Mathematische Grundlagen – Kurz & Gut[1]

[1] Frei nach Folien von Alex Schickedanz und David Veith

Hinweise

- Hier werden Grundlagen der Datenstrukturen-Vorlesung rekapituliert.
- Es ersetzt nicht die Leiden des jungen Informatikers im Selbststudium.
- Probiert die Verfahren mal aus.
- Es gibt Bücher, Skripte, Videos, ...
- Das Lernzentrum hat sich zum Lernen in der Gruppe bewährt. :)
- Wenn gar nichts geht, dann holt euch Hilfe!

Der Logarithmus

- Der Logarithmus beantwortet folgende Fragestellung:
$$2^x = 8 \quad \rightarrow \quad \text{2 hoch wie viel ist 8?}$$
- Wir schreiben dies als $\log_2(8)$. Die Antwort ist $x = 3$.
- Rechenregeln:
  - $\log_a(b \cdot c) = \log_a(b) + \log_a(c)$
  - $\log_a(\frac{b}{c}) = \log_a(b) - \log_a(c)$
  - $\log_a(b^c) = c \log_a(b)$
  - $\log_a(b) = \frac{\log_c(b)}{\log_c(a)}$
- Oft unterteilen wir schwere Probleme in kleinere Mengen, die wir dann effizienter lösen können wie bei Quick- oder Mergesort.
- Wenn wir eine Eingabe der Länge n immer weiter halbieren, dann können wir dies nur $\log_2(n)$ mal tun.
- $\log(n)$ wächst langsamer als jedes $n^{1/k}$ für beliebige k.

Summenformeln

Was sind Summenformeln?

Faulheit obsiegt: Niemand will $1 + 2 + 3 + 4 + 5 + \cdots + 100$ aufschreiben!

Schneller und schöner: $\sum_{i=1}^{100} i$

Es gilt übrigens

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

Beweis?

$$\begin{array}{rcl} 1 + 100 & = & 101 \\ 2 + 99 & = & 101 \\ \vdots & & \vdots \\ 100 + 1 & = & 101 \\ \hline 100 \cdot 101/2 & = & 5050 \end{array}$$

Ist das ein Beweis? **Nein!**

Induktion I

Wie prüfen wir, dass $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$ gilt?

Wir beweisen mit vollständiger Induktion.

Zunächst müsst ihr den **Induktionsanker** bzw. die **Induktionsbasis** zeigen.

Mit Hilfe der Induktionsvoraussetzung zeigt ihr durch den **Induktionsschritt** oder -**schluss**, dass eure Annahme auch für beliebiges n gilt.

Induktion II

Basisfall: Für $n = 1$ gilt:

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

Somit haben wir gezeigt, dass die Formel für $n = 1$ gilt.

Induktionsschritt: Dazu gehen wir von $n \to n+1$.
(Es geht auch von $n-1 \to n$).

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

Beobachtung: Wir haben den schon bekannten Teil abgespalten und mussten nur zeigen, dass der neue Summand die Formel zum gewünschten Ziel bringt.

Induktion III

Induktion kann auf vielen Fragen angewandt werden.

Im späteren Studiumsverlauf eine einfache und sehr wertvolle Methode.

Viele Algorithmen verhalten sich beim Hinzufügen eines weiteren Elementes zur Eingabe so wie unsere Summenformel.

Weitere Summenformeln

Es gibt noch viele weitere geschlossene Formeln für Summen. Beispiele:

$$\sum_{k=1}^{n} c = n \cdot c \qquad \text{(Summe über die Konstante } c\text{)}$$

