Übungsserie 1
Datenstrukturen & Algorithmen

Universität Bern

Frühling 2018

Übungsserie 1

b
UNIVERSITÄT
BERN

- Aufgabe 1: Mathematische Grundlagen
- Aufgabe 2: Sortieralgorithmen, Pseudocode und Schleifeninvarianten zu Suchalgorithmen
- Aufgabe 3: Experimentelle Evaluation von Sortieralgorithmen

Aufgabe 1 (Grundlagen)

> Summen
> Produkte
> Induktionsbeweise

> Möglichst genau und strukturiert aufschreiben!
> Siehe Anhang A im Buch

Summen

> Notation

$$\sum_{j=1}^{n} a_j = a_1 + a_2 + \ldots + a_n$$

> Ausklammern

$$\sum_{j=1}^{n} (c \cdot a_j + b_j + d) = c \cdot \sum_{j=1}^{n} a_j + \sum_{j=1}^{n} b_j + n \cdot d$$

> Bsp:

$$\sum_{i=1}^{4} i^2 = 1 + 4 + 9 + 16$$

Produkte

> Notation

$$\Pi_{j=1}^{n} a_j = a_1 \cdot \ldots \cdot a_n$$

> Produkte

$$\Pi_{i=1}^{n} a_i b_i = (\Pi_{i=1}^{n} a_i) \cdot (\Pi_{i=1}^{n} b_i)$$

> Nicht vergessen: Produktregel

$$\Pi_{i=1}^{n} a^{b_i} = a^{b_1} a^{b_2} \cdots a^{b_n} = a^{b_1 + \ldots + b_n} = a^{\sum_{i=1}^{n} b_i}$$

Nützliche Formeln

- Arithmetische Reihe

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

- Geometrische Reihe

$$\sum_{k=1}^{n} pq^k = p\frac{q^{n+1}-1}{q-1}$$

Induktiv beweisen

- Ziel: Man will eine Aussage A für alle natürlichen Zahlen `n` beweisen.

- Bsp: Für alle natürlichen Zahlen `n` stimmt die Aussage

  `A(n)` = «es gibt mindestens n Primzahlen»

- Wie vorgehen?

Induktiv beweisen

- Aussagen einzeln testen:
  - A(1) «Es gibt mindestens 1 Primzahl» {2}
  - A(2) «Es gibt mindestens 2 Primzahlen» {2,3}
  - A(3) {2,3,5}
  - A(4) {2,3,5,7}
  - ...

Induktiv beweisen

- Idee: Angeben wie man für beliebige `n` die Aussage `A(n)` benutzt um die Aussage `A(n+1)` zu konstruieren (**Induktionsschritt**):

  `A(n) => A(n+1)`

- Dann genügt es zu zeigen, dass für ein kleines `n` (typisch `n=1`) die Aussage `A(n)` gilt (**Induktionsverankerung**). Alle anderen Aussagen folgen:

`A(1)`: Verankerung → `A(1) => A(2) => A(3) => A(4)...`

`A(n) => A(n+1)`

`A(n) => A(n+1)`

`A(n) => A(n+1)`

Beispiel 0

- `A(n)` = «es gibt mindestens `n` Primzahlen»
- **Induktionsverankerung**

  `A(1)`: «es gibt mindestens `eine` Primzahl» `{7}`

- **Induktionsschritt** Benutze `A(n)` um `A(n+1)` zu beweisen.

  `A(n)`: *Angenommen es gibt mindestens* `n` *Primzahlen.*
  Wähle die `n` kleinsten Primzahlen:

  $$p_1, \ldots, p_n$$

  $$p = p_1 \cdot \ldots \cdot p_n + 1$$

  Beachte: $p_1, \ldots, p_n$ teilen `p` nicht!

