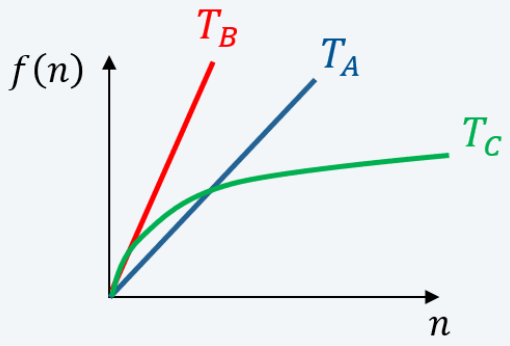
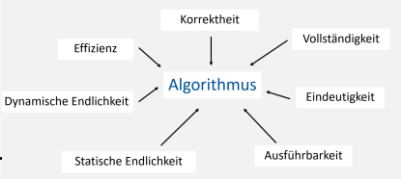
Studiengang Software Engineering
Fakultät für Informatik, Kommunikation, Medien
Campus Hagenberg

8 Laufzeitkomplexität von Algorithmen



8.1 Begriff und Abgrenzung



Laufzeit, Speicherplatz und andere Aspekte (z. B. Struktur, Energiebedarf) sind Eigenschaften eines Algorithmus, die seine Qualität kennzeichnen (siehe Kapitel 1)

Unter dem Begriff Komplexität (engl. *complexity*) verstehen wir i. S. d. Informatik den für die Ausführung – manchmal auch den für das Verstehen bzw. für die Korrektheitsprüfung – eines Algorithmus erforderlichen Aufwand

mehr Komplexität bedeutet mehr Aufwand und umgekehrt

Unter Aufwand verstehen wir in erster Linie den Speicherbedarf und die Laufzeit (Rechenzeit), die ein Algorithmus (oder Programm) zu seiner Ausführung benötigt

Deshalb sprechen wir von der Speicherkomplexität und der Laufzeitkomplexität eines Algorithmus

Höhere Strukturkomplexität führt zu höherem Testaufwand -- mehr Bedingungen

Begriff und Abgrenzung

Den Begriff Komplexität können wir also mit „Aufwand“ in Bezug setzen

Laufzeitkomplexität (Zeitaufwand) – dieses Kapitel

- Wie lange braucht ein Algorithmus, um ein Ergebnis zu liefern?
- Von welchen Parametern hängt die Laufzeit ab?
- Wie ändert sich die Laufzeit wenn sich die „Problemgröße“ ändert?

Parameter hier: Alles was eine Auswirkung auf die Laufzeit haben kann

-- Algo für 1000 Elemente
-- wie verändert sich die Laufzeit bei 2000 Elementen

Speicherkomplexität (Speicheraufwand)

- Wie viel Speicher braucht ein Algorithmus?
- Von welchen Parametern hängt der Speicherbedarf ab?

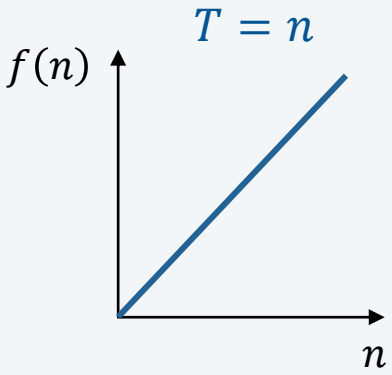
Strukturkomplexität (Test-/Verständnisaufwand) – Kapitel 1

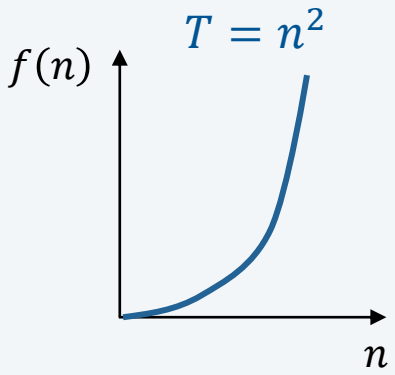
- Wie viele Verzweigungen sind in einem Algorithmus enthalten?

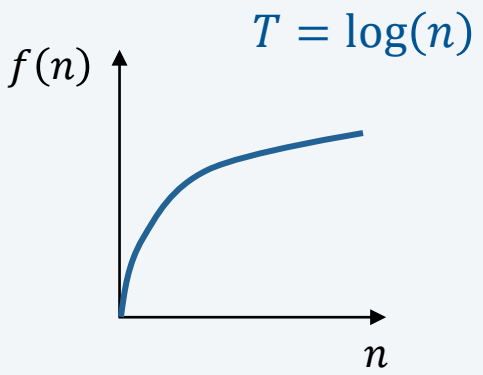
Laufzeitkomplexität

Die Laufzeitkomplexität ist eine Funktion einer Problemgröße n die den „Zeitaufwand“ zur Lösung der Aufgabe beschreibt n z.B. Anzahl Elemente im Feld

Laufzeit $T = f(n)$







Typische Problemgrößen:

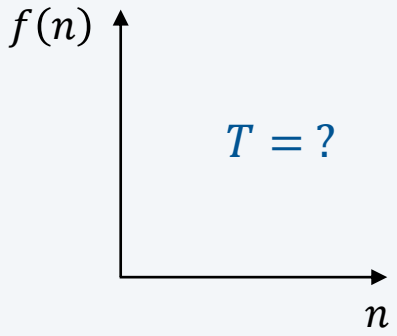
- Anzahl der Elemente eines Felds oder einer Matrix
- Anzahl der Knoten einer verketteten Liste oder eines Baums
- Länge einer Zeichenkette
- Grad eines Polynoms (Anzahl der Koeffizienten)

