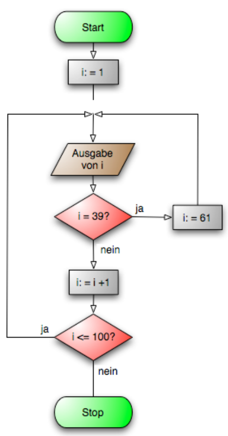
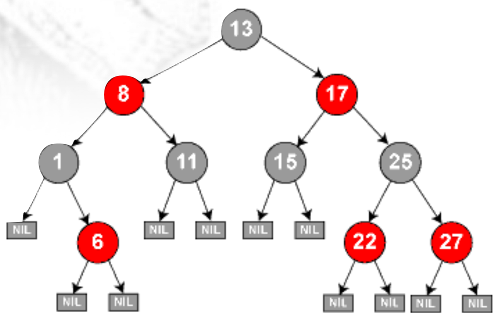
FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Algorithmen und Datenstrukturen





Version 1.26 vom 22, April 2021

Fachhochschule Aachen

Lehrgebiet für Angewandte Informatik

Prof. Dr. rer. nat. Jörg Striegnitz

1. Grundlagen

1. Grundlagen

1.14. **Vergleich von Algorithmen**
1.14.1. Einführung
1.14.2. Die Landau-Notation

1.15. Rekursionsgleichungen
1.15.1. Einfache Strategien
1.15.2. Das *Master-Theorem*
1.15.3. Erzeugende Funktionen

1. Grundlagen

1. Grundlagen

1.14. Vergleich von Algorithmen
1.14.1. Einführung
1.14.2. Die Landau-Notation

1.15. Rekursionsgleichungen
1.15.1. Einfache Strategien
1.15.2. Das *Master-Theorem*
1.15.3. Erzeugende Funktionen

Effizienz von Algorithmen

Wiederholung!

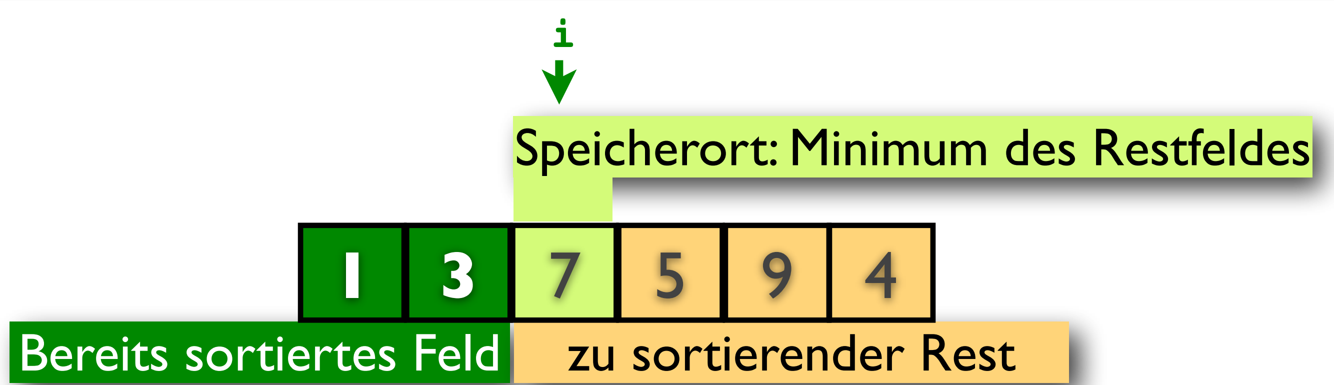
- **Kategorien**
  - Rechenzeit / Anzahl der atomaren Rechenschritte
  - Speicherplatzbedarf
  - Anzahl der Zugriffe auf Sekundärspeicher (z.B. Festplatten, Bänder)
  - Kommunikationsaufwand (verteilte Algorithmen)

- **Komplexität**
  - abhängig von Eingabedaten (Größe / Anzahl der Elemente)
  - **Platzkomplexität**
    - RAM ist begrenzt (und teuer)
    - Platz kann auf Kosten von Laufzeit eingespart werden
  - **Laufzeitkomplexität**
    - Ergebnisse sollten rechtzeitig vorliegen
    - Laufzeit kann auf Kosten von Platz eingespart werden
  - üblicherweise **asymptotische** Betrachtung