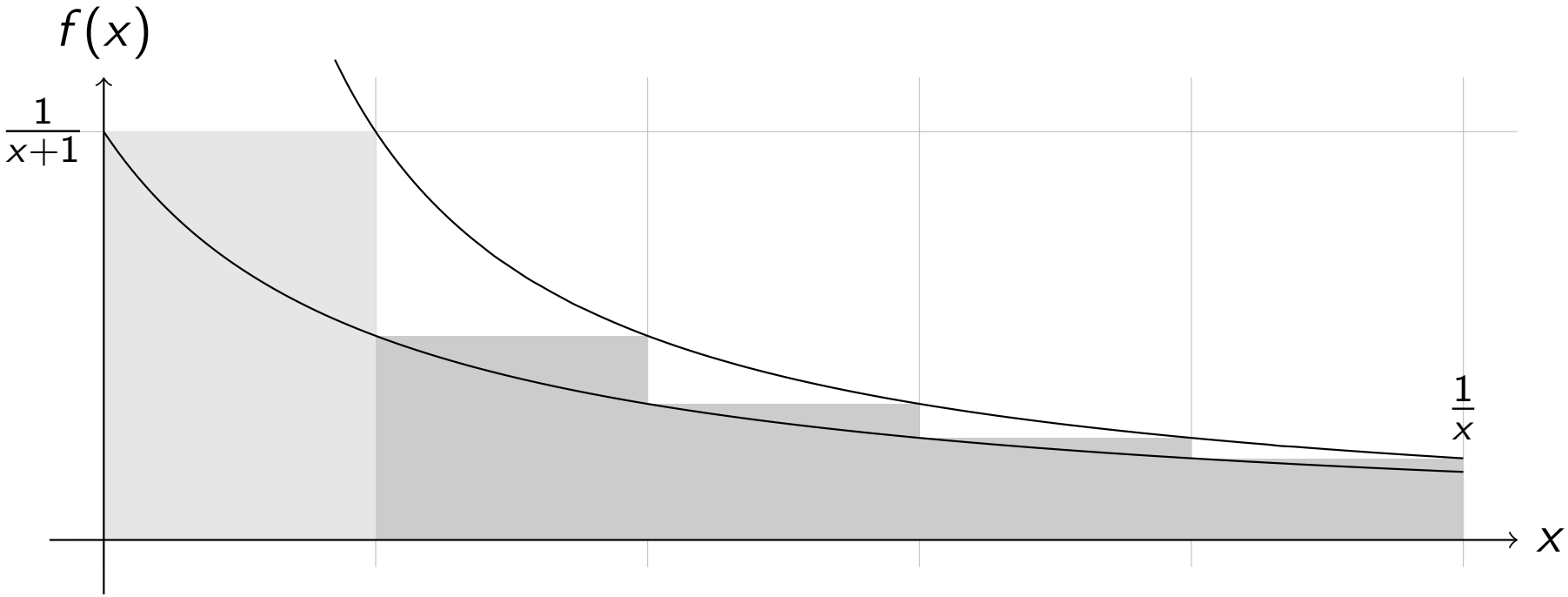
$$\sum_{k=1}^{n} 2k = n(n+1) \qquad \text{(Summe über gerade Zahlen)}$$

$$\sum_{k=1}^{n} k^2 = \frac{1}{6} n(n+1)(2n+1) \qquad \text{(Summe über Quadrate)}$$

$$\sum_{k=1}^{n} q^k = \frac{1-q^{n+1}}{1-q} \qquad \text{(Geometrische Summe)}$$

$$\sum_{k=1}^{n} \frac{1}{k} = H_n = \Theta(\log(n)) \qquad \text{(Harmonische Summe)}$$

Harmonische Summe



$$\sum_{k=1}^{n} \frac{1}{k} \leq 1 + \int_1^n \frac{1}{x} \mathrm{d}x = \log(n) + 1$$

$$\sum_{k=1}^{n} \frac{1}{k} \geq 1 + \int_1^n \frac{1}{x+1} \mathrm{d}x = \log(n+1) - \log(2) + 1$$

Die asymptotische Notation / Landau-Symbole

$f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ seien nicht-negative Funktionen.

- Die Groß-Oh Notation: $f = \mathcal{O}(g) \Leftrightarrow$ Es gibt eine positive Konstante $c > 0$ und eine natürliche Zahl $n_0 \in \mathbb{N}$, so dass

  $$f(n) \leq c \cdot g(n)$$

  für alle $n \geq n_0$ gilt: f wächst höchstens so schnell wie g.
- $f = \Omega(g) \Leftrightarrow g = \mathcal{O}(f)$: f wächst mindestens so schnell wie g.
- $f = \Theta(g) \Leftrightarrow f = \mathcal{O}(g)$ und $f = \Omega(g)$:
  f und g wachsen gleich schnell.
- Die Klein-Oh Notation: $f = o(g) \Leftrightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$:
  f wächst langsamer als g.
- $f = \omega(g) \Leftrightarrow \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$: f wächst schneller als g.

