Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2016

Dr. Tobias Lasser

Computer Aided Medical Procedures
Technische Universität München

Notizen

Programm heute

Notizen

Wozu Effizienz?

"Wenn das Programm zu langsam läuft, kaufen wir einfach einen schnelleren Rechner!"

- der schnellere Rechner ist eventuell trotzdem nicht schnell genug
- auch wenn es schnell genug läuft, Energie sparen (z.B. wegen Akkulaufzeit!)

Notizen

Komplexität von Algorithmen

Interessante Fragen bei Algorithmus A:

- wie viel Speicherplatz benötigt Algorithmus?
- wie viel Rechenzeit benötigt Algorithmus?

→ Komplexitätsanalyse

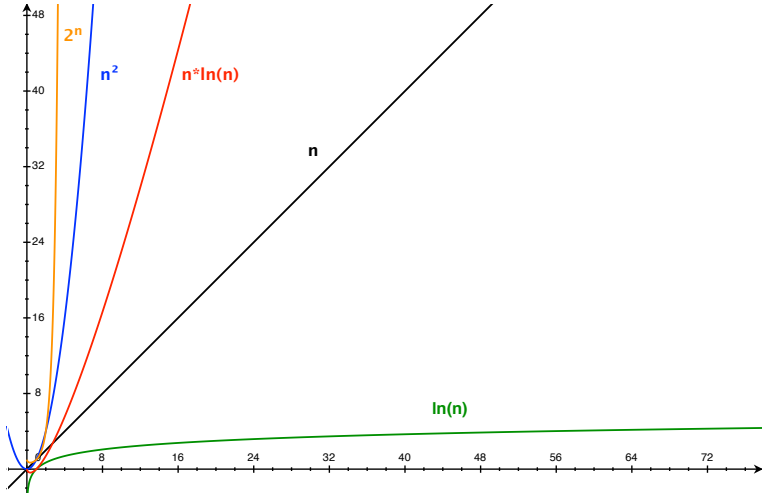
Wesentlicher Faktor bei Komplexität: Größe der Eingabe n

- benötigter Speicher abhängig von n
- benötigte Rechenzeit abhängig von n

→ Funktion T zur Beschreibung des Wachstumsverhaltens von A

Notizen

Typische Wachstumsraten



Notizen

Wachstumsraten illustriert

Annahme: Rechner mit 1GHz Taktfrequenz

- eine Operation benötigt 1ns (nano-Sekunde)
- n bezeichne Anzahl Operationen

n	$T(n) = \ln(n)$	$T(n) = n$	$T(n) = n\ln(n)$	$T(n) = n^2$	$T(n) = 2^n$	$T(n) = n!$
10	$0.003\mu s$	$0.01\mu s$	$0.033\mu s$	$0.1\mu s$	$1\mu s$	3.63ms
20	$0.004\mu s$	$0.02\mu s$	$0.086\mu s$	$0.4\mu s$	1ms	77.1 years
30	$0.005\mu s$	$0.03\mu s$	$0.147\mu s$	$0.9\mu s$	1s	$8.4 \cdot 10^{15}$years
40	$0.005\mu s$	$0.04\mu s$	$0.213\mu s$	$1.6\mu s$	18.3min	
50	$0.006\mu s$	$0.05\mu s$	$0.282\mu s$	$2.5\mu s$	13 days	
100	$0.007\mu s$	$0.1\mu s$	$0.644\mu s$	$10\mu s$	$4 \cdot 10^{13}$years	
1000	$0.010\mu s$	$1.0\mu s$	$9.966\mu s$	1ms		
10000	$0.013\mu s$	$10\mu s$	$130\mu s$	100ms		
100000	$0.017\mu s$	0.1ms	1.67ms	10s		
$1 \cdot 10^6$	$0.020\mu s$	1ms	19.93ms	16.7min		
$1 \cdot 10^7$	$0.023\mu s$	0.01s	0.23s	1.16 days		
$1 \cdot 10^8$	$0.027\mu s$	0.1s	2.66s	115.7 days		
$1 \cdot 10^9$	$0.030\mu s$	1s	29.9s	31.7 years		

