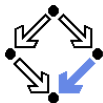


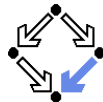
Berechenbarkeit und Komplexität

Komplexität von Algorithmen

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

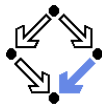
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
<http://www.risc.uni-linz.ac.at>





Wie wächst der Aufwand für die Ausführung eines Algorithmus mit wachsender Problemgröße?

- **Problemgröße:**
 - Eigenschaft der Eingabe, die ein Maß für den Aufwand darstellt.
- **Zeitkomplexität** eines Algorithmus:
 - Zeitaufwand als Funktion der Problemgröße.
 - Grenzverhalten: **asymptotische Zeitkomplexität.**
 - Problemgröße gegen unendlich.
- **Raumkomplexität** eines Algorithmus:
 - Speicheraufwand als Funktion der Problemgröße.
 - Grenzverhalten: **asymptotische Raumkomplexität.**
- **worst-case Komplexität:**
 - Maximale Komplexität für alle Eingaben der gegebenen Problemgröße.
- **erwartete Komplexität:**
 - Durchschnittliche Komplexität über alle Eingaben einer Problemgröße.



Ordnung der Komplexitätsfunktion

Im allgemeinen ist man nur an der “Größenordnung” einer Komplexitätsfunktion interessiert.

- $g(n) = \mathcal{O}(f(n))$

“ $g(n)$ ist von der Ordnung $f(n)$ ”

- $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$

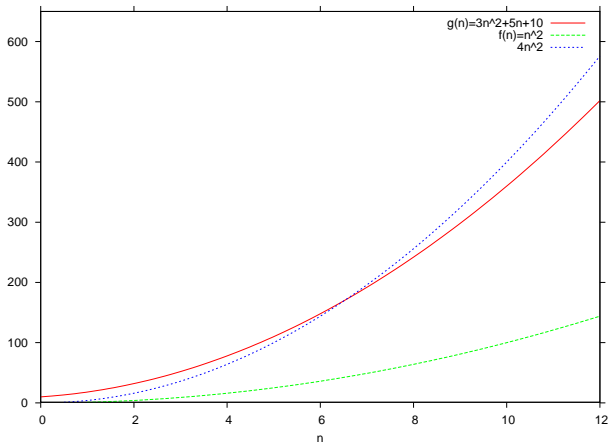
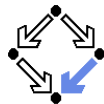
- g ist eine Komplexitätsfunktion.

- f beschreibt die “Größenordnung” von g .

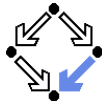
- $g(n) = \mathcal{O}(f(n)) : \Leftrightarrow \exists c \in \mathbb{R}_{>0} : \exists N \in \mathbb{N} : \forall n \geq N : g(n) \leq c \cdot f(n)$

- Ab einer gewissen Eingabegröße N wächst f mindestens so schnell wie g (d.h. ergibt mit einer gewissen Konstante c multipliziert einen mindestens so großen Wert).

Additive und multiplikative Konstanten spielen bei der Ordnung einer Komplexitätsfunktion keine Rolle.



Es gilt $g(n) = \mathcal{O}(f(n))$.



■ $6n^2 + 7n = \mathcal{O}(n^2)$

■ Wir setzen $c := 7$ und $N := 7$.

■ Es gilt für alle $n \geq 7$:

$$6n^2 + 7n \leq 6n^2 + n \cdot n = 6n^2 + n^2 = 7n^2$$

■ $\log_a n = \mathcal{O}(\log_b n)$ (für alle $a, b \in \mathbb{R}_{\geq 0}$)

■ Wir setzen $c := \log_a b$ und $N := 0$.

■ Es gilt für alle $n \geq 0$:

$$\log_a n = \log_a b \cdot \log_b n$$

■ $2^n + n^2 = \mathcal{O}(2^n)$

■ Wir setzen $c := 2$ und $N := 4$.

■ Es gilt für alle $n \geq 4$:

$$2^n + n^2 \stackrel{(*)}{\leq} 2^n + 2^n = 2 \cdot 2^n$$

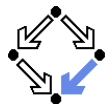
■ Lemma (*): $\forall n \geq 4 : n^2 \leq 2^n$.

