Komplexität von Algorithmen

OOPM, Ralf Lämmel

Motivierendes Beispiel

- Algorithmus
  - Eingabe: ein Zahlen-Feld *a* der Länge *n*
  - Ausgabe: Durchschnitt
- Fragen:
  - **Was ist die Laufzeit des Algorithmus?**
  - Speicherverbrauch ist offensichtlich “gering”.

```
sum = 0;
i = 0;
while (i < n) {
  sum += a[i];
  i++;
}
return sum/n
```

Komplexitätsanalyse

- Unmittelbare Ziele
  - Vorhersage der Kosten
  - Verifikation von Kosten
- Weitere Ziele
  - Auswahl eines besseren Algorithmus
  - Feststellung eines optimalen Algorithmus
- Im Kontext von Qualitätseigenschaften von Algorithmen:
  - Korrektheit
  - **Effizienz**
  - Verständlichkeit
  - ...

Zeitliche Komplexität: Wie bestimmt man sie?

* *Vergleich der Ausführung zweier Algorithmen (Programme)*: unerwünschte Abhängigkeit von Eingabe, Maschine, Rechnerauslastung, ...
* Vorhersage der auszuführenden Anweisungen in Abhängigkeit von der Eingabe: unerwünschte Abhängigkeit von Sprache, Compiler, Maschine, u.a.
* **Wie oben, aber Zählen von "wesentlichen" *Basisoperationen* - z.B.: arithmetische und Boolesche Operationen sowie Vergleiche.**

Wir lassen ***hier*** Zugriffe auf Speicher (Variablen) unbeachtet.

(Zeit-) Komplexität =
Anzahl der Basisoperationen
(in Abhängigkeit von Eingabe (-größe))

Zählen der Operationen für die Berechnung des Durchschnitts

Program	# Operationen
1. sum = 0;	0
2. i = 0;	0
3. while (i < n) {	n+1
4. sum += a[i];	n
5. i++;	n
6. }	
7. return sum/n	1
Summe	**3n + 2**

(C) 2007-2018, Ralf Lämmel, Universität Koblenz-Landau

Was ist die Zeitkomplexität von Matrizenmultiplikation?

- Basisoperationen:
  - Multiplizieren von Matrizeneinträgen
- Komplexität: $m * p * n$
- Zum Beispiel:
  - $m=n=p$
  - Komplexität: n^3
- **Komplexität hängt nur von Eingabegröße ab.**

Wir lassen ***hier*** andere Vergleiche und Inkrementierung unbeachtet.

- Eingabe: a_{m*n}, b_{n*p}
- Ausgabe: c_{m*p}

for $(i=0;\ i<m;\ i{+}{+})$
  for $(j=0;\ j<p;\ j{+}{+})$
    $c_{ij} = 0$
    for $(k=0;\ k<n;\ k{+}{+})$
      $c_{ij} = c_{ij} + a_{ik}\, b_{kj}$

Verschiedene Zeitkomplexitäten

* **T(n)** - *Genaue* Anzahl der Operationen
* **T_W(n)** - Komplexität im schlechtesten Fall (engl.: worst-case complexity): die größte Anzahl von Basisoperationen unten allen möglichen Eingaben der Größe n.
* **T_A(n)** - Komplexität im Durchschnitt (engl.: *average complexity*): die durchschnittliche Anzahl von Basisoperationen für alle möglichen Eingaben der Größe n.

Was ist die Zeikomplexität von Linearer Suche?

```
boolean linear(int[] a, int x)
    r = false
    i = 0
    i < a.length && !r
        r = a[i] == x
        i = i + 1
    return r
```

Basisoperationen: ***hier*** nur Vergleiche mit gesuchtem Element.