Fragestellungen bei der Komplexitätsanalyse

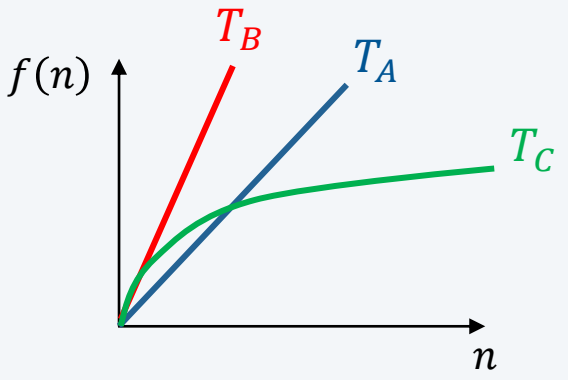
Zur Laufzeitkomplexität eines Algorithmus

- Wie schnell ist ein bestimmter Algorithmus (z.B. ein Sortieralgorithmus)?

Wie verhält sich die Laufzeit des Algorithmus in Abhängigkeit von der Problemgröße?



Wie verhält sich die Laufzeit eines Algorithmus im Vergleich mit anderen Algorithmen?



- Welchen Wert hat eine Optimierung?

-- z.B. Lohnt sich eine Optimierung vor einer Auslieferung der Software

Analyse der Laufzeitkomplexität

Gegeben: Algorithmus

Beispiel: Sequenzielle Suche in einem Feld:
günstig: gesuchtes Element an erster Stelle
ungünstig: letzte Stelle

Gesucht: seine Laufzeit im günstigsten, ungünstigsten und durchschnittlichen Fall in Abhängigkeit von einer oder mehreren Problemgrößen n, [m, x, y, …]

- Grobanalyse (Laufzeitabschätzung)
  - Ermittlung der Anzahl erforderlicher Schleifendurchläufe (iterative Algorithmen) oder Prozeduraufrufen (rekursive Algorithmen)
  - Details wie einzelne Anweisungen oder Ausdrucksauswertungen (sofern nicht essenziell) bleiben unberücksichtigt
  - Analyse unabhängig vom verwendeten Prozessortyp und der verwendeten Programmiersprache
- Feinanalyse (Laufzeitberechnung)
  - Analyse jeder einzelnen Anweisung und jeder Ausdrucksauswertung
  - Berechnung der Laufzeit bezogen auf einen bestimmten Prozessortyp und ggf. der eingesetzten Programmiersprache (Compiler)

8.2 Grobanalyse (Laufzeitabschätzung)

Vorgehensweise

- Bestimmen der für das Laufzeitverhalten wesentlichen Problemgröße
- Bestimmen der minimalen, maximalen und durchschnittlichen Anzahl der wesentlichen algorithmischen Schritte (z. B. Suchschritte, Schleifendurchläufe) in Abhängigkeit der Problemgröße
  - günstiger Fall (*best case*): der Algorithmus kommt mit einer minimalen Anzahl von Arbeitsschritten aus
  - ungünstiger Fall (*worst case*): erfordert eine maximale Anzahl von Arbeitsschritten
  - durchschnittlicher Fall (*average case*): die Anzahl der Arbeitsschritte repräsentiert den typischen, über viele Anwendungen hinweg betrachteten und gemittelten Anwendungsfall

Schleifenstrukturen und ihre Auswirkung auf das Laufzeitverhalten

Beispiel 1: Zählschleife

```
F(↓n: int): int
  var i, fn: int
begin
  fn := 1
  for i := 1 to n do
    fn := fn * i
  end
  return fn
end F
```

Parameter n ist die Problemgröße

Anzahl der Schleifendurchläufe immer genau n
d. h. die Laufzeit T steigt linear mit n
Verdoppelung von n ⇒Verdoppelung der Laufzeit

Beispiel 2: Geschachtelte Schleifen

```
for i := 1 to n do
  for j := 1 to n do
    A
  end
end
```

für jeden äußeren Schleifendurchlauf gibts weitere n innere Durchläufe
-- quadratisch

Anweisung A wird n^2 mal durchlaufen,
d. h. die Laufzeit T steigt mit dem Quadrat der Problemgröße n
Verdoppelung von n ⇒ Vervierfachung der Laufzeit

Schleifenstrukturen und ihre Auswirkung auf das Laufzeitverhalten

Beispiel 3: Abweisschleife

```
P(↓n: int): int
  var p: int
begin
  p := 1
  while p ≤ n do
    p := p * 2
  end
  return p
end P
```

Anweisung wird $log_2(n)$ Mal durchlaufen
d. h. die Laufzeit T steigt mit Logarithmus von n
Verdoppelung von n ⇒ Erhöhung der Laufzeit um konstanten Betrag

bei 2er log ist das 1

Beispiel 4: Geschachtelte Schleifen

innere Schleife immer zuerst n mal durchlaufen, dann n-1 dann n-2 ..... n-n

```
for i := 1 to n do
  for j := i to n do
    A
  end
end
```

Anweisung A wird $\frac{n(n+1)}{2}$ Mal durchlaufen,
d. h. die Laufzeit T steigt mit dem Quadrat der Problemgröße n
Verdoppelung von n ⇒ Vervierfachung der Laufzeit

