



Informatik I

Komplexität von Algorithmen


G. Zachmann
 Clausthal University, Germany
zach@in.tu-clausthal.de

Leistungsverhalten von Algorithmen

- **Speicherplatzkomplexität:** Wird primärer & sekundärer Speicherplatz effizient genutzt?
- **Laufzeitkomplexität:** Steht die Laufzeit im akzeptablen / vernünftigen / optimalen Verhältnis zur Aufgabe?
- Theorie: liefert **untere Schranke**, die für jeden Algorithmus gilt, der das Problem löst.
- Spezieller Algorithmus liefert **obere Schranke** für die Lösung des Problems.
- Erforschung von oberen und unteren Schranken: Effiziente Algorithmen und Komplexitätstheorie (Zweige der Theoretischen Informatik)


G. Zachmann Informatik I - WS 05/06 Komplexität 2



Laufzeit

- **Definition:**
 Die **Laufzeit $T(x)$** eines Algorithmus A bei Eingabe x ist definiert als die **Anzahl von Basisoperationen**, die Algorithmus A zur Berechnung der Lösung bei Eingabe x benötigt.
- Laufzeit = Funktion der Größe der Eingabe
- Definition für **Eingabegröße** ist abhängig vom Problem

G. Zachmann Informatik I - WS 05/06 Komplexität 3



Laufzeitanalyse

- Sei P ein gegebenes Programm und x Eingabe für P , $|x|$ Länge von x , und $T_P(x)$ die Laufzeit von P auf x .
- Beschreibe Aufwand eines Algorithmus als Funktion **der Größe des Inputs** (kann verschieden gemessen werden):
 $T_P(n)$ = Laufzeit des Programms P für Eingaben der Länge n
- **Der beste Fall (best case):** Oft leicht zu bestimmen, kommt in der Praxis jedoch selten vor:

$$T_P(n) = \inf\{T_P(x) \mid |x| = n, x \text{ Eingabe für } P\}$$
- **Der schlechteste Fall (worse case):** Liefert garantierte Schranken, meist **relativ** leicht zu bestimmen. Oft zu pessimistisch:

$$T_P(n) = \sup\{T_P(x) \mid |x| = n, x \text{ Eingabe für } P\}$$

G. Zachmann Informatik I - WS 05/06 Komplexität 4

Kostenmaße

- **Einheitskostenmaß:** Annahme, jedes Datenelement belegt unabhängig von seiner Größe denselben Speicherplatz (in Einheitsgröße).
 - Damit: Größe der Eingabe bestimmt durch Anzahl der Datenelemente
 - Beispiel: Sortierproblem
- **Logarithmisches Kostenmaß (Bit-Komplexität):** Annahme, jedes Datenelement belegt einen von seiner Größe (logarithmisch) abhängigen Platz
 - Größe der Eingabe bestimmt durch die Summe der Größen der Elemente
 - Erinnerung: für $n > 0$ ist die # Bits zur Darstellung von $n = \lceil \log_2(n+1) \rceil$
 - Beispiel: Zerlegung einer gegebenen großen Zahl in Primfaktoren
- **Ab jetzt immer Einheitskostenmaß**

G. Zachmann Informatik 1 - WS 05/06 Komplexität 5

Beispiel Minimum-Suche

- Eingabe : Folge von n Zahlen (a_1, a_2, \dots, a_n) .
- Ausgabe : Index i , so daß $a_i \leq a_j$ für alle Indizes $1 \leq j \leq n$.
- Beispiel:
 - Eingabe: 31, 41, 59, 26, 51, 48
 - Ausgabe: 4

```
def min(A):
    min = 1
    for j in range(2, len(A)):
        if A[j] < A[min]:
            min = j
```

G. Zachmann Informatik 1 - WS 05/06 Komplexität 6

```
def min(A):
    min = 0
    for j in range(1, len(A)):
        if A[j] < A[min]:
            min = j
```

	Kosten	Anzahl
<code>min = 0</code>	c_1	1
<code>for j in range(1, len(A)):</code>	c_2	$n - 1$
<code>if A[j] < A[min]:</code>	c_3	$n - 1$
<code>min = j</code>	c_4	$n - 1$