Grenzwerte

Grenzwerte sollten das Wachstum voraussagen (L'Hospital hilft)!

Der Grenzwert der Folge $\frac{f(n)}{g(n)}$ existiere und es sei $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$.

- Wenn $c = 0$, dann ist $f = o(g)$. (f wächst langsamer als g.)
- Wenn $0 \leq c < \infty$, dann ist $f = \mathcal{O}(g)$.
  (f wächst höchstens so schnell wie g.)
- Wenn $0 < c < \infty$, dann ist $f = \Theta(g)$.
  (f und g wachsen gleich schnell.)
- Wenn $0 < c \leq \infty$, dann ist $f = \Omega(g)$.
  (f wächst mindestens so schnell wie g.)
- Wenn $c = \infty$, dann ist $f = \omega(g)$. (f wächst schneller als g.)

Die Notation einprägsam aufgelistet

- $f = o(g)$ ist gleichbedeutend mit der Relation $f < g$.
- $f = \mathcal{O}(g)$ ist gleichbedeutend mit der Relation $f \leq g$.
- $f = \Theta(g)$ ist gleichbedeutend mit der Relation $f = g$.
- $f = \Omega(g)$ ist gleichbedeutend mit der Relation $f \geq g$.
- $f = \omega(g)$ ist gleichbedeutend mit der Relation $f > g$.

Achtung: Notationsfallen

- $5n = \mathcal{O}(n)$ und $5n = \mathcal{O}(n^2)$. Gilt dann $\mathcal{O}(n) = \mathcal{O}(n^2)$?
  $\mathcal{O}(n) = \mathcal{O}(n^2)$ ergibt nicht viel Sinn. Sinnvoller ist hier eine Interpretation als Mengen: $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$.
  So ergibt auch die Schreibweise $5n \in \mathcal{O}(n)$ Sinn.
- $\log_2(n) = \Theta(\log(n))$ und $\log_4(n) = \Theta(\log(n))$, aber $4^{\log_2(n)} \neq \Theta(4^{\log_4(n)})$.
  Landau-Symbole verstecken Konstanten, aber nicht im Exponenten.

Eine kleine Wachstums-Hierarchie

- konstanter Wert (kein Wachstum)
- $\log(\log(n))$
- $\log(n)$
- $n^{1/k}$
- n
- $n \cdot \log(n)$
- n^k mit $k > 1$
- a^n mit $a > 1$
- $n!$
- n^n

Landau-Symbole sind keine Laufzeit

Landau-Symbole reduzieren eine Funktion auf ihr asymptotisches Verhalten.
Die Funktion kann aber verschiedenes modellieren, z.B.

- die Laufzeit
- den Speicherplatz
- I/O-Zugriffe auf Festplatten
- einen komplizierten Teil einer Formel, z.B.
  $\ln(n!) = n\ln(n) - n + \mathcal{O}(\ln(n))$

Laufzeit

- Viele Informatiker möchten oder müssen programmieren.
- Qualität der Programme entscheidet über den Erfolg.
- Als Spezialisten sollten wir anderen Hilfestellung geben können...
- ... und die Qualität von Programmcode bewerten.
- Wir brauchen ein Modell!

Modell des Rechners

- Jeder Befehl soll eine Zeiteinheit verbrauchen.
- Jeder Speicheraufruf soll nur konstant viel Zeit benötigen.
- Wir kümmern uns nicht um die Sprache!
- Z.B. fast jeder C++ Befehl besteht aus einer Menge an Befehlen in Assembler.
- Wir zählen die Anzahl der Befehle in **Abhängigkeit der Eingabegröße**.
- Hat sich in der Praxis bewährt, um Code einzuordnen.
- Die gezählten Befehle ergeben eine Summe in Abhängigkeit von n.
- Wir verwenden die Landau-Notation um uns das Leben einfacher zu machen und Laufzeiten leichter vergleichen zu können.

