

Algorithmen und Datenstrukturen

Kapitel 1

Algorithmen & Algorithmenanalyse

Frank Heitmann
heitmann@informatik.uni-hamburg.de

14. Oktober 2015

Der Sprung ins Wasser...

Worum geht es bei

Algorithmen und Datenstrukturen

?

Und warum brauch ich das?!

Was nützt das?

- Spaß!
- Grob abschätzen, ob etwas überhaupt *möglich ist*.
- Grob abschätzen, wie *teuer* etwas wird/werden kann.
- Lösungsvorschläge und deren Kosten verstehen.
- Lösungen selbst erarbeiten können!
- *Abstrahieren!*

Organisatorisches 1

Zur Vorlesung:

- Immer Mittwochs in Hörsaal A Chemie
 - 0915 - 1045 Vorlesung mit Lecture2Go-Aufzeichnung
 - 1045 - 1100 Pause
 - 1100 - 1145 Wiederholung, Übung, Vertiefung, ...
- Boxen für besonders Wichtiges / für Interessierte

Das Buch zur Vorlesung:

- Cormen et al. 'Introduction to Algorithms', McGraw-Hill.
- Oder jedes andere Algorithmen-Buch (+Folien)

Organisatorisches 2

Zu den Übungsgruppen und Übungszetteln:

- Beginn diesen Mittwoch. Einige Besonderheiten:
 - Gruppe 01 und 04 am ZBH in Raum 16
 - Gruppe 03 startet erst um 1240 Uhr (bis 1410)
 - Gruppe 04 startet um 1400 Uhr (bis 1530)
- Neuen Aufgabenzettel alle zwei Wochen i.A. Mo-Mi
- Bearbeiten in 2er- oder 3er-Gruppen.
- Abgabe ca. zwei Wochen später am Montag bis 16 Uhr in die Abgabebox im 1. Stock von Haus C (vor C-201).
- Besprechung in den Übungen in der Woche.
- Alle Gruppenmitglieder müssen den eigenen Lösungsvorschlag präsentieren können.
- 50% der Punkte aller Zettel sind nötig.

Los ...

Begriff: Datentyp & Datenstruktur

Definition

Ein *Datentyp* ist eine Menge von Werten (z.B. \mathbb{N}) und Operationen darauf (z.B. $+$).

Definition

Bei einer *Datenstruktur* sind die Daten zusätzlich in bestimmter Weise angeordnet und in bestimmter Weise wird Zugriff und Verwaltung ermöglicht. (Beispiele: Array, Liste, Stack, Graph)

Bemerkung

Abstrakte Definition/Beschreibung/Darstellung erfolgt mittels Abstrakter Datentypen/Datenstrukturen (ADT). Dabei ist die *Signatur* die algebraische Spezifikation des Datentyps. Die *Algebra* wird als Datentyp zur Signatur bezeichnet.

Ein Problem...

Folgendes Problem: Gegeben sei eine (lange) Liste mit Namen. Wir wollen herausfinden ob Max Mustermann auf der Liste steht.

Wie machen wir das?

Wie machen wir das *algorithmisch*?

Und was ist noch mal ein *Algorithmus*?

Begriff: Algorithmus

Definition

Ein *Algorithmus* ist ein *endlich* und *präzise beschriebenes* Verfahren, das *Eingabewerte* in *Ausgabewerte* umwandelt. Es ist (i.A.) *deterministisch* und der Ausgang ist *determiniert*. Die einzelnen *Schritte* sind zudem *elementar/atomar* und *effektiv ausführbar*. Meist wird noch die *Termination* sowie die *Korrektheit* des Verfahrens verlangt (beides muss bewiesen werden!). I.A. soll ein Algorithmus ein Problem *lösen*. Eine *Instanz* ist dabei eine mögliche Eingabe (bspw. zwei Zahlen, die addiert werden sollen).

Bemerkung

Wir benutzen nachfolgend übliche Pseudocode-Bausteine und Konventionen. (for, while, Zuweisung, if-then, case, ...)

... eine Lösung

Algorithmus 1 Lineare Suche

```
1: for  $i = 0$  to  $n$  do  
2:   if  $a[i] == \text{max}$  mustermann then  
3:     return true  
4:   end if  
5: end for  
6: return false
```

Ist diese Lösung *gut*?