  Entweder `p` ist eine Primzahl => `A(n+1)` ist wahr

  Oder `p` ist keine Primzahl => `p` ist teilbar durch mindestens eine Primzahl, die nicht $p_1, \ldots, p_n$ ist => `A(n+1)` ist wahr

  => Es gibt mindestens `n+1` Primzahlen, `A(n+1)` ist wahr.

Induktiv Beweisen

> Struktur

1. Angabe der Aussage die zu beweisen ist
2. Beweis per Induktion:
   1. **Induktionsverankerung** Basisfall einsetzen
   2. **Induktionsschritt** Verwende Aussage `A(n)` für `A(n+1)`

> Typisch beim Induktionsschritt:

- Aussage `A(n+1)` aufschreiben
- Einen Teil abkapseln, der aussieht wie `A(n)`
- `A(n)` auf diesen Teil anwenden.

> Siehe auch folgende Beispiele

Beispiel 1

Zu zeigen: $\sum_{k=1}^{n} k = \frac{n\cdot(n+1)}{2}$
Die Aussage wird per Induktion über n bewiesen:
Verankerung ($n = 1$): Man kann die Aussage überprüfen, indem man $n = 1$ einsetzt:

$$\sum_{k=1}^{1} k = \frac{1 \cdot 2}{2}$$

$$1 = 1$$

Induktionsschritt: Angenommen, die Aussage stimmt für n, also

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

Dann beweisen wir sie für $n+1$:

$$\sum_{k=1}^{n+1} k = (n+1) + \sum_{k=1}^{n} k$$

Setzt man die Aussage für n ein, erhält man

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

was nach folgenden Umformungen der Aussage für $n+1$ entspricht.

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

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

Beispiel 2

Zu zeigen: sind $a_1, ..., a_n > 0 \in \mathbb{R}$, dann gilt $\sqrt[n]{a_1 \cdot a_2 \cdot \dots \cdot a_n} \leq \frac{a_1+a_2+...+a_n}{n}$
Um dies zu beweisen, zeigen wir folgendes Hilfsresultat.
Hilfsresultat: Gilt für reelle positive Zahlen $b_1 ... b_n$

$$\prod_{k=1}^{n} b_k = 1, \ so\ gilt \ \sum_{k=1}^{n} b_k \geq n$$

Das Hilfsresultat wird per Induktion über die Anzahl Faktoren n bewiesen.
Verankerung$(n = 1)$: Durch einsetzen ist die Aussage schnell überprüft:

$$\prod_{k=1}^{1} b_k = b_1$$

Induktionsschritt: Angenommen die Aussage stimmt für n, dann beweisen wir sie für $n+1$: Es gelte $\prod_{k=1}^{n+1} b_k = 1$, dann ist zu zeigen, dass $\sum_{k=1}^{n+1} b_k \geq n+1$. Anwendung der Aussage für n: die n Zahlen $(b_1 \cdot b_2)$, b_3, b_4, ..., b_{n+1} haben die Eigenschaft, dass ihr Produkt 1 ist und die Formel für n kann angewandt werden:

$$b_1 \cdot b_2 + \sum_{k=3}^{n+1} b_k \geq n$$

Um auf die Aussage für $n+1$ zu kommen machen wir folgende Umformungen: b_1 und b_2 können immer so gewählt werden, dass $b_1 \leq 1 \leq b_2$. Wir benutzen die Tatsche, dass falls $b_1 \leq 1 \leq b_2$ folgt, dass $(1-b_1) \cdot (b_2 - 1) \geq 0$ und damit $b_1 + b_2 > 1 + b_1 \cdot b_2$. Setzt man dies ein gilt:

$$b_1 \cdot b_2 + \sum_{k=3}^{n+1} b_k \geq n$$

$$b_1 \cdot b_2 + 1 + \sum_{k=3}^{n+1} b_k \geq n+1$$

$$b_1 + b_2 + \sum_{k=3}^{n+1} b_k \geq n+1$$

Benutzen des Hilfsresultats: Sei $G = \sqrt[n]{a_1 \cdot ... \cdot a_n}$. Setzt man $b_i = a_i/G$ gilt $b_1 \cdot ... \cdot b_n = 1$. Aus dem Hilfsresultat folgt

$$\sum_{i=1}^{n} b_i \geq n \Leftrightarrow \frac{1}{G} \sum_{i=1}^{n} a_i \geq n \Leftrightarrow \frac{\sum_{i=1}^{n} a_i}{n} \geq G$$

