestens ist dann geben Sie eine **asymptotisch untere Schranke** g an. Diese Schranke ist im besten Fall sehr groß (in dem Sinne dass jede größere, asymptotisch untere Schranke g' schon asymptotisch gleich ist zu g).

Asymptotisch untere Schranken II

Wenn wir eine (Kosten)funktion g fixieren, dann ist die Menge der Funktionen für die g eine asymptotisch untere Schranke ist

$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} \mid g \text{ ist asymptotisch untere Schrank von } f\}$$

Formalisiert man diese Idee, dann ist g eine asymptotisch untere Schranke von f, wenn es einen Zeitpunkt n_0 und eine positive Konstanten c gibt sodass für alle Zeitpunkte $n > n_0$ gilt:

$$0 \leq c \cdot g(n) \leq f(n)$$

Zusammenfassung

Es ist $f \in \Theta(g)$, also g asymptotisch gleich zu f, genau dann wenn es einen Zeitpunkt n_0 und positive Konstanten c_1, c_2 gibt sodass

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ für alle } n > n_0$$

Es ist $f \in \mathcal{O}(g)$, also g asymptotisch obere Schranke zu f, genau dann wenn es einen Zeitpunkt n_0 und eine positive Konstante c gibt sodass

$$0 \leq f(n) \leq c \cdot g(n) \text{ für alle } n > n_0$$

Es ist $f \in \Omega(g)$, also g asymptotisch untere Schranke zu f, genau dann wenn es einen Zeitpunkt n_0 und eine positive Konstante c gibt sodass

$$0 \leq c \cdot g(n) \leq f(n) \text{ für alle } n > n_0$$

Notation

In der Community hat sich folgende Notation durchgesetzt (ob es uns passt oder nicht).

- $f = \Theta(g)$ statt $f \in \Theta(g)$
- $f = \mathcal{O}(g)$ statt $f \in \mathcal{O}(g)$
- $f = \Omega(g)$ statt $f \in \Omega(g)$

Beispiele I

Der Umgang mit asymptotischem Wachstum muss eingeübt werden. Wir geben hier einige Beispiele an. Arbeiten Sie diese Beispiele als Übung nach.

- $42 = \Theta(1)$ also wächst 42 asymptotisch gleich schnell wie 1
- Für beliebige positive Funktionen f und positive Konstanten c ist $c \cdot f = \Theta(f)$
- $42 \cdot n + 11 = \Theta(n)$ also wächst $42 \cdot n + 11$ asymptotisch gleich schnell wie n
- Für ein Polynom $f(n) = a_k \cdot n^k + \ldots + a_0$ mit $a_k > 0$ ist $f = \Theta(n^k)$. Also wächst ein Polynom von Grad k asymptotisch so schnell wie n^k.
- $n^k = \mathcal{O}(n^{k+1})$ aber $n^k \neq \Theta(n^{k+1})$. Also ist n^{k+1} eine asymptotisch obere Grenze von n^k aber n^k wächst asymptotisch nicht so schnell wie n^{k+1}.
- $n + \log(n) = \Theta(n)$ also wächst der Logarithmus langsamer als linear
- $n \cdot \log(n) \neq \mathcal{O}(n)$ also wächst der Logarithmus schneller als eine konstante Funktion

Beispiele II

Die in dieser Vorlesung wichtigsten, vorkommenden Größenordnungen sind:

$$\mathcal{O}(1) \subsetneq \mathcal{O}(\log n) \subsetneq \mathcal{O}(n) \subsetneq \mathcal{O}(n \log n) \subsetneq \mathcal{O}(n^2) \subsetneq \mathcal{O}(n^3) \subsetneq \mathcal{O}(2^n)$$

UNIVERSITÄT BONN

Min Sort (out of place) Laufzeitabschätzung

Ziel ist es eine (möglichst kleine) asymptotisch obere Schranke für Min Sort (out of place) zu finden.

```
1 Erstelle neues Array 𝔹 = [B_0, B_1, ..., B_{n-1}] der Länge n
2 Def. (Multi)menge 𝕊 = {A_0, ..., A_{n-1}}
3 for i = 0, ..., n - 1 do
4     Finde A_k ∈ 𝕊 minimal
5     Entferne A_k aus 𝕊 und platziere es in B_i.
6 return 𝔹
```

Für Zeile 1, 2 und 5 sind keine Vergleiche nötig. Es gibt insgesamt n Schleifendurchläufe. Pro Schleifendurchlauf muss in Zeile 4 ein minimales Element gefunden werden. Das ist z.B. möglich, indem durch die Elemente iteriert wird um den Index des kleinsten Elements zu bestimmen. Dazu sind höchstens $n-1$ Vergleiche nötig. Insgesamt sind somit höchstens $n(n-1) \leq n^2$ Vergleiche nötig. Somit liegt die Anzahl der benötigten Vergleiche von Min Sort (out of place) in $\mathcal{O}(n^2)$.