Geht das *besser*?

und eine andere Lösung

Algorithmus 2 Binäre Suche

```
1: while  $first \leq last \wedge idx < 0$  do  
2:    $m = first + ((last - first)/2)$   
3:   if  $a[m] < p$  then  
4:      $first = m + 1$   
5:   else if  $a[m] > p$  then  
6:      $last = m - 1$   
7:   else  
8:      $idx = m$   
9:   end if  
10: end while  
11: return  $idx$ 
```

Ist das besser?

Welche Voraussetzungen haben wir hier?

Sortieren

Definition (Das Sortierproblem)

Eingabe: Eine Sequenz $\langle a_1, a_2, \dots, a_n \rangle$ von n Zahlen.

Gesucht: Eine Permutation $\langle a'_1, a'_2, \dots, a'_n \rangle$ der Eingabesequenz mit $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Anmerkung

Ob die Reihenfolge zweier Elemente a_i, a_j ($i < j$) mit $a_i = a_j$ beibehalten werden soll oder nicht, ist nicht gesagt. Bleibt sie erhalten, d.h. ist $a'_p = a_i$, $a'_q = a_j$ und $p < q$, so nennt man das Verfahren *stabil*.

Ferner heißt ein Sortierverfahren *in-place* (oder *in situ*), wenn der zusätzlich benötigte Speicherbedarf unabhängig von der gegebenen Sequenz ist.

Bedeutung des Sortierens

Daten zu Sortieren ist ein fundamentales Problem:

- Manchmal ist Sortieren das Wesen der Anwendung
- Sortieren als Vorstufe (z.B. für das Suchen)
- Sortieren als Unterroutine (z.B. Painter's Algorithm)
- Viele verschiedene (Algorithmen-)Techniken
- Beweisbarkeit einer nichttrivialen unteren Schranke
- Sortiert wird ständig:
 - Musikliste nach Künstlern oder Titeln
 - Ankommende Pakete über die Datenleitung
 - ...

Pause to Ponder...

Wem fällt ein *Brute-Force-Algorithmus* zum Sortieren ein?

Sortieren mit Maximumbestimmung (MaxSort)

Algorithmus 3 Sortieren mit Max

```
1: for  $i = n$  downto 1 do  
2:    $idx = \max(A)$   
3:    $B[i] = A[idx]$   
4:    $A[idx] = 0$   
5: end for  
6: return  $B$ 
```

Bestimmung des Maximums

Algorithmus 4 Find Maximum

```
1:  $max = 1$ 
2: for  $i = 2$  to  $n$  do
3:   if  $a[i] > a[max]$  then
4:      $max = i$ 
5:   end if
6: end for
7: return  $max$ ;
```

InsertionSort: Die Idee

Beim *Sortieren durch Einfügen* (InsertionSort) wird ähnlich wie beim Spielkarten sortieren vorgegangen:

- Starte mit der leeren linken Hand
 - Nimm eine Karte und füge sie an der richtigen Position in der linken Hand ein. Dazu
 - Vergleiche diese neue Karte von rechts nach links mit den Karten, die schon auf der linken Hand sind.
 - Sobald eine kleinere Karte erreicht wird, füge ein.
- ⇒ Zu jedem Zeitpunkt sind die Karten auf der linken Hand sortiert.
- ⇒ Die Karten auf der linken Hand sind jeweils die obersten Karten des Haufens.

InsertionSort: Der Algorithmus

Algorithmus 5 InsertionSort($A[1 \dots n]$)

```
1: for  $j = 2$  to  $n$  do  
2:    $key = A[j]$   
3:    $i = j - 1$   
4:   while  $i > 0$  und  $A[i] > key$  do  
5:      $A[i + 1] = A[i]$   
6:      $i = i - 1$   
7:   end while  
8:    $A[i + 1] = key$   
9: end for
```

Zusammenfassung / Diskussion

Wir kennen jetzt

- Lineares Suchen und binäres Suchen
- MaxSort und InsertionSort

Aber:

- Ist ein Verfahren *besser* als ein anderes?
- Was *messen* wir?

Algorithmenanalyse

Wir behandeln nachfolgend *Zeit-* und *Platzkomplexität*. Eine (elementare) Anweisung zählt dabei als eine Zeiteinheit. Eine benutzte (elementare) Variable als eine Platzeinheit.

Zeit- und Platzbedarf wird *abhängig von der Länge n der Eingabe* gezählt. Wir arbeiten hierfür mit Funktionen $f : \mathbb{N} \rightarrow \mathbb{N}$ bzw. $f : \mathbb{N} \rightarrow \mathbb{R}$.