Varianten

> Induktion über mehrere Variablen
  — Z. B. `A(i,j)`
    – Verankerung z. B. `A(0,0)`
  — Verschiedene Induktionsschritte möglich
    – `A(i,0) => A(i+1,0) & A(i,j) => A(i,j+1)`
    – `A(0,j) => A(0,j+1) & A(i,j) => A(i+1,j)`
    – …

> Manchmal nützlich
  — Benutze dass `A(j)` wahr ist für alle `j = 0,…,i` statt nur `A(i)` wahr.

Achtung!

- **Induktionsverankerung** ist zentral!
  - Auch wenn trivial, nie weglassen
  - **Beweis nur gültig mit (korrekter) Verankerung**

- Sonst besteht die Welt nur aus Elefanten!

Behauptung: Wenn sich unter n Tieren ein Elefant befindet, dann sind alle diese Tiere Elefanten.
Beweis durch vollständige Induktion:
Verankerung: n=1 : Wenn von einem Tier eines ein Elefant ist, dann sind alle diese Tiere Elefanten.
Induktionsvoraussetzung: Die Behauptung sei richtig für alle natürlichen Zahlen kleiner oder gleich n.
Induktionsschritt: Sei unter n+1 Tieren eines ein Elefant. Wir stellen die Tiere so in eine Reihe, dass sich dieser Elefant unter den ersten n Tieren befindet. Nach Induktionsannahme sind dann alle diese ersten n Tiere Elefanten.
Damit befindet sich aber auch unter den letzten n Tieren ein Elefant, womit diese auch alle Elefanten sein müssen.
Also sind alle n+1 Tiere Elefanten.
Der Fehler?
Der Induktionsschritt funktioniert tatsächlich, aber nur für n>1, die Induktionsvoraussetzung war aber gezeigt für n=1
Im Fall n+1=2 kann man den Elefanten zwar so stellen, dass er bei den ersten n=1 Tieren steht. Folglich sind alle Tiere unter den ersten n=1 Tieren Elefanten. Aber deshalb befinden sich unter den "letzten" n Tieren nicht notwendig Elefanten.
Könnte man bei n=2 die Verankerung zeigen wäre der Beweis korrekt.

Aufgabe 2: Theoretische Aufgabe

UNIVERSITÄT BERN

> **Teilaufgaben 1 & 2** Insertion-Sort & Merge-Sort durchspielen.
  — Wie in der Vorlesung oder im Buch in Figure 2.14

> **Teilaufgaben 3 & 4** Pseudocode für lineares & binäres Suchen, Korrektheitsbeweis mittels Schleifeninvariante

Schleifeninvariante

- Ein generelles Konzept um Aussagen über einen Algorithmus zu beweisen

- Beschrieben im Buch in **Kapitel 2**
  - Am Beispiel Insertion-Sort

Schleifeninvariante

UNIVERSITÄT BERN

- Wie «beweist» man, dass ein Programm korrekt ist?
- Beispiel Primzahltest:

```
isPrime(n)
  j=2
  while j < n
    if n% j == 0
      break
    j= j+1

  return !(j < n)
```

Schleifeninvariante

> Schleifeninvariante spiegelt verhalten einer typischen Schleife wieder

```
isPrime(n)
  j=2
  while j < n
    if n% j == 0
      break
    j= j+1

  return !(j < n)
```

Schleife wird betreten, Variablen initialisiert

Einzelne Iteration

Terminierungs-bedingungen

„Definition“ Schleifeninvariante

> Schleifeninvariante spiegelt verhalten einer typischen Schleife wieder

```
isPrime(n)
  j=2
  while j < n
    if n% j == 0
      break
    j= j+1

  return !(j < n)
```

