OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

Algorithmen und Datenstrukturen

- Grundlagen (Komplexität) -

Prof. Dr. Klaus Volbert

Wintersemester 2018/19
Regensburg, 08. Oktober 2018

Komplexität (Zeit/Platz)

- Uniforme Komplexität (Einheitsmaß)
  - Jeder Befehl, der ausgeführt wird, entspricht einem Schritt (konstant)
  - Sei A ein Algorithmus für eine RAM mit Eingabe x
    - $T_A(x)$: Anzahl der Schritte, die A bei Eingabe von x durchführt
    - $S_A(x)$: Anzahl der Speicherzellen, die A bei Eingabe von x benutzt
- Nachteil der uniformen Komplexität
  - Befehle mit unterschiedlicher Bit-Komplexität werden gleich bewertet
- Logarithmische Komplexität (Logarithmisches Maß)
  - Jeder Befehl, der ausgeführt wird, wird mit der Bit-Länge der Operanden des Befehls gewichtet (entspricht der Bit-Komplexität)
  - Anstelle einer Speicherzelle wird die Bit-Länge der in einer Speicherzelle abgespeicherten größten Zahl verwendet
  - Kennzeichnung analog zu oben: ${T_A}^{\log}(x)$ bzw. ${S_A}^{\log}(x)$
- Beispiel: Addition und Multiplikation von zwei n-Bit Zahlen

Worst Case, Average Case, Best Case I

- Komplexität wird über alle Eingaben unterschieden nach
  - Worst Case: Laufzeit im schlechtesten Fall (Wesentlich!)
  - Average Case: Laufzeit im Mittel (Mittelwert)
  - Best Case: Laufzeit im besten Fall (Eher uninteressant)
- Beispiel: Addition um eins im Binärsystem
  - Eingabe: $\text{bin}(a) = (a_{n-1}, \ldots, a_0)$, $a \in \mathbb{N}$, $a_i \in \{0,1\}$
    (Binärdarstellung einer natürlichen Zahl)
  - Ausgabe: $\text{bin}(a) + 1$
  - Algorithmus:
    Falls $(a_{n-1}, \ldots, a_0) = (1, \ldots, 1)$
    Gib $(1,0 \ldots, 0)$ aus // n Nullen
    Sonst
    Ermittle i mit $a_i = 0$ und $a_j \neq 0$ für alle $j < i$
    Gib $(a_{n-1}, \ldots, a_{i+1}, 1,0, \ldots, 0)$ aus // i Nullen
  - Laufzeit: Worst Case: $n + 1$ Schritte // Wann?
    Best Case: 1 Schritt // Wann?

Worst Case, Average Case, Best Case II

- Average Case bei Addition um 1 im Binärsystem:
  - Wie viele mögliche Eingaben für a gibt es? 2^n
  - Wie viele Eingaben gibt es für den Worst Case (1. Fall)? 1
    - Laufzeit in diesem Fall: $n+1$ Schritte
  - Wie viele Eingaben gibt es mit $a_i = 0$ und $a_j \neq 0$ für alle $j < i$? 2^{n-i-1}
    - Laufzeit in diesen Fällen: $(i+1)$ Schritte
  - Insgesamt ergibt sich für die durchschnittliche Laufzeit:

$$\frac{\sum_{i=0}^{n-1}\left(2^{n-i-1}\cdot(i+1)\right)+n+1}{2^n} \text{ Schritte}$$

  - Es gilt:

$$\sum_{i=0}^{n-1} 2^{n-i-1}\cdot(i+1) = \sum_{i=1}^{n} 2^{n-i}\cdot i$$

Worst Case, Average Case, Best Case III

- Weiter gilt:

$$\sum_{i=1}^{n} 2^{n-i} \cdot i = 2^{n-1} \cdot 1 + 2^{n-2} \cdot 2 + \ \ldots + 2^0 \cdot n$$

$$= 2^{n-1} + 2^{n-2} + \cdots + 2^0 \quad \text{(Zeile 1)}$$
$$+ 2^{n-2} + \cdots + 2^0 \quad \text{(Zeile 2)}$$
$$\ddots \qquad \vdots \qquad \vdots$$
$$+ 2^0 \quad \text{(Zeile n)}$$