Algorithmenanalyse...

Algorithmus 6 Lineare Suche

```
1: for  $i = 0$  to  $n$  do
2:   if  $a[i] == \text{max mustermann}$  then
3:     return true
4:   end if
5: end for
6: return false
```

Wir behandeln nachfolgend *Zeit-* und *Platzkomplexität* (im uniformen Maß). Eine (elementare) Anweisung zählt dabei als eine Zeiteinheit. Eine benutzte (elementare) Variable als eine Platzeinheit.

Zeit- und Platzbedarf wird *abhängig von der Länge n der Eingabe* gezählt. Wir arbeiten hierfür mit Funktionen $f : \mathbb{N} \rightarrow \mathbb{R}$.

O-Notation - Motivation

Wir werden nachfolgend *konstante Faktoren* ignorieren. Diese 'verschwinden' in der *O*-Notation (auch *Landau*-Notation).

- ⇒ Konzentration auf das wesentliche.
- ⇒ Abstrahiert von verschiedenen Rechnerarchitekturen.
- ⇒ Guter Vergleich von Algorithmen möglich (praxisbewährt).
- ⇒ Ferner werden wir 'anfängliche Schwankungen' ignorieren können.
- ABER: $5 \cdot n$ und $5000 \cdot n$ wird als 'im Prinzip' gleich angesehen werden!

Beware ...

Anmerkung

Der Sinn der O -Notation zur Analyse der Laufzeit und des Speicherbedarfs von Algorithmen wird später klarer. Gleich folgen erstmal Definitionen und Beispiele dazu, bevor die Anwendung auf Algorithmen folgt.

O-Notation - Definition

Definition (O-Notation (und Verwandte))

- $O(g(n)) = \{f \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)|\}$
- $\Omega(g(n)) = \{f \mid \exists c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)|\}$
- $o(g(n)) = \{f \mid \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \leq c \cdot |g(n)|\}$
- $\omega(g(n)) = \{f \mid \forall c \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : |f(n)| \geq c \cdot |g(n)|\}$
- $\Theta(g(n)) = \{f \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c_1 \cdot |g(n)| \leq |f(n)| \leq c_2 \cdot |g(n)|\}$

Bemerkung

f ist dabei stets eine Funktion von \mathbb{N} nach \mathbb{R} , also $f : \mathbb{N} \rightarrow \mathbb{R}$. (Daher überall die **Betragsstriche!**) In der Literatur gibt es hier Varianten. Bei der tatsächlichen Algorithmenanalyse machen diese aber kaum einen Unterschied.

Wichtige Funktionen

Bemerkung

Wichtige Funktionen in der Algorithmik/Algorithmenanalyse:

- Konstante Funktionen
- Logarithmen (\log , \ln)
- Wurzelfunktionen (\sqrt{n})
- Polynome (beachte: $\sqrt{n} = n^{1/2}$)
- Exponentialfunktionen (2^n)
- ... Kombinationen davon

In welchem Zusammenhang stehen diese bzgl. der O -Notation?
(\Rightarrow Hausaufgabe!)

Zum Logarithmus

Bemerkung

Mit \log werden wir stets eine Variante des Logarithmus zur Basis 2 meinen, nämlich:

$$\log(n) := \begin{cases} 1 & , \text{ falls } n \leq 1 \\ \lfloor \log_2(n) \rfloor + 1 & , \text{ sonst.} \end{cases}$$

Damit ist $\log(n)$ die Länge der Binärdarstellung einer natürlichen Zahl n . (Die Anzahl von Speicherstellen/Schritten ist stets eine natürliche Zahl!) Andere Logarithmen werden nur selten benötigt und sind bis auf einen konstanten Faktor ohnehin gleich. Die Rechengesetze wie z.B. $\log(x \cdot y) = \log(x) + \log(y)$ und $\log(x^r) = r \cdot \log(x)$ sind oft hilfreich.

O-Notation - Beispiele I

Definition (Θ - Wiederholung)

$$\Theta(g(n)) = \{f \mid \exists c_1, c_2 \in \mathbb{R}^+ \exists n_0 \in \mathbb{N} \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

Beispiel

Wir wollen $\frac{1}{2}n^2 - 3n \in \Theta(n^2)$ zeigen. Wir suchen also Konstanten c_1, c_2, n_0 , so dass

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

für alle $n \geq n_0$ erfüllt ist. Teilen durch n^2 führt zu

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

. Dies kann man z.B. mit $c_1 = \frac{2}{8}, c_2 = 1, n_0 = 24$ erfüllen.