- Zusammen: Zeit

$$T(n) = c_1 + (n - 1) \cdot (c_2 + c_3 + c_4) \leq \bar{c} \cdot n$$
- Eingabegröße = Größe des Arrays

G. Zachmann Informatik 1 - WS 05/06 Komplexität 7

Weiteres Beispiel für Aufwandsberechnung

- Wir betrachten folgende Funktion $f1$, die $1! \cdot 2! \cdot \dots \cdot (n-2)! \cdot (n-1)!$ berechnet

```
def f1(n):
    r = 1
    while n > 0:
        i = 1
        while i < n:
            r *= i
            i += 1
        n -= 1
    return r
```

n	Z(n)	V(n)	M(n)	I(n)
1	2	2	0	1
2	3	5	1	1
3	5	3	1	3
4	8	6	3	6
5	12	10	6	10
6	17	15	10	15
7	23	21	15	21
8	30	28	21	28
9	38	36	28	36
10	47	45	36	45

- Exakte Bestimmung des Aufwandes:
 - M = Anzahl Mult, I = Anzahl Inkr.,
 - V = Anzahl Vergleiche, Z = Anzahl Zuweisungen

G. Zachmann Informatik 1 - WS 05/06 Komplexität 8

■ Anzahl Mult $M(n)$

$$M(n) = (n-1) + M(n-1) = (n-1) + (n-2) + M(n-2)$$

$$= \sum_{k=1}^{n-1} k = \frac{(n-1)(n-2)}{2}$$

■ Anzahl der Inkrementierungen: $I(n) = n + M(n+1)$,
 woraus folgt:

$$I(n) = \frac{n(n+1)}{2}$$

■ Die Anzahl der Vergleiche

$$V(n) = I(n+1) = \frac{(n+1)(n+2)}{2}$$

■ Die Anzahl benötigter Zuweisungen $Z(n)$ ist gleich

$$Z(n) = 1 + n + I(n) = 1 + \frac{n(n+3)}{2}$$

```

r = 1
while n > 0 :
    i = 1
    while i < n:
        r *= i
        i += 1
    n -= 1
return r
    
```

G. Zachmann Informatik 1 - WS 05/06 Komplexität 9

Rechenmodell / Algorithmisches Modell

- Für eine präzise mathematische Laufzeitanalyse benötigen wir ein Rechenmodell, das definiert
 - Welche Operationen zulässig sind.
 - Welche Datentypen es gibt.
 - Wie Daten gespeichert werden.
 - Wie viel Zeit Operationen auf bestimmten Daten benötigen.
- Formal ist ein solches Rechenmodell gegeben durch die Random Access Maschine (RAM).
 - RAMs sind Idealisierung von 1-Prozessorrechner mit einfachem aber unbegrenzt großem Speicher.

G. Zachmann Informatik 1 - WS 05/06 Komplexität 10

Basisoperationen und deren Kosten

- Definition:** Als **Basisoperationen** bezeichnen wir
 - Arithmetische Operationen** – Addition, Multiplikation, Division, Ab-, Aufrunden, auf Zahlen fester Längen (z.B. 64 Bit = Double);
 - Datenverwaltung** – Laden, Speichern, Kopieren von Datensätzen fester Größe;
 - Kontrolloperationen** – Verzweigungen, Sprünge, Funktionsaufruf.
- Kosten:** Zur Vereinfachung nehmen wir an, daß jede dieser Operationen bei allen Operanden gleich viel Zeit benötigt (im Einheitskostenmaß)
 - Überwiegend unabhängig von der verwendeten Programmiersprache
 - Ablesbar aus Pseudocode oder Programmstück
 - Exakte Definition ist nicht bedeutend

G. Zachmann Informatik 1 - WS 05/06 Komplexität 11

Beispiele

- einen Ausdruck auswerten
- einer Variablen einen Wert zuweisen
- Indizierung in einem Array
- Aufrufen einer Methode / Funktion mit Parametern
- Verlassen einer Methode / Funktion

G. Zachmann Informatik 1 - WS 05/06 Komplexität 12

Beispiel für Wachstum von Funktionen

$f_1(n) = \frac{1}{3}n^2,$
 $f_2(n) = \frac{1}{4}n^2,$
 $f_3(n) = n,$
 $f_4(n) = \frac{1}{3}n^2 + 2,$
 $f_5(n) = 2^n.$