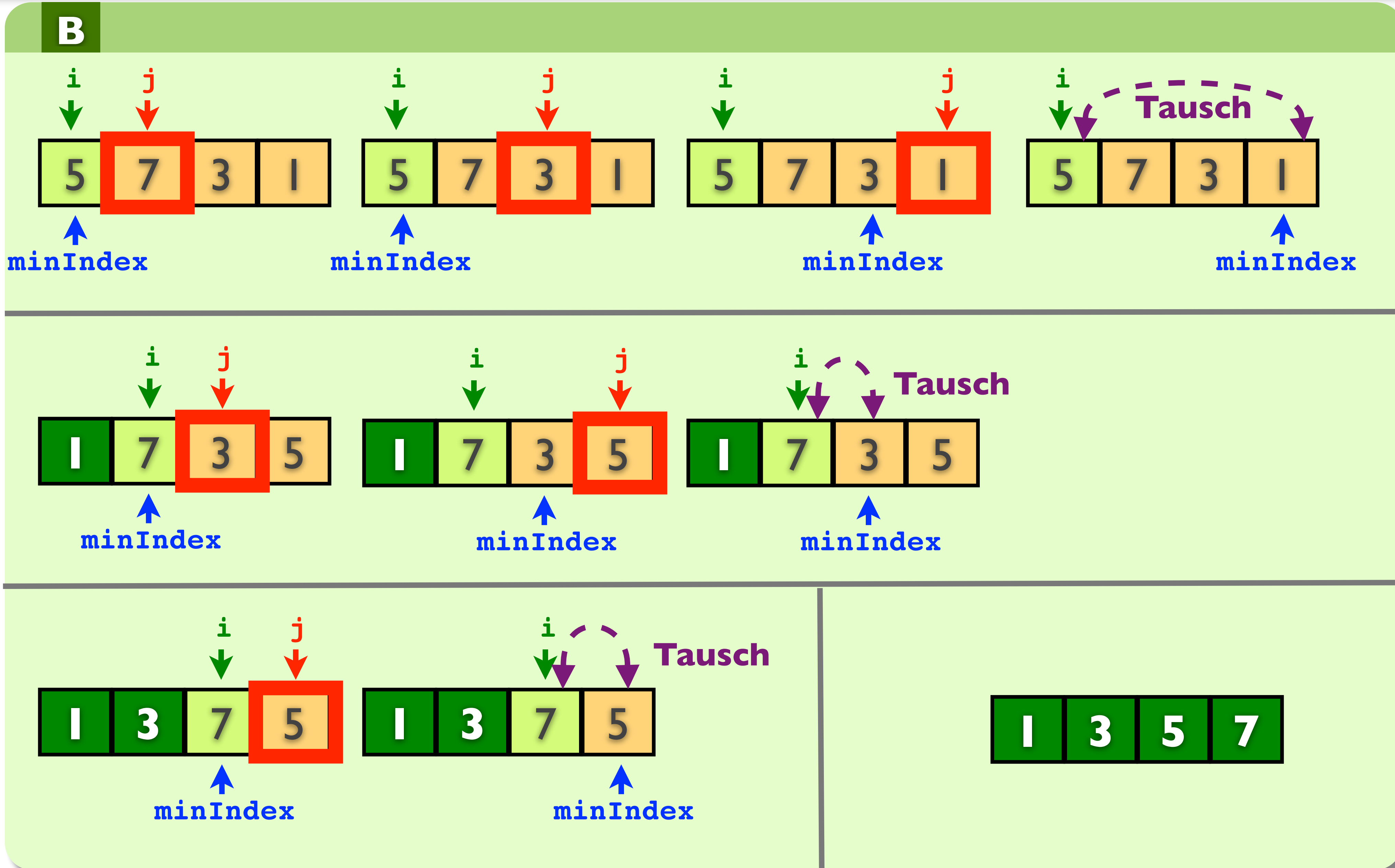
Beispiel: Sortieren durch Auswählen

Die **Laufzeitkomplexität** des folgenden Algorithmus soll bestimmt werden:

```java
public class SelectionSort {
  public static void sort(int feld[]) {
    int minIndex; // Index des aktuellen Minimum
    int t;        // Hilfsvariable für Tausch
    for (int i=0 ; i<feld.length - 1 ; i++) {
      minIndex = i;
      for (int j=i+1 ; j<feld.length ; j++) // Minimum in Rest suchen
        if (feld[j]<feld[minIndex]) minIndex = j;
      t=feld[i]; feld[i]=feld[minIndex] ; feld[minIndex]=t;
    }
  }
}
```



Selection Sort



Komplexität `Selection Sort`

Schätzen wir ab:

- Die Rechenzeit $T(n)$ hängt ab von der Zahl der zu sortierenden Elemente n
- Sie wird bestimmt von der

Zahl der **Vergleiche** (compares): $C(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$

Zahl der **Vertauschungen** (exchanges): $E(n) = n - 1$

Bezeichnet man den relativen Aufwand für Vergleiche mit α_1 und den der Vertauschungen mit α_2, so ergibt sich der Gesamtaufwand T wie folgt:

$$
\begin{aligned}
T(n) &= \alpha_1 C(n) + \alpha_2 E(n) \\
&= \alpha_1 \frac{n(n-1)}{2} + \alpha_2 (n-1) \\
&\cong \alpha_1 \frac{n^2}{2} \quad \text{für große } n \quad \left(n \gg \frac{\alpha_2}{\alpha_1}\right)
\end{aligned}
$$

Das **asymptotische Verhalten** von $T(n)$ entspricht dem von n^2. Man sagt: $T(n)$ hat **quadratische Komplexität**.

