Fragestellung: Kostenanalyse von Algorithmen

Aufgabe: Bewertung von Algorithmen bzgl. ihrer Effizienz d.h. Kosten/Aufwand an Ressourcen wie:

- Laufzeit (primäres Interesse)
- Speicher
- Kommunikation

nicht: Entwicklungs-/Wartungsaufwand ($\rightarrow$ Software Engineering)

Gesucht ist eine Systematik zur Aufwandsbewertung für datenabhängige Algorithmen.

Absolute Kosten

Bestimmung der absoluten Kosten durch Messung der Laufzeit für bestimmte Testdaten in bestimmten Ausführungsumgebungen (Sprache, System).

```
long start = System.nanoTime();
// zu messender Code, z.B. Methodenaufruf
long ende = System.nanoTime();
long dauerMS = (ende - start) / 1000000; // Millisekunden
```

$\Rightarrow$ Konkrete Aussage zu konkretem Fall („Benchmark“)

Achtung: Zeitmessung unterliegt Schwankungen (Systemeinflüsse)

$\Rightarrow$ Mehrfachmessung und statistische Auswertung erforderlich

Sinnvolles Vorgehen bei Entwicklung (nur) für bestimmte Szenarien, problematisch ist die Übertragung auf andere Fälle (Daten, Umgebungen, . . .).

Relative Kosten

Relative Kosten machen allgemeine Aussagen

- in Abhängigkeit von kennzeichnenden Problemgrößen (z.B. Anzahl der Elemente)
- durch Bereichsangaben (bester vs. schlechtester Fall)
- durch Abstraktion von spezifischen Einflüssen (z.B. Umgebung).

Daraus sind keine konkreten Aussagen ableitbar, nur relative (Vergleiche).

Lineare Suche: Algorithmus & Implementierung

Aufgabe: Suche nach Element in Sequenz
Verfahren: Naiv durch lineare Suche

- durchlaufe Sequenz (hier: Array) der Reihe nach,
- bis Wert gefunden (Rückgabe Position) oder Ende (Rückgabe -1)

```
public static int sucheLinear(int[] a, int x) {
  for (int i = 0; i < a.length; ++i) {
    if (a[i] == x) {
      return i;
    }
  }
  return -1;
}
```

Frage: Wie „gut“ (hier: effizient) ist die Lösung?

Lineare Suche: Analyse

relative Kosten durch Zählen der Operationen im Code:

```
public static int sucheLinear(int[] a, int x) { // 2 Op.
  for (int i = 0;                               // 1 Op.
       i < a.length;                            // 1 Op. pro Iter.
       ++i) {                                   // 1 Op. pro Iter.
    if (a[i] == x) {                            // 3 Op. pro Iter.
      return i;                                 // 1 Op. bei Fund
    }
  }
  return -1;                                    // 1 Op. bei Nichtfund
}
```

$\Rightarrow$ insgesamt (etwa) 5 Op. pro Iteration und 4 Op. einmalig

Hinweis: Weitere „versteckte“ Operationen;
aus Quellcode nur ungefähre Angabe ablesbar (ggf. nicht deterministisch)

Automatisierte Analyse

Zahl der Operationen $\mathrm{op}(C)$ in Code C nach festem Schema:

- in Folge von Anweisungen $A_1,\ldots,A_k$:

$$\mathrm{op}(A_1 \ldots A_k) = \sum_{i=1}^{k} \mathrm{op}(A_i)$$

- in Fallunterscheidung:

$$\mathrm{op}(\text{if } (B)\ A_t \text{ else } A_f) = \mathrm{op}(B) + \begin{cases} \max(\mathrm{op}(A_t), \mathrm{op}(A_f)) \\ \min(\mathrm{op}(A_t), \mathrm{op}(A_f)) \end{cases}$$

- in Schleife mit n Iterationen:

$$\mathrm{op}(\text{while } (B)\ A) = n \cdot (\mathrm{op}(B) + \mathrm{op}(A)) + \mathrm{op}(B)$$