Schlimmster Fall: Das Element ist nicht zu finden.

```
boolean linear(int[] a, int x)
  r = false
  i = 0
  i < a.length && !r
      r = a[i] == x
      i = i + 1
  return r
```

Vergleiche alle n Elemente.
$T_W(n) = n$

Durchschnittlicher Fall

- Fall 1: *x* nicht in *a*, siehe $T_W(n)$
- Fall 2: *x* kommt in *a* vor.
  - ***Annahme***: alle Elemente verschieden in *a*.
  - Es gibt ein *i* so dass *x* auf *i*-ter Position mit $0 \leq i < n$.
    - Wahrscheinlichkeit für beliebiges *i*: $1/n$
    - $i+1$ Vergleiche benötigt
    - Für alle i: $1 + 2 + ... + n =$ $\mathbf{n (n + 1) / 2}$
    - Im Durchschnitt: $\mathbf{(n + 1) / 2}$

Durchschnittlicher Fall

- Zu kombinieren:
  - Fall 1: n
  - Fall 2: $(n + 1) / 2$
- ***Annahme***: q ist Wahrscheinlichkeit für Fall 2.
- $T_A(n) = (1 - q)\, n + q\, (n + 1) / 2$
- Zum Beispiel: q = 0.5
  - $T_A(n) = 1/2\ n + 1/4\ (n + 1) = 3/4\ n + 1/4$
  - ≈ 3/4 aller Elemente werden verglichen.

Speicherkomplexität

- Speicherplatz für Programm hier vernachlässigt.
- Speicherplatz für Eingabe
  - Konstant für skalare Werte
  - Feldgröße für Felder
- **Speicherkomplexität = Extra Speicherplatz für**
  - **Arbeitsspeicher**
  - **Ausgabe**
- S(*n*): Speicherkompl. in Abh. von Eingabegröße
- Analog: $S_W(n)$ und $S_A(n)$
- Algorithmus “in place” für konstantes S(*n*)

(C) 2007-2018, Ralf Lämmel, Universität Koblenz-Landau

Was ist die Speicherkomplexität von MergeSort?

- MergeSort benötigt Arbeitsspeicher.
- Implementation:

```
public static void mergeSort(int[] a) {
    int[] temp = new int[a.length];
    mergeSort(a,temp,0,a.length-1);
}
```


- $S(n) = n$

Optimalität von Algorithmen

* Ein Algorithmus ist ***optimal*** wenn es keinen Algorithmus für das gleiche Problem gibt, dessen Komplexität besser ist.

* Beachte:

  Mehrere Dimensionen: $T_W(n)$, $T_A(n)$, $S_W(n)$, $S_A(n)$

* Welches ist der optimale Sortieralgorithmus?

Von Termen über der Eingabegröße *n* zu Komplexitätsklassen

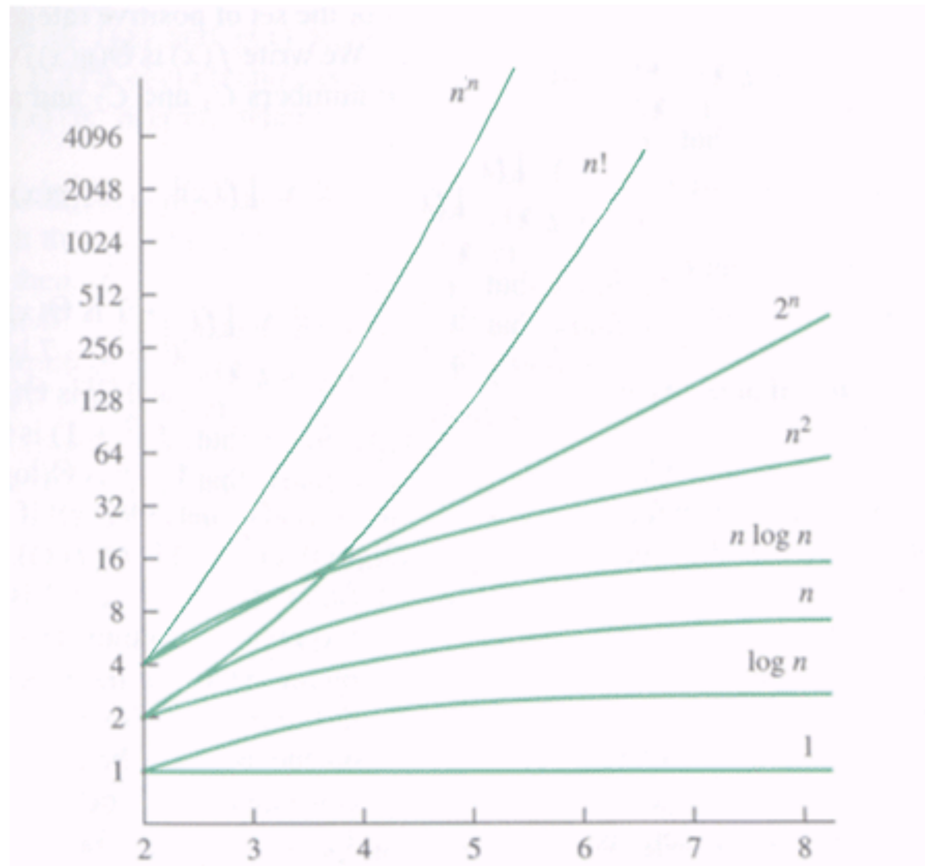
Probleme beim Zählen von Basisoperationen