O-Notation - Variante

Satz

$$\textcircled{1} \quad f(n) \in O(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

$$\textcircled{2} \quad f(n) \in o(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$\textcircled{3} \quad f(n) \in \Omega(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} > 0$$

$$\textcircled{4} \quad f(n) \in \omega(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

Bemerkung

Mit obigen Äquivalenzen und mit Kenntnissen der Limesbildung sind einige der unten folgenden Sätze schnell zu zeigen (und evtl. schneller als mit der obigen, ursprünglichen Definition).

O-Notation - Beispiele II

Satz / Definition

$$f(n) \in O(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} < \infty$$

Beispiel

Wir wollen $\frac{1}{2}n^2 - 3n \in O(n^2)$ zeigen.

$$\lim_{n \rightarrow \infty} \frac{1/2n^2 - 3n}{n^2} = \lim_{n \rightarrow \infty} \frac{1}{2} - \frac{3}{n} = \frac{1}{2}$$

Ähnlich zeigt man $\frac{1}{2}n^2 - 3n \in \Omega(n^2)$ woraus das gleiche Ergebnis wie oben folgt, wie wir gleich sehen werden.

Merkhilfe

Bemerkung

Eine kleine *Merkhilfe* (nicht mehr!)

- $f \in O(g) \approx f \leq g$
- $f \in \Omega(g) \approx f \geq g$
- $f \in \Theta(g) \approx f = g$
- $f \in o(g) \approx f < g$
- $f \in \omega(g) \approx f > g$

Wir sagen, dass f *asymptotisch kleiner gleich*, *größer gleich*, *gleich*, *kleiner bzw. größer ist als* g , wenn $f \in O(g)$, $f \in \Omega(g)$, $f \in \Theta(g)$, $f \in o(g)$ bzw. $f \in \omega(g)$ gilt.

Wichtige Eigenschaften

Satz

- 1 $f \in X(g)$ und $g \in X(h)$ impliziert $f \in X(h)$
($X \in \{O, \Omega, \Theta, o, \omega\}$)
- 2 $f \in X(f)$ ($X \in \{O, \Omega, \Theta\}$)
- 3 $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$

Beweis (für 1. und $X = O$)

Sei $f \in O(g)$ und $g \in O(h)$, dann gibt es c, n_0 , so dass $f(n) \leq c \cdot g(n) \forall n \geq n_0$ und ebenso c', n'_0 , so dass $g(n') \leq c' \cdot h(n') \forall n' \geq n'_0$. Sei nun $n''_0 = \max(n_0, n'_0)$. Dann gelten $\forall n'' \geq n''_0$ beide Ungleichungen und für diese n'' gilt dann $f(n'') \leq c \cdot g(n'') \leq c \cdot c' \cdot h(n'')$ also $f(n'') \leq c \cdot c' \cdot h(n'')$ und da $c \cdot c'$ eine Konstante ist, folgt $f \in O(h)$.

Weitere wichtige Eigenschaften

Satz

- 1 $g \in O(f) \Leftrightarrow f \in \Omega(g)$
- 2 $g \in o(f) \Leftrightarrow f \in \omega(g)$
- 3 $g \in \Theta(f) \Leftrightarrow g \in O(f) \cap \Omega(f)$
- 4 $o(f) \subseteq O(f)$
- 5 $\omega(f) \subseteq \Omega(f)$
- 6 $\omega(f) \cap o(f) = ?$
- 7 $\Omega(f) \cap O(f) = ?$

Beweis (für 4.)

Sei $g \in o(f)$, dann gilt $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$. Um $g \in O(f)$ zu zeigen, muss $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = c$ für eine Konstante c gezeigt werden. Mit $c = 0$ folgt dies sofort.

Noch mehr wichtige Eigenschaften

Satz

- 1 $f, g \in O(h) \Rightarrow f + g \in O(h)$
- 2 $f \in O(g), c \in \mathbb{R}^+ \Rightarrow c \cdot f \in O(g)$
- 3 $f \in O(h_1), g \in O(h_2) \Rightarrow f \cdot g \in O(h_1 \cdot h_2)$

Beweis (für 3.)