■ Induktionsbasis ($n = 4$): $4^2 = 16 = 2^4$.

■ Induktionsschritt: $(n + 1)^2 = n^2 + 2n + 1 \leq n^2 + 2n + 2n = n^2 + 4n \leq$

$$n^2 + n \cdot n = n^2 + n^2 \stackrel{\text{Ind.hyp.}}{\leq} 2 \cdot 2^n = 2^{n+1}.$$

Bedeutung der Asymptotischen Komplexität



Macht die Entwicklung immer schnellerer Computer die Entwicklung asymptotisch schnellerer Algorithmen überflüssig?

- Verschiedene Algorithmen zur Lösung des gleichen Problems:

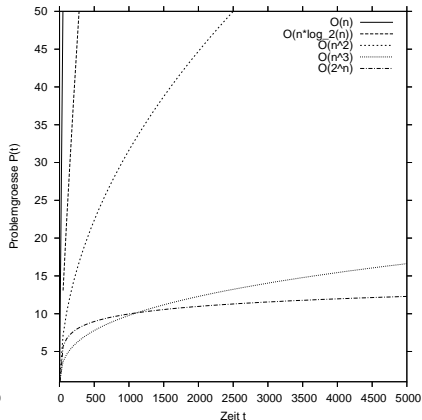
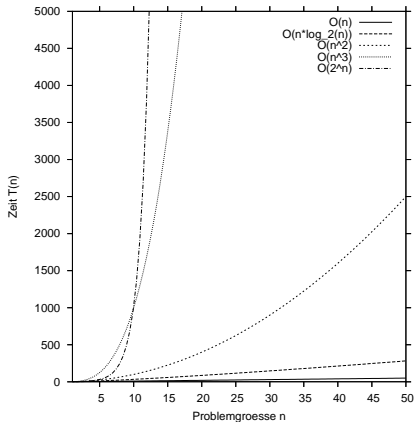
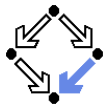
Algorithmus	Zeitkomplexität	Maximale Problemgröße n		
		1s	1m	1h
A_1	$\mathcal{O}(n)$	1.000	60.000	3.600.000
A_2	$\mathcal{O}(n \log n)$	140	4893	250.000
A_3	$\mathcal{O}(n^2)$	31	244	1897
A_4	$\mathcal{O}(n^3)$	10	39	153
A_5	$\mathcal{O}(2^n)$	9	15	21

- 10 mal schnellerer Computer kommt auf den Markt:

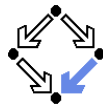
Algorithmus	Zeitkomplexität	Maximale Problemgröße n			Zunahme
		1s	1m	1h	
A_1	$\mathcal{O}(n)$	10.000	600.000	36.000.000	*10
A_2	$\mathcal{O}(n \log_2 n)$	1.003	39.311	1.736.782	*7 (\approx)
A_3	$\mathcal{O}(n^2)$	100	774	6000	*3.16 ($\sqrt{10}$)
A_4	$\mathcal{O}(n^3)$	21	84	330	*2.15 ($\sqrt[3]{10}$)
A_5	$\mathcal{O}(2^n)$	13	19	25	+3.3 ($\log_2 10$)

Problemgröße wächst nur deutlich bei niedriger Laufzeitkomplexität!

Bedeutung der Asymptotischen Komplexität



Multiplikation ganzer Zahlen

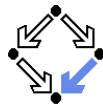


Wie schnell kann man zwei n -stellige Zahlen x und y multiplizieren?

- Klassischer Algorithmus: $\mathcal{O}(n^2)$.
 - n Multiplikationen $r_i \leftarrow x \cdot y_i$ der Komplexität $\mathcal{O}(n)$.
 - Jede Multiplikation besteht aus n Multiplikationen $x_j \cdot y_i$.
 - $n - 1$ Additionen $r_1 + \dots + r_n$ der Komplexität $\mathcal{O}(n)$.
 - Jede Addition enthält maximal $2n$ Additionen zweier Ziffern.
- Karatsuba und Ofman (1962): $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.59})$
 - Grundidee: “divide und conquer” Prinzip.
 - Führt die Multiplikation zweier n -stelliger Zahlen zurück auf drei Multiplikationen zweier $n/2$ -stelliger Zahlen.