Asymptotische Laufzeit

- Wir ermitteln die asymptotische Laufzeit durch Gernzwertbetrachtung.
- Durch unsere neue Notation spielen die Konstanten (theoretisch) keine Rolle mehr.

$$400n = \mathcal{O}(n) = 5n$$

- Wir betrachten nur noch dominierende Therme.

$$n^2 + n = \mathcal{O}(n^2)$$

- Dadurch können jedoch große Konstanten verborgen werden.

$$10^{80}n = \mathcal{O}(n)$$

Rekursion

- 1. Definition von Rekursion: siehe Rekursion.
- Eine Funktion $f(n)$ ruft sich zur Lösungsfindung selbst wieder auf mittels neuem Aufruf $f(n')$.
- Typischerweise tritt irgendwann eine Abbruchbedingung ein (z.B. nur noch ein Element).
- Uns interessiert hautpsächlich die Laufzeitbestimmung in dieser Vorlesung.

Laufzeitbestimmung bei Rekursion

- Liegt die Rekursion in der Form $T(n) = a \cdot T\left(\frac{n}{b}\right) + t(n)$ vor, kann das Mastertheorem angewandt werden.
- Sonst muss die Rekursion mittels Expansion und Vermutung gelöst werden.

Zusammenfassung: Das Mastertheorem

Die Rekursion

$$T(1) = c,\ T(n) = a \cdot T\left(\frac{n}{b}\right) + t(n)$$

sei zu lösen. n ist eine Potenz der Zahl $b > 1$ und $a \geq 1$, $c > 0$ gelte.

(a) Wenn $t(n) = \mathcal{O}\left(n^{(\log_b a)-\varepsilon}\right)$ für eine Konstante $\varepsilon > 0$, dann ist $T(n) = \Theta(n^{\log_b a})$.

(b) Wenn $t(n) = \Theta(n^{\log_b a})$, dann ist $T(n) = \Theta(n^{\log_b a} \cdot \log_b n)$.

(c) Wenn $t(n) = \Omega\left(n^{(\log_b a)+\varepsilon}\right)$ für eine Konstante $\varepsilon > 0$ und $a \cdot t\left(\frac{n}{b}\right) \leq \alpha t(n)$ für eine Konstante $\alpha < 1$, dann $T(n) = \Theta(t(n))$.

Hinweis: Das Mastertheorem ordnet die Rekursion einem exakten asymptotischen Wachstum zu (Θ). Wenn die Rekursion jedoch nur eine obere Schranke für die Laufzeit oder den Speicher eines Algorithmus modelliert, dann erhält man für diese keine Θ Zuordnung.

Mastertheorem - Kochrezept zum Ersten

- Rekursion der Form $T(n) = a \cdot T\left(\frac{n}{b}\right) + t(n)$
- Beispiel: $T(n) = 4 \cdot T\left(\frac{n}{2}\right) + n$
- **Schritt 1**: Identifizierung von a, b, $t(n)$
- $a = 4$, $b = 2$, $t(n) = n$
- **Schritt 2**: Bilde $\log_b(a) = \log_2(4) = 2$
- **Schritt 3**: Wettrennen zwischen $t(n)$ und $n^{\log_b(a)}$
- $\lim_{n\to\infty} \frac{t(n)}{n^{\log_b(a)-\varepsilon}} = \lim_{n\to\infty} \frac{n}{n^{\log_2(4)-\varepsilon}} = \lim_{n\to\infty} \frac{n}{n^{2-\varepsilon}} = 0$
- $\Rightarrow \quad t(n) = \mathcal{O}\left(n^{\log_b(a)-\varepsilon}\right)$ für $0 < \varepsilon < 1$
- $n^{\log_2(4)}$ gewinnt das Rennen und dominiert somit die Laufzeit.
- Dies ist der 1. Fall und somit $T(n) = \Theta(n^2)$