- Erinnerung Geometrische Reihe:

$$\sum_{k=0}^{m} q^k = \frac{q^{m+1} - 1}{q - 1} \text{ für } q \neq 1$$

- Damit folgt:

$$\sum_{i=1}^{n} 2^{n-i} \cdot i = \frac{2^{n-1+1} - 1}{2 - 1} + \frac{2^{n-2+1} - 1}{2 - 1} + \ \ldots + \frac{2^{n-n+1} - 1}{2 - 1}$$

Worst Case, Average Case, Best Case IV

- Es folgt schließlich:

$$\sum_{i=1}^{n} 2^{n-i} \cdot i = (2^n - 1) + (2^{n-1} - 1) + \ ... + (2^1 - 1)$$

$$= \sum_{i=1}^{n} 2^i - n = 2^{n+1} - 2 - n$$

- Für die durchschnittliche Laufzeit folgt:

$$\frac{(2^{n+1} - 2 - n) + (n+1)}{2^n} = 2 - \frac{1}{2^n} \text{ Schritte}$$

- Zusammenfassend:
  - Worst Case: $n + 1$ Schritte
  - Average Case: 2 Schritte (asymptotisch)
  - Best Case: 1 Schritt
- Positiv: Durchschnitt liegt nah am Best Case (nicht immer so!)

Worst Case, Average Case, Best Case V

- Sei A ein Algorithmus für eine RAM mit Eingabe x über einem Alphabet Σ
- Laufzeit im schlechtesten Fall (Worst Case)

$$T_A{}^{wc}(n) = \max_{x \in (\Sigma)^n} \{T_A(x)\}$$

- Laufzeit im besten Fall (Best Case)

$$T_A{}^{bc}(n) = \min_{x \in (\Sigma)^n} \{T_A(x)\}$$

- Laufzeit im Durchschnitt (Average Case)

$$T_A{}^{ac}(n) = \frac{1}{|\Sigma|^n} \sum_{x \in (\Sigma)^n} T_A(x)$$

Zeitkomplexität von C, C++, C#, Java - Programmen

- Alle ausgeführten Anweisungen werden „mit 1“ gezählt
- Anweisungen sind
  - Zuweisungen
  - Auswahl-Anweisungen (if-then-else, switch-case)
  - Iterationen (for, while-do, do-while)
  - Sprünge (break, continue, goto, return)
  - Funktionsaufrufe/Methodenaufrufe
- Bei Iterationen und Funktionsaufrufen/Methodenaufrufen
  - Alle im Rahmen der Iteration oder der Funktion/Methode ausgeführten Anweisungen werden natürlich auch gezählt (iterativ oder rekursiv)
- Vorsicht:
  - Funktions-/Methodenaufrufe in höheren Programmiersprachen sind nicht sofort ersichtlich (Beispiele: Konstruktor, Destruktor, Boolesche Abfragen, …)
- Einfache und komplexe Zuweisungen werden nicht unterschieden
  - Ziel: Ermittlung des asymptotischen Laufzeitverhaltens