Grobanalyse des Laufzeitverhaltens

Beispiel: Binäre Suche in Feldern

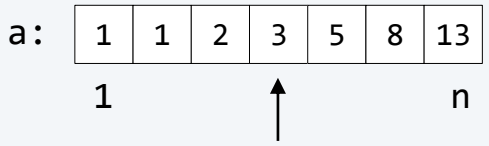
- Gegeben: ein aufsteigend sortiertes Zahlenfeld mit n Elementen
- Gesucht: Index von x im Feld (index = 0 wenn x nicht in Feld enthalten ist)

Lösungsidee:

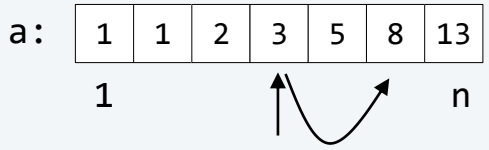
- Es wird geprüft, ob der gesuchte Schlüsselwert mit dem des mittleren Elementes übereinstimmt
- Wenn das nicht der Fall ist, muss (durch den Vergleich des gesuchten Schlüsselwerts mit dem Wert der Schlüsselkomponente des gerade betrachteten Elements) festgestellt werden,
  - ob in der „linken Hälfte“
  - oder in der „rechten Hälfte“

  des Behälters analog (ev. rekursiv) weitergesucht werden muss

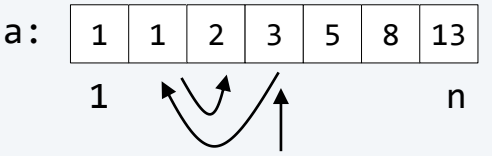
Beispiel: x = 3



Beispiel: x = 8



Beispiel: x = 2



Grobanalyse des Laufzeitverhaltens

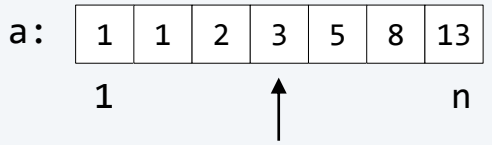
Algorithmus für die binäre Suche

```
IndexOf(↓a: array[1:n] of int ↓x: int): int
  var
    index, m, first, last: int
begin
  index := 0; first := 1; last := n
  while index = 0 and first ≤ last do
    m := (first + last) div 2
    if x = a[m] then
      index := m
    elsif x < a[m] then
      last := m - 1
    else -- x > a[m]
      first := m + 1
    end
  end
  return index
end IndexOf
```

Grobanalyse des Laufzeitverhaltens

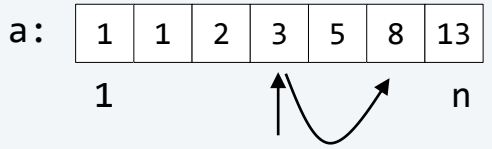
Anzahl der Suchschritte hängt offensichtlich neben n auch von x ab

Beispiel x = 3



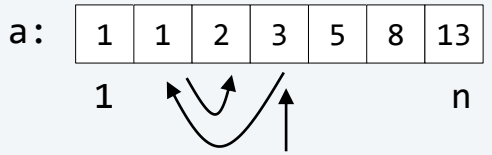
$\Rightarrow$ 1 Schritt

Beispiel x = 8



$\Rightarrow$ 2 Schritte

Beispiel x = 2



$\Rightarrow$ 3 Schritte

Wie oft wird Schleife in Abhängigkeit von n und x durchlaufen?

Minimum: $S_{min}(n)$, Maximum: $S_{max}(n)$, Durchschnitt: $S_{avg}(n)$

Grobanalyse des Laufzeitverhaltens

Minimal erforderliche Anzahl von Suchschritten (Schleifendurchläufen)

- Anzahl der Schleifendurchläufe im günstigsten Fall (wir finden das gesuchte Element auf Anhieb):
- $S_{min}(n) = 0$ für $n = 0$
- $S_{min}(n) = 1$ für $n \geq 1$ gesuchter Wert an Stelle $\frac{1+n}{2}$

Grobanalyse des Laufzeitverhaltens

Maximal erforderliche Anzahl von Suchschritten (Schleifendurchläufen)

- Anzahl der Schleifendurchläufe im ungünstigsten Fall
- Ausprobieren für ein paar kleine Werte; dann Gesetzmäßigkeit ableiten

dann braucht man einen Schleifendurchlauf mehr

$S_{max}(1) = 1$	2^0 Elemente
$S_{max}(2) = 2$	2^1 Elemente
$S_{max}(3) = 2$	
$S_{max}(4) = 3$	2^2 Elemente
$S_{max}(5) = 3$	
$S_{max}(6) = 3$	
$S_{max}(7) = 3$	
$S_{max}(8) = 4$	2^3 Elemente

Folgerung: Wenn die Anzahl der Elemente n um eine 2er-Potenz steigt, dann steigt $S_{max}(n)$ um 1

Für $2^k \leq n < 2^{k+1}$ ist

$$S_{max}(n) = k + 1$$

also

$$S_{max}(n) = \lfloor log_2(n) \rfloor + 1$$

Grobanalyse des Laufzeitverhaltens

Durchschnittlich erforderliche Anzahl von Suchschritten (Schleifen-durchläufen)