Verbesserung auch von klassischen Algorithmen möglich.

Karatsuba-Algorithmus



Wir nehmen an, die Anzahl der Stellen n ist eine Potenz von 2.

Karatsuba($\downarrow n, \downarrow x, \downarrow y, \uparrow z$):

if $n = 1$ **then**

$z \leftarrow x_1 \cdot y_1$

else

$a \leftarrow x_{1\dots n/2}; b \leftarrow x_{n/2+1\dots n}$

$c \leftarrow y_{1\dots n/2}; d \leftarrow y_{n/2+1\dots n}$

Karatsuba($\downarrow n/2, \downarrow a + b, \downarrow c + d, \uparrow u$)

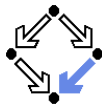
Karatsuba($\downarrow n/2, \downarrow a, \downarrow c, \uparrow v$)

Karatsuba($\downarrow n/2, \downarrow b, \downarrow d, \uparrow w$)

$z \leftarrow v \cdot \beta^n + (u - v - w) \cdot \beta^{n/2} + w$

end Karatsuba.

Nimmt an, dass die Additionen $a + b$ und $c + d$ keinen Übertrag zu $n + 1$ Stellen ergeben (siehe Skriptum für die Behandlung auch dieses Falls).



Analyse des Karatsuba-Algorithmus

- **Korrektheit:** sei β die Basis der Zahlendarstellung.

- $x = a \cdot \beta^{n/2} + b$

- $y = c \cdot \beta^{n/2} + d$

- $$\begin{aligned}x \cdot y &= (a \cdot \beta^{n/2} + b) \cdot (c \cdot \beta^{n/2} + d) \\&= ac \cdot \beta^n + (ad + bc) \cdot \beta^{n/2} + bd \\&= ac \cdot \beta^n + ((a + b) \cdot (c + d) - ac - bd) \cdot \beta^{n/2} + bd \\&= v \cdot \beta^n + (u - v - w) \cdot \beta^{n/2} + w\end{aligned}$$

- **Laufzeit:**

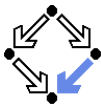
$$T(n) = \begin{cases} k, & \text{wenn } n = 1 \\ 3T(n/2) + kn & \text{wenn } n > 1 \end{cases}$$

- k konstanter Faktor.

- $k \dots$ Obergrenze für die Addition im "then"-Zweig.

- $kn \dots$ Obergrenze für Additionen/Verschiebungen im "else"-Zweig.

Lösung der Rekursionsrelation notwendig.



Analyse des Karatsuba-Algorithmus

■ **Behauptung:** $T(n) = 3kn^{\log_2 3} - 2kn$.

■ Induktionsbasis $n = 1$:

$$T(1) = k = 3k - 2k = 3k \cdot 1^{\log_2 3} - 2k \cdot 1.$$

■ Induktionsschritt $n = 2\bar{n}$:

$$\begin{aligned} T(2\bar{n}) &= 3T(\bar{n}) + 2k\bar{n} \\ &= 3(3k\bar{n}^{\log_2 3} - 2k\bar{n}) + 2k\bar{n} \\ &= 3k \cdot 3 \cdot \bar{n}^{\log_2 3} - 6k\bar{n} + 2k\bar{n} \\ &= 3k \cdot 2^{\log_2 3} \cdot \bar{n}^{\log_2 3} - 4k\bar{n} \\ &= 3k \cdot (2\bar{n})^{\log_2 3} - 2k \cdot (2\bar{n}). \end{aligned}$$

Für die Zeitkomplexität $T(n)$ des Karatsuba-Algorithmus gilt also $T(n) = \mathcal{O}(n^{\log_2 3})$.

Lösung linearer Rekursionsrelationen



Wie kommt man zur Lösung der Rekursionsrelation?