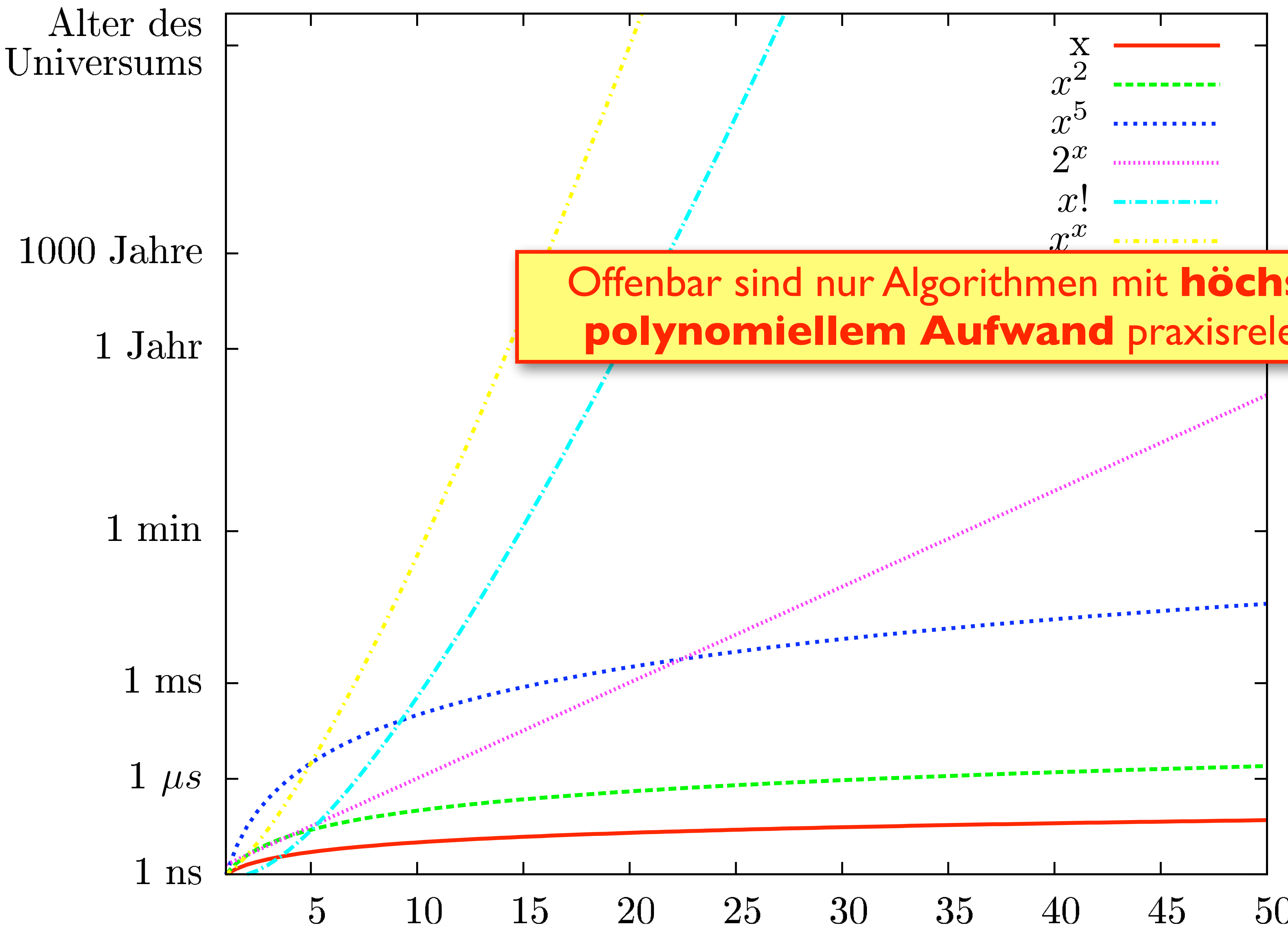
Kostenmaße - siehe RAM

- **Einheitskostenmaß**
  - Jeder Instruktion wird ein konstanter Aufwand zugeordnet - unabhängig vom Operanden
- **Logarithmisches Kostenmaß**
  - auch Bit-Kostenmaß genannt
  - Kosten sind abhängig von der Bit-Länge des Operanden
- **Leider gibt es kein standardisiertes Kostenmaß!**
  - Meist: möglichst realistische Abschätzung unter Berücksichtigung von
    - Adressierungsart (immediate, direkt, indirekt)
    - Art des Befehls
      - MUL und DIV in der Regel teurer als ADD und SUB
      - Bedingte Sprünge aufwendiger als direkte Sprünge

Komplexitätsklassen

Schranke	Bezeichnung	Beispiel
1	konstant	Elementarer Befehl
$\log(\log n)$	doppelt logarithmisch	Interpolationssuche
$\log n$	logarithmisch	Binäre Suche
n	linear	Lineare Suche, Minimum einer Folge
$n \log n$	überlinear	Effiziente Sortierverfahren (z.B. QuickSort)
n^2	quadratisch	Einfache Sortierverfahren
n^3	kubisch	Inversion von Matrizen, CYK-Parsing
n^k	polynomiell vom Grad k	lineare Programmierung
2^n	exponentiell	Erschöpfende Suche (Backtracking)
$n!$	Fakultät	Travelling Salesman Problem
n^n		

Wachstumsverhalten einiger Komplexitätsklassen



Offenbar sind nur Algorithmen mit **höchstens polynomiellem Aufwand** praxisrelevant!

Hinweise zum Logarithmus

- **Bei logarithmischen Komplexitätsklassen spielt die Basis keine Rolle,** denn der Übergang zu einer anderen Basis bedeutet die Multiplikation mit konstantem Faktor:

$$\log_b x = \frac{1}{\log_a b} \cdot \log_a x$$

- **Wir verwenden künftig folgende Schreibweisen:**

$$\begin{array}{lll} \mathrm{ld} & := & \log_2 \\ \ln & := & \log_e \\ \lg & := & \log_{10} \end{array}$$

ld	:=	$\log_2$	dualer Logarithmus
ln	:=	$\log_e$	natürlicher Logarithmus
lg	:=	$\log_{10}$	dekadischer Logarithmus

Vorsicht vor experimentell ermittelten Komplexitäten!

Vorsicht!

- **2001: Rechner mit 500 MHz**

N	Laufzeit
5	0,535s
10	1,1s
33	4,389s

Annahme: **lineares** Wachstum

$$T_{500Mhz}(n) \cong n$$

- **2006: Rechner mit 5 GHz / n=2000**

Unter der Annahme (lineares Wachstum) ließe sich die Laufzeit wie folgt abschätzen:

$$T_{5Ghz}(2000) = \frac{T_{500Mhz}(10)*200}{10}$$

(200 mal die alte Laufzeit für n=10 / Faktor 10 schnellerer Rechner - ca. **22s**).

Tatsächliches Laufzeitverhalten:

$$T_{500MHz}(n) = n^2 + 100n$$

Damit: tatsächliche Laufzeit bei 430s (ca. **7 min**)

1. Grundlagen

1. Grundlagen

1.14. Vergleich von Algorithmen
1.14.1. Abstrakte Maschinen
1.14.2. Die Landau-Notation

1.15. Rekursionsgleichungen
1.15.1. Einfache Strategien
1.15.2. Das *Master-Theorem*
1.15.3. Erzeugende Funktionen

Landau-Notation I

- **Die Laufzeit wird oft durch Funktion $T(n)$ beschrieben werden**
  (siehe Beispiel *Selection Sort* - n ist Größe der Eingabe)
- **Meist interessiert nur das asymptotische Verhalten von $T(n)$**
  (also das Verhalten für sehr große n)
  - Konstante Faktoren werden dann vernachlässigt
  - Zugleich sollen aber möglichst enge Schranken gefunden werden
- **Mathematisches Hilfsmittel hierzu: Landau-Notation**
  (auch **O-Notation** genannt)

Edmund Georg Hermann Landau
*14.02.1877 - †19.02.1938

Deutscher Mathematiker, der sich um die analytische Zahlentheorie verdient gemacht hat.

Landau-Notation II

D Landau-Symbole

Sei $f : \mathbb{N} \to \mathbb{R}^+$ eine Funktion, dann ist

$$O(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : \quad g(n) \leq c \cdot f(n)\}$$

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : \quad g(n) \geq c \cdot f(n)\}$$

$$\Theta(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c_1 > 0\ \exists c_2 > 0\ \exists n_0 > 0\ \forall n \geq n_0 : \quad c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$$

Sprechweisen:

$g \in O(f)$: f ist **obere Schranke** von g
g wächst höchstens so schnell wie f

$g \in \Omega(f)$: f ist **untere Schranke** von g
g wächst mindestens so schnell wie f

$g \in \Theta(f)$: f ist die **Wachstumsrate** von g
g wächst wie f

D Landau-Symbole

Sei $f : \mathbb{N} \to \mathbb{R}^+$ eine Funktion, dann ist

$$O(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : \quad g(n) \leq c \cdot f(n)\}$$

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : \quad g(n) \geq c \cdot f(n)\}$$

$$\Theta(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c_1 > 0\ \exists c_2 > 0\ \exists n_0 > 0\ \forall n \geq n_0 : \quad c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$$

Schreibweisen:

Anstelle von $g \in O(f)$ schreibt man oft $g(n) \in O(f(n))$ - formal nicht ganz korrekt, aber man kann auf die Definition von Hilfsfunktionen verzichten; statt

$$g \in O(f) \text{ mit } f(n) = n^2$$

schreibt man dann kurz:

$$g(n) \in O(n^2)$$

Die Schreibweisen $g(n) = O(n^2)$ und $g \prec f$ sind ebenfalls gebräuchlich.