Min Sort (out of place) Speicherabschätzung

Ziel ist es eine (möglichst kleine) asymptotisch obere Schranke für Min Sort (out of place) zu finden.

```
1 Erstelle neues Array 𝔹 = [B_0, B_1, ..., B_{n-1}] der Länge n
2 Def. (Multi)menge 𝕊 = {A_0, ..., A_{n-1}}
3 for i = 0, ..., n - 1 do
4     Finde A_k ∈ 𝕊 minimal
5     Entferne A_k aus 𝕊 und platziere es in B_i.
6 return 𝔹
```

Lokale Variablen die eine Größe unabhängig von n haben, verbrauchen insg. c viel Speicher. In Zeilen 1 und 2 werden zwei Variablen angelegt, die jeweils n Objekte speichern. Wir wissen noch nicht wie groß der Speicherbedarf von Mengen ist aber wir haben in der vergangenen Vorlesung eine Möglichkeit gesehen das Problem mit Speicheraufwand n zu lösen. Also benötigt man $2 \cdot n \cdot (\text{Gr. eines Objekts}) + c$ viel Speicher. Somit liegt der Speicherverbrauch von Min Sort (out of place) in $\mathcal{O}(2 \cdot n \cdot G + c) = \mathcal{O}(n)$.

UNIVERSITÄT BONN

Laufzeit- und Speicherverbrauch von Mergesort

Algorithm: mergesort

Eingabe : Array $\mathbb{A} = [A_0, \ldots, A_{k-1}]$ der Länge k

Ausgabe : Sortierte Kopie von $\mathbb{A}$

return `mergesort(`$\mathbb{A}$`)`

Funktion `mergesort(`$\mathbb{X} = [X_0, \ldots, X_{k-1}]$`)`
- **if** $k > 1$ **then**
  - Erstelle Ganzzahl $t = \lfloor k/2 \rfloor$
  - $\mathbb{L} = [X_0, \ldots, X_t]$
  - $\mathbb{R} = [X_{t+1}, \ldots, X_{k-1}]$
  - $\mathbb{L}_{\text{sort}}$ = `mergesort(`$\mathbb{L}$`)`
  - $\mathbb{R}_{\text{sort}}$ = `mergesort(`$\mathbb{R}$`)`
  - **return** `merge(`$\mathbb{L}_{sort}, \mathbb{R}_{sort}$`)`
- **else**
  - **return** $\mathbb{X}$

Algorithm: mergesort (Forts.)

Funktion `merge(`$\mathbb{L} = [L_0, \ldots, L_{l-1}], \mathbb{R} = [R_0, \ldots, R_{r-1}]$`)`
- Erstelle neues Array $\mathbb{Y}$ der Länge $l + r$
- Erstelle Ganzzahlen $i = 0$ und $j = 0$
- **while** $i < l$ *und* $j < r$ **do**
  - **if** $L_i \leq R_j$ **then**
    - $Y_{i+j} = L_i$ und i++
  - **else**
    - $Y_{i+j} = R_j$ und j++
- **while** $i < l$ **do**
  - $Y_{i+j} = L_i$ und i++
- **while** $j < r$ **do**
  - $Y_{i+j} = R_j$ und j++
- **return** $\mathbb{Y}$

Die Laufzeit von `merge(`$[L_0, \ldots, L_{l-1}], [R_0, \ldots, R_{r-1}]$`)` liegt in $\mathcal{O}(l + r)$. Der Speicherverbrauch von `merge` liegt ebenfalls in $\mathcal{O}(l + r)$.

Die **Rekusrsiontiefe** von `mergesort` beträgt $\log_2(n)$, denn in jedem Schritt wird die Länge der Inputarrays halbiert.