Best, Worst, Average Case

Durch Fallunterscheidungen und Schleifen unterschiedliche Resultate möglich:

- best case: Minimal mögliche Zahl von Operationen
- worst case: Maximal mögliche Zahl von Operationen
- average/expected case: Durchschnittliche/erwartete Zahl von Operationen

Meist nur worst case betrachtet, da er Mindestgarantie für die Güte des Algorithmus liefert.

average case oft schwer zu ermitteln

Lineare Suche: Best, Worst, Average Case

Kosten der linearen Suche: 5 Op. pro Iteration und einmalig

- bei Fund: 4 Op.
- bei Nichtfund: 4 Op.

Anzahl der Iterationen entspricht

- bei Fund: Position des (ersten) „Treffers“ im Feld
- bei Nichtfund: Anzahl der Elemente

Bei Fund:

- best case: Erstes El. ist Treffer $\Rightarrow$ 1 It. $\Rightarrow 5 \cdot 1 + 3 = 8$ Op.
- worst case: Letztes El. ist Treffer $\Rightarrow$ n It. $\Rightarrow 5 \cdot n + 3$ Op.
- avg. case: Mittel der Fälle $\Rightarrow (n+1)/2$ It. $\Rightarrow (5 \cdot n + 11)/2$ Op.

Bei Nichtfund stets n Iterationen $\Rightarrow 5 \cdot n + 5$ Op.

Kritische Operationen

Die bisherige Annahme *alle Operationen sind gleich teuer* ist

- falsch: z.B. Speicherzugriffe sehr(!) viel teurer
- unpraktisch: Einzeloperationen umständlich zu zählen (und im Einzelnen meist irrelevant)

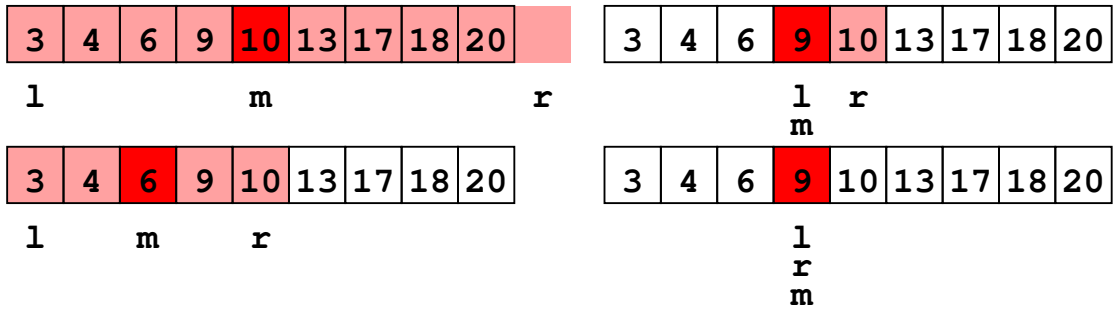
Vereinfachende Annahme: Nur folgende Operationen sind „interessant“:

- Speicherzugriffe (z.B. Lesen/Schreiben von Elementen)
- Operationen auf Problemdaten (z.B. Vergleiche von Elementen)

Binäre Suche: Algorithmus

Binäre Suche: erfordert sortierte(!) Sequenz; dann:

1. Suchintervall zu Beginn gesamte Sequenz
2. Bestimme Mitte und vergleiche Suchwert mit dortigem Wert
   - wenn kleiner gleich: reduziere Intervall auf linke Hälfte
   - wenn größer: reduziere Intervall auf rechte Hälfte
3. wenn Suchintervall von Länge 1: Position gefunden