in Abhängigkeit von der Eingabegröße

* **Wie vergleicht man Terme über *n*?**
* Außerdem (nicht betrachtet):
  * Verschiedene Festlegungen zu den Basisoperationen
  * Abhängigkeit von Compiler und Plattform
  * Interne Operationen werden eventuell übersehen

(C) 2007-2018, Ralf Lämmel, Universität Koblenz-Landau

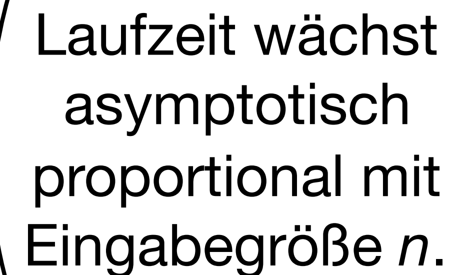
Ziel: Vergleich asymptotischen Wachstums



Haben wir lineare, quadratische, exponentielle, ... **Abhängigkeit** der Anzahl der Operationen bzw. der Grösse des Speichers **von Grösse der Eingabe**?

Erneute Betrachtung der Komplexität des Durchschnitts

Program	# Operationen
• sum = 0;	0
• i = 0;	0
• while (i < n) {	n+1
• sum += a[i];	n
• i++;	n
• }	
• return sum/n	1
Summe	**3n + 2**



Konstante Faktoren

* Betrachten wir zwei Komplexitäten (“Funktionen”):
  - a) 2n
  - b) $2.5n$

Wir vernachlässigen konstante Fakoren.
(Intuition: Parallelisierung von Operationen beseitigt den Unterschied.)

$2n$ “=” $2.5n$

Asymptotisches Wachstum

- Betrachten wir zwei Komplexitäten (“Funktionen”):
  - a) $n^3/2$
  - b) $5n^2$
- Kubische Funktionen wachsen schneller als quadratische.
  - Für einige kleine n ist b) größer.
  - Für genügend große n ist a) größer.

Übungsaufgabe:

Bestimme das n so dass $n^3/2$ “>” $5n^2$

$n^3/2$ “>” $5n^2$

(C) 2007-2018, Ralf Lämmel, Universität Koblenz-Landau

Asymptotisches Wachstum

* Betrachten wir zwei Komplexitäten ("Funktionen"):
  - $3n^3 - 2n^2$
  - $4n^3$

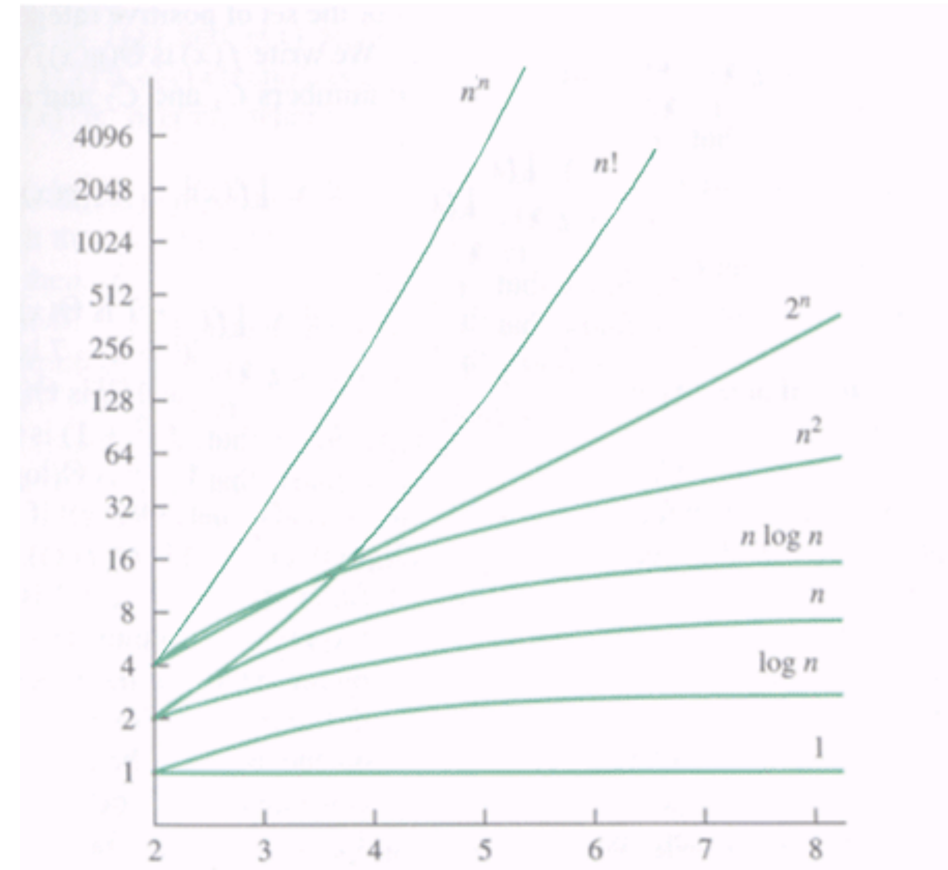
Bei der Abbildung von Komplexitätsfunktionen auf Komplexitätsklassen vernachlässigen wir auch Summanden von kleinerem asymptotischen Wachstum.

$3n^3 - 2n^2$ "=" $3n^3$ "=" $4n^3$

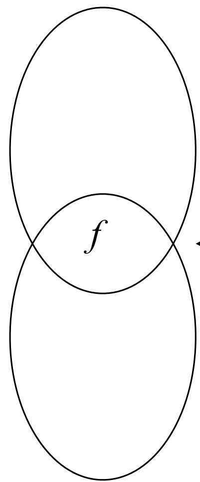
(C) 2007-2018, Ralf Lämmel, Universität Koblenz-Landau

Komplexitätsklassen

* Konstant: 1
* Logarithmisch: log n
* Linear: n
* Quadratisch: n^2
* Polynomial: n^k, $k > 1$
* Exponentiell: a^n, $a > 1$



Schranken für asymptotisches Wachstum



$\Omega(f)$ enthält Funktionen die mindestens so schnell wie f wachsen.
(f ist eine *untere Schranke* für Komplexitätsfunktionen.)

$\Theta(f)$ enthält Funktionen die genauso schnell wie f wachsen.
(f ist ein *asymptotisch enges Mass*.)

$O(f)$ enthält Funktionen die nicht schneller als f wachsen.
(f ist eine *obere Schranke* für Komplexitätsfunktionen.)

Aussprache: **(Big) oh (*O*)**
(Big) theta (Θ)
(Big) omega (Ω)

O - Notation für obere Schranke f(n)

- $O(f(n)) =$

  $\{ g: N \to N \mid$

  $\exists n_0 > 0.\ \exists c > 0.\ \forall n \geq n_0.$

  $g(n) \leq c * f(n) \}$

- Anmerkungen
  - $k(n) \in O(f(n))$ steht für:

    $f(n)$ ist eine asymptotische obere Schranke für $k(n)$.
  - Sprechweise "$k(n)$ ist $O(f(n))$"
  - Analog: Ω und Θ

Wir betrachten Funktionen über den natürlichen Zahlen, weil wir so die Eingabegröße ausdrücken.

Wir sind an kleinen, oberen Schranken interessiert.