- **Satz:** seien a, b, n natürliche Zahlen. Die Lösung von

$$T(n) = \begin{cases} b, & \text{wenn } n = 1 \\ aT(n/c) + bn & \text{wenn } n > 1 \end{cases}$$

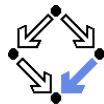
für eine Potenz n von c ist

$$T(n) = \begin{cases} \mathcal{O}(n), & \text{wenn } a < c \\ \mathcal{O}(n \log_c n), & \text{wenn } a = c \\ \mathcal{O}(n^{\log_c a}), & \text{wenn } a > c \end{cases}$$

- Beweis: siehe Skriptum.

Nützlich für die Analyse von linearen “divide and conquer” Algorithmen.

Lineare “divide and conquer”-Algorithmen



$$T(n) = \begin{cases} \mathcal{O}(n), & \text{wenn } a < c \\ \mathcal{O}(n \log_c n), & \text{wenn } a = c \\ \mathcal{O}(n^{\log_c a}), & \text{wenn } a > c \end{cases}$$

■ Lineare “divide und conquer”-Algorithmen:

- Problem wird in a Teilprobleme zerlegt.
- Jedes Teilproblem hat Größe n/c .
- Aufwand für Zerlegung und Kombination der Teilergebnisse ist linear.

■ Beispiel $c = 2$: Teile halber Größe.

- $a = 1$ Teil: $T(n) = \mathcal{O}(n)$.
- $a = 2$ Teile: $T(n) = \mathcal{O}(n \log_2 n)$.
- $a = 3$ Teile: $T(n) = \mathcal{O}(n^{\log_2 3})$.
- $a = 4$ Teile: $T(n) = \mathcal{O}(n^{\log_2 4}) = \mathcal{O}(n^2)$.
- $a = 8$ Teile: $T(n) = \mathcal{O}(n^{\log_2 8}) = \mathcal{O}(n^3)$.

Die Anzahl der Teile geht in die Laufzeitkomplexität logarithmisch ein.

Komplexität von RAM Programmen



Abhängig von den Kosten, die für die Ausführung einer Instruktion angenommen werden.

■ Uniformes Kostenkriterium:

- Jedes Register benötigt eine Raumeinheit.
- Jede Instruktion benötigt eine Zeiteinheit.

■ Logarithmisches Kostenkriterium:

- Anzahl der für ein Register benötigten Speicherzellen ist abhängig von der Länge $l(n)$ des Inhalts n des Registers:

$$l(n) := \begin{cases} \lfloor \log |n| \rfloor + 1, & \text{wenn } n \neq 0 \\ 1, & \text{wenn } n = 0 \end{cases}$$

Das uniforme Kostenkriterium ist die Default-Annahme bei der Analyse der Komplexität von RAM-Programmen.

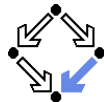
Logarithmische Komplexität



Wir betrachten die Auswirkungen des logarithmischen Kostenkriteriums.

- **Logarithmische Raumkomplexität:** $\sum l(x_i)$
 - Summe über alle verwendeten Register i (einschließlich Akkumulator).
 - x_i ist dabei die betragsmäßig größte Zahl, die während der Berechnung je in Register i gespeichert wird.
- **Logarithmische Kosten** für Operanden:
 - Zeit $t(a)$ für den Zugriff auf den Operanden a :
$$t(=i) := l(i)$$
$$t(i) := l(i) + l(c(i))$$
$$t(*i) := l(i) + l(c(i)) + l(c(c(i)))$$
- **Logarithmische Kosten** für Instruktionen:
 - Zeit für die Ausführung der Instruktion.
 - Annahme: Zeit ist proportional zur Länge des Operanden.

Logarithmische Komplexität

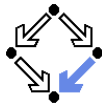


Die logarithmischen Kosten für jede RAM-Instruktion:

Instruktion	Kosten
LOAD a	$t(a)$
STORE i	$l(c(0)) + l(i)$
STORE $*i$	$l(c(0)) + l(i) + l(c(i))$
ADD a	$l(c(0)) + t(a)$
SUB a	$l(c(0)) + t(a)$
MULT a	$l(c(0)) + t(a)$
DIV a	$l(c(0)) + t(a)$
READ i	$l(input) + l(i)$
READ $*i$	$l(input) + l(i) + l(c(i))$
WRITE a	$t(a)$
JUMP b	1
JGTZ b	$l(c(0))$
JZERO b	$l(c(0))$
HALT	1

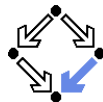
Auch Multiplikation und Division gelten als lineare Operationen (?!).