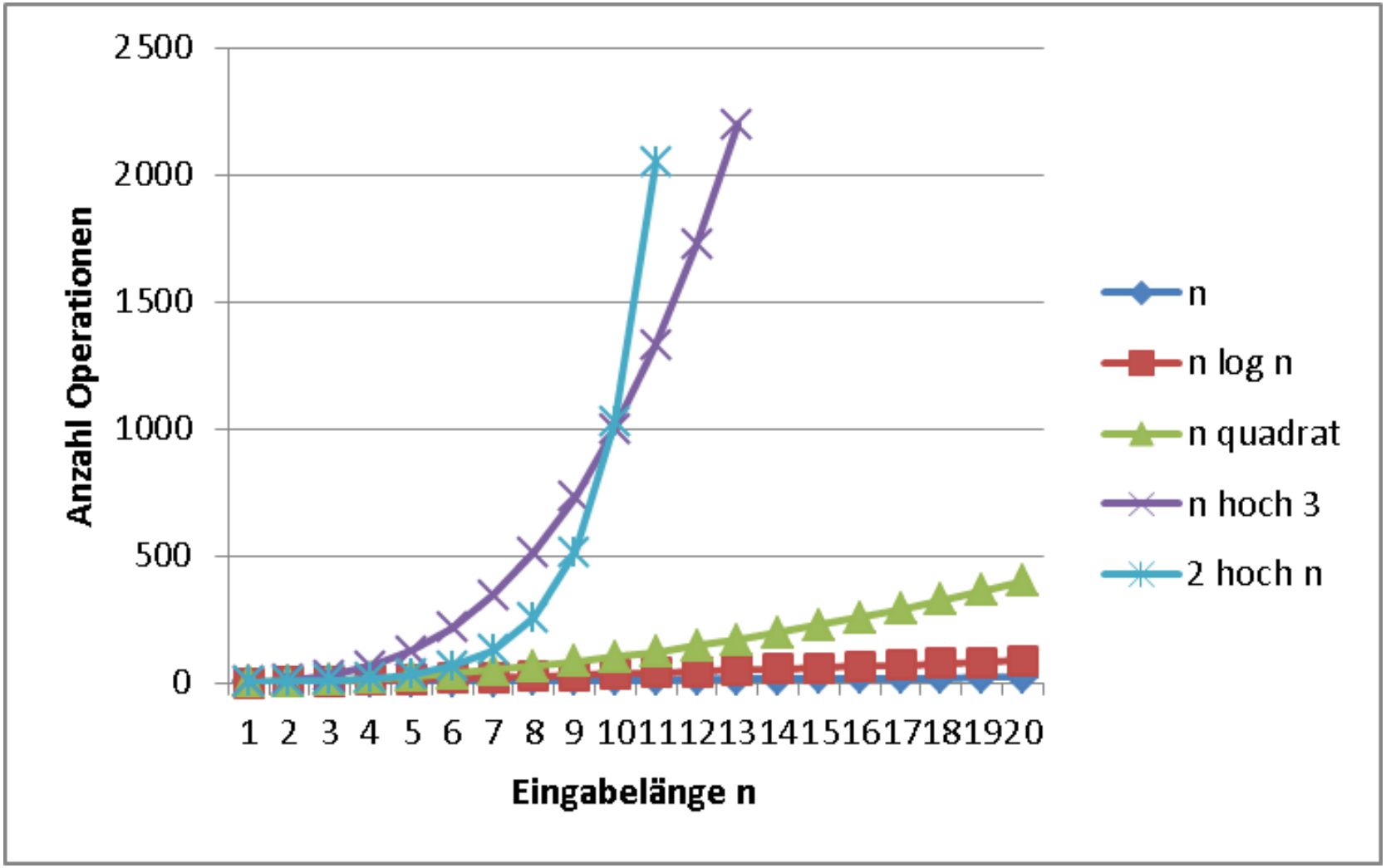
Beispiel: Laufzeitkomplexität

- Annahme:
  - Rechner kann 1.000 Operationen pro Sekunde ausführen (1.000 Hz)
  - Die folgende Tabelle zeigt jeweils die mögliche Problemgröße (Wert für n) bei vorgegebener Rechenzeit

Komplexität		1 Sekunde	1 Minute	1 Stunde	1 Tag	1 Woche
Linear	n	1.000	60.000	3.600.000	86.400.000	604.800.000
Super-Linear	$n \log_2 n$	140	4.895	204.094	3.943.234	24.631.427
Polynomiell	n^2	31	244	1897	9.295	24.592
	n^3	10	39	153	442	845
Exponentiell	2^n	9	15	21	26	29

- Effekt der technischen Entwicklung ist
  - bei linearem Laufzeitverhalten: ideal
  - bei polynomiellem Laufzeitverhalten: akzeptabel
  - bei exponentiellem Laufzeitverhalten: sehr schlecht

Wesentliche Wachstumsfunktionen



Asymptotische Kostenmaße
(Landau-Symbole, O-Notation)