Binäre Suche: Implementierung

```
public static int suchePosBinaer(int[] a, int x, int li, int re) {
  while (li < re) {      // min eine Position
    int m = (li + re) / 2; // Mitte: li <= m < re
    if (x <= a[m]) {     // Wert liegt links
      re = m;
    }
    else {               // Wert liegt rechts
      li = m + 1;
    }
  }
  return li;             // Wert liegt hier
}                        // oder laege hier
public static int sucheBinaer(int[] a, int x) {
  int p = suchePosBinaer(a, x, 0, a.length);
  return p < a.length && x == a[p] ? p : -1; // gefunden?
}
```

Binäre Suche: Implementierung rekursiv

Binäre Suche lässt sich elegant rekursiv implementieren:

```
public static int suchePosBinaerRek(int[] a, int x, int li, int re) {
 if (li == re) {          // genau eine Position
   return li;             // Position gefunden
 }
 int m = (li + re) / 2; // Mitte: li <= m < re
 return x <= a[m] ? suchePosBinaerRek(a, x, li, m) :
                    suchePosBinaerRek(a, x, m + 1, re);
}
```

Binäre Suche: Analyse

- Länge des Intervalls wird in jedem Schritt halbiert
  (auf-/abgerundet $\rightarrow$ konstant, ignorierbar)
- pro Schritt ist der Aufwand konstant (in $\mathcal{O}(1)$; Definition folgt)
- nach max. k Schritten: Länge ist 1 ($li == re$) $\rightarrow$ fertig

Frage: was ist k? oder: wie oft n halbieren, bis 1?
oder: wie oft 1 verdoppeln, bis n?

Antwort: Logarithmus von n (zur Basis 2)

Algorithmisches Muster: Teile-und-Herrsche bzw. divide-and-conquer
(auch in vielen anderen Zusammenhängen angewendet)

Komplexität eines Algorithmus

Situation: Relative Kosten nicht unbedingt absolut interpretierbar:

z.B. kann Algorithmus mit relativen Kosten $5 \cdot n + 4$ (für kleines n) geringere abs. Kosten haben als Alg. mit relativen Kosten $3 \cdot n + 2$

Alternative Fragestellung: Entwicklung der (relativen&absoluten) Kosten bei Vergrößerung („Skalierung“) des Problems

Beispiel: Wenn Problemgröße vervierfacht wird, wird Laufzeit

- gleich bleiben?
- verdoppelt?
- vervierfacht?
- versechzehnfacht?
- . . . ?

Komplexität vs. Laufzeit

Die folgenden Tabelle gibt einen Eindruck der Laufzeit in Abhängigkeit von der Problemgröße (Spalten) und der Laufzeitfunktion (Zeilen):

	10	20	30	40	50	60
n	0,00001 sek	0,00002 sek	0,00003 sek	0,00004 sek	0,00005 sek	0,00006 sek
n^2	0,0001 sek	0,0004 sek	0,0009 sek	0,0016 sek	0,0025 sek	0,0036 sek
n^3	0,001 sek	0,008 sek	0,027 sek	0,064 sek	0,125 sek	0,216 sek
n^5	0,1 sek	3,2 sek	24,3 sek	1,7 min	5,2 min	13,0 min
2^n	0,001 sek	1 sek	17,9 min	12,7 Tage	35,7 Jahre	366 Jahrhund.
3^n	0,059 sek	58 min	6,5 Jahre	3855 Jahrhund.	2×10^8 Jahrhund.	$1,3 \times 10^{13}$ Jahrhund.

Annahme der Tabelle: CPU schafft ein Megaflop /Sekunde

Komplexität vs. schnellere Rechner

Die folgenden Tabelle untersucht den Effekt schnellerer Rechner (Spalten) auf die in einer Stunde lösbaren Probleminstanzen in Abhängigkeit von der Laufzeitfunktion (Zeilen):

Laufzeit-funktion	derzeitiger Computer	100 mal schneller Computer	1000 mal schneller Computer
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31{,}6N_2$
n^3	N_3	$4{,}64N_3$	$10N_3$
n^5	N_4	$2{,}5N_4$	$3{,}98N_4$
2^n	N_5	$N_5 + 6{,}64$	$N_5 + 9{,}97$
3^n	N_6	$N_6 + 4{,}19$	$N_6 + 6{,}29$

Cave: Gewaltiger Unterschied zwischen polynomiellen und exponentiellen Laufzeitfunktionen!