Beispiel



RAM-Programm für die Funktion $f(n) = \begin{cases} n^n, & \text{wenn } n > 0 \\ 0, & \text{sonst} \end{cases}$

read r_1	READ	1	Eingabe von $c(1)$	
if $r_1 \leq 0$ then	LOAD	1	$c(0) \leftarrow c(1)$	
write 0	JGTZ	else	wenn $c(0) > 0$, else	
else	WRITE	=0	Ausgabe von 0	
$r_2 \leftarrow r_1$	JUMP	halt		
$r_3 \leftarrow r_1 - 1$	else:	LOAD	1	$c(0) \leftarrow c(1)$
while $r_3 > 0$ do	STORE	2	$c(2) \leftarrow c(0)$	
$r_2 \leftarrow r_2 \cdot r_1$	LOAD	1	$c(0) \leftarrow c(1)$	
$r_3 \leftarrow r_3 - 1$	SUB	=1	$c(0) \leftarrow c(0) - 1$	
write r_2	STORE	3	$c(3) \leftarrow c(0)$	
	while:	LOAD	3	$c(0) \leftarrow c(3)$
	JGTZ	body	wenn $c(0) > 0$, body	
	JUMP	done		
	body:	LOAD	2	$c(0) \leftarrow c(2)$
	MULT	1	$c(0) \leftarrow c(0) \cdot c(1)$	
	STORE	2	$c(2) \leftarrow c(0)$	
	LOAD	3	$c(0) \leftarrow c(3)$	
	SUB	=1	$c(0) \leftarrow c(0) - 1$	
	STORE	3	$c(3) \leftarrow c(0)$	
	JUMP	while		
	done:	WRITE	2	Ausgabe von $c(2)$
	halt:	HALT		



■ Zeitkomplexität:

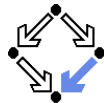
- Bestimmt durch die Zeitkomplexität der MULT-Instruktionen:
 - Schleife führt $n - 1$ Iterationen aus.
 - In Iteration i enthält Akkumulator n^i und r_2 enthält n .
- Uniformes Kostenkriterium: $(n - 1) \cdot 1 = \mathcal{O}(n)$
- Logarithmisches Kostenkriterium: $\mathcal{O}(n^2 \log n)$
 - i -te MULT-Instruktion: $l(n^i) + l(n) \approx i \cdot \log n + \log n = (i + 1) \log(n)$
 - Alle MULT-Instruktionen: $\sum_{i=1}^{n-1} (i + 1) \log n = \mathcal{O}(n^2 \log n)$

■ Raumkomplexität:

- Inhalt der Register 0 bis 3.
- Uniformes Kostenkriterium: $3 = \mathcal{O}(1)$
- Logarithmisches Kostenkriterium: $\mathcal{O}(n \log n)$
 - n^n ist die größte gespeicherte Zahl.
 - $l(n^n) \approx n \log n$

Das logarithmische Kostenkriterium ist realistischer, wenn die beteiligten Zahlen nicht mehr in eine einzelne Speicherzelle passen.

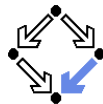
Beispiel



RAM-Programm, das alle Wörter aus 1en und 2en akzeptiert, die aus gleich vielen 1en wie 2en bestehen.