- Größenordnung der Komplexität in Abhängigkeit der Eingabegröße
  - Best Case, Worst Case, Average Case
- Seien $f, g: \mathrm{IN} \to \mathrm{IR}^+$ zwei beliebige Funktionen, dann definieren wir
  - Groß-O-Notation (O-Notation, O-Kalkül), Obere Schranke:

g wächst höchstens wie f

$$O\big(f(n)\big) = \{\, g(n) \mid \exists\, c \in \mathrm{IR}^+, n_0 \in \mathrm{IN}: \forall\, n \geq n_0, g(n) \leq c \cdot f(n) \,\}$$

- Verwendung des Logarithmus im O-Kalkül:
  - Mit $\log(n)$ ist $\log_2(n)$ gemeint, d.h. der Logarithmus zur Basis 2
  - Zielbasis a ist unwesentlich, da:

$$\log_b n = \frac{\log_a n}{\log_a b} = \frac{1}{\log_a b} \log_a n \;\; (\log_a b \text{ ist konstant})$$

Anmerkungen zur O-Notation

- Abstrakte Abschätzung zur Klassifizierung
  - Extrahiert dominante Terme (Optimierung von Konstanten oft ineffektiv)
  - Abstrahiert von Konstanten (Versteckte Konstanten können groß sein!)
  - Vernachlässigt „geringfügige“ Terme
- Rechenregeln

$$\lim_{n\to\infty} \frac{g(n)}{f(n)} \leq c \Rightarrow g(n) \in O(f(n))$$

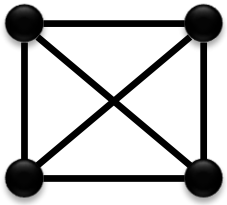
  - $\forall f: f(n) \in O(f(n))$
  - $\forall f: \forall c \in \mathrm{IR}^+: c\, f(n) \in O\big(f(n)\big),\ c + f(n) \in O\big(f(n)\big)$
  - $\forall g \in O\big(f(n)\big): \forall c \in \mathrm{IR}^+: g(n) + g(n) \in O\big(f(n)\big),\ c\, g(n) \in O\big(f(n)\big)$
  - $\forall g_1 \in O\big(f_1(n)\big): \forall g_2 \in O\big(f_2(n)\big): g_1(n) + g_2(n) \in O\big(f_1(n) + f_2(n)\big)$
  - $\forall g_1 \in O\big(f_1(n)\big): \forall g_2 \in O\big(f_2(n)\big): g_1(n) + g_2(n) \in O(\max(f_1(n), f_2(n)))$
  - $\forall g_1 \in O\big(f_1(n)\big): \forall g_2 \in O\big(f_2(n)\big): g_1(n)\, g_2(n) \in O\big(f_1(n)\, f_2(n)\big)$

Beispiele O-Notation

- Wie lautet die minimale obere Schranke für $3n^3 + 4n^2 + 5n + 42$?
  - $3n^3 + 4n^2 + 5n + 42 \in O(n^3)$, da:
  - Aus $3n^3 + 4n^2 + 5n + 42 \leq c \cdot n^3$ folgt:

$$3 + \frac{4}{n} + \frac{5}{n^2} + \frac{42}{n^3} \leq 3 + 4 + 5 + 42 = 54, \text{d. h. } c \geq 54$$

- Anzahl der Kanten in einem vollständigen Graph mit n Knoten ist $O(n)$?
  - Falsch, da:

$n = 4$  6 Kanten

  - Anzahl Kanten, die ein Graph mit n Knoten hat, ist:

$$n - 1 + n - 2 + \cdots + 0 = \sum_{i=0}^{n-1} i = \frac{(n-1)n}{2} = \frac{n^2 - n}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \in O(n^2)$$

Erweiterungen der O-Notation

- Seien $f, g: \mathrm{IN} \to \mathrm{IR}^+$ zwei beliebige Funktionen, dann definieren wir
  - Groß-Omega-Notation, Untere Schranke:

g wächst mindestens wie f

$$\Omega(f(n)) = \{\, g(n) \mid \exists\, c \in \mathrm{IR}^+, n_0 \in \mathrm{IN}: \forall\, n \geq n_0, g(n) \geq c \cdot f(n) \,\}$$