Mastertheorem - Kochrezept zum Zweiten

- Rekursion der Form $T(n) = a \cdot T\left(\frac{n}{b}\right) + t(n)$
- Beispiel: $T(n) = 16 \cdot T\left(\frac{n}{4}\right) + n^2$
- **Schritt 1**: Identifizierung von a, b, $t(n)$
- $a = 16$, $b = 4$, $t(n) = n$
- **Schritt 2**: Bilde $\log_b(a) = \log_4(16) = 2$
- **Schritt 3**: Wettrennen zwischen $t(n)$ und $n^{\log_b(a)}$
- $\lim_{n\to\infty} \frac{t(n)}{n^{\log_b(a)}} = \lim_{n\to\infty} \frac{n^2}{n^{\log_4(16)}} = \lim_{n\to\infty} \frac{n^2}{n^2} = 1$
- $\Rightarrow \quad t(n) = \Theta\left(n^{\log_b(a)}\right)$
- $n^{\log_4(16)}$ und n^2 kommen beide zeitgleich ins Ziel.
- Dies ist der 2. Fall und somit $T(n) = \Theta(n^2 \cdot \log_4(n))$

Mastertheorem - Kochrezept zum Dritten

- Rekursion der Form $T(n) = a \cdot T\left(\frac{n}{b}\right) + t(n)$
- Beispiel: $T(n) = 2 \cdot T\left(\frac{n}{4}\right) + n$
- **Schritt 1**: Identifizierung von a, b, $t(n)$
- $a = 2$, $b = 4$, $t(n) = n$
- **Schritt 2**: Bilde $\log_b(a) = \log_4(2) = 0.5$
- **Schritt 3**: Wettrennen zwischen $t(n)$ und $n^{\log_b(a)}$
- $\lim_{n\to\infty} \frac{t(n)}{n^{\log_b(a)+\varepsilon}} = \lim_{n\to\infty} \frac{n}{n^{\log_4(2)+\varepsilon}} = \lim_{n\to\infty} \frac{n}{n^{0.5+\varepsilon}} = \infty$
- $\Rightarrow \quad t(n) = \Omega\left(n^{\log_b(a)+\varepsilon}\right)$ für $0 < \varepsilon < 0.5$
- Und $a \cdot t(\frac{n}{b}) = \frac{1}{2}n \leq \alpha t(n)$ für $\frac{1}{2} \leq \alpha < 1$
- $t(n)$ gewinnt das Rennen und dominiert somit die Laufzeit.
- Dies ist der 3. Fall und somit $T(n) = \Theta(n)$

Mastertheorem - Abschluss

- Das Mastertheorem ist immer die Lösung nach Schema F und ein wenig Mathematik
- Die Welt könnte so schön sein, wären da nicht Rekursionsgleichungen wie $T(n) = T(n-4) + 3$.
- Was nun? Die -4 Ignorieren? Dann wäre dieser Fall gar nicht lösbar ($\log_1(n)$ ist nicht definiert).
- Wir müssen uns die Lösung anhand des Verhaltens der Gleichung herleiten.
- Nutze Expansion und stelle eine Vermutung an.

Rekursionsgleichung mit Expansion lösen

- Gegeben sei $T(n) = T(n-4) + 3$.
- Dann ist $T(n-4) = T((n-4)-4) + 3 = T(n-8) + 3$
- Erste Auflösung:

$$T(n) = (T(n-8) + 3) + 3 = T(n-8) + 2 \cdot 3$$

- Dies führe ich jetzt fort: $T(n) = T(n-12) + 3 \cdot 3$
- Das Muster: $T(n) = (n - 4k) + 3k$
- Wie oft muss ich das machen?
  Bis n den Basisfall erreicht, z.B. bei $n \leq 0$.
- Von n kann man $k = \frac{n}{4}$ mal 4 abziehen.
- Somit ist die Lösung $T(n) = T(0) + \frac{n}{4} \cdot 3 = \frac{3}{4}n$ falls $T(0) = 0$.
- Achtung: Der Basisfall könnte auch teuer sein. Für $T(0) = n^2$ erhält man $T(n) = n^2 + \frac{3}{4}n$.

Rekursionsgleichung mit Expansion lösen II

- Gegeben sei $T(n) = T(n-2) + n$.
- Dann ist $T(n-2) = T((n-2)-2) + (n-2)$
- Erste Auflösung:

$$T(n) = (T((n-2)-2) + (n-2)) + n = T(n-4) + n + (n-2)$$

- Der Wert von n verringert sich bei jedem Aufruf um 2.
- Also $n-2$, $n-4$, ...
- Muster: $T(n) = T(n-2k) + \sum_{i=0}^{k-1}(n-2i)$.
- Wie oft muss ich das machen? $k = \frac{n}{2}$ mal um $T(0)$ zu erreichen.
- Somit ergibt sich

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

- Diese Technik erfordert etwas mehr Übung.