$r_2 \leftarrow 0$	LOAD	=0	$c(0) = 0$
read r_1	STORE	2	$c(2) = c(0)$
while $r_1 \neq 0$ do	READ	1	Eingabe von r_1
if $r_1 \neq 1$	while: LOAD	1	$c(0) \leftarrow c(1)$
then $r_2 \leftarrow r_2 - 1$	JZERO	done	wenn $c(0) = 0$, done
else $r_2 \leftarrow r_2 + 1$	LOAD	1	$c(0) \leftarrow c(1)$
read r_1	SUB	=1	$c(0) \leftarrow c(0) - 1$
if $r_2 = 0$ then write 1	JZERO	one	wenn $c(0) = 0$, one
	LOAD	2	$c(0) \leftarrow c(2)$
	SUB	=1	$c(0) \leftarrow c(0) - 1$
	STORE	2	$c(2) = c(0)$
	JUMP	read	
	one: LOAD	2	$c(0) \leftarrow c(2)$
	ADD	=1	$c(0) \leftarrow c(0) + 1$
	STORE	2	$c(2) = c(0)$
	read: READ	1	Eingabe von $c(1)$
	JUMP	while	
	done: LOAD	2	$c(0) \leftarrow c(2)$
	JZERO	write	wenn $c(0) = 0$, write
	HALT		
	write: WRITE	=1	Ausgabe von 1
	HALT		

Beispiel

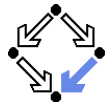


Wort aus n Zeichen wird gelesen.

- n Iterationen.
- In jeder Iteration Addition von 1 zu einer Zahl kleiner n .
- Maximaler Absolutbetrag n aller gespeicherten Zahlen.

	Uniforme Kosten	Logarithmische Kosten
Zeitkomplexität	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Raumkomplexität	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

Komplexität von RAM/RASP-Simulationen

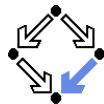


Die folgenden Sätze gelten sowohl für das uniforme als auch für das logarithmische Kostenkriterium.

- **Satz:** Für jedes RAM-Programm mit Zeitkomplexität $T(n)$ gibt es eine Konstante k und ein äquivalentes RASP-Programm mit Zeitkomplexität $kT(n)$.
 - Beweis: das zum RAM-Programm P äquivalente RASP-Programm P' enthält für jede Instruktion von P maximal 6 Instruktionen. Unter dem uniformen Kostenkriterium ist die Zeitkomplexität von P' also $6T(n)$. Durch Analyse von P' kann man zeigen, dass für das logarithmische Kostenkriterium die Zeitkomplexität von P' gleich $(6 + 11 \cdot l(r)) \cdot T(n)$ ist (r ist der bei der Simulation angenommene Verschiebungsfaktor).
- **Satz:** Für jedes RASP-Programm mit Zeitkomplexität $T(n)$ gibt es eine Konstante k und ein äquivalentes RAM-Programm mit Zeitkomplexität $kT(n)$.

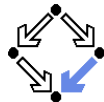
Die Simulation von RAM durch RASP und umgekehrt kostet nur einen konstanten Zeitfaktor.

Komplexität von Turing-Maschinen



- Die (**worst-case**) **Zeitkomplexität** $T(n)$ einer Turing-Maschine M :
 - $T(n)$ ist die maximale Anzahl der Übergänge zu einer neuen Konfiguration bei einer Berechnung, die mit einem beliebigen Eingabewort der Länge n beginnt.
 - Terminiert die Berechnung für ein Eingabewort der Länge n nicht, ist $T(n)$ undefiniert.
- Die (**worst-case**) **Raumkomplexität** $S(n)$ einer Turing-Maschine M :
 - $S(n)$ ist die maximale Distanz, die sich einer der L/S-Köpfe während einer Berechnung, die mit einem beliebiges Eingabewort der Länge n beginnt, vom linken Bandende entfernt.
 - Gibt es ein Eingabewort der Länge n , für das sich der L/S-Kopf unbegrenzt nach rechts bewegt, ist $S(n)$ undefiniert.
- Die **Zeit/Raumkomplexität einer nicht-deterministischen T.-M.** M :
 - Die maximale Zeit/Raumkomplexität, die M für eine Eingabe der Länge n (für eine beliebige Folge von nichtdeterministischen Wahlen) erreichen kann.

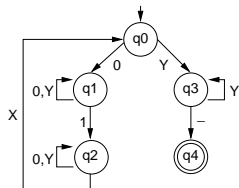
Beispiel



$M_1 = (Q, \Sigma, \Gamma, q_0, F, \delta)$ zur Erkennung der Wörter $0^m 1^m$ ($m \in \mathbb{N}$).