  - Groß-Theta-Notation, Exakte Schranke:

g wächst wie f

$$\Theta(f(n)) = \{\, g(n) \mid g(n) \in O(f(n)) \text{ und } g(n) \in \Omega(f(n)) \,\}$$

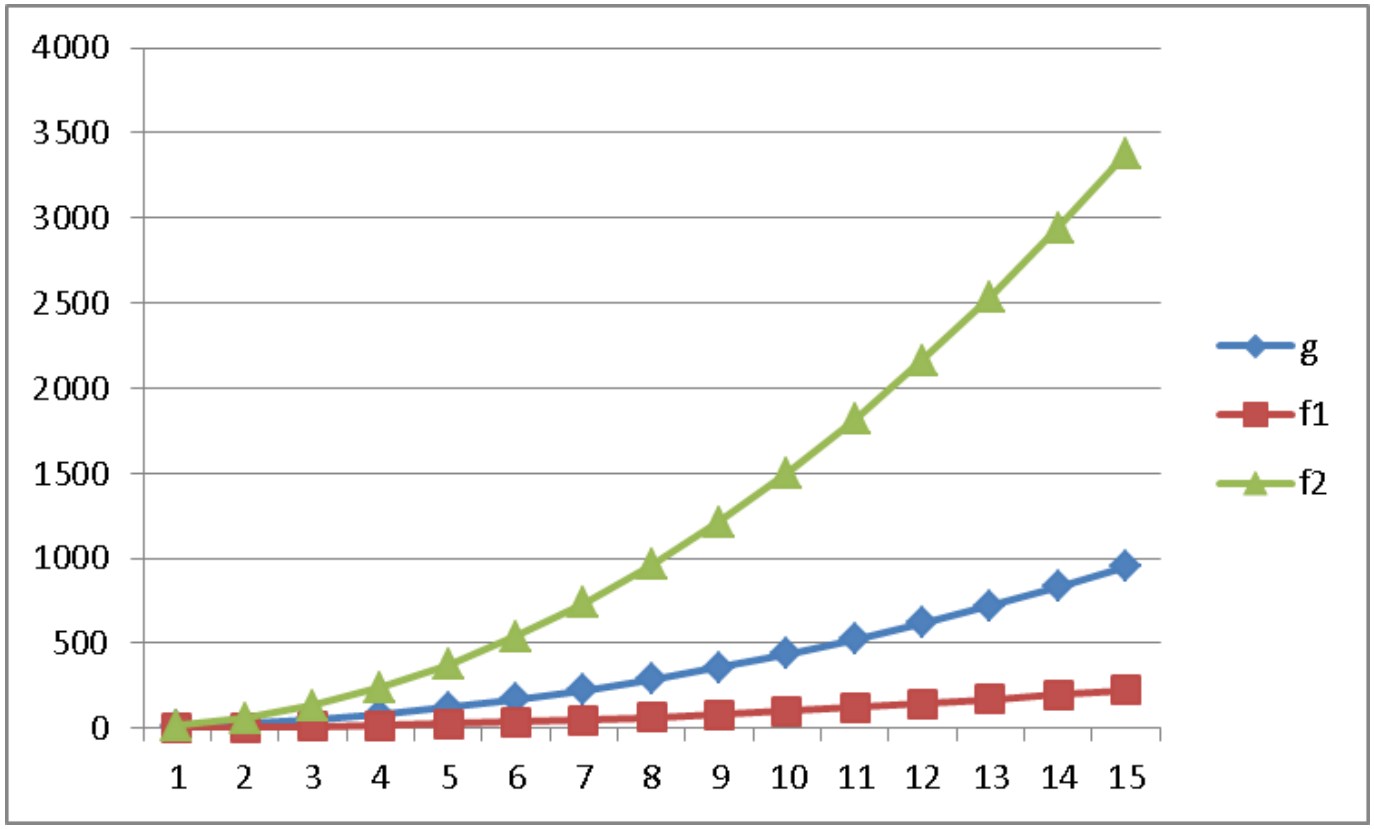
  - Seltener verwendet: o wie O, aber $<$ statt $\leq$ und ω wie Ω, aber $>$ statt $\geq$

- Schreibweise:
  - Statt $g(n) \in O(f(n))$ wird auch $g(n) = O(f(n))$ (analog für Ω und Θ)
  - Vorsicht beim Umgang, da es sich nicht um ein echtes „=“ handelt !!!