Sei $f \in O(h_1)$ und $g \in O(h_2)$, dann gibt es c, n_0 , so dass $f(n) \leq c \cdot h_1(n) \forall n \geq n_0$ und ebenso c', n'_0 , so dass $g(n') \leq c' \cdot h_2(n') \forall n' \geq n'_0$. Daraus folgt $f(n'') \cdot g(n'') \leq c \cdot c' \cdot h_1(n'') \cdot h_2(n'') \forall n'' \geq \max(n_0, n'_0)$, also $f \cdot g \in O(h_1 \cdot h_2)$.

Zur Übung

Zur Übung

- *Formaler* Nachweis der obigen Eigenschaften (übt das Formale).
- Sei g eine unserer 'üblichen Funktionen'. Man überlege sich für jedes $X \in \{O, \Omega, \Theta, o, \omega\}$ ein f mit $f \in X(g)$ (baut Intuition auf).

Algorithmus 1: Lineare Suche

Algorithmus 7 Algorithmus 1

```
1: for  $i = 0$  to  $n$  do
2:   if  $a[i] == \text{max mustermann}$  then
3:     return true
4:   end if
5: end for
6: return false
```

Analyse

Laufzeit ist linear in der Länge n des Arrays, d.h. in $O(n)$ oder genauer sogar in $\Theta(n)$. (*Korrektheit* ist noch zu zeigen. Dazu später mehr...)

Algorithmus 2: Fakultät

Algorithmus 8 $\text{fac}(n)$

```
1:  $res = 1$   
2: for  $i = 1$  to  $n$  do  
3:    $res = res \cdot i$   
4: end for  
5: return  $res$ 
```

Analyse

Die Laufzeit ist in $O(n)$. Aber Achtung! Dies ist **exponentiell in der Größe der Eingabe!** Diese ist nämlich nur in $O(\log n)$.

Vorgehen

Vorgehen, wenn wir einen Algorithmus analysieren:

- 1 Überlegen bzgl. welcher *Kenngroße* der Eingabe wir messen wollen. Diese kann sich von der Größe der Eingabe unterscheiden! (Beispiel: Anzahl Knoten eines Graphen).
 - 2 Laufzeit/Speicherbedarf bzgl. dieser Kenngroße ausdrücken.
 - 3 Nochmal überlegen, ob dies die Aussage des Ergebnisses verfälscht (so wie bei der Fakultätsberechnung eben).
- ⇒ I.A. sollte sich die eigentliche Eingabegröße leicht durch die Kenngroße ausdrücken lassen und der Unterschied sollte nicht zu groß sein.
- + Beim Graphen mit n Knoten ist die Adjazenzmatrix in $O(n^2)$.
 - Ist eine Zahl n die Eingabe, so ist die Eingabegröße in $O(\log n)$.
Eine Laufzeit von $O(n)$ wäre also exponentiell in der Eingabe!

Algorithmus 3: Bestimmung des Maximums

Algorithmus 9 Find Maximum

```
1:  $max = 1$ 
2: for  $i = 2$  to  $n$  do
3:   if  $a[i] > a[max]$  then
4:      $max = i$ 
5:   end if
6: end for
7: return  $max$ ;
```

Analyse

Laufzeit ist linear in der Länge n des Arrays, d.h. in $O(n)$ oder genauer sogar in $\Theta(n)$.

Algorithmus 4: MaxSort

Algorithmus 10 Sortieren mit Max

```
1: for  $i = n$  downto 1 do  
2:    $idx = \max(A)$   
3:    $B[i] = A[idx]$   
4:    $A[idx] = 0$   
5: end for  
6: return  $B$ 
```

Laufzeit ist in $O(n^2)$.

Algorithmus 5: InsertionSort

Algorithmus 11 InsertionSort($A[1 \dots n]$)

```
1: for  $j = 2$  to  $n$  do
2:    $key = A[j]$ 
3:    $i = j - 1$ 
4:   while  $i > 0$  und  $A[i] > key$  do
5:      $A[i + 1] = A[i]$ 
6:      $i = i - 1$ 
7:   end while
8:    $A[i + 1] = key$ 
9: end for
```

Laufzeit ist ebenfalls in $O(n^2)$.

Algorithmus 6: Binäre Suche

Algorithmus 12 Binäre Suche

```
1: while  $first \leq last \wedge idx < 0$  do  
2:    $m = first + ((last - first)/2)$   
3:   if  $a[m] < p$  then  
4:      $first = m + 1$   
5:   else if  $a[m] > p$  then  
6:      $last = m - 1$   
7:   else  
8:      $idx = m$   
9:   end if  
10: end while  
11: return  $idx$ 
```

Laufzeit ist in $O(\log n)$.