Zwei Fälle

$$S_{avg}(n) = S_{max}(n) = \lfloor log_2(n) \rfloor + 1 \quad \text{wenn } x \notin a$$

$$S_{avg}(n) = ? \quad \text{wenn } x \in a$$

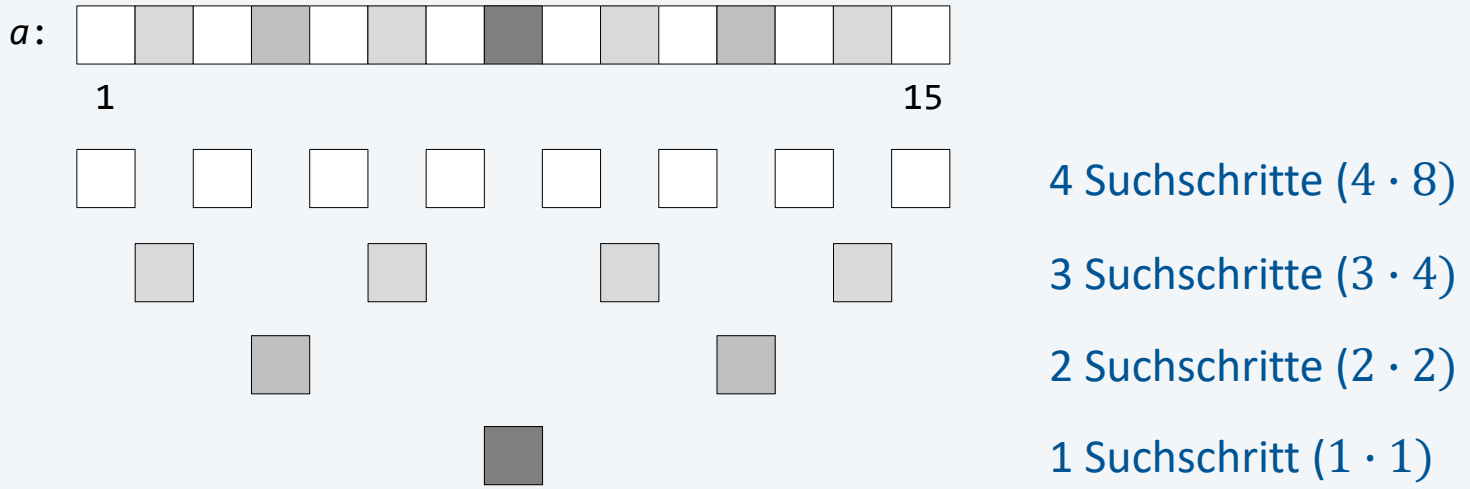
Lösung:

- Alle Elemente des Feldes einmal suchen
- Für jede Elementsuche die Anzahl der Schleifendurchläufe ermitteln und die Summe aller Schleifendurchläufe durch die Anzahl der Elemente dividieren $S_{avg}(n) = \frac{sum(n)}{n}$ (1)
- wir betrachten Felder der Länge $n = 2^k - 1$ mit ganzzahligem k

damit kann man immer durch zwei Teilen und ein Wert in der Mitte bleibt übrig
-- nicht bei jeder ungeraden Zahl, normalerweise geht das nur 1 mal

Grobanalyse des Laufzeitverhaltens

Beispiel für $n = 15$, $(15 = 2^4 - 1\ )$



$$sum(15) = 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 4 + 4 \cdot 8 = 49$$

$$sum(15) = 1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + 4 \cdot 2^3 = 49$$

Weitere Beispiel für $n = 31$ und $n = 63$

$$sum(31) = 1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + 4 \cdot 2^3 + 5 \cdot 2^4$$

$$sum(63) = 1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + 4 \cdot 2^3 + 5 \cdot 2^4 + 6 \cdot 2^5$$

Grobanalyse des Laufzeitverhaltens

Allgemein (für $n = 2^k - 1$)

$$sum\left(2^k - 1\right) = \sum_{i=1}^{k} i \cdot 2^{i-1} = \ \ldots = (k-1) \cdot 2^k + 1 \qquad (2)$$

Formel aus chatgpt :D

Beispiel für $k = 4$

$$sum(15) = sum(2^4 - 1) = (4-1) \cdot 2^4 + 1 = 3 \cdot 16 + 1 = 49$$

Aus (1) und (2) folgt

$$s_{avg}(n) = s_{avg}\left(2^k - 1\right) = \frac{sum\left(2^k - 1\right)}{2^k - 1} = \frac{(k-1) \cdot 2^k + 1}{2^k - 1}$$

- Für große k gilt $S_{avg}(n) \approx (k-1)$ $(+1\ , -1$ vernachlässigen)
- Im schlechtesten Fall ist die binäre Suche nur geringfügig langsamer als im Durchschnitt.

Da n=2^(k-1) kann man das Ergebnis umformen und kommt auf eine Laufzeit von T(n) = log2(n)

8.3 Feinanalyse (Laufzeitberechnung)

- Alle Anweisungen und Ausdrucksauswertungen werden berücksichtigt
  - Wie oft werden diese ausgeführt?
  - Wie lange dauert die Ausführung?
- Wir rechnen dabei nicht mit echten Ausführungszeiten, denn die sind prozessorabhängig, sondern mit Zeiteinheiten bezogen auf eine Referenzoperation (z. B. die Wertzuweisung = 1.0) fikitive Zeiteinheit
- Ausführungszeiteinheiten für einen fiktiven Prozessor, z. B.:

Operation	Dauer
Wertzuweisung	1.0
Vergleich	1.6
Addition / Subtraktion	0.8
Multiplikation	2.3
Division / Restbildung	4.9
Einfache Indizierung	2.1

Referenzoperation

Vorgehensweise

(1) Ermitteln, wie oft jede Anweisung/jeder Ausdruck ausgeführt wird

```
x := 1    i := 1
while i ≤ n do
  x := x * i
  i := i + 1
end
```