Beispiel zur Θ-Notation

- Betrachte $g(n) = 4n^2 + 3n + 7$
- Es gilt $g(n) = \Theta(n^2)$, da für alle n:

$$f_1(n) := n^2 \leq g(n) \leq 15n^2 =: f_2(n)$$



Beispiel MaxTeilSum

- Eingabe: $a_0, \ldots, a_{n-1} \in \mathbb{Z}$ (n ganze Zahlen)
- Ausgabe: Maximale Teilsumme, d.h.

$$s = \max_{0 \le i \le j \le n-1} \sum_{k=i}^{j} a_k$$

- Anwendungen
  - Erkennung von grafischen Mustern
  - Analyse von Aktienkursen
    (täglich neue Kurse: Ermittlung bester Ein-/Ausstiegszeitpunkt)
- Beispiel:
  - Eingabe: -13, 25, 34, 12, -3, 7, -87, 28, -77, 11
  - Ausgabe: 75 (ergibt sich aus $i = 1, j = 5$)

MaxTeilSum1

- Naiver Algorithmus: Durchlaufen aller Varianten

```
int MaxTeilsum1(int a[],int n) {
      int i, j, k, sum, max = int.MinValue;

      for (i=0; i<n; i++) {
            for (j=i; j<n; j++) {
                  sum=0;
                  for (k=i; k<=j; k++) sum += a[k];
                  if (sum > max) max = sum;
            }
      }

      return max;
}
```

- Laufzeit: $T_1{}^{wc}(n) = O(n^3)$ (kubisch)

Laufzeitanalyse MaxTeilSum1

- Zu verarbeitende Eingabedaten: n ganze Zahlen
- Laufzeit in Abhängigkeit der Eingabe sei ${T_1}^{wc}(n)$, dann gilt:

$$ {T_1}^{wc}(n) = 1 + \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} \left(\sum_{k=i}^{j} 1 + 2 \right) = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} (j - i + 3) = \sum_{i=0}^{n-1} \sum_{k=0}^{n-1-i} (k + 3) $$

$$ = \sum_{i=0}^{n-1} \sum_{k=0}^{i} k + \sum_{i=0}^{n-1} \sum_{k=0}^{i} 3 = \sum_{i=0}^{n-1} \frac{i(i+1)}{2} + \sum_{i=0}^{n-1} 3(i+1) $$

$$ = \frac{1}{2} \left(\sum_{i=0}^{n-1} i^2 + \sum_{i=0}^{n-1} i \right) + 3 \sum_{i=0}^{n-1} i + n $$

$$ = \frac{1}{2} \left(\frac{(n-1)n(2(n-1)+1)}{6} + \frac{(n-1)n}{2} \right) + \frac{3(n-1)n}{2} = \ldots = \Theta(n^3) $$

MaxTeilSum2

- Verbesserung: Statt immer wieder vollständige Summe zu berechnen: Erweiterung um den aktuellen Summanden

```
int MaxTeilsum2(int a[],int n) {
      int i, j, sum, max = int.MinValue;

      for (i=0; i<n; i++) {
            sum=0;
            for (j=i; j<n; j++) {
                  sum += a[j];
                  if (sum > max) max = sum;
            }
      }

      return max;   }
```

- Laufzeit: $T_2{}^{wc}(n) = O(n^2)$ (quadratisch)

MaxTeilSum3

- Verbesserung: Berechne lokales und globales Maximum. Das globale Maximum ist das Maximum über alle lokalen Maxima. Das lokale Maximum ist an einer Position entweder der Wert oder der Wert plus Werte „von links“.

```
int MaxTeilsum3(int a[],int n) {
      int i, s, max = int.MinValue, aktSum = 0;

      for (i=0; i<n; i++) {
            s = aktSum + a[i];
            if (s > a[i]) aktSum = s;
            else aktSum = a[i];
            if (aktSum > max) max = aktSum;
      }

      return max;   }
```

- Laufzeit: $T_3{}^{wc}(n) = O(n)$ (linear)