G. Zachmann Informatik 1 - WS 05/06 Komplexität 13

Funktionenklassen

- Ziel: Konstante Summanden und Faktoren dürfen bei der Aufwandsbestimmung vernachlässigt werden.
- Gründe:
 - Man ist an **asymptotischem Verhalten** für große Eingaben interessiert
 - Genaue Analyse kann technisch oft sehr aufwendig oder unmöglich sein
 - Lineare Beschleunigungen sind ohnehin immer möglich (schnellere Hardware)
- Idee:
 - Komplexitätsmessungen mit Hilfe von Funktionenklassen. Etwa $O(f)$ sind die Funktionen, die (höchstens) in der Größenordnung von f sind.
- **Groß-O-Notation:** Mit O -, Ω - und Θ -Notation sollen obere, untere bzw. genaue Schranken für das Wachstum von Funktionen beschrieben werden.

G. Zachmann Informatik 1 - WS 05/06 Komplexität 14

"Groß-O"

- Sei $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$
- Definition **Groß-O**: Die **Ordnung von f** (*the order of f*) ist die Menge

$$O(f(n)) = \{t : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \mid \exists c \in \mathbb{R}_{>0} \exists n_0 \in \mathbb{N} \forall n \geq n_0 : t(n) \leq c \cdot f(n)\}$$
- Definition **Groß-Omega**: die Menge Ω ist wie folgt definiert:

$$\Omega(f(n)) = \{t : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \mid \exists c \in \mathbb{R}_{>0} \exists n_0 \in \mathbb{N} \forall n \geq n_0 : t(n) \geq c \cdot f(n)\}$$
- Definition **Groß-Theta**: Die **exakte Ordnung Θ** von $f(n)$ ist definiert als:

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$$
- Terminologie: O , Ω , Θ , heißen manchmal auch **Landau'sche Symbole**

G. Zachmann Informatik 1 - WS 05/06 Komplexität 15

Veranschaulichung der O-Notation

- Die Funktion f gehört zur Menge $O(g)$, wenn es positive Konstante c , n_0 gibt, so daß $f(n)$ ab n_0 unterhalb $cg(n)$ liegt

G. Zachmann Informatik 1 - WS 05/06 Komplexität 16

Veranschaulichung der Ω -Notation

- Die Funktion f gehört zur Menge $\Omega(g)$, wenn es positive Konstante c , n_0 gibt, so daß $f(n)$ ab n_0 unterhalb $cg(n)$ liegt

G. Zachmann Informatik 1 - WS 05/06 Komplexität 17

Veranschaulichung der Θ -Notation

- Die Funktion f gehört zur Menge $\Theta(g)$, wenn es positive Konstante c_1 , c_2 , und n_0 gibt, so daß $f(n)$ ab n_0 zwischen $c_1g(n)$ und $c_2g(n)$ "eingepackt" werden kann

G. Zachmann Informatik 1 - WS 05/06 Komplexität 18

Bemerkungen zu den O-Notationen

- In manchen Quellen findet man leicht abweichende Definitionen, etwa $O(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \text{es gibt positive Konstanten } a \text{ und } b \text{ mit } f(n) \leq ag(n) + b \text{ für alle } n\}$
- Für die relevantesten Funktionen f (etwa die monoton steigenden f nicht kongruent 0) sind diese Definitionen äquivalent.
- Schreibweise (leider) oft: $f = O(g)$ statt $f \in O(g)$
- Minimalität:** Die angegebene Größenordnung muß **nicht** minimal gewählt sein
- Asymptotik:** Wie groß n_0 ist bleibt unklar (kann sehr groß sein)
- „Verborgene Konstanten“:** Die Konstanten c und n_0 haben für kleine n großen Einfluß.