Anzahl
1
u + 1
u
u

u = Anzahl der Schleifendurchläufe

(2) Ausführungszeiten für jede Anweisung/jeden Ausdruck ermitteln

```
x := 1    i := 1
while i ≤ n do
  x := x * i
  i := i + 1
end
```

Anzahl	Dauer
1	2.0
u + 1	1.6
u	3.3
u	1.8

(3) Gesamtausführungszeit ermitteln

$$2.0 \cdot 1 + 1.6(u + 1) + 3.3u + 1.8u = 3.6 + 6.7u$$

Beispiel: Feinanalyse „binäre Suche in Feldern“

(1) Häufigkeit der Anweisungsausführung ermitteln

```
BinarySearch1(↓a[1:n] ↓x ↑index)
begin
  min := 1; max := n; index := 0
  while (index = 0)
         and (min ≤ max) do
    m := (min + max) / 2
    if x = a[m] then
      index := m
    elsif x < a[m] then
      max := m - 1
    else
      min := m + 1
    end
  end
end BinarySearch1
```

Anzahl

das AND

```
    1
u + 1
    u
    u
    u
    1
u - 1
    v

u – 1 - v
```

+1 ist wenn index ungleich 0

u weil bei Kurzschlussauswertung dann nicht mehr geprüft wird

Annahme, dass Element gefunden wird

Beispiel: Feinanalyse „binäre Suche in Feldern“

(2) Ausführungszeiten für jede Anweisung/jeden Ausdruck ermitteln

```
BinarySearch1(↓a[1:n] ↓x ↑index)
begin
  min := 1; max := n; index := 0
  while (index = 0)
         and (min ≤ max) do
    m := (min + max) / 2
    if x = a[m] then
      index := m
    elsif x < a[m] then
      max := m - 1
    else
      min := m + 1
    end
  end
end BinarySearch1
```

	Anzahl	Dauer
	1	3.0
	u + 1	1.6
	u	1.6
	u	6.7
	u	3.7
	1	1.0
	u - 1	3.7
	v	1.8
	u – 1 - v	1.8

(3) Gesamtausführungszeit ermitteln: $-0.1 + 19.1u$

Funktion ist nur ab 1 sinnvoll
u ist mindestens 1, da wir Anfangs definiert haben dass wir das Element finden (u = Anzahl der Schleifen= durchläufe wir fordern dass u min. 1)

Wie könnte man Algorithmus schneller machen?

```
BinarySearch1(↓a[1:n] ↓x ↑index)
begin
  min := 1; max := n; index := 0
  while (index = 0)
         and (min ≤ max) do
    m := (min + max) / 2
    if x = a[m] then
      index := m
    elsif x < a[m] then
      max := m - 1
    else
      min := m + 1
    end
  end
end BinarySearch1
```

wird bei jedem Schleifendurchlauf geprüft

wird bei jedem Schleifendurchlauf geprüft

kann man mit einem break "beheben"
-- maximale Optimierung auf LK

Optimierungsmöglichkeiten:

- Aus Schleife springen wenn Element gefunden
- Zuerst `if x < a[m]` prüfen und die Prüfung `x = a[m]` zum Schluss

Beispiel: Feinanalyse „binäre Suche in Feldern“

Feinanalyse des Optimierungsversuches für „binäre Suche in Feldern

u: Anzahl der Schleifendurchläufe

```
BinarySearch2(↓a[1:n] ↓x ↑index)
begin
  min := 1; max := n
  while true do
    if min > max then
      index := 0; return
    end
    m := (min + max) / 2
    if x < a[m] then
      max := m – 1
    elsif x > a[m] then
      min := m + 1
    else
      index := m; return
    end
  end
end BinarySearch2
```

	Anzahl	Dauer
	1	2
	u	
	0	
Gesamt		

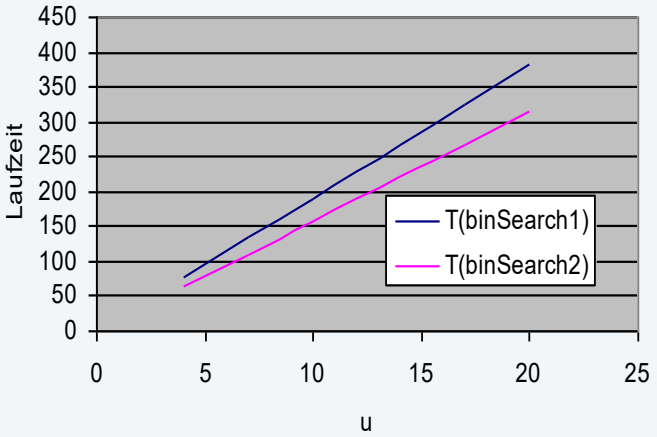
Vergleich der beiden Implementierungsvarianten

$$T_1 = -0.1 + 19.1u$$

$$T_2 = 1.2 + 15.65u$$

-- Bei der Annahme v = u/2

n	u	T_1	T_2
10	4	77	64
100	7	134	111
1000	10	191	158
10000	14	268	220
100000	17	325	267
1000000	20	382	314