Beispiel MaxTeilSum3

Folge	i	aktSum	max
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11		0	-2147483648
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	0	-13	-13
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	1	25	25
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	2	59	59
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	3	71	71
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	4	68	71
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	5	75	75
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	6	-12	75
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	7	28	75
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	8	-49	75
-13, 25, 34, 12, -3, 7, -87, 28, -77, 11	9	11	75

MaxTeilSum3 (Korrektheit I)

- Behauptung:
  - Nach dem i.ten Schleifendurchlauf gilt (Schleifeninvariante):

$$\text{aktSum} = \max_{0 \le l \le i} \sum_{k=l}^{i} a_k\,, \qquad \text{max} = \max_{0 \le l \le m \le i} \sum_{k=l}^{m} a_k$$

- IA: $i = 0$:
  - In der Schleife wird $s = \text{aktSum} + a[i] = 0 + a[0] = a[0]$
  - Da s nicht größer als $\text{a[i]} = a[0]$ wird $\text{aktSum} = \text{a[i]} = \text{a[0]}$
  - 1. Fall: $\text{aktSum} > \text{max}$, dann wird $\text{max} = \text{aktSum}$
  - 2. Fall: aktSum ist kleinster int-Wert,
    dann hatte max den Wert schon bei Initialisierung
- IV: Behauptung gilt für i-1

MaxTeilSum3 (Korrektheit II)

- IS: $i-1 \to i$:
  - Nach IV gilt nach dem $(i-1)$.ten Schleifendurchlauf (Schleifeninvariante):

$$\text{aktSum} = \max_{0 \le l \le i-1} \sum_{k=l}^{i-1} a_k \text{ und } \max = \max_{0 \le l \le m \le i-1} \sum_{k=l}^{m} a_k$$

  - Es wird $s = \text{aktSum} + a[i]$
  - 1. Fall: $s > a[i]$, dann $\text{aktSum} = s$
  - 2. Fall: sonst, dann $\text{aktSum} = a[i]$

$$\Big\} \quad \text{aktSum} = \max_{0 \le l \le i} \sum_{k=l}^{i} a_k$$

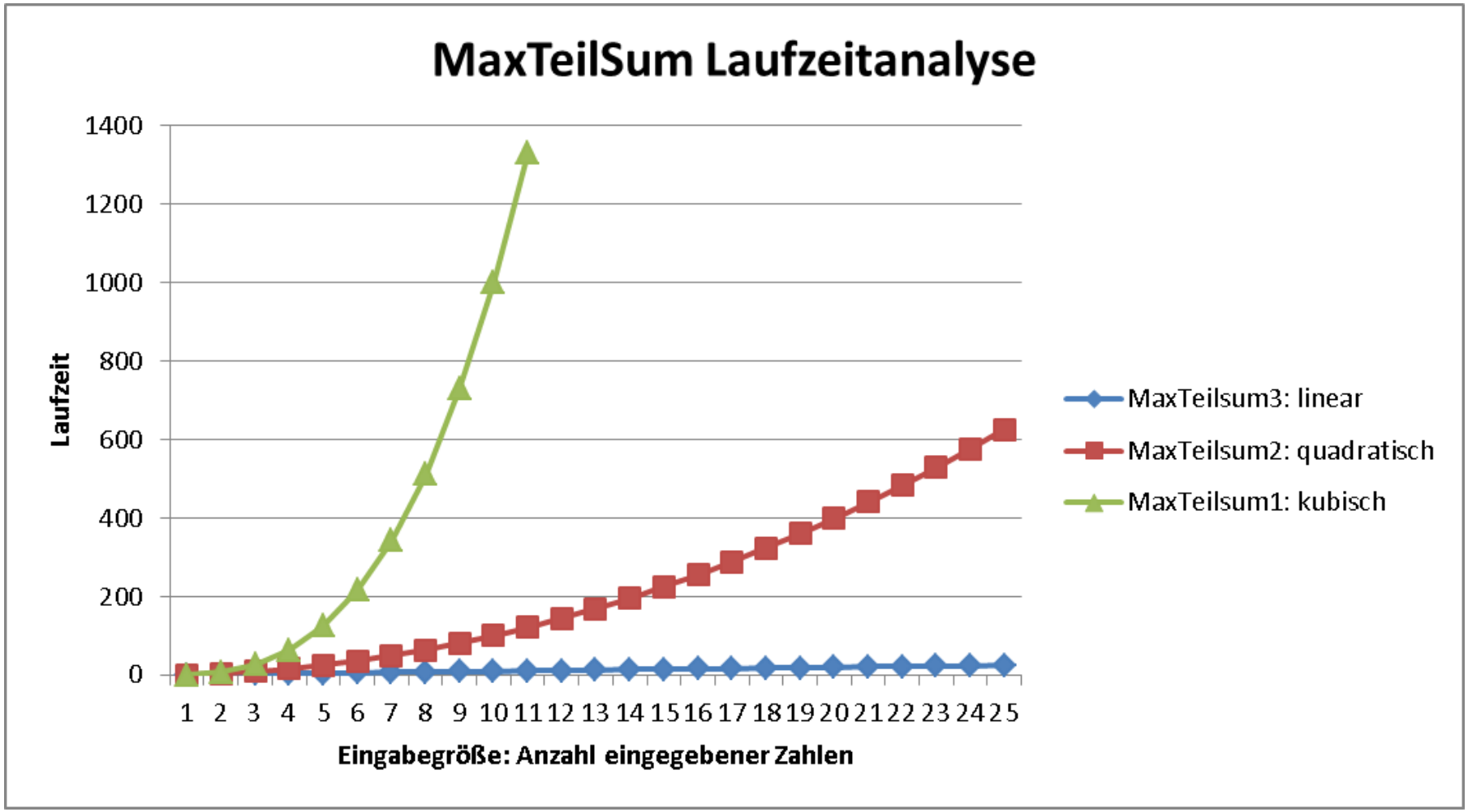
  - Falls $\text{aktSum} > \max$, dann $\max = \text{aktSum}$, d.h.