Algorithmus 7: Brute-Force-Algorithmen

Das Mengenpartitionsproblem

Gegeben sei eine Menge $S \subseteq \mathbb{N}$. Gesucht ist eine Menge $A \subseteq S$, so dass $\sum_{x \in A} x = \sum_{x \in \bar{A}} x$ gilt.

Algorithmus 13 Suchraum durchsuchen

```
1: for all  $A \subseteq S$  do  
2:   if  $\sum_{x \in A} x = \sum_{x \in \bar{A}} x$  then  
3:     return true  
4:   end if  
5: end for  
6: return false
```

Laufzeit ist in $O(2^{|S|})$.

Eine kleine Warnung zum Schluss

Wichtige Anmerkung

Die O -Notation 'verschluckt' Konstanten. Wenn die zu gross/klein sind, dann kann dies das Ergebnis verfälschen! In dem Fall ist dann eine genauere Analyse (ohne O -Notation) nötig. I.A. hat sich die O -Notation aber bewährt, weil Extreme wie $10^6 \cdot n$ und $10^{-10} \cdot 2^n$ in der Praxis kaum vorkommen.

Kurz: $O(2^n)$ ist nicht zwingend *immer* schlimmer als $O(n)$ aber auf lange Sicht (d.h. bei wachsenden Eingabelängen) auf jeden Fall und im Allgemeinen (und bei allem, was einem so i.A. in der Praxis begegnet) ist $O(2^n)$ eben doch schlimmer als $O(n)$.

Sequentielle Algorithmen - Zusammenfassung

Ein paar Merkgeln für die bisherigen Laufzeiten:

- Konstant - Selten. Es wird nicht mal die ganze Eingabe betrachtet! - Aber *Grundoperationen* kosten nur konstant viel!
- Logarithmisch - Bei wiederholten Halbierungen oder wenn man mit der Höhe eines Baumes arbeitet.
- Lineare Laufzeiten, wenn man sich jedes Element der Eingabe einmal (oder: eine konstante Anzahl von Malen) ansieht.
- Quadratisch - jedes Element mit jedem anderen vergleichen.
- Höhere Polynome - ?
- Exponentiell - wenn man jede Möglichkeit durchprobiert.

Zusammenfassung

- Begriffe: Datentyp, Datenstruktur, Algorithmus
- Zeit- und Platzkomplexität ('gute' Algorithmen), wobei wir das uniforme Kostenmaß nutzen (jeder (elementare) Schritt eine Zeiteinheit, jede (elementare) Variable eine Platzeinheit)
- Zwei Definitionen für die O -Notation (und Verwandte)
- Wichtige Eigenschaften der O -Notation
- Wichtige Funktionen und ihre Klassifizierung bzgl. der O -Notation
- Lineares Suchen und Binäres Suchen
- MaxSort und InsertionSort

Themen der Vorlesung

Thema 2

Wir untersuchen den *Zeit- und Platzbedarf von Algorithmen*, wobei wir das *uniforme Kostenmaß* nutzen (jeder (elementare) Schritt eine Zeiteinheit, jede (elementare) Variable eine Platzeinheit). Zeit- und Platzbedarf wird *abhängig von der Länge n der Eingabe* gezählt. Hierzu nutzen wir Funktionen $f : \mathbb{N} \rightarrow \mathbb{N}$, bzw. $f : \mathbb{N} \rightarrow \mathbb{R}$.

Thema 1 & 3

Weitere (Ober-)Themen der Vorlesung sind bekannte Algorithmen für Probleme lesen, verstehen und anwenden zu können, sowie neue Algorithmen entwerfen, deren Korrektheit beweisen und ihre Laufzeit analysieren zu können.

Ausblick: Eine andere Art von Algorithmen

Algorithmus 14 $fac(n)$

```
1: if  $n == 0$  then  
2:   return 1  
3: else  
4:   return  $n \cdot fac(n - 1)$   
5: end if
```

Dies ist ein *rekursiver* Algorithmus... unsere bisherigen Techniken sind hier bei der Analyse wenig hilfreich.

Ausblick

Nächstes Mal:

- Rekurrenzgleichungen
- Korrektheit von Algorithmen
- Dabei: Weiteres zu Suchen und Sortieren
- Datenstrukturen