Laufzeit- und Speicherverbrauch von Mergesort

Wir schätzen die Laufzeit und den Speicherverbrauch wie folgt ab. In der ersten Rekursionstiefe rufen wir zweimal `merge` auf mit Arrays die maximal $\frac{n}{2}$ Element enthalten. In der zweiten Rekursionstiefe rufen wir doppelt so oft `merge` auf mit Arrays die maximal halb so groß sind. Das sind in der zweiten Rekursionstiefe vier Aufrufe mit Arrays die nicht größer sind als $\frac{n}{4}$. Insgesamt ist die Rekursionstiefe höchstens $\lceil \log_2(n) \rceil$. Das ergibt somit

$$
\begin{aligned}
&\frac{n}{2} + \frac{n}{2} \\
&+ \frac{n}{4} + \frac{n}{4} + \frac{n}{4} + \frac{n}{4} \\
&+ \frac{n}{8} + \frac{n}{8} + \frac{n}{8} + \frac{n}{8} + \frac{n}{8} + \frac{n}{8} + \frac{n}{8} + \frac{n}{8} \\
&+ \ldots \\
&= n + n + n + \ldots = \lceil \log_2(n) \rceil \cdot n
\end{aligned}
$$

Kosten der Sortieralgorithmen I

Wie teuer sind Sortierverfahren im schlimmsten Fall? Genauer:

- Wie viele Vergleiche sind maximal nötig?
- Wie viel Speicherplatz ist maximal für die Bearbeitung (zusätzlich) nötig?

Verfahren	Anzahl Vergleiche	Zusätzlicher Speicherbedarf
Min Sort out of place	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Min Sort in place	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort (unoptimiert)	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$
Merge Sort (optimiert)	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n)$

Kosten der Sortieralgorithmen II

Wie teuer sind Sortierverfahren mindestens? Genauer:

- Wie viele Vergleiche sind minimal nötig (bei variablem Input)?
- Wie viel Speicherplatz ist mindestens für die Bearbeitung (zusätzlich) nötig?

Mit geschicktem Knobeln und Wissen über den Logarithmus, findet man die asymptotisch untere Schranke $n \log n$. Daraus folgt, dass die jedes Sortierverfahren mindestens in $\Omega(n \log n)$ liegen. Wir wissen aber auch, dass Merge Sort in $\mathcal{O}(n \log n)$ liegt. Also benötigen die günstigsten Sortieralgorithmen $\Theta(n \log n)$ viele Vergleiche.

Wir haben gesehen, dass Min Sort in place höchstens $\mathcal{O}(1)$ Speicher verbraucht. Also verbrauchen die günstigsten Sortieralgorithmen $\Theta(1)$ viel Speicher.

Ausblick

Die Laufzeitanalyse von Algorithmen ist ein sehr wichtiges Teilgebiet der theoretischen Informatik. Sie werden in der Vorlesung **Algorithmen und Berechnungskomplexität** und den Folgemodulen tiefer in die Materie einsteigen.

Aber auch in anderen Vorlesungen werden Sie hier und da die Laufzeit oder den Speicherverbrauch von Algorithmen asymptotisch abschätzen.

Haben Sie Fragen?

Zusammenfassung

Mit Landausymbole vergleicht man die Effizienz von Algorithmen bei wachsendem Input

Der Vergleich ist unabhängig von der Implementierung und Hardware

Rückblick und Ausblick

Zusammenfassung

Wir haben unser erstes Ziel erreicht

Sie haben die grundlegenden Werkzeuge in Theorie und Praxis erlernt um über Sortieralgorithmen zu sprechen und um diese zu implementieren

Zusammenfassung

Wir fassen die vergangenen Inhalte kurz zusammen

Stellen Sie Ihre Fragen vergangenen Inhalt hier, in Ihrer Lerngruppe, in der Lernbetreuung und in den Übungsgruppen

Bishere und kommende Inhalte

- ✓ Einleitung
- ✓ Motivation: Theorie und Praxis um Sortierverfahren zu studieren
  - ✓ Imperative Programmierung (in C++)
  - ✓ Algorithmen vollständig beschreiben
  - ✓ Speicherverbrauch und Laufzeiten analysieren
- → Motivation: Theorie und Praxis um Graphen zu modellieren
  - Graphen und Algorithmen auf Graphen
  - Abstrakte Datentypen und Datenstrukturen
  - Objektorientierte Programmierung (in C++)

Haben Sie Fragen?