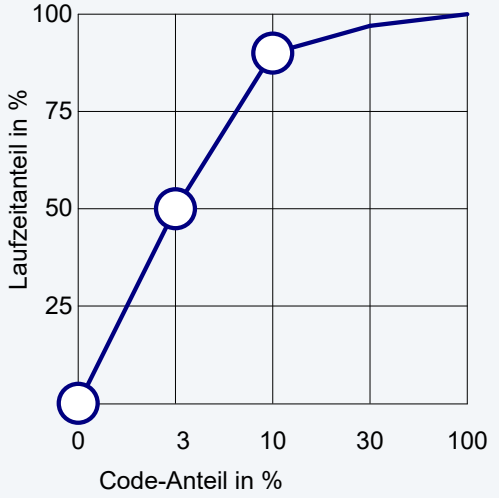
Ergebnis

- Optimierung `BinarySearch`$_2$ führt zu etwas schnellerer Lösung
- Lohnt sich die Verbesserung wirklich, wenn Rechenzeit für eine Wertzuweisung ca. 0.001 μsec ist?
- Differenz bei n = 10 000 ist 48*0.001 μ sec
  bringt Verbesserung von 0.048 μsec ⇒ Verbesserung lohnt sich nicht!

8.4 Laufzeitmessung

- Schätzungen sind oft zu ungenau
- Feinanalyse ist nur bei Teilen eines Softwaresystems leistbar
- Studie von Hennessy & Patterson: 3 % des Codes sind für 50 % der Laufzeit verantwortlich

Code	Laufzeit
3 %	50 %
10 %	90 %



- Optimierung bringt nichts, wenn sie an den falschen Stellen vorgenommen wird

Instrumentalisierung zur Laufzeitmessung

Mögliches Schema für Zeitmessung:

```
t := Time()
A
Done(↓1 ↕t)
if … then
  B
  Done(↓2 ↕t)
else
  C
  Done(↓3 ↕t)
end
D
Done(↓4 ↕t)
```

```
Done(↓nr: int ↕t: real)
begin
  e := Time()
  cnt[nr] := cnt[nr] + 1
  tm[nr] := tm[nr] + (e - t)
  t := Time()
end Done
```

Berücksichtigung von Messfehlern

- Zeit für Aufrufe von `Time()` wird auch gemessen
- Korrektur durch „overhead“

```
overhead := 0
for i := 1 to 1000 do
  t1 := Time()
  t2 := Time()
  overhead := overhead + (t2 - t1)
end
overhead := overhead / 1000
```

```
Done(↓nr: int ↕t: real)
begin
  e := Time()
  cnt[nr] := cnt[nr] + 1
  tm[nr] := tm[nr] + (e - t) - overhead
  t := Time()
end Done
```

8.5 Asymptotische Laufzeitkomplexität

- Bisher detaillierte Betrachtung des Laufzeitverhaltens (erlaubt genauen Vergleich, aber die Faktoren hängen von der Implementierung ab)
- Exakte Analyse ist oft zu schwierig bzw. zu aufwändig
- Oft will man nur wissen, welcher Algorithmus für große *n* günstiger ist, z.B. beim Suchen, beim Sortieren
  - Wachstum der Laufzeit in Abhängigkeit einer Problemgröße?
  - Skalierbarkeit: Wenn Laufzeit für kleine Problemgrößen (z.B. $n = 100$) ausreichend ist, ist sie es auch für große Problemgrößen (z.B. $n = 1.000.000$)?

- Typische Frage: Wie verhält sich ein Algorithmus, wenn man *n* verdoppelt, verzehnfacht, etc.

Vergleich zweier Algorithmen hinsichtlich ihres Laufzeitverhaltens

Gegeben sind zwei Algorithmen mit folgendem Laufzeitverhalten

$$T_1(n) = 1000n + 500$$
$$T_2(n) = n^2 + 2n + 10$$

Vergleich der Laufzeiten

ab ca. $n = 1000$ ist T_2 langsamer als T_1

Asymptotische Laufzeitkomplexität

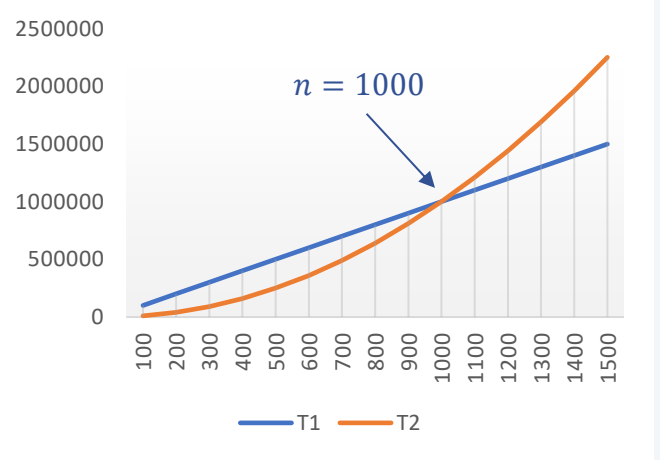
T_1 wächst linear: $T_1(n) \in O(n)$

T_2 wächst quadratisch: $T_2(n) \in O(n^2)$

$T(n) \in O\big(f(n)\big)$ bedeutet:

die Funktion T wächst wie die Funktion $f(n)$

$T(n)$ ist von der Ordnung $f(n)$



n	T_1	T_2
1	1.500	13
10	10.500	130
100	100.500	10.210
1.000	1.000.500	1.002.010
10.000	10.000.500	100.020.010

=> Für große n ist T1 immer schneller!