Pseudocode I

Was ist eigentlich Pseudocode?

- Eine Abstraktion einer möglichen Implementierung.
- Aber nicht zu abstrakt! Wir wollen noch die Laufzeit anhand des Pseudocodes bestimmen können.
- Sie müssen dafür ein Gefühl entwickeln.

- Hier: C++ artiger Pseudocode
- Wir beschränken uns auf die ganz einfachen Sprachkonstrukte: Variablen, Datentyp (int, long, float, double, String), if-else, for, while, Funktionsaufrufe und manchmal Pointer.
- Pseudocode darf auch mathematische Ausdrücke enthalten, z.B. a = 2^{20}.

Hinweis: Mit dem LaTeX-Paket **algorithm2e** kann man schöne Pseudocodes schreiben.

Pseudocode II

Pseudocode sollte richtig eingerückt
und Klammern verwendet werden!

Falsch:

```
for(i = 1; i <= n; i++)
  {
    if(i % 2 == 0){
  if(i % 3 == 0)
    counter1++;
    counter2++;
}
 }
```

Besser:

```
for(i = 1; i <= n; i++)
{
  if(i % 2 == 0) {
    if(i % 3 == 0) {
      counter1++;
    }
    counter2++;
  }
}
```

Was fehlt hier noch?

Pseudocode III

Auch in Pseudocode sollten Variablen initialisiert werden!

Beispiel:

```
c = x;
while(c < n)
{
  c++;
}
```

- Für `x = 0` ist die Laufzeit ?$\Theta(n)$.
- Für `x = n` ist die Laufzeit ?$\Theta(1)$.

Verwende keine dirty tricks

Beispiele:

- `a+++++b**p` ist ein gültiger C++ Ausdruck, aber nicht sehr verständlich. Er wird übrigens als `((a++)++)+(b*(*p))` ausgewertet.
- `c = n % 2 == 1 ? 0 : 1;` und `c = (int) (n % 2 != 1);` entsprechen beide `if(n ungerade) { c = 0; } else { c = 1; }`

Pseudocode IV

Standarddatentypen können oft weggelassen, aber komplexere Datenstrukturen müssen angegeben werden

- Standarddatentypen: `int`, `float`, `bool`, `String`, `char`
- komplexere Datenstrukturen: Arrays, Stacks, Queues, Lists, Sets, etc.

Beispiele:

- `a = 1;` ist OK, da klar ist, dass `a` hier vom Typ `int` sein soll.
- `a; a.push(1);` ist nicht OK, da hier nicht klar ist, ob `a` ein Stack, eine Queue oder eine Liste ist. Deshalb:

  ```
  queue a;
  a.push(1);
  ```

  Dabei ist nicht wichtig, ob das eine selbst implementierte Queue ist oder die aus der C++ Standard Bibliothek, d.h. `std::queue` ist nicht nötig.

Pseudocode Analyse I

Wir geben die Laufzeit als Anzahl von Operationen an,
d.h. wir versuchen diese zu zählen oder abzuschätzen.

input : Eine Liste L von positiven Integern und eine Zahl x.
output: *true*, falls x in L ist, sonst *false*.

```
1 foreach a in L do
2 |   if a == x then
3 |   |   return true;
4 |   end
5 end
6 return false;
```

- Welche Operationen sollen hier gezählt werden?
- In jedem Schleifendurchlauf holen wir uns ein a aus der Liste und machen einen Vergleich.
  $\Rightarrow$ Insgesamt $T(n) < 2n + 1 = \mathcal{O}(n)$ Operationen.