Erläuterungen zur O-Notation:

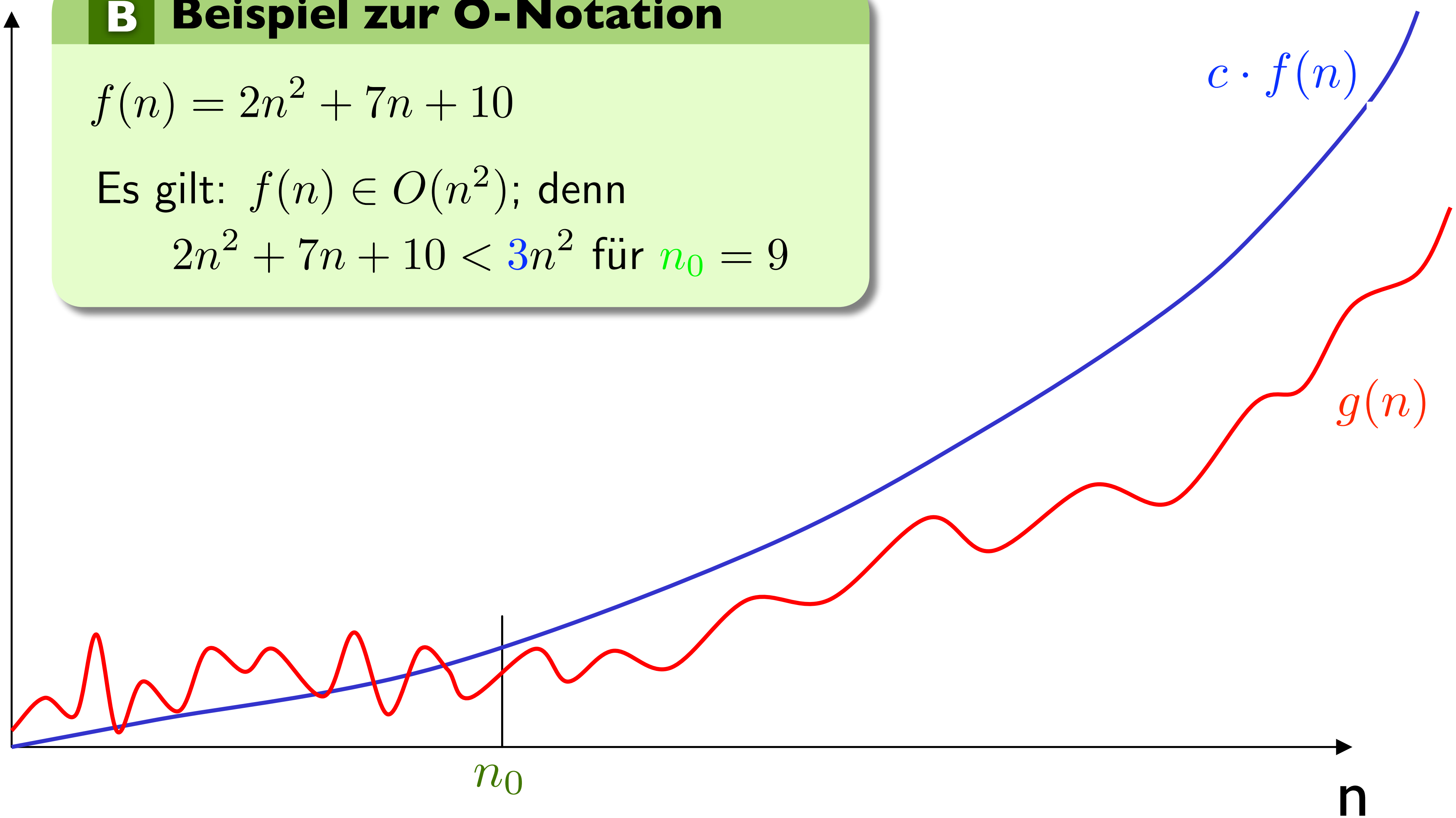
$$O(f) = \{g : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists c > 0\ \exists n_0 > 0\ \forall n \geq n_0 : \quad g(n) \leq c * f(n)\}$$