Tabelle adaptiert von "The Algorithm Design Manual", S. Skiena, Springer

Notizen

Wiederholung: Insertion Sort

Input: Feld A[0..n-1] von n natürlichen Zahlen
Output: Feld A aufsteigend sortiert
InsertionSort(A):

```
1   for j=1 to länge(A)-1 {
2     key = A[j];
3     // füge A[j] in sortierte Liste A[0..j-1] ein
4     i = j-1;
5     while (i >= 0 && A[i] > key) {
6       A[i+1] = A[i];
7       i = i-1;
8     }
9     A[i+1] = key;
10  }
```

Notizen

Komplexität von Insertion Sort

Speicher:

- Feld A (Länge n), eine extra Variable (key)

Zeit:

- Annahme: jeder elementare Verarbeitungsschritt in Zeile i benötigt konstante Zeit $c_i \in \mathbb{R}$
- Beschreibe Laufzeit mittels

$$\text{Laufzeitfunktion } T : \mathbb{N} \to \mathbb{R}, \quad T : n \mapsto T(n)$$

- $T(n)$ beschreibt Wachstumsverhalten in Abhängigkeit von Eingabegröße n

Notizen

Laufzeit Insertion Sort I

Kostenübersicht:

Zeile	Kosten	wie oft?
1	c_1	
2	c_2	
3	0	Kommentar
4	c_4	
5	c_5	
6	c_6	
7	c_7	
8	0	Blockabschluß
9	c_9	
10	0	Blockabschluß

InsertionSort(A):

```
1 for j=1 to länge(A)-1 {
2     key = A[j];
3     // füge A[j] in sortierte
      // Liste A[0..j-1] ein
4     i = j-1;
5     while (i >= 0 && A[i] > key) {
6       A[i+1] = A[i];
7       i = i-1;
8     }
9     A[i+1] = key;
10 }
```

t_j bezeichnet Anzahl der Abfragen der while-Bedingung in Zeile 5 für Durchlauf j

Notizen

Laufzeit Insertion Sort II

Berechnung der Laufzeit $T(n)$:

- $T(n)$ ist Summe aller Laufzeiten der ausgeführten elementaren Verarbeitungsschritte
- Beispiel: Schritt 1 (Zeile 1) kostet c_1 und wird n mal ausgeführt

$$\begin{aligned} T(n) = {} & c_1 n + c_2(n-1) + c_4(n-1) \\ & + c_5 \sum_{j=1}^{n-1} t_j + c_6 \sum_{j=1}^{n-1} (t_j - 1) \\ & + c_7 \sum_{j=1}^{n-1} (t_j - 1) + c_9(n-1) \end{aligned}$$

- Beobachtung: $T(n)$ hängt stark von tatsächlichen Eingabe ab!

Notizen

Laufzeit Insertion Sort III

Laufzeit $T(n)$ im besten Fall (best case)?

- best case: Feld A ist bereits sortiert bevor Algorithmus aufgerufen wird
- in while-Schleife (Zeile 5) wird Bedingung nur einmal abgefragt
  - es ist nämlich A[i] <= key
  - also $t_j = 1$ für $j = 1, \ldots, n-1$
- Also:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) \\ &\quad + c_5(n-1) + c_6 \cdot 0 \\ &\quad + c_7 \cdot 0 + c_9(n-1) \\ &= \underbrace{(c_1 + c_2 + c_4 + c_5 + c_9)}_{=:a} n - \underbrace{(c_2 + c_4 + c_5 + c_9)}_{=:b} \\ &= an - b \qquad\qquad (a, b \text{ Konstanten}) \end{aligned}$$

Notizen

Laufzeit Insertion Sort IV

Laufzeit $T(n)$ im schlechtesten Fall (worst case)?

- worst case: Feld A ist bereits absteigend sortiert bevor Algorithmus aufgerufen wird
- in while-Schleife (Zeile 5) muß mit allen Elementen A[0..j-1] verglichen werden
  - es ist nämlich A[i] > key
  - also $t_j = j + 1$ für $j = 1, \ldots, n-1$
- Also:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) \\ &\quad + c_5\left(\tfrac{n(n+1)}{2} - 1\right) + c_6\left(\tfrac{n(n-1)}{2}\right) \\ &\quad + c_7\left(\tfrac{n(n-1)}{2}\right) + c_9(n-1) \end{aligned}$$

da Gauss'sche Summenformel:

$\sum_{j=1}^{n-1}(j+1) = \frac{n(n+1)}{2} - 1$ und $\sum_{j=1}^{n-1} j = \frac{n(n-1)}{2}$

Notizen

Laufzeit Insertion Sort V

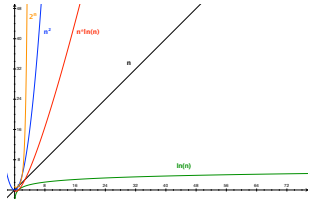
Fortsetzung worst case $T(n)$:

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\tfrac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\tfrac{n(n-1)}{2}\right) + c_7\left(\tfrac{n(n-1)}{2}\right) + c_9(n-1) \\
&= \underbrace{\left(\tfrac{c_5}{2} + \tfrac{c_6}{2} + \tfrac{c_7}{2}\right)}_{=:a} n^2 + \underbrace{\left(c_1 + c_2 + c_4 + \tfrac{c_5}{2} - \tfrac{c_6}{2} - \tfrac{c_7}{2} + c_9\right)}_{=:b} n \\
&\quad - \underbrace{\left(c_2 + c_4 + c_5 + c_9\right)}_{=:c} \\
&= an^2 + bn - c
\end{aligned}
$$

Notizen

Zusammenfassung Insertion Sort

- tatsächliche Kosten wurden abstrahiert in Konstanten c_i
  - der tatsächliche Wert der Konstanten ist abhängig von Compiler und Rechner
  - einfaches Modell hierfür: RAM-Modell (siehe 5.2)
- detaillierte Laufzeitanalyse ist aufwendig, schon im sehr einfachen Beispiel
- die wesentliche Information steckt im Grad des Wachstums bzw. im asymptotischen Verhalten der Laufzeit bei wachsendem n (siehe 5.3)
  - im Beispiel hier: linear im best case, quadratisch im worst case



Notizen

Programm heute

Notizen

RAM-Modell

- einfaches Rechnermodell für Laufzeitanalyse
- Ziel: Abschätzung der erforderlichen Ressourcen zur Ausführung eines Algorithmus
  - exakte Zeit der Ausführung ist nicht relevant
- Modell: Random Access Machine, kurz RAM

Notizen

Annahmen im RAM-Modell

- nur sequentielle Ausführungen
  - das heißt nur 1 Prozessor, keine Parallelität
- alle Daten liegen direkt zugreifbar im Speicher ($\rightarrow$ RAM)
- jeder Speicherzugriff dauert gleich lang
  - das ist in Wahrheit nicht der Fall! (Speicher-Hierarchie)
- alle elementaren Verarbeitungsschritte benötigen eine Zeiteinheit
  - Wertzuweisung
  - Arithmetische Operationen: +, -, *, /, %, ceil, floor
  - Vergleichsoperationen: $<$, $>$, $!=$
  - Kontrollflußoperationen: **if**, **else**

Notizen

Beispiel: Laufzeit mit RAM-Modell

- Laufzeit von Algorithmus ermittelt über Anzahl der elementaren Verarbeitungsschritte
- Beispiel:

Algorithmus	Anzahl Schritte
X = X+1;	

for i=1 **to** n {
  X = X+1;
}	

for i=1 **to** n {
  for j=1 **to** n {
  X = X+1;
  }
}	

Notizen

Beispiel 2: Laufzeit mit RAM-Modell

Komplexeres Beispiel:

Algorithmus A bestehend aus drei Teilen A1, A2, A3:

$$A = A1; A2; A3$$

mit

A1	A2	A3

X = X+1;
Y = Y+1;
Z = Z+1;
	for i=1 **to** n {
	X = X+Y;
	Y = Y*Z;
	}
		for i=1 **to** n {
		for j=1 **to** n {
		X = X+1;
		Y = Y+2;
		Z = Z+3;
		}
		}

Gesamt ist für A: $T(n) = 3 + 2n + 3n^2$

Notizen

Beispiel 2: Laufzeit mit RAM-Modell

$$T(n) = 3 + 2n + 3n^2$$

- wie ist das Wachstum von $T(n)$?
- Beispiel-Werte für n:

$n = 2$	$T(2) = 3 + 4 + 12$
$n = 10$	$T(10) = 3 + 20 + 300$
$n = 100$	$T(100) = 3 + 200 + 30000$
$n = 1000$	$T(1000) = 3 + 2000 + 3000000$