Ω - Notation für untere Schranke

- $\Omega(f(n)) =$

  $\{ g: N \rightarrow N \mid$

  $\exists n_0 > 0.\ \exists c > 0.\ \forall\ n \geq n_0.$

  $c*f(n) \leq\ g(n) \}$

- $k(n) \in \Omega(f(n))$ steht für:

  $f(n)$ ist eine asymptotische untere Schranke für $k(n)$.

Wir sind an großen, unteren Schranken interessiert.

Θ - Notation für asymptotisch enges Maß

- $\Theta(f(n)) =$

  $\{ g: N \rightarrow N \mid$

  $\exists\, n_0 > 0.\ \exists\, c_1, c_2 > 0.\ \forall\, n \geq n_0.$

  $c_1 * f(n) \leq g(n) \leq c_2 * f(n) \}$

Rechenregeln für die O-Notation

- $f(n) \in O(f(n))$
- $c * O(f(n)) \in O(f(n))$
- $O(f(n)) * O(g(n)) \in O(f(n) * g(n))$
- $O(f(n) * g(n)) \in O(f(n)) * O(g(n))$
- ...

Wenn sowohl S und S' Mengen sind in der Notation $S \in S'$, dann ist dies gemeint: $\forall f \in S.\ f \in S'$.

Was ist die Komplexität von Sortieralgorithmen?

	$T_W(n)$	$S_W(n)$
SelectionSort	?	?
MergeSort	?	?
...	...	...

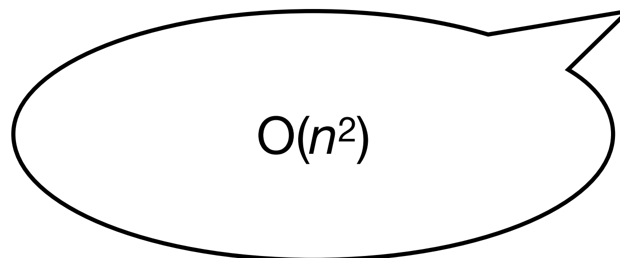
Was ist die Zeitkomplexität von ***SelectionSort***?

```
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        // Find least element among the remaining ones
        int least = i;
        for (int j = i + 1; j < a.length; j++)
            if (a[j] < a[least])
                least = j;
        // Swap current element with least one
        if (least != i) {
            int swap = a[i];
            a[i] = a[least];
            a[least] = swap;
        }
    }
}
```

Was ist die Zeitkomplexität von ***SelectionSort***?

```
for (int i=0; i<a.length; i++) {
    int least = i;
    for (int j=i+1; j<a.length; j++)
        if (a[j] < a[least])
            ...
    ...
}
```

- **Zählen der Vergleiche**
- Feld der Größe n
- Äußere Schleife $i=0..n-1$
- Innere Schleife $j=i+1..n-1$
- Vergleiche: $(n-1) + (n-2) + ... + 1 = n * (n - 1) / 2$



Was ist die Zeitkomplexität von ***MergeSort***?

```
public static void mergeSort(int[] a, int[] temp, int min, int max) {
    // Cease on trivial sorting problem
    if (!(min<max))
        return;

    // Divide
    int middle = ( min + max ) / 2 ;

    // Solve smaller problems recursively
    mergeSort(a,temp,min,middle);
    mergeSort(a,temp,middle+1,max);

    // Merge via temporary array
    merge(a,temp,min,middle,max);
}
```

Was ist die Zeitkomplexität von ***MergeSort***?

- Merge: $T_W(n) = O(n)$
- MergeSort:
  - $T_W(n) \quad = 2\, T_W(n/2) + O(n)$
  - $\quad\quad\quad = n \log n$ (per “Mastertheorem”)

(C) 2007-2018, Ralf Lämmel, Universität Koblenz-Landau

Übersicht Sortieralgorithmen

	$T_W(n)$	$S_W(n)$
SelectionSort	$\Theta(n^2)$	$\Theta(1)$
MergeSort	$\Theta(n \log n)$	O(n)
...	...	...

Geht es schneller als *n* log *n*?
Geht *n* log *n* auch ohne Speicheraufwand?
Warum verwenden wir SelectionSort?