$$\max = \max \left\{ \max_{0 \le l \le m \le i-1} \sum_{k=l}^{m} a_k \, , \text{aktSum} \right\} = \max_{0 \le l \le m \le i} \sum_{k=l}^{m} a_k$$

  - Nach dem letzten Schleifendurchlauf gilt: $i = n-1$, d.h.

$$\max = \max_{0 \le l \le m \le i} \sum_{k=l}^{m} a_k = \max_{0 \le l \le m \le n-1} \sum_{k=l}^{m} a_k$$

Übersicht
Laufzeitverhalten MaxTeilSum



Komplexität von Algorithmen

- Ein Algorithmus A hat die Komplexität $O(f(n))$, wenn

$$T_A{}^{wc}(n) = O(f(n))$$

- Komplexität von nun an: Worst Case Laufzeitkomplexität
- Statt $T_A{}^{wc}(n)$ wird auch gerne einfach nur $T(n)$ verwendet
- Vorgehen
  - Bestimme $T(n)$ für den Algorithmus möglichst exakt (Bestimmung genauer Konstanten ist oft schwierig/aufwendig)
  - Schätze $T(n)$ mit Hilfe der O-Notation ab
- Allgemein gilt:
  - Je kleiner die obere Schranke, desto besser
  - Je größer die untere Schranke, desto besser

Übersicht Komplexitätsklassen

Menge	Laufzeit	Beispiele
$O(1)$	Konstant	Abfrage eines Wertes an einer bestimmten Stelle in einem Feld
$O(\log n)$	Logarithmisch	Binäre Suche in einem Feld (Halbierungsprinzip)
$O(n)$	Linear	Suche eines Wertes in einem unsortierten Feld, Fibonacci iterativ
$O(n \log n)$	Super-Linear	Sortieren mit guten Algorithmen (z.B. Mergesort)
$O(n^k)$	Quadratisch, Kubisch, … Allgemein: Polynomiell	Sortieren mit einfachen Algorithmen (z.B. Bubblesort)
$O(2^n)$	Exponentiell	Rekursive Variante zur Berechnung der Fibonacci-Folge, SAT
$O(n!)$	Faktoriell	Problem des Handlungsreisenden (Traveling Salesman Problem, TSP)