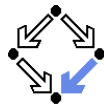
$Q = \{q_0, q_1, q_2, q_3, q_4\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{\sqcup, 0, 1, X, Y\}$, $F = \{q_4\}$

δ	\sqcup	0	1	X	Y
q_0	—	(q_1, X, R)	—	—	(q_3, Y, R)
q_1	—	$(q_1, 0, R)$	(q_2, Y, L)	—	(q_1, Y, R)
q_2	—	$(q_2, 0, L)$	—	(q_0, X, R)	(q_2, Y, L)
q_3	(q_4, \sqcup, R)	—	—	—	(q_3, Y, R)
q_4	—	—	—	—	—



- $T(n) = \mathcal{O}(n^2)$ (für Eingabewort w der Länge n):
 - Hauptschleife wird höchstens $(n+1)/2$ mal durchlaufen.
 - Ersetzt in jedem Durchlauf zwei Symbole von w durch X und Y .
 - Jeder Schleifendurchlauf: $2n+2$ Schritte.
 - $n+1$ Schritte für die Suche nach rechts, n Schritte für die Suche nach links, 1 Schritt für die Bewegung zum nächsten Symbol.
 - Schlussbehandlung: n Schritte.
 - In Summe $(2n+2) \cdot (n+1)/2 + n = (n+1)^2 + n$ Schritte.
- $S(n) = \mathcal{O}(n)$:
 - L/S -Kopf bewegt sich höchstens über eine Position von w hinaus.

Komplexität von RAM/RASP-Simulationen



- Zwei Funktionen $f_1, f_2 : \mathbb{N} \rightarrow \mathbb{N}$ sind in **polynomialer Relation**:
 - Es gibt Polynome $p_1(x)$ und $p_2(x)$ mit ganzzahligen Koeffizienten, sodass für alle $n \in \mathbb{N}$ gilt:

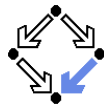
$$f_1(n) \leq p_1(f_2(n)) \text{ und } f_2(n) \leq p_2(f_1(n))$$

- Beispiele:

- $2n^2$ und n^5 sind in polynomialer Relation.
 - $p_1(x) = 2x, p_2(x) = x^3$.
 - $2n^2 \leq 2n^5$ und $n^5 \leq (2n^2)^3$
- a^n und b^n sind in polynomialer Relation (für $a, b > 1$):
 - $p(x) = x^{\lceil \log_b a \rceil}$.
 - $a^n = (b^{\log_b a})^n = (b^n)^{\log_b a} \leq p(b^n)$
- n^2 und 2^n sind nicht in polynomialer Relation:
 - Es gibt kein Polynom $p(x)$ sodass $p(n^2) \geq 2^n$, für alle n .

Kriterium, dass sich zwei Funktionen nicht mehr als um eine "polynomiale Transformation" voneinander unterscheiden.

Komplexität von RAM/RASP-Simulationen



- **Satz:** Das RAM/RASP Modell unter dem logarithmischen Kostenkriterium und das Turing-Maschinen-Modell sind in polnomialer Relation.
 - Zu jeder RAM/RASP R gibt es eine Turing-Maschine M , sodass die Komplexitätsfunktionen von R und M in polynomialer Relation stehen, und umgekehrt.
- Uniformes Kostenkriterium: jede Turing-Maschine mit Komplexität $\mathcal{O}(T(n))$ wird durch RAM mit Komplexität $\mathcal{O}(T^2(n))$ simuliert.
 - Umgekehrung gilt nicht: RAM kann durch iteratives Quadrieren in $\mathcal{O}(n)$ Schritten die Zahl 2^{2^n} berechnen ($2^{2^n} = (2^{2^{n-1}})^2$).
 - Turing-Maschine braucht 2^n Zellen, nur um diese Zahl zu speichern.
 - RAM benötigt dafür nur 1 Register.

Unter dem logarithmischen Kostenkriterium unterscheidet sich die Komplexität von RAM und Turing-Maschine "nicht essentiell".