B Beispiel zur O-Notation

$$f(n) = 2n^2 + 7n + 10$$

Es gilt: $f(n) \in O(n^2)$; denn

$$2n^2 + 7n + 10 < 3n^2 \text{ für } n_0 = 9$$



Erläuterungen zur Θ-Notation

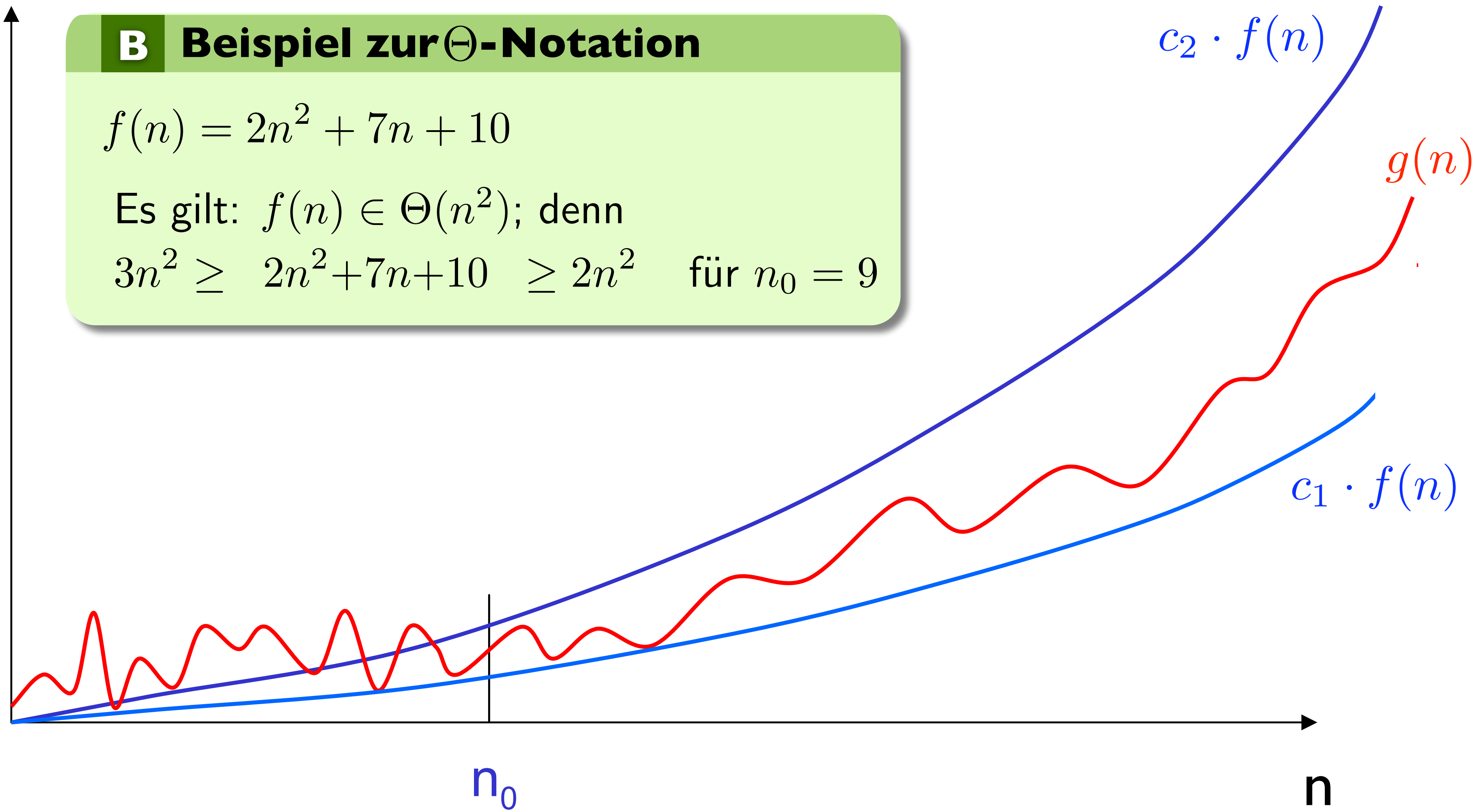
$$\Theta(f) = \{g : \mathbb{N} \to \mathbb{R}^+ \mid \exists c_1 > 0\ \exists c_2 > 0\ \exists n_0 > 0\ \forall n \geq n_0 : \quad c_1 \cdot f(n) \leq g(n) \leq c_2 \cdot f(n)\}$$

B Beispiel zur Θ-Notation

$f(n) = 2n^2 + 7n + 10$

Es gilt: $f(n) \in \Theta(n^2)$; denn

$3n^2 \geq \ 2n^2+7n+10 \ \geq 2n^2$ für $n_0 = 9$



Rechenregeln

1. Linearität

Falls $g(n) = \alpha \cdot f(n) + \beta$ mit $\alpha, \beta \in \mathbb{R}^+$ und $f \in \Omega(1)$, so gilt:

$$g \in O(f)$$

2. Addition

$$f + g \in O(\max\{f, g\}) = \begin{cases} O(g) & \text{falls } f \in O(g) \\ O(f) & \text{falls } g \in O(f) \end{cases}$$

3. Multiplikation

$$a \in O(f) \wedge b \in O(g) \quad \Rightarrow \quad a \cdot b \in O(f \cdot g)$$

4. Grenzwert

Falls der Grenzwert

$$\lim_{n \to \infty} \frac{g(n)}{f(n)}$$

existiert, so ist $g \in O(f)$ bzw. $g \prec f$.
Der Umkehrschluss gilt nicht.

Nützlich zum Vergleich!

Addition, Multiplikation und Maximumbildung sind bildweise zu verstehen; z.B.

$$(f+g)(n) = f(n) + g(n)$$

1. Linearität

1 ist **untere Schranke** von f

Falls $g(n) = \alpha \cdot f(n) + \beta$ mit $\alpha, \beta \in \mathbb{R}^+$ und $f \in \Omega(1)$, so gilt:

$$g \in O(f)$$

Beweis: Wegen $f \in \Omega(1)$ existieren $c' > 0$ und $n' > 0$, so dass

$$\forall n \geq n' : f(n) \geq c' \cdot 1$$

Wähle jetzt $c = \alpha + \frac{\beta}{c'}$ und $n_0 = n'$, dann gilt

$$\begin{aligned}
\forall n \geq n_0 : g(n) &= \alpha \cdot f(n) + \beta \\
&= \left(\alpha + \frac{\beta}{f(n)}\right) \cdot f(n) \\
&\leq \left(\alpha + \frac{\beta}{c'}\right) \cdot f(n)
\end{aligned}$$

(* da $c' \leq f(n)$ für $n \geq n' = n_0$ und somit $\frac{\beta}{c'} \geq \frac{\beta}{f(n)}$ *)

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

Also ist $g \in O(f)$

Zeitbedarf von Algorithmen

D (Laufzeit-)Komplexität von Algorithmen

Sei A ein Algorithmus und beschreibe $T_A(n)$ die Laufzeit von A in Abhängigkeit von der Größe der Eingabe n. Dann gilt: **A hat die Komplexität $O(g)$**, falls $T_A \in O(g)$.