G. Zachmann Informatik 1 - WS 05/06 Komplexität 19

Beispiel Min-Search

- Behauptung: unser Minimum-Search-Algo besitzt Laufzeit $\Theta(n)$.
- Erinnerung: $T(n) \approx c_1 \cdot n + c_2$

```
def min(A):
    min = 1
    for j in range(2, len(A)):
        if A[j] < A[min]:
            min = j
```

- Zum Beweis ist zu zeigen:
 - Es gibt ein c_2 und n_2 , so daß die Laufzeit von Min-Search bei allen Eingaben der Größe $n \geq n_2$ immer höchstens c_2n ist. (Groß-O)
 - Es gibt ein c_1 und n_1 , so daß für alle $n \geq n_1$ eine Eingabe der Größe n existiert, bei der Min-Search mindestens Laufzeit c_1n besitzt. (Omega)

G. Zachmann Informatik 1 - WS 05/06 Komplexität 20

Beispiele zu Funktionsklassen

- Ist $n^2 \in O(n^3)$?
 - Gesucht: $c \in \mathbb{R}^+$ und $n_0 \in \mathbb{N}$, so daß Bed. erfüllt, also $\forall n > n_0 : n^2 \leq cn^3$

$$\Leftrightarrow 1 \leq c \cdot n \Leftrightarrow n \geq \frac{1}{c}$$
 Wähle $c = 1, n_0 = 1$
- Ist $n^3 \in O(n^2)$?
 - Gesucht: $c \in \mathbb{R}^+, n_0 \in \mathbb{N}$, so daß $\forall n > n_0 : n^3 \leq cn^2$

$$\Leftrightarrow \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \forall n > n_0 : n \leq c$$
 Widerspruch!
- $f(n) = n \log n \notin O(n)$

G. Zachmann Informatik 1 - WS 05/06 Komplexität 21

Der Groß-O-Kalkül

- Zunächst ein paar einfache "Rechen"-Regeln:

$$f \in O(f)$$

$$O(O(f)) = O(f)$$

$$k \cdot O(f) = O(k \cdot f) = O(f) \text{ für konstantes } k$$

$$O(f) + k = O(f + k) = O(f) \text{ für konstantes } k$$

G. Zachmann Informatik 1 - WS 05/06 Komplexität 22

Additionsregel

- Lemma, Teil 1:** Für beliebige Funktionen f und g gilt:

$$f + g \in O(f + g) = O(\max\{f, g\})$$
- Zu beweisen: nur das rechte "="
- Zu beweisen: jede der beiden Mengen ist jeweils in der anderen Menge enthalten
- " \subseteq ": Sei $t(n) \in O(f(n) + g(n)) \Rightarrow$

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \forall n \geq n_0 : t(n) \leq c \cdot (f(n) + g(n))$$
 Abschätzung nach oben:

$$c \cdot (f(n) + g(n)) \leq c \cdot 2 \cdot \max\{f(n), g(n)\}$$
 Mit $\bar{c} = 2 \cdot c$ gilt $t(n) \leq \bar{c} \cdot \max\{f(n), g(n)\}$
 Also ist $t(n) \in O(\max\{f(n), g(n)\})$

G. Zachmann Informatik 1 - WS 05/06 Komplexität 23

" \supseteq ": Sei $t(n) \in O(\max\{f(n), g(n)\})$

$$\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \forall n \geq n_0 : t(n) \leq c \cdot \max\{f(n), g(n)\}$$

Abschätzung nach oben:

$$\max\{f(n), g(n)\} \leq f(n) + g(n)$$

Also ist Bedingung für $t \in O(f(n) + g(n))$ mit denselben c, n_0 erfüllt

G. Zachmann Informatik 1 - WS 05/06 Komplexität 24

■ **Lemma, Teil 2:** Für beliebige Funktionen f und g gilt:

$$O(f) + O(g) = O(f + g)$$

■ Additionsregel findet Anwendung bei der Berechnung der Komplexität, wenn Programmteile hintereinander ausgeführt werden.

G. Zachmann Informatik 1 - WS 05/06 Komplexität 25

Multiplikationsregel

■ **Lemma:** Für beliebige Funktionen f und g gilt:

$$f \cdot g \in O(f \cdot g) = O(f) \cdot O(g)$$

■ Multiplikationsregel findet Anwendung bei der Berechnung der Komplexität, wenn Programmteile ineinander geschachtelt werden (Schleifen)

G. Zachmann Informatik 1 - WS 05/06 Komplexität 26

Teilmengenbeziehungen

■ **Lemma:** Es gelten die folgenden Aussagen:

- $O(f) \subseteq O(g) \Leftrightarrow f \in O(g)$
- $O(f) = O(g) \Leftrightarrow f \in O(g)$ und $g \in O(f)$
- $O(f) \subset O(g) \Leftrightarrow f \in O(g)$ und $g \notin O(f)$