$\rightarrow$ quadratischer Term dominiert

Notizen

Programm heute

Notizen

Komplexität

Beispiele zeigen: Term mit höchstem Exponent dominiert
Wachstumsrate

Präzise Notation → Landau-Symbole

Notizen

Landau-Symbol Θ

Landau-Symbol Θ

Sei $g : \mathbb{R} \to \mathbb{R}$ eine Funktion. Das Landau-Symbol $\Theta(g)$ ist definiert als die Menge

$$\Theta(g) := \{ f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c_1, c_2 > 0, n_0 \in \mathbb{N} \text{ so dass für alle } n \geq n_0 \text{ gilt: } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$

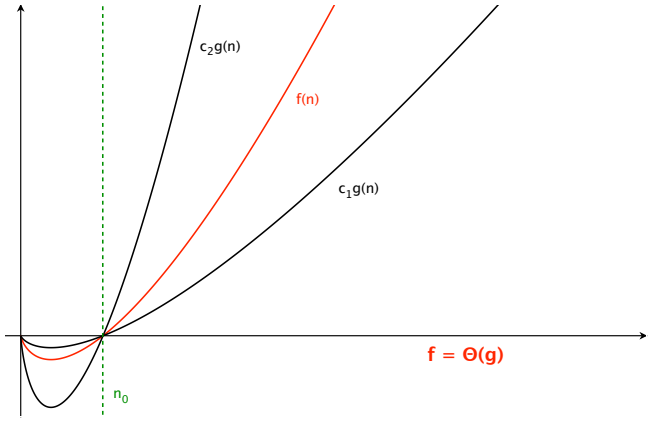
Für $f : \mathbb{R} \to \mathbb{R}$ mit $f \in \Theta(g)$ schreiben wir kurz: $f = \Theta(g)$.

- f ist also bis auf konstanten Faktor im wesentlichen "gleich" der Funktion g für $n \geq n_0$
- man sagt auch: g ist asymptotisch scharfe Schranke von f (von oben und unten)
- Kurznotation: $f = \Theta(g)$ oder $f(n) = \Theta(g(n))$

Notizen

Landau-Symbol Θ

$$\Theta(g) = \{ f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c_1, c_2 > 0, n_0 \in \mathbb{N} \text{ so dass für alle } n \geq n_0 \text{ gilt: } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \}$$



Notizen

Beispiel: Laufzeit Insertion Sort I

- Laufzeit für worst case von Insertion Sort:

$$T(n) = an^2 + bn - c,$$

wobei $a = \frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}$, $b = c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_9$ und $c = c_2 + c_4 + c_5 + c_9$

- weitere Annahme: alle Konstanten $c_i = 1$ für $i = 1, \ldots, 9$

$$\longrightarrow \quad T(n) = \tfrac{3}{2}n^2 + \tfrac{7}{2}n - 4$$

Notizen

Beispiel: Laufzeit Insertion Sort II

$$T(n) = \tfrac{3}{2}n^2 + \tfrac{7}{2}n - 4$$

Behauptung: $T(n) = \Theta(n^2)$

- Zu zeigen: finde $c_1, c_2 > 0$ und $n_0 \in \mathbb{N}$ mit

$$0 \leq c_1 n^2 \leq T(n) \leq c_2 n^2 \qquad \text{für } n \geq n_0$$

- linke Seite:

$$0 \leq c_1 n^2 \leq \tfrac{3}{2}n^2 + \tfrac{7}{2}n - 4 \quad \left| \tfrac{1}{n^2} \right.$$

$$0 \leq c_1 \leq \tfrac{3}{2} + \tfrac{7}{2n} - \tfrac{4}{n^2}$$

Wähle z.B. $n_0 = 1$, dann $c_1 = 1$.

- rechte Seite: $n_0 = 1$ und $\frac{3}{2} + \frac{7}{2n} - \frac{4}{n^2} \leq 5 = c_2$ ✓

Notizen

Beispiel: Polynom 2. Grades

Allgemeiner:

$$f(n) = an^2 + bn + c$$

für $a, b, c \in \mathbb{R}$ und $a > 0$.

Behauptung: $f(n) = \Theta(n^2)$

- wähle $c_1 = \frac{a}{4}$, $c_2 = \frac{7a}{4}$ sowie $n_0 = 2 \cdot \max\left(\frac{|b|}{a}, \sqrt{\frac{|c|}{a}}\right)$
- es folgt: $0 \leq c_1 n^2 \leq an^2 + bn + c \leq c_2 n^2$ für $n \geq n_0$ ✓

Notizen

Beispiel: Polynome vom Grad d

Noch allgemeiner:

$$p(n) = \sum_{i=0}^{d} a_i n^i$$

mit $a_i \in \mathbb{R}$ und $a_d > 0$ (Polynom vom Grad d).

Dann gilt:

$$p(n) = \Theta(n^d)$$

Notizen

Landau-Symbol O

Landau-Symbol O

Sei $g : \mathbb{R} \to \mathbb{R}$ eine Funktion. Das Landau-Symbol $O(g)$ ist definiert als die Menge

$$O(g) := \{f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass für alle } n \geq n_0 \text{ gilt: } 0 \leq f(n) \leq cg(n)\}$$

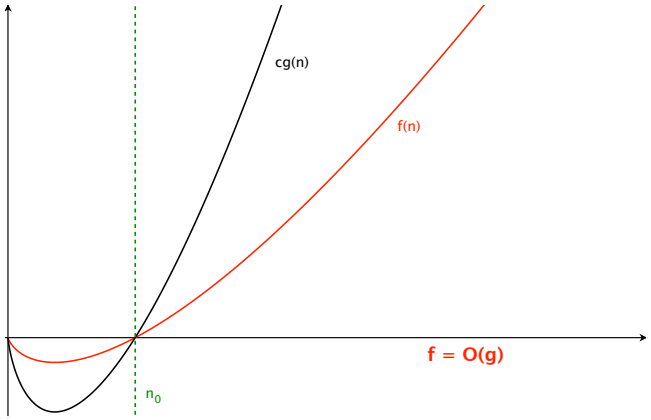
Für $f : \mathbb{R} \to \mathbb{R}$ mit $f \in O(g)$ schreiben wir kurz: $f = O(g)$.

- man sagt auch: g ist asymptotisch obere Schranke von f
- auch genannt: "gross-O-Notation"
- aus $f = \Theta(g)$ folgt automatisch $f = O(g)$

Notizen

Landau-Symbol O

$$O(g) = \{f : \mathbb{R} \to \mathbb{R} : \text{ es existieren } c > 0 \text{ und } n_0 \in \mathbb{N} \text{ so dass für alle } n \geq n_0 \text{ gilt: } 0 \leq f(n) \leq cg(n)\}$$



Notizen

Beispiel: lineare Funktion

$$f(n) = 60n + 12$$

- **Behauptung:** $f(n) = O(n)$
  - tatsächlich:

$$0 \leq 60n + 12 \leq cn \quad \Big| \frac{1}{n}$$
$$0 \leq 60 + \frac{12}{n} \leq c$$

  also z.B. $n_0 = 12$ und $c = 61$ ✓

- **Behauptung:** $f(n) = O(n^2)$
  - tatsächlich:

$$0 \leq 60n + 12 \leq cn^2 \iff 0 \leq \frac{60}{n} + \frac{12}{n^2} \leq c$$

  also z.B. $n_0 = 1$ und $c = 73$ ✓

Notizen

Landau-Symbole Θ und O

- Landau-Symbole beschreiben Wachstumsverhalten von Funktionen
- $f = \Theta(g)$ bedeutet: g ist asymptotisch scharfe Schranke von f
- $f = O(g)$ bedeutet: g ist asymptotisch obere Schranke von f
- Landau-Symbol Θ ist "exakt"
- Landau-Symbol O kann zu "grosszügig" sein (siehe vorige Folie)
- Landau-Symbol O beschreibt Laufzeit im schlechtesten Fall (worst case)

Notizen

Effizienz von Algorithmen I

O-Notation erlaubt Kategorisierung der Effizienz von Algorithmen

- $O(1)$: konstante Laufzeit
  - unabhängig von Problemgröße
  - *Beispiel:* Löschen von erstem Element in verketteter Liste
- $O(\log n)$: logarithmische Laufzeit
  - Laufzeit wächst langsamer als Problemgröße
  - typisch für Divide & Conquer Algorithmen (s. Kapitel 6)
  - *Beispiel:* Suchen in sortierter Liste (Binäre Suche, s. Kapitel 8)
- $O(n)$: lineare Laufzeit
  - Laufzeit wächst vergleichbar zur Problemgröße
  - jedes Eingabe-Element erfordert $O(1)$ Arbeit
  - *Beispiele:* Suchen in unsortierter Liste, Löschen von Element in sequentieller Liste

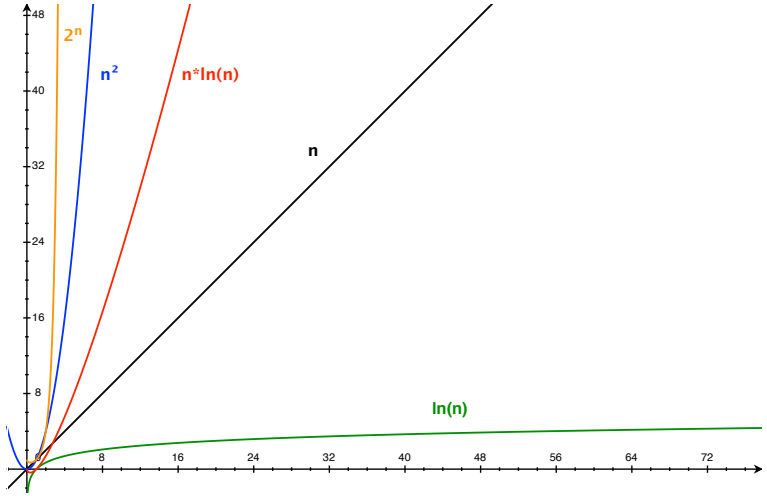
Notizen

Effizienz von Algorithmen II

- $O(n \log n)$: "loglinear" Laufzeit
  - Laufzeit wächst schneller als Problemgröße
  - typisch für Divide & Conquer Algorithmen (s. Kapitel 6)
  - *Beispiele:* Quicksort (s. Kapitel 6), FFT (s. Kapitel 10)
- $O(n^2)$: quadratische Laufzeit
  - typisch für Algorithmen, die Element paarweise kombinieren
  - *Beispiele:* Insertion Sort, Matrix-Vektor Multiplikation (s. Kapitel 10)
- $O(n^3)$: kubische Laufzeit
  - *Beispiel:* Matrix-Matrix Multiplikation
- $O(2^n)$: exponentielle Laufzeit
  - auch als "unlösbar" bezeichnet (intractable)
  - *Beispiel:* Traveling Salesman (kürzeste Route, so dass alle Städte exakt einmal besucht)

Notizen

Wiederholung: Wachstumsraten



Notizen

Wiederholung: Wachstumsraten illustriert

Annahme: Rechner mit 1GHz Taktfrequenz

- eine Operation benötigt 1ns (nano-Sekunde)
- n bezeichne Anzahl Operationen

n	$T(n) = \ln(n)$	$T(n) = n$	$T(n) = n\ln(n)$	$T(n) = n^2$	$T(n) = 2^n$	$T(n) = n!$
10	$0.003\mu s$	$0.01\mu s$	$0.033\mu s$	$0.1\mu s$	$1\mu s$	3.63ms
20	$0.004\mu s$	$0.02\mu s$	$0.086\mu s$	$0.4\mu s$	1ms	77.1 years
30	$0.005\mu s$	$0.03\mu s$	$0.147\mu s$	$0.9\mu s$	1s	$8.4 \cdot 10^{15}$years
40	$0.005\mu s$	$0.04\mu s$	$0.213\mu s$	$1.6\mu s$	18.3min	
50	$0.006\mu s$	$0.05\mu s$	$0.282\mu s$	$2.5\mu s$	13 days	
100	$0.007\mu s$	$0.1\mu s$	$0.644\mu s$	$10\mu s$	$4 \cdot 10^{13}$years	
1000	$0.010\mu s$	$1.0\mu s$	$9.966\mu s$	1ms		
10000	$0.013\mu s$	$10\mu s$	$130\mu s$	100ms		
100000	$0.017\mu s$	0.1ms	1.67ms	10s		
$1 \cdot 10^6$	$0.020\mu s$	1ms	19.93ms	16.7min		
$1 \cdot 10^7$	$0.023\mu s$	0.01s	0.23s	1.16 days		
$1 \cdot 10^8$	$0.027\mu s$	0.1s	2.66s	115.7 days		
$1 \cdot 10^9$	$0.030\mu s$	1s	29.9s	31.7 years		