- **Hinweis:** Komplexität meint von jetzt an „Laufzeitkomplexität".
- **Laufzeitanalyse** (**Komplexitätsanalyse**):
  - suche nach geeignetem $T_A(n)$
  - Abschätzen von $T_A(n)$ mithilfe der O-Notation

    Abschätzen = Extraktion des dominanten Teils + Weglassen von Koeffizienten; z.B.:

    - $T(n) = 60n^3 + 50n^2 + 70 \quad\Rightarrow\quad T(n) \in \mathrm{O}(n^3)$
    - $T(n) = \mathrm{ld(n)} + 1 \Rightarrow \quad T(n) \in \mathrm{O}(\log n)$

    Manchmal ist es sinnvoll, zum Vergleich die Koeffizienten des dominanten Terms beizubehalten; z.B.:

    - $T_1(n) = 60n^2 + 4n \quad\Rightarrow\quad T_1(n) = 60n^2 + \mathrm{O(n)}$
    - $T_2(n) = \quad 4n^2 + 75 \quad\Rightarrow\quad T_2(n) = \quad 4n^2 + \mathrm{O}(1)$

Komplexität vs. Laufzeit

B Beispiel zur O-Notation

$$T_A(n) = \frac{n \cdot (n-1)}{2} \qquad T_A \in O(n^2) \qquad T_A(n) = \frac{1}{2} \cdot n^2 + O(n)$$

$$\frac{n \cdot (n-1)}{2} = \frac{1}{2}(n^2 - n) \le n^2 - n \le n^2$$

$\Rightarrow$ mit $n_0 = 1$ und $c = 1$ ist die Definition der O-Notation erfüllt.

$$T_B(n) = n^2 + 2n \qquad T_B \in O(n^2) \qquad T_B(n) = n^2 + O(n)$$

$$n^2 + 2n \le n^2 + 2n^2 = 3n^2$$

$\Rightarrow$ mit $n_0 = 1$ und $c = 3$ ist die Definition der O-Notation erfüllt.

Also:

- Algorithmus A ist zwar etwas ***schneller*** als Algorithmus B,
- aber beide Algorithmen gehören zu derselben **Komplexitätsklasse** (sind also ***gleich gut***)

Regeln für die Laufzeitanalyse I

- **Elementare Anweisungen (z.B. RAM-Befehl)**
  - **Sind in $O(1)$**
  - **Achtung!**
    in höheren Programmiersprachen sind nicht alle Anweisungen elementar!
    Beispiel: Menge **`M`**, Objekt **`O`**: **`M.isElem(O)`**

- **Anweisungssequenzen**
  - Sei „**`A; B`**“ eine Folge von Anweisungen und $T_{\mathtt{A}} \in O(f)$ und $T_{\mathtt{B}} \in O(g)$ dann hat die Hintereinanderausführung die Komplexität

$$T = T_{\mathtt{A}} + T_{\mathtt{B}} \in O(\max(f, g))$$

  - Maßgeblich ist der größte Aufwand!

Regeln für die Laufzeitanalyse II

- **Schleifen**
  - Allgemein: Summe über die einzelnen Durchläufe (oft rekursiv!)
  - Falls Laufzeit T_S Schleifenrumpf unabhängig von jeweiligem Durchlauf, so $T = T_S * k$; k Gesamtzahl der Durchläufe

- **Bedingte Anweisungen**
  - **`if C then A else B`**
  - sei $T_{\mathtt{A}} \in O(f)$, $T_{\mathtt{B}} \in O(g)$ und $T_{\mathtt{C}} \in O(h)$; dann ergibt sich die Laufzeit als $O(h + O(g + f))$

- **Funktionsaufrufe**
  - **nicht-rekursiv**
    - Aufwand für Funktion separat ermitteln + konstanter Overhead für Aufruf
  - **rekursiv**
    - nicht trivial zu ermitteln - Lösen von Rekursionsgleichungen notwendig (evtl. später ...)

Genauer hingesehen - transitives O, kleinstes O

- **Wenn $f \in O(g)$ und $g \in O(h)$, so $f \in O(h)$**

- **Sei $T(n)=3 \cdot n^2$, dann ist**
  - $T(n) \in O(\mathrm{n}^2)$
  - $T(n) \in O(\mathrm{n}^3)$
  - ...
  - $T(n) \in O(2^n)$
  - ...
  - $T(n) \in O(n!)$
  - $T(n) \in O(n^n)$
  - ...

- **Interessant ist natürlich nur die minimale obere Schranke** (hier $O(n^2)$ - das „minimale $O(g)$“)

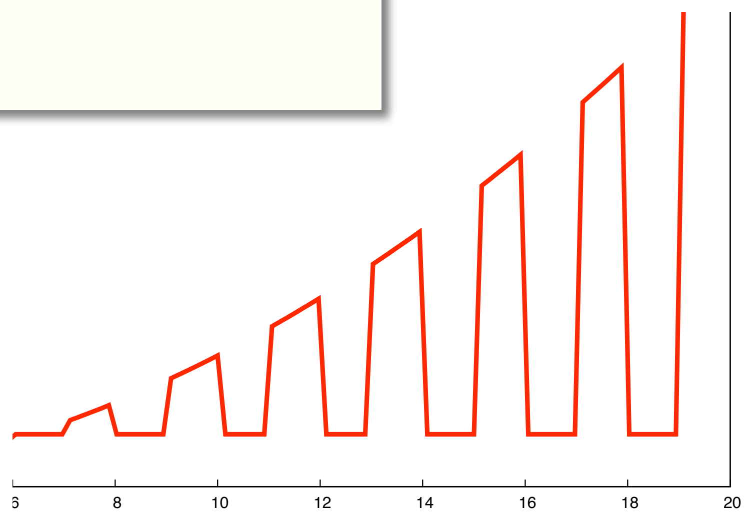
- **Achtung**: Das Wort „minimal“ wird oft weggelassen

- **Warum nicht immer die genauere Θ-Notation?**
  (Warum der Aufwand mit „kleinsten oberen Schranken“?)

- **Betrachten wir folgende JAVA-Methode:**

```
public void seltsam(int[] feld) {
    n = feld.length;
    if (n%2 == 0) {
      // tausche erstes und letztes Element => O(1)
    } else {
      // sortiere Feld mit SelectionSort    => O(n²)
    }
}
```

- **minimale obere Schranke: $O(n^2)$**
- **maximale untere Schranke: $\Omega(1)$**
- **Aber:**
  - Zeitbedarf in keiner Menge $\theta(g)$ vorhanden!



Die Fibonacci-Funktion

- **Leonardo de Pisa (auch Fibonacci genannt) italienischer Mathematiker (um 1200)**
- **Berühmte Kaninchenaufgabe:**
  - Start: 1 Paar Kaninchen
  - Jedes Paar wirft nach zwei Monaten ein neues Kaninchenpaar
  - dann monatlich jeweils ein Paar
  - wie viele Kaninchenpaare gibt es in einem Jahr, wenn kein Kaninchen vorher stirbt?
  - 1,1,2,3,5,8,13, ...
  - Anzahl im n-ten Monat:

$$fib(n) = \begin{cases} 0 & \text{falls } n = 0 \\ 1 & \text{falls } n = 1 \\ fib(n-1) + fib(n-2) & \text{sonst} \end{cases}$$

Komplexität ?