> Schleifeninvariante: Eigenschaft, die wahr ist
- Beim Betreten der Schleife (Initialisierung)
- Zu Beginn jeder Iteration (Fortsetzung)
- Beim Verlassen der Schleife (Terminierung)

„Definition“ Schleifeninvariante

UNIVERSITÄT BERN

> Eigenschaft = Eigenschaft des Programmzustandes
> Zustand eines Programms:
- — Tupel von Werten `(w_1,...,w_n)`

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j - 1].
4      i = j - 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i - 1
8      A[i + 1] = key
```

- — Bsp: *`(key,A,i,j)`*
- — *Beispiel Schleifeninvariante Loop 1-8 (siehe auch Buch)*

$$I(key, A, i, j) = A[1] < A[2] < .... < A[j-1]$$

Beispiel (Schleifeninvariante)

UNIVERSITÄT BERN

```
isPrime(n)
  j=2
  while j < n
    if n% j == 0
      break
    j= j+1

  return !(j < n)
```

- Zustand : `(j,n)`
- Gute Schleifeninvariante: *«Keine Zahl* `> 1, < j` *teilt* `n`, *bei Schleifenterminierung teilt* `j n`*»*

Beweis (Schleifen Invariante)

u^b UNIVERSITÄT BERN

- *«Keine Zahl* `> 1, < j` *teilt* `n`*, bei Schleifenterminierung teilt* `j` `n`*»*

```
isPrime(n)
  j=2
  while j < n
    if n% j == 0
      break
    j= j+1

  return !(j < n)
```

- Zu beweisen: Eigenschaft wahr
  - Beim Betreten der Schleife (Initialisierung)
  - Zu Beginn jeder Iteration (Fortsetzung)
  - Beim Verlassen der Schleife (Terminierung)

Beweis (Schleifen Invariante)

UNIVERSITÄT BERN

- *«Keine Zahl* `> 1, < j` *teilt* `n`*, bei Schleifenterminierung teilt* `j n`*»*

```
isPrime(n)
  j=2
  while j < n
    if n% j == 0
      break
    j= j+1

  return !(j < n)
```

- Wahr beim Betreten der Schleife (Initialisierung)
  - `j=2`: Keine natürliche Zahl `> 1, < 2` teilt `n` – es gibt keine.

Beweis (Schleifen Invariante)

UNIVERSITÄT BERN

- *«Keine Zahl $> 1, < j$ teilt n, bei Schleifenterminierung teilt j n»*

```
isPrime(n)
  j=2
  while j < n
    if n% j == 0
      break
    j= j+1

  return !(j < n)
```

- Zu Beginn jeder Iteration (Fortsetzung)
  - **Induktion:** Angenommen wahr zu Beginn der Iteration j.

    «Keine Zahl $> 1, < j$ teilt n»

    Wenn die Schleife nicht terminiert, wissen wir, dass j auch nicht n teilt. Also zu Beginn der Schleife $j+1$ gilt, «Keine Zahl $> 1, < j+1$ teilt n»

Beweis (Schleifen Invariante)

- *«Keine Zahl* `> 1, < j` *teilt* `n`*, bei Schleifenterminierung teilt* `j` *n»*

```
isPrime(n)
  j=2
  while j < n
    if n% j == 0
      break
    j= j+1

  return !(j < n)
```

- Wahr beim Verlassen der Schleife (Terminierung)
  - Zwei Möglichkeiten
    - **`break`** wurde erreicht. Zu Beginn der Iteration war die SI wahr. `j` wurde nicht verändert, also immer noch wahr
    - Die Iteration verlief normal, wir haben bereits gezeigt, dass die SI wahr bleibt.
  - In beiden Fällen teilt bei Terminierung `j` `n`.

Schleifeninvariante verwenden