Komplexitätsklassen

Zusammenfassung von (Kosten-)Funktionen zu Mengen von Funktionen „gleichen Wachstumsverhaltens“ („gleicher Krümmung“)

Benennung der Komplexitätsklassen mit Landau-Symbolen:

- $\mathcal{O}(f)$: Menge aller Funktionen, die höchstens wie f wachsen
- $\Omega(f)$: Menge aller Funktionen, die mindestens wie f wachsen
- $\Theta(f)$: Menge aller Funktionen, die wie f wachsen

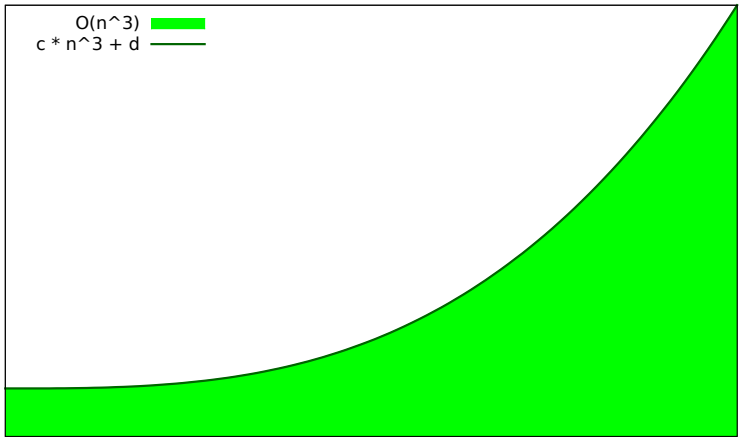
Groß-O

Definition 2.1

$$\mathcal{O}(f) := \{g \mid \exists c > 0, d \in \mathbb{R} : g(n) \leq c \cdot f(n)+d\}$$

anschaulich: $\mathcal{O}(f)$ enthält alle Funktionen g, die sich von (von positivem Vielfachen) der oberer Schranke f nach oben begrenzen lassen

(positives Vielfaches, weil $c \leq 0$ sinnlos wäre)



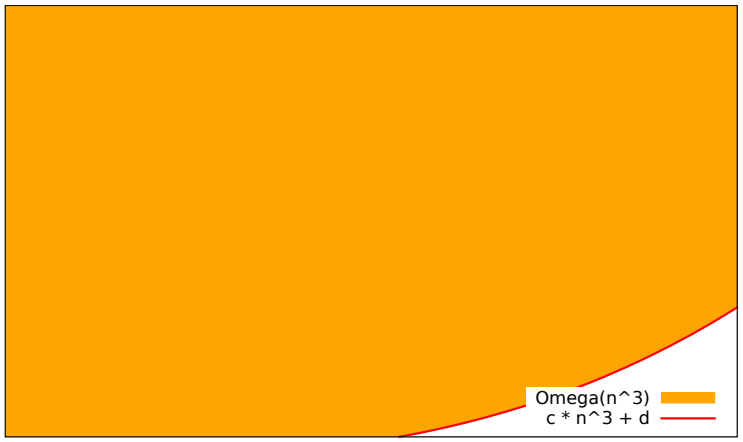
Groß-Omega

Definition 2.2

$$\Omega(f) := \{g \mid \exists c > 0, d \in \mathbb{R} : \ c \cdot f(n)+d \leq g(n)\}$$

anschaulich: $\Omega(f)$ enthält alle Funktionen, die sich von (positivem Vielfachen von) unterer Schranke f nach unten begrenzen lassen

(positives Vielfaches, weil $c \leq 0$ sinnlos wäre)