Fibonacci: Rekursiver Algorithmus

```
public static int fib(int n) {
   if (n<=1) return n;
   else return fib(n-1) + fib(n-2);
}
```

- **Komplexität von `fib`:** $T_{\texttt{fib}}(n) \in O(2^n)$
- **Beweis:**

1. $fib(n)$ ist positiv: $\forall n > 0 : \; fib(n) > 0$
2. $fib(n)$ ist für $n > 2$ streng monoton steigend: $fib(n) < fib(n+1)$
3. Abschätzung nach oben: Für alle $n > 3$ gilt

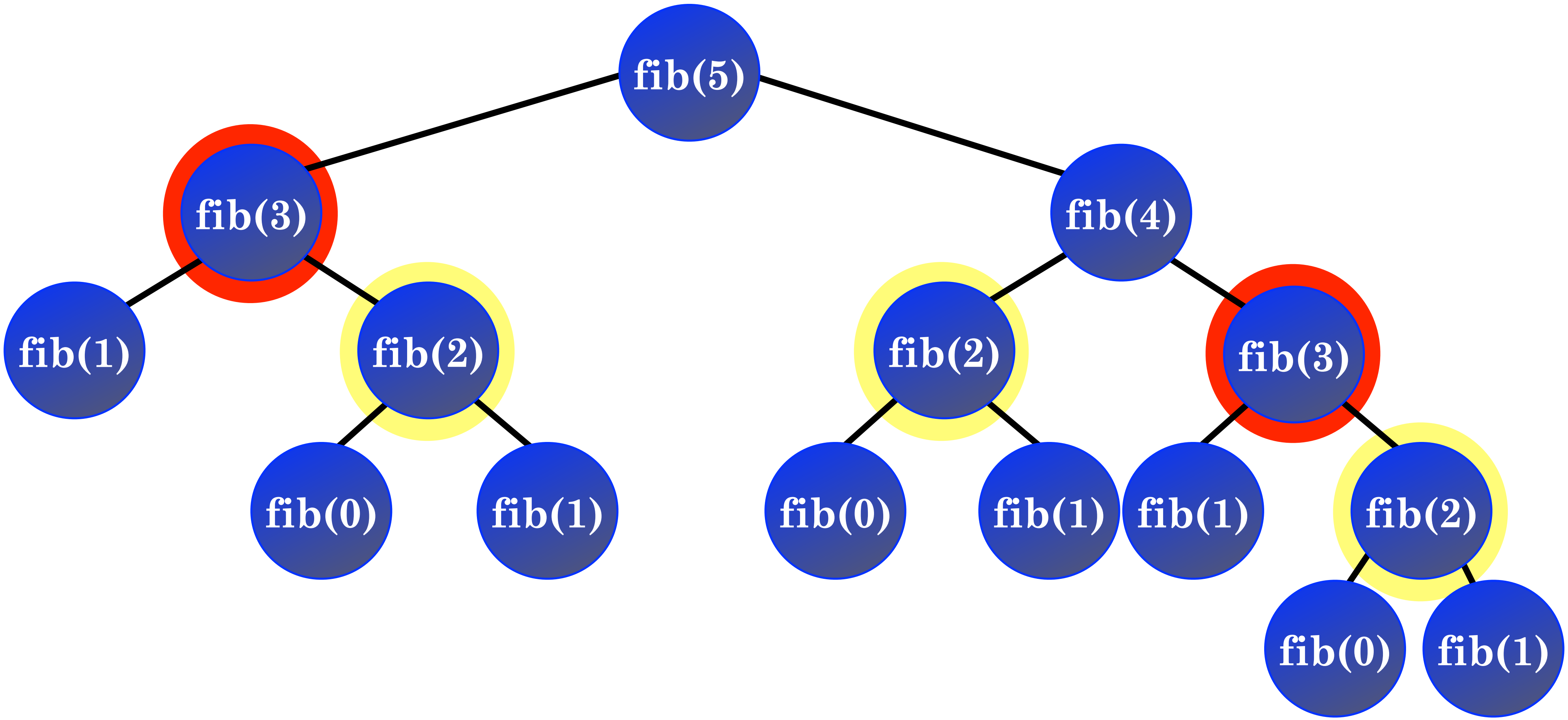
$$
\begin{array}{rcl}
fib(n) & = & fib(n-1) + fib(n-2) \\
 & & (*\;\; fib(n-2) < fib(n-1) \Rightarrow 2 \cdot fib(n-1) > fib(n-2) + fib(n-1) \;\; * \\
 & < & 2 \cdot fib(n-1) \\
 & & (*\;\; fib(n-1) = fib(n-2) + fib(n-3) \;\; *) \\
 & < & 4 \cdot fib(n-2) \\
 & < & \cdots \\
 & < & 2^{n-1} \cdot fib(1) = \frac{1}{2} \cdot 2^n
\end{array}
$$

Also gilt: $fib(n) \in O(2^n)$

$\Theta(c^n)$ mit $c = \frac{1+\sqrt{5}}{2}$

Berechnungsbaum für $\mathrm{fib}(5)$

- **Zahlreiche Redundanzen in der Berechnung:**



- **Offenbar wird z.B. $\mathrm{fib}(2)$ dreimal berechnet - das kann man vermeiden, indem man sich Zwischenergebnisse merkt!**

Fibonacci: Iterativer Algorithmus

- **Idee**: jede Fibonacci-Zahl wird aus der Summe der beiden Vorgänger gebildet - merke diese Vorgänger in Variablen **`fib_new`** und **`fib_old`**

```java
public static int fib_it(int n) {
   int p_previous=0, // fib(0)  - Vorvorgänger
       previous=1;   // fib(1)  - Vorgänger
   int current=0;    // Hier wird das Ergebnis abgelegt
   for (int i=0 ; i<n ; ++i) {
     // Die beiden Vorgänger aktualisieren
     p_previous = previous;
     previous   = current;
     // Ergebnis ermitteln
     // -> fib(i) = fib(i-1) + fib(i-2)
     current = current + p_previous;
   }
   return current;
}
```