Tabelle adaptiert von "The Algorithm Design Manual", S. Skiena, Springer

Notizen

Konstanten in Landau-Symbolen

- Landau-Symbole Θ, O enthalten Konstanten c_1, c_2 bzw. c
  $\rightarrow$ für asymptotisches Verhalten ($n \rightarrow \infty$) vernachlässigbar

- Aber: ist n klein, spielen die Konstanten sehr wohl eine Rolle!
  - es kann angebracht sein einen eigentlich komplexeren Algorithmus einzusetzen, da bei kleinem n die Konstanten mehr zählen
  - Beispiel: Insertion Sort

Notizen

Rechenregel für O-Notation I

Addition in O-Notation

Seien A_1, A_2 zwei Algorithmen mit Laufzeiten

$$T_1(n) = O(f(n)), \qquad T_2(n) = O(g(n))$$

für zwei Funktionen $f, g : \mathbb{R} \rightarrow \mathbb{R}$. Dann hat der Algorithmus $A = A_1; A_2$ (sequentielle Ausführung von A_1, A_2) die Komplexität

$$T(n) = T_1(n) + T_2(n) = O\big(\max(f(n), g(n))\big)$$

Beispiel: Sei $A_1 = O(\log n)$, $A_2 = O(n^2)$. Dann ist für $A = A_1; A_2$

$$A = O\big(\max(\log n, n^2)\big) = O(n^2)$$

Notizen

Rechenregel für O-Notation II

Zu zeigen:

$$T(n) = T_1(n) + T_2(n) = O(\max(f(n), g(n)))$$

- da $T_1(n) = O(f(n))$, gibt es $c_1 > 0$ und $n_1 \in \mathbb{N}$ mit

$$T_1(n) \leq c_1 f(n) \qquad \text{für } n \geq n_1$$

- da $T_2(n) = O(g(n))$, gibt es $c_2 > 0$ und $n_2 \in \mathbb{N}$ mit

$$T_2(n) \leq c_2 g(n) \qquad \text{für } n \geq n_2$$

- Setze $n_0 := \max(n_1, n_2)$, dann ist für $n \geq n_0$

$$\begin{aligned} T_1(n) + T_2(n) &\leq c_1 f(n) + c_2 g(n) \\ &\leq (c_1 + c_2) \max(f(n), g(n)) \end{aligned}$$

- das bedeutet $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$ ✓

Notizen

Komplexität der elementaren Bausteine

- Elementarer Verarbeitungsschritt:
  - $O(1)$
- Sequenz:
  - Addition in O-Notation
- Bedingter Verarbeitungsschritt:
  - Maximum von Komplexität von if und else Block, sowie
  - $O(1)$ für Auswertung der Bedingung
- Wiederholung:
  - Anzahl Wiederholungen multipliziert mit Komplexität Schleifenkörper, sowie
  - Anzahl Wiederholungen multipliziert mit $O(1)$ für Auswertung der Schleifenbedingung

Notizen

Komplexität eines Algorithmus

Algorithmus besteht aus elementaren Bausteinen

→ Komplexität eines ganzen Algorithmus ergibt sich aus

- Komplexität der elementaren Bausteine
- und der Addition in O-Notation

Notizen

Komplexität Operationen elementarer Datenstrukturen

- sequentielle Liste:
  - elementAt $O(1)$, insert $O(n)$, erase $O(n)$
- verkettete Liste:
  - elementAt $O(n)$, insert $O(n)$, erase $O(n)$
- Stack als sequentielle Liste:
  - push, pop, top alle $O(1)$
- Stack als verkettete Liste:
  - push, pop, top alle $O(1)$
- Queue als verkettete Liste:
  - enqueue, dequeue beide $O(1)$

Notizen

Zusammenfassung

1. Einführung
2. Grundlagen von Algorithmen
3. Grundlagen von Datenstrukturen
4. Grundlagen der Korrektheit von Algorithmen
5. Grundlagen der Effizienz von Algorithmen
   - Motivation
   - Das RAM-Modell
   - Landau-Symbole

Notizen

Notizen