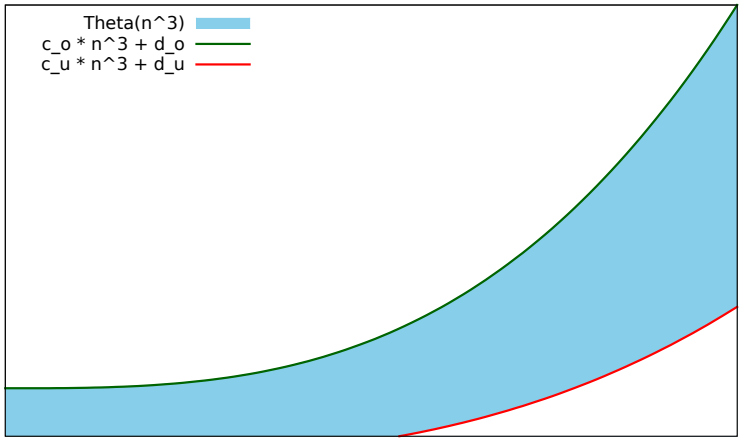


Groß-Theta

Definition 2.3

$$\Theta(f) := \Omega(f) \cap \mathcal{O}(f)$$

anschaulich: $\Theta(f)$ enthält alle Funktionen, die sich von positiven Vielfachen einer(!) Funktion f nach unten und oben begrenzen lassen.



Wichtige Komplexitätsklassen

meist wird obere Schranke der Kosten gesucht; untere selten relevant

Bezeichnungen:

$\mathcal{O}(0)$	„kostenlos“
$\mathcal{O}(1)$	konstant
$\mathcal{O}(\log n)$	logarithmisch
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \cdot \log n)$	„$n \cdot \log n$“
$\mathcal{O}(n^2)$	quadratisch
$\mathcal{O}(n^3)$	kubisch
$\mathcal{O}(n^k)$	polynomiell
$\mathcal{O}(a^n)$	exponentiell ($\rightarrow$ in Praxis unbrauchbar)

Logarithmus: Rechenregeln

1. $\log_a (x_1 \cdot x_2) = \log_a x_1 + \log_a x_2$
2. $\log_a x^y = y \cdot \log_a x$
3. $\frac{\log_a x}{\log_b x} = \log_a b$
4. $\frac{\log_a x}{\log_a y} = \log_y x$

Herleitungen:

1. $a^{\log_a (x_1 \cdot x_2)} = x_1 \cdot x_2 = a^{\log_a x_1} \cdot a^{\log_a x_2} = a^{\log_a x_1 + \log_a x_2}$
2. $a^{\log_a x^y} = x^y = \left(a^{\log_a x}\right)^y = a^{y \cdot \log_a x}$
3. $\log_a x = \log_a b^{\log_b x} = \log_b x \cdot \log_a b$
4. folgt aus 3. für $b = y$

Komplexitätsklassen: Rechenregeln

Summe vereint Komplexitätsklassen:

$$\mathcal{O}(f+g) = \mathcal{O}(f) \text{ falls } g \in \mathcal{O}(f)$$

Produkt überträgt sich auf Komplexitätsklassen:

$$\tilde{f} \in \mathcal{O}(f) \wedge \tilde{g} \in \mathcal{O}(g) \Rightarrow (\tilde{f} \cdot \tilde{g}) \in \mathcal{O}(f \cdot g)$$

Komplexitätsklassen: Konsequenzen I

konstante Faktoren vernachlässigbar:

$$\mathcal{O}(c \cdot f) = \mathcal{O}(f)$$

(da $c \in \mathcal{O}(1)$)

„kleinere“ Terme in Polynom vernachlässigbar:

$$\mathcal{O}(a_k \cdot n^k + \cdots + a_0 \cdot n^0) = \mathcal{O}(n^k)$$

(da Summe und alle anderen Komplexitätsklassen in $\mathcal{O}(n^k)$ enthalten)