Pseudocode Analyse II

```
1  Function search(sortedArray A, x, left = 0, right = —A—-1)
2      if right ¡ left then                              /* 1 Vergleich */
3          return false;
4      end
5      mid = left + ⌊(right − left)/2⌋; /* ein paar Operationen */
6      if A[mid] < x then                                /* 1 Vergleich */
7          return search(A, x, mid+1, right);  /* 1 rek. Aufrauf */
8      else if A[mid] > x then                           /* 1 Vergleich */
9          return search(A, x, left, mid-1);    /* 1 rek. Aufrauf */
10     else
11         return true;
12     end
13 end
```

Bei jedem Aufruf machen wir maximal konstant viele Operationen und maximal einen rekursiven Aufruf, wobei sich $right - left$ halbiert

$\Rightarrow$ Insgesamt $T(n) = T(\frac{n}{2}) + c = \mathcal{O}(\log(n))$ Operationen.

Datenstrukturen

- Eine Datenstuktur ist ein Container (siehe z.B. C++ Container Klassen), der ein Datum (=Einzahl von Daten) d erhält und dieses Datum in immer dem gleichen Muster verarbeitet.
- Wir erzeugen Wissen über die Anordnung von Daten, um die Effizienz unserer Datenstruktur zu erhöhen.
- Dazu erzwingen wir auf den Daten eine Ordnung (=Struktur), die uns schnelleren Zugriff und Veränderungen auf den Daten effizient ermöglichen soll.
- Die Datenstruktur stellt uns Operationen bereit.
- Achtung: Nicht jede Datenstruktur kennt jede Operation.
- Datenstukturen sind auf die jeweilige Aufgabe zugeschnitten und machen sich die Besonderheiten der Aufgabe zu Nutze.

Das Array

- Die Basisdatenstruktur schlechthin.
- Hat eine feste Größe und ist somit nicht dynamisch.
- Zugriff auf ein Element erfolgt sehr schnell über den Index.
- In der Informatik ist der erste Index typischerweise die Zahl 0.
- Kann (vereinfacht gesagt) nur die Operationen lesen und schreiben (effizient) ausführen.
- Wenn das Array sortiert ist, kann Suchen in $\mathcal{O}(\log(n))$ durchgeführt werden.

Einfach verkettete Listen

- Mehrere Daten sollen in einem Verbund gespeichert werden.
- Größe bekannt? Dann nutzen wir meist Arrays.
- Was aber, wenn die Größe nicht konstant ist?
- Oder wir wollen Elemente beliebig zwischen anderen Elementen einfügen, löschen, ...?
- Dafür sind Arrays nicht gut geeignet. Bei Löschen und Einfügen eines neuen Elements ist die Laufzeit $\mathcal{O}(n)$!
- Deshalb: Listen.

Einfach verkettete Listen

- Listen haben eine variable Größe.
- Wir können Elemente in konstanter Zeit einfügen.
- Wir können das aktuelle Element in konstanter Zeit löschen
- Leider dauert das Suchen lange. Schlimmstenfalls $\mathcal{O}(n)$ Zeit.

Queue

- Prinzip: FIFO - Warteschlange.
- Gut, um Elemente nacheinander abzuarbeiten.
- Implementierung beispielsweise über ein Array und zwei Pointer.
- Einfügen wird mit der Operation `enqueue(ele e)` aufgerufen.
- Ein Element wird mit der Operation `dequeue()` aus der Warteschlange entnommen, sofern diese nicht leer.
- In der Praxis gibt es oft die Möglichkeit erst nur zu schauen und dann zu entfernen.

Stack

- Prinzip: LIFO - Stapel auf dem Schreibtisch.
- Implementierung beispielsweise durch ein Array mit einem Pointer.
- Ein Element wird mit der Operation `push(ele x)` auf dem Stack abgelegt.
- Mit der Operation pop() wird das oberste Element vom Stack zurück gegeben.
- Zwei Stack können eine Queue effizient simulieren.
- Wird oft in Hardware (Programmstack, ...) verwendet.

War das etwa jetzt schon alles!?

- Das war gerade erst der Anfang, mehr bei:
- Algorithmen - Eine Einführung (ISBN: 3486590022)
- Mathematik für Informatiker: Ein praxisbezogenes Lehrbuch (ISBN: 3834818569)
- Python 3: Lernen und professionell anwenden (ISBN: 3826694562)
- Datenstrukturen Vorlesung vom Sommersemester 2017.
- Und viele weitere, z.B. in der Bibliothek.