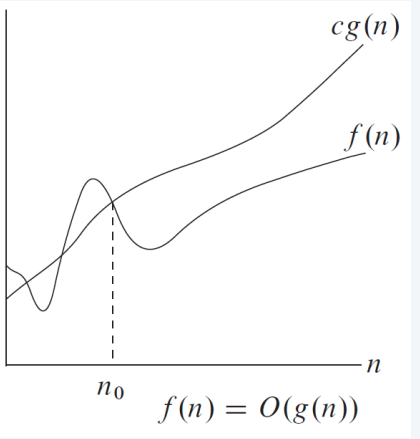
8.4.1 Notationen

Die O-Notation (Obere Schranke)

Wenn f und g Funktionen sind, die natürliche Zahlen auf positive reelle Zahlen abbilden, so ist f von der Ordnung g, wenn f ein Element folgender Menge von Funktionen ist:

$$O\big(g(n)\big) = \{\, f(n) : \text{es gibt positive Konstanten } n_0 \text{ und } c, \text{ so dass für alle } n \geq n_0 \text{ gilt: } 0 \leq f(n) \leq c \cdot g(n)\}$$

- Konstanten $n_0 > 0$ und $c > 0$ können beliebig groß sein!
- Man schreibt auch $f(n) \in O\big(g(n)\big)$ oder kürzer und weniger exakt $f(n) = O\big(g(n)\big)$



Die O-Notation

Beispiel: Lineare Laufzeitkomplexität

- Annahme: für $T(n) = 7n + 64$ gilt $T(n) \in O(n)$
- Zu zeigen: es existieren n_0 und c, so dass für $n > n_0$ gilt $T(n) \leq c \cdot n$

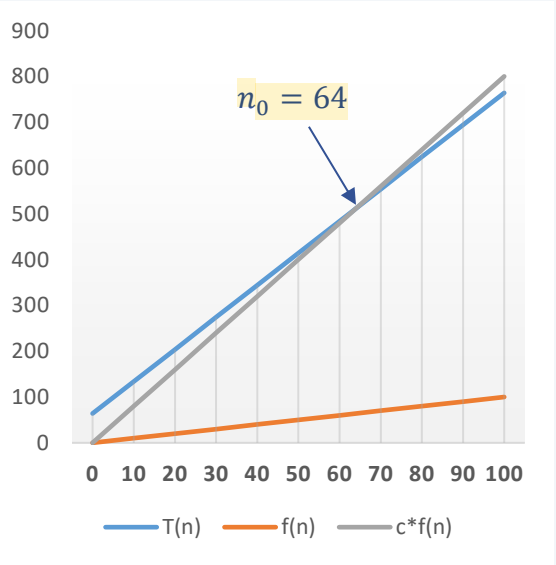
$$7n + 64 \leq c \cdot n$$

- Damit $c \cdot n$ stärker als $7n + 64$ wächst, muss c mindestens 8 sein

$$7n + 64 \leq 8 \cdot n$$

- Gilt für alle $n \geq 64$, daraus folgt

$T(n) \in O(n)$ mit $n_0 = 64$ und $c = 8$



Die O-Notation

Weitere Beispiele (ohne Beweise)

$n^2 + 2n + 10 \in O(n^2)$

$(5n^2 + 2n + 10) \cdot n \in O(n^3)$

$\log(n) \cdot (2n + 10) \in O(n \cdot log(n))$

Anmerkung:

- Bei Polynomen zählt immer nur höchste Potenz
- Konstante Faktoren werden nicht angeschrieben