- **Komplexität von `fib_it`:** $T_{\texttt{fib\_it}}(n) \in \Theta(n)$

Rekursion vs. Iteration

- **Rekursive Formulierung oft**
  - eleganter / näher an mathematischer Formulierung
  - einfacher zu lesen
- **Iterative Lösungen sind oft effizienter**
  - aber komplizierter
- **Rekursion**
  - mächtiges, allgemeines Programmierprinzip
  - es gibt Sprachen, die Schleifen nur über Rekursion realisieren können (z.B. Haskell)
  - Jede rekursive Lösung kann in eine iterative überführt werden!
  - Rekursion muss nicht ineffizient sein - **tail recursion** (endständige Rekursion)

Fibonacci: Tail Recursion / Tail Calls

- **Idee:** Nach rekursivem Aufruf werden Werte auf Stapel nicht mehr verwendet
  - der aktuelle Aufrufkontext kann damit überschrieben werden! (Compiler muss entsprechende Optimierung unterstützen)
  - Ergebnis wird in Argument „akkumuliert“

```java
public static int fib_tr(int n) {
  return fib_tr_help(n,0,1);
}

private static int fib_tr_help(int n,int current,int previous) {
  if (n<1) return current;
  else return fib_tr_help(n-1, current+previous, current);
}
```

- **Komplexität von `fib_tr`:** $T_{\texttt{fib\_tr}}(n) \in \Theta(n)$

Beispiel: Tail-Call Optimization

B Tail-Call Optimization (Elimination)

```
  int fib_tr_help(int n,int current,int previous) {
    if (n<1) return current;
    else return fib_tr_help(n-1, current+previous, current);
  }
```

Tail-Call Optimization

```
int fib_tr_help_opt(int n,int current,int previous) {
  while (true) {
    if (n<1) return current;
    else {
      // return fib_tr_help(n-1, current+previous, current);
      n = n - 1 ; int current_merk = current;
      current = current + previous;
      previous = current_merk;
    }
  }
}
```

Worst, Best und Average Case

- **Man unterscheidet den Zeitbedarf**
  - im schlechtesten Fall (**worst case**)
  - im Mittel (**average case**)
  - im besten Fall (**best case**)

- **Bei der Komplexitätsanalyse wird meist der schlechteste oder der mittlere Fall betrachtet**

- **Average Case (Erwartungswert)**
  - Mittelwert des Aufwandes über alle möglichen Eingaben; gewichtet mit Auftrittswahrscheinlichkeit

Worst, Best und Average Case - Beispiel (Teil 1)

B Multiplikation von zwei n-Bit Binärzahlen

Pseudo-Code:

```
Multipliziere(x,y):
  ergebnis=0 ; i=0;
  solange(i<n) {
     wenn(i-tes Bit von x == 1){
        y um i Bits nach links shiften
        ergebnis += geshiftetes  y;
     }
     i++;
  }
```

- **Aufwand** = #Additionen der Bits
  = #Einsen in der Binärdarstellung von x
- **Best case** $\mathrm{x} = 0 \quad \Rightarrow \quad 0$ Additionen
- **Worst case** $\mathrm{x} = 2^n - 1 \quad \Rightarrow \quad n$ Additionen
- ***Average case ?***

B Multiplikation von zwei n-Bit Binärzahlen

Anzahl der möglichen Eingaben $N = 2^n$

Anzahl der Eingaben mit k Einsen: $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

$$
\begin{aligned}
S &= \sum_{k=0}^{n} k \cdot \binom{n}{k} = \sum_{k=0}^{n} \frac{k \cdot n!}{k!(n-k)!} = \sum_{k=0}^{n} \frac{n(n-1)!}{(k-1)!(n-1-k+1)!} = \sum_{k=0}^{n} n \cdot \binom{n-1}{k-1} \\
&= n \cdot \left(\binom{n-1}{-1} + \sum_{k=1}^{n} \binom{n-1}{k-1} \right) = n \cdot \left(0 + \sum_{k=1}^{n} \binom{n-1}{k-1} \right) = n \cdot \sum_{k=0}^{n-1} \binom{n-1}{k} \\
&= n \cdot \sum_{k=0}^{n-1} \binom{n-1}{k} 1^k \cdot 1^{n-1-k} = n \cdot (1+1)^{n-1} = n \cdot 2^{n-1}
\end{aligned}
$$

Mittlerer Aufwand $= \frac{S}{N} = \frac{n \cdot 2^{n-1}}{2^n} = \frac{n}{2}$

Abschließende Beispiele I

- **Schleife**

```
for (int i=0 ; i<n; ++i) {
  a[i]=0;
}
```

$O(n)$

- **Geschachtelte Schleife**

```
for (int i=0 ; i<n; ++i)
    for (int j=0 ; j<n; ++j)
      a[i][j]=0;
```

$O(n^2)$

- **Hintereinanderausführung**

```
for (int k=0; k<n ; ++k)
  c[k]=-1;
for (int i=0 ; i<n; ++i) {
  for (int j=0 ; j<n; ++j)
    a[i][j]=0;
```

$O(n^2)$

Abschließende Beispiele II

- **Bedingte Anweisung**

```
if (x<100)
    y=x;
else
  for (int i=0 ; i<n ; ++i)
    if (a[i]>3) y+=a[i];
```

$O(n)$

- **Innere Schleife hängt vom Lauf ab**

```
for (int i=0; i<n ; ++i)
  for (j=0 ; j<i ; ++j)
    k++;
```

$O(n^2)$

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

Abschließende Beispiele III

- **Aussprung**

```
for (int i=0 ; i<n ; ++i)
  if (i>2)
    return;
```

$O(1)$

- **Schleifenvariable wird in jedem Lauf halbiert**

```
while (i>0) {
  x++;
  i>>1; // Shift nach rechts um 1 Stelle
        // = Division durch 2
}
```

$O(\log_2 n)$

$$
\begin{aligned}
T(n) &= 1 + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \\
&= 1 + 1 + T\left(\left\lfloor \frac{\frac{n}{2}}{2} \right\rfloor\right) = 2 + T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) \\
&= 3 + T\left(\left\lfloor \frac{n}{8} \right\rfloor\right) \\
&\vdots \\
&= \log_2(n)
\end{aligned}
$$