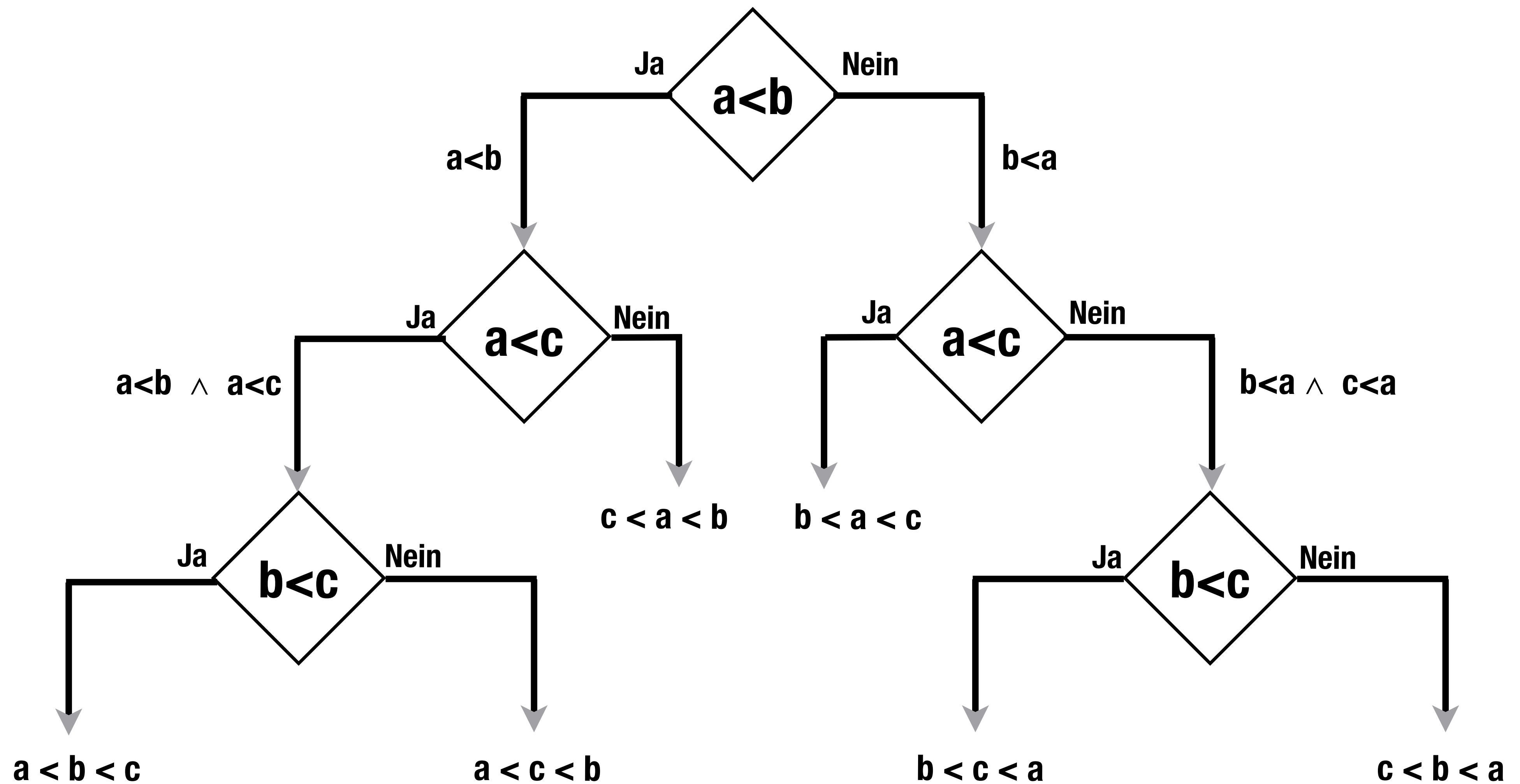
Satz: Jeder ***vergleichsbasierte*** Algorithmus zum Sortieren braucht $\Omega(n \log n)$ Vergleiche im schlimmsten Fall. Also ist $n \log n$ eine untere Schranke für T_w.

Bedeutung von Schrankenaussagen

- ***Obere Schranke* (O(*f*(*n*))**: Um zu zeigen, dass ein Problem (z.B. Sortieren) in f(*n*) gelöst werden kann, genügt es ***einen*** Algorithmus mit dieser Komplexität anzugeben.