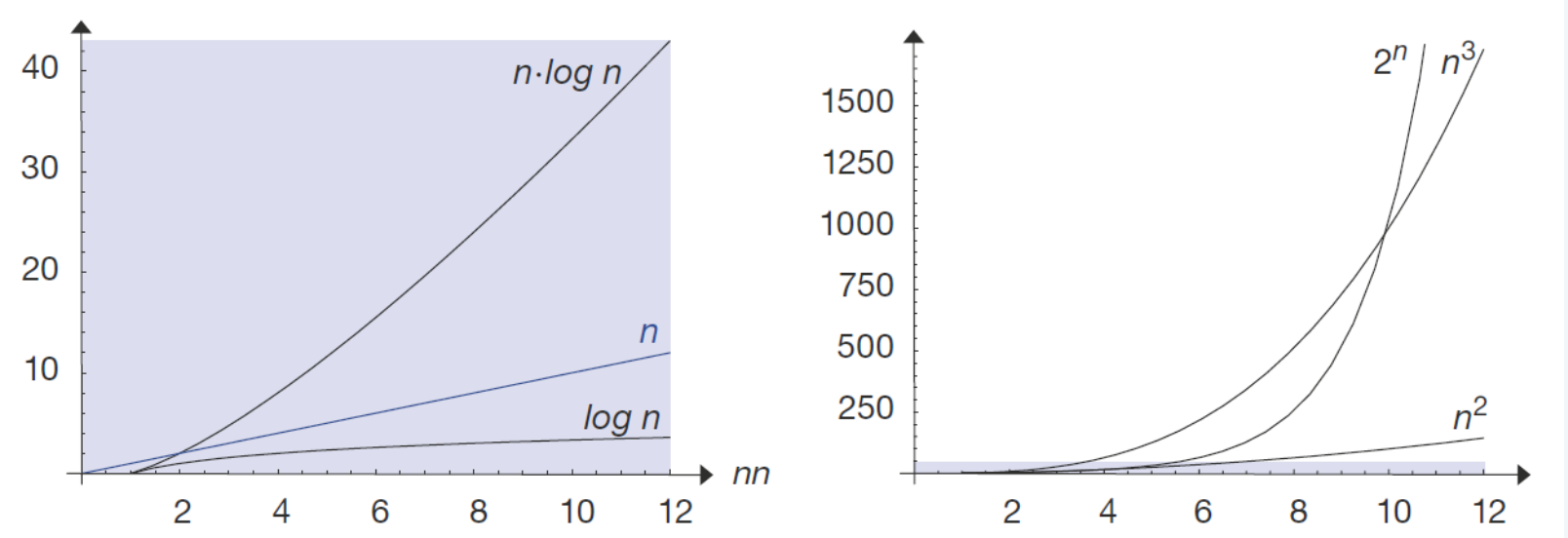
Komplexitätsklassen

Klasse	Bezeichnung	Beschreibung
$O(1)$	konstant	Laufzeit ist unabhängig von der Problemgröße, optimaler Fall, der praktisch nicht auftritt
$O(\log n)$	logarithmisch	Verdoppelung der Problemgröße bewirkt Anstieg der Laufzeit um *log* 2, also um eine Konstante (um 1 für *ld*), sehr günstig und daher erstrebenswert, z.B. *binäre Suche*
$O(n)$	linear	Verdoppelung der Problemgröße bewirkt Verdoppelung der Laufzeit, immer noch zufrieden stellend, z.B. *sequenzielle Suche*
$O(n \log n)$	-	Fast so gut wie linear, weil *log n* im Verhältnis zu *n* klein ist, z.B. gute Sortierverfahren wie *Quicksort*

Komplexitätsklassen

Klasse	Bezeichnung	Beschreibung
$O(n^2)$	quadratisch	Verdoppelung der Problemgröße bewirkt Vervierfachung der Laufzeit, ungünstig, z.B. schlechte Sortierverfahren wie *Bubblesort*
$O(n^3)$	kubisch	Verdoppelung der Problemgröße bewirkt Verachtfachung der Laufzeit, sehr unbefriedigend, z.B. einfache *Matrizenmultiplikation*
$O(2^n)$	exponentiell	Verdoppelung der Problemgröße bewirkt Quadrierung (weil $2^{2n} = (2^n)^2$) der Laufzeit, katastrophal, z.B. *Backtracking* oder *Türme von Hanoi*

Wachstumskurven der Komplexitätsklassen



Komplexitätsklassen – Vergleich

Annahme: 1 Schritt dauert 1 μsec:

n	$O(n)$	$O(n \cdot \log_2 n)$	$O(n^2)$	$O(2^n)$
10	10 μs	33 μs	0.1 ms	1 ms
20	20 μs	86 μs	0.4 ms	1 sec
50	50 μs	280 μs	2.5 ms	35 Jahre
100	100 μs	660 μs	10 ms	10^{16} Jahre
200	200 μs	1.5 ms	40 ms	10^{46} Jahre
500	500 μs	4.4 ms	250 ms	10^{137} Jahre
1000	1 ms	9.9 ms	1 sec	10^{287} Jahre

- Das Universum existiert seit ca. 10^{10} Jahren
- Beim Urknall gestarteter $O(2^n)$ -Algorithmus wäre heute gerade bei $n = 88$

Asymptotische LK von Algorithmen*systemen*

Gegeben sind $T_1(n) \in O\big(f_1(n)\big)$ und $T_2(n) \in O\big(f_2(n)\big)$

Sequenz von Algorithmen(teilen): $T_1(n) + T_2(n) \in O\left(max\big(f_1(n), f_2(n)\big)\right)$

- In einer Sequenz ist die Gesamtkomplexität durch die Komplexität des langsamsten Algorithmus bestimmt
- Sequenz von linear, logarithmisch und quadratisch ergibt $O(n^2)$

Schachtelung von Algorithmen(teilen): $T_1(n) \cdot T_2(n) \in O\big(f_1(n) \cdot f_2(n)\big)$

- Bei geschachtelten Algorithmen ergibt sich die Gesamtkomplexität durch Multiplikation der Einzelkomplexitäten
- Schachtelung von linear, logarithmisch und quadratisch ergibt $O\big(n^3 \cdot log_2(n)\big)$

8.6 Minimale Laufzeitkomplexität/Problemkomplexität

Unter dem Begriff der Problemkomplexität oder minimalen Laufzeitkomplexität verstehen wir die Laufzeitkomplexität, die jener Algorithmus mit dem geringsten Rechenaufwand zur Lösung einer Aufgabenstellung, die zu einer bestimmten Problemklasse gehört, aufweist.

Deshalb wird die Problemkomplexität auch als minimale Laufzeitkomplexität bezeichnet

Fragestellung zur Problemkomplexität

- Wie schnell kann man prinzipiell sortieren, suchen, …?

Minimale Laufzeitkomplexität/Problemkomplexität

Minimale Laufzeitkomplexität ausgewählter Problemklassen:

- Suche in sortierten Feldern $O\left(log_2(n)\right)$ (Binäre Suche)
- Suche in Feldern nach arithmetischem Schlüssel $O(1)$ (Hashing)
- Berechnung eines Polynoms der Ordnung $O(n)$ (Hornerschema)
- Multiplikation zweier n-stelligen Zahlen $O\left(n^{log_2(3)}\right) \approx O(n^{1.58})$ (Karatsuba, Ofman)
- Suche einer Zeichenkette der Länge m in einer Zeichenkette der Länge n $O(m+n)$ (Boyer, Moore)
- Multiplikation zweier $n \times n$ Matrizen $O\left(n^{log_2(7)}\right) \approx O(n^{2.81})$ (Strassen)
- Sortieralgorithmen, die auf Vergleich der Werte beruhen $O\left(n \cdot log_2(n)\right)$