Komplexitätsklassen: Konsequenzen II

konstanter Faktor c/Summand d in Argument zu Potenz/Exponent/Logarithmus vernachlässigbar:

$$\mathcal{O}((c \cdot n + d)^k) = \mathcal{O}(n^k)$$
$$\mathcal{O}(a^{c \cdot n + d}) = \mathcal{O}(a^n)$$
$$\mathcal{O}(\log_a (c \cdot n + d)) = \mathcal{O}(\log_a n)$$

Wahl der Basis von Exp./Log. vernachlässigbar:

$$\mathcal{O}(a^n) = \mathcal{O}(b^n) \quad \text{und} \quad \mathcal{O}(\log n) := \mathcal{O}(\log_a n) = \mathcal{O}(\log_b n)$$

(da $\log_a n = \log_a b \cdot \log_b n$ und $\log_a b$ eine Konstante)

Komplexitätsklassen: Hierarchie

$\forall 0 < x < y,\ 1 < z,\ 1 < a$:

$$\mathcal{O}(0) \subset \mathcal{O}(1) \subset \mathcal{O}(\log n) \subset \overbrace{\mathcal{O}(n^x) \subset \mathcal{O}(n^y)}^{\mathcal{O}(n) \subset \mathcal{O}(n \cdot \log n) \subset \mathcal{O}(n^z)} \subset \mathcal{O}(a^n)$$

wegen:

- $n^x \cdot n^{y-x} = n^y$ und $n^{y-x} > 1$ $(\Leftarrow y > x)$
- Wachstum von Exp./Log. vs. Potenzen

Komplexitätsklasse: Bestimmung per Regeln

Beispiel 2.1

Funktion: $g(n) = 3 \cdot n \cdot \log_2(n+1) - 2 \cdot n + 6$

Anwendung von Streichregeln:

$$
\begin{aligned}
g(n) &\in \mathcal{O}(3 \cdot n \cdot \log_2(n+1) - 2 \cdot n + 6) && \text{(Einsetzen)} \\
&= \mathcal{O}(3 \cdot n \cdot \log_2 n - 2 \cdot n) && \text{(konst. Summanden streichen)} \\
&= \mathcal{O}(n \cdot \log_2 n - n) && \text{(konst. Faktoren streichen)} \\
&= \mathcal{O}(n \cdot \log n - n) && \text{(Basis streichen)} \\
&= \mathcal{O}(n \cdot \log n) && \text{(schwächere Terme streichen)}
\end{aligned}
$$

Komplexitätsklasse: Bestimmung per Defintion

Beispiel 2.2

Funktion: $g(n) = 3 \cdot n \cdot \log_2(n+1) - 2 \cdot n + 6$

Anwendung der Definition (Vermutung: $g(n) \in \mathcal{O}(n \cdot \log n)$):
suche $c > 0, d \in \mathbb{R}: \; g(n) \leq c \cdot (n \cdot \log_2 n) + d$ (für $n > 0$)

$$
\begin{aligned}
g(n) &= 3 \cdot n \cdot \log_2(n+1) - 2 \cdot n + 6 \\
&< 3 \cdot n \cdot \log_2(n+1) + 6 \\
&\leq 3 \cdot n \cdot \log_2(2n) + 6 \\
&= 3 \cdot n \cdot (\log_2 n + 1) + 6 \\
&= 3 \cdot n \cdot \log_2 n + 3 \cdot n + 6 \\
&\leq 3 \cdot n \cdot \log_2 n + 3 \cdot n \cdot \log_2 n + 3 + 6 \\
&= 6 \cdot n \cdot \log_2 n + 9
\end{aligned}
$$

Komplexität der Suchverfahren

lineare Suche hat Komplexität $\mathcal{O}(n)$

$\rightarrow$ kein Verfahren geringerer Komplexität für unsortierte Sequenzen, da ggf. jedes der n Elemente zu betrachten ist

binäre Suche hat Komplexität $\mathcal{O}(\log n)$