■ **Beweis von Teil 1:**

- " \Rightarrow ": trivial
- " \Leftarrow ": $f(n) \in O(g(n))$.
 Also $\exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} \forall n \geq n_0 : f(n) \leq c \cdot g(n)$
 Zu zeigen: jedes $t(n) \in O(f(n))$ ist auch in $O(g(n))$
 Sei also $t(n) \in O(f(n))$.
 Per Def gilt $\exists c' \in \mathbb{R}^+, n'_0 \in \mathbb{N} \forall n \geq n'_0 : t(n) \leq c' f(n)$
 Wähle $n_0 = \max\{n_0, n'_0\}$ und $\bar{c} = c \cdot c'$
 Damit gilt $t(n) \leq \bar{c} \cdot g(n)$
 Also $t(n) \in O(g(n))$

G. Zachmann Informatik 1 - WS 05/06 Komplexität 27

■ Teil 2 & 3 : analog

G. Zachmann Informatik 1 - WS 05/06 Komplexität 28

Transitivität von Groß-O

- **Lemma:** Falls $f(n) \in O(g(n))$ und $g(n) \in O(h(n))$, dann ist $f(n) \in O(h(n))$.
- **Beweis:**
 Sei $f(n) \in O(g(n))$ und $g(n) \in O(h(n))$
 $\Rightarrow \exists c', c'' \in \mathbb{R}^+, n'_0, n''_0 \in \mathbb{N} \forall n \geq \max\{n'_0, n''_0\} :$
 $f(n) \leq c' \cdot g(n) \leq c' \cdot c'' \cdot h(n)$
 \Rightarrow Behauptung

G. Zachmann Informatik 1 - WS 05/06 Komplexität 29

Einfache Beziehungen

- **Lemma:** Für alle $m \in \mathbb{N}$ gilt $O(n^m) \subseteq O(n^{m+1})$.
- **Beweis:** Übung
- **Satz:** Sei $p(n) := a_m n^m + a_{m-1} n^{m-1} + \dots + a_1 n + a_0$, wobei $a_i \in \mathbb{R}_{>0}$ für $0 \leq i \leq m$. Dann gilt $p(n) \in O(n^m)$.
- **Insbesondere:** Es seien p_1 und p_2 Polynome vom Grad d_1 bzw. d_2 , wobei die Koeffizienten vor n^{d_1} und n^{d_2} positiv sind. Dann gilt:
 - $p_1 \in \Theta(p_2) \Leftrightarrow d_1 = d_2$
 - $p_1 \in O(p_2) \Leftrightarrow d_1 \leq d_2$
 - $p_1 \in \Omega(p_2) \Leftrightarrow d_1 \geq d_2$

G. Zachmann Informatik 1 - WS 05/06 Komplexität 30

- Für alle k, k fest, gilt: $n^k \in O(2^n)$
- Für alle $k > 0$ und $\epsilon > 0$ gilt: $\log^k n \in O(n^\epsilon)$
- $2^{n/2} \in O(2^n)$
- Für beliebige positive Zahlen $a, b \neq 1$ gilt:
 $f(n) = \log_a n \in O(\log_b n)$
 - Insbesondere:
 $O(\log_b n) = O(\log_2 n)$
- **Beweis:** Übungsaufgabe. (Gleichzeitig ein Beleg, daß die Analysis-Vorlesung Anwendung hat.)

G. Zachmann Informatik 1 - WS 05/06 Komplexität 31

- **Bemerkung:** Groß-O definiert keine totale Ordnungsrelation auf der Menge aller Funktionen $\mathbb{R} \rightarrow \mathbb{R}$
- **Beweis:** Es gibt positive Funktionen f und g so, daß $f(n) \notin O(g(n))$ und auch $g(n) \notin O(f(n))$.
 Wähle zum Beispiel $f(n) = \sin(n) + 1$ und $g(n) = \cos(n) + 1$.
- **Definition "polynomielle Zeit" :**
 wir sagen, ein Algorithmus A mit Komplexität $f(n)$ braucht höchstens **polynomielle Rechenzeit (polynomial time)**, falls es ein Polynom $P(n)$ gibt, so daß $f(n) \in O(P(n))$. A braucht höchstens **exponentielle Rechenzeit (exponential time)**, falls es eine Konstante $a \in \mathbb{R}^+$ gibt, so daß $f(n) \in O(a^n)$.