- ***Untere Schranke* (Ω(*f*(*n*))**: Um zu zeigen, dass ein Problem nicht schneller als f(*n*) gelöst werden kann, muss man zeigen, dass ***alle*** möglichen Algorithmen nicht schneller sein können.

Ein Entscheidungsbaum zum Sortieren einer Liste mit drei verschiedenen Elementen a, b, c durch binäre Vergleiche

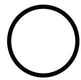


Sortieralgorithmen mögen noch mehr Vergleiche bemühen.

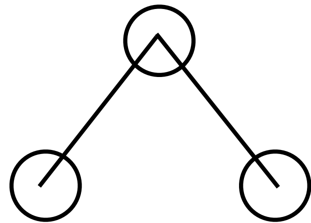
Der Entscheidungsbaum

- Jeder Knoten vergleicht zwei Elemente.
- Der Baum ist binär wegen der Entscheidungen.
- Alle möglichen Permutationen müssen unterschieden werden.
- Also gibt es **n! Blätter** bei n Elementen in der Eingabe.
- **Was ist die Tiefe eines Binärbaumes mit n! Blättern?**

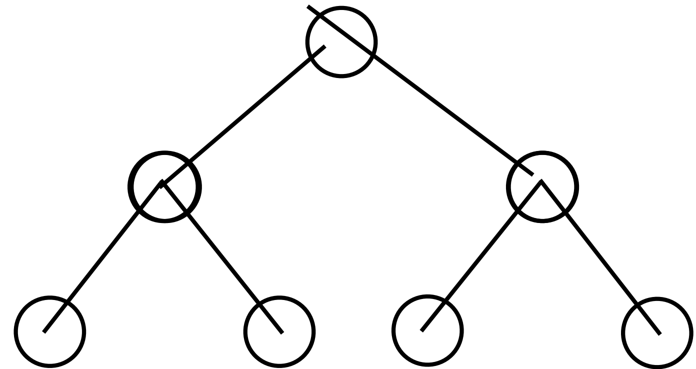
Zusammenhang zwischen Höhe und Anzahl der Blätter



Höhe: 0
Knoten: 1
Blätter: 1



Höhe: 1
Knoten: 3
Blätter: 2



Höhe: 2
Knoten: 7
Blätter: 4

Höhe: h
Knoten: $2^{h+1} - 1$
Blätter: 2^h

Wenn der Baum nicht vollstândig ist, dann ist die Höhe größer bei gleicher Blattzahl.

Höhe: $\log_2(n!)$
Blätter: n!

http://de.wikipedia.org/wiki/Bin%C3%A4rbaum

Vergleichsbasiertes Sortieren

- $T_W(n)$

  $\geq$ "längster Pfad"

  $\geq \log(n!)$

  $= \log n + \ldots + \log 1$

  $\geq \log n + \ldots + \log (n/2)$

  $\geq n/2 (\log n/2)$

  $= \Omega(n \log n)$

- Jeder vergleichsbasierte Algorithmus braucht $\Omega(n \log n)$ Vergleiche im schlimmsten Fall.

Komplexität rekursiver Algorithmen

- Ein rekursiver Algorithmus ohne Schleifen hat die Komplexität f(x) wenn f(x) die Anzahl der rekursiven Aufrufe in Abhängigkeit von der Eingabe x beschreibt.
- Beispiel n!
  - n rekursive Aufrufe

Komplexität Fibonacci

- Definition
  - fib(n) =
    - 0, falls n = 0
    - 1, falls n = 1
    - fib(n-2) + fib(n-1), n >= 2
- Für *n* > 2 zwei direkt rekursive Aufrufe
- **Was ist die resultierende Komplexität?**

Komplexität Fibonacci

- Anzahl der Aufrufe insgesamt
  - Exponentiell steigend (zur Basis 2) mit *n*
- Fazit: fib(n) ist $O(2^n)$

Zusammenfassung

Die Kosten in Bezug auf Laufzeit und Speicherverbrauch sind wichtige Eigenschaften von algorithmischen Problemen bzw. konkreten Implementationen.

(C) 2007-2018, Ralf Lämmel, Universität Koblenz-Landau