G. Zachmann Informatik 1 - WS 05/06 Komplexität 32

Beweishilfe

Lemma: Es gilt:

- (1) $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c, c > 0 \Rightarrow O(f(n)) = O(g(n))$
- (2) $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow O(f(n)) \subset O(g(n))$

Inbesondere hat man dann in Fall (1): $f(n) \in \Theta(g(n))$

G. Zachmann Informatik 1 - WS 05/06 Komplexität 33

Wie überprüft man diesen Limes?

Satz 7.44. (3. Regel von de l'Hôpital: "x → ∞") Seien f und g auf dem Intervall $[a, \infty[$ differenzierbar und es gelte $\lim_{x \rightarrow \infty} f(x) = \lim_{x \rightarrow \infty} g(x) = 0$ (bzw. $= \infty$). Es existiere $\lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)} =: L$. Dann existiert auch $\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)}$ und ist gleich L . Kurz:

$$\lim_{x \rightarrow \infty} \frac{f(x)}{g(x)} = \lim_{x \rightarrow \infty} \frac{f'(x)}{g'(x)}$$

G. Zachmann Informatik 1 - WS 05/06 Komplexität 34

Hierarchie von Größenordnungen

Größenordnung	Name
$O(1)$	konstante Funktionen
$O(\log n)$	logarithmische Funktionen
$O(\log^2 n)$	quadratisch logarithmische Funktionen
$O(n)$	lineare Funktionen
$O(n \log n)$	$n \log n$ - wachsende Funktionen
$O(n^2)$	quadratische Funktionen
$O(n^3)$	kubische Funktionen
$O(n^k)$	polynomielle Funktionen (k konstant)

f heißt **polynomiell beschränkt**, wenn es ein Polynom p mit $f \in O(p)$ gibt.

f wächst **exponentiell**, wenn es ein $k > 0$ gibt mit $f \in \Theta(k^n)$.

G. Zachmann Informatik 1 - WS 05/06 Komplexität 35

Skalierbarkeiten

- Annahme: 1 Rechenschritt \cong 0.001 Sekunden \rightarrow Maximale Eingabelänge bei gegebener Rechenzeit:

Laufzeit T(n)	1 Sekunde	1 Minute	1 Stunde
n	1000	60000	3600000
$n \log n$	140	4895	204094
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

- Annahme: Wir können einen 10-fach schnelleren Rechner verwenden \rightarrow Statt eines Problems der Größe p kann in gleicher Zeit dann berechnet werden:

Laufzeit T(n)	neue Problemgröße
n	$10p$
$n \log n$	fast $-10p \approx -10 \frac{\log p}{\log(10p)}$
n^2	$3.16p \approx \sqrt{10}p$
n^3	$2.15p \approx \sqrt[3]{10}p$
2^n	$3.32 \cdot p \approx \log_{10} p$

G. Zachmann Informatik 1 - WS 05/06 Komplexität 36

Allg. Bestimmung des Zeitaufwands mit Groß-O

- Sei A ein Programmstück, dann ist Zeitaufwand T(A) im Fall:
 - A ist einfache Anweisung oder arithm./log. Ausdruck → $T(A) = const \in O(1)$
 - A ist Folge von Anweisungen → Additionsregel anwenden
 - A ist if-Anweisung →
 - (a) `if cond: B` → $T(A) = T(cond) + T(B)$
 - (b) `if cond: B else: C` → $T(A) = T(cond) + \max(T(B), T(C))$
 - A ist eine Schleife (while, for, ...) →

$$T(A) = \sum_{\text{Umlauf } i} T(\text{Anweisungen } i) + T(\text{Terminierungsbedingung } i)$$
 oft einfach
 $T(A) = \# \text{Umläufe} \cdot (T(\text{Anweisungen}) + T(\text{Terminierungsbedingung}))$
 - A ist Rekursion → später

G. Zachmann Informatik 1 - WS 05/06 Komplexität 37

Beispiele zur Laufzeitabschätzung

Problem: `prefixAverages1(X)`
 Eingabe: Ein Array X von n Zahlen
 Ausgabe: Ein Array A von Zahlen, so daß gilt: A[i] ist das arithmetische Mittel der Zahlen X[0], ..., X[i]

Algorithmus:

```

for i in range(0, n):
    a = 0
    for j in range(0, i+1):
        a += X[j]
    A[i] = a / (i + 1)
return A
  
```

Annotations: $O(1)$ for `a = 0`, `A[i] = a / (i + 1)`, and `return A`. The inner loop is $O(i)$ for each `i`. The total complexity is $O(n^2)$.

G. Zachmann Informatik 1 - WS 05/06 Komplexität 38

Exkurs

- Das eben gestellte Problem kann man auch effizienter lösen

Algorithmus `prefixAverages2(X)`

```

s = 0.0
for i in range(0, n):
    s += X[i]
    A[i] = s / (i + 1)
return A
  
```

Annotations: The inner loop is $O(1)$. The total complexity is $O(n)$.

G. Zachmann Informatik 1 - WS 05/06 Komplexität 39

Average-Case-Komplexität

- Nicht leicht zu handhaben, für die Praxis jedoch relevant
- Sei $p_n(x)$ Wahrscheinlichkeitsverteilung, mit der Eingabe x mit Länge n auftritt
- Mittlere (erwartete) Laufzeit:

$$\bar{T}(n) = \sum_{x, |x|=n} T(x) p_n(x)$$
- Wichtig:
 - Worüber wird gemittelt?
 - Sind alle Eingaben der Länge n gleichwahrscheinlich?
 - Oft: Annahme der Gleichverteilung aller Eingaben x der Länge n
 - Dann ist $p_n(x) \equiv 1/N$, $N = \text{Anzahl aller mögl. Eingaben der Länge } n$
- $$\bar{T}(n) = \frac{1}{N} \sum_{x, |x|=n} T(x)$$

G. Zachmann Informatik 1 - WS 05/06 Komplexität 40

Beispiel

- Taktzahl (Anzahl Bitwechsel) eines **seriellen Addierers** bei Addition von 1 zu einer in Binärdarstellung gegebenen Zahl i der Länge n , d.h. $0 \leq i \leq 2^{n-1}$.
- Sie beträgt 1 plus der Anzahl der Einsen am Ende der Binärdarstellung von i .
- **Worst Case:** $n + 1$ Takte
 - **Beispiel:** Addition von 1 zu 111...1.
- **Average Case:**
 - Wir nehmen eine Gleichverteilung auf der Eingabemenge an.
 - Es gibt 2^{n-k} Eingaben der Form $(x, \dots, x, 0, 1, \dots, 1)$ wobei $k-1$ Einsen am Ende stehen.
 - Hinzu kommt die Eingabe $i = 2^n - 1$, für die das Addierwerk $n+1$ Takte benötigt.

G. Zachmann Informatik 1 - WS 05/06 Komplexität 41

- Die **average-case Rechenzeit** $\bar{T}(n)$ beträgt also:

$$\bar{T}(n) = \frac{1}{2^n}((n+1) + \sum_{1 \leq k \leq n} 2^{n-k}k)$$
- Es ist

$$\begin{aligned} \sum_{1 \leq k \leq n} 2^{n-k}k &= n2^{n-n} + \dots + 2 \cdot 2^{n-2} + 1 \cdot 2^{n-1} \\ &= 2^0 + \dots + 2^{n-3} + 2^{n-2} + 2^{n-1} \\ &\quad + 2^0 + \dots + 2^{n-3} + 2^{n-2} \\ &\quad + 2^0 + \dots + 2^{n-3} \\ &\quad \vdots \\ &\quad + 2^0 \\ &= (2^n - 1) + \dots + (2^1 - 1) \\ &= 2^{n+1} - 2 - n \end{aligned}$$

G. Zachmann Informatik 1 - WS 05/06 Komplexität 42

- Demnach ist

$$\bar{T}(n) = 2^{-n}(2^{n+1} - 2 - n + (n+1)) = 2 - 2^{-n}$$
- Es genügen also im Durchschnitt 2 Takte, um eine Addition von 1 durchzuführen.

G. Zachmann Informatik 1 - WS 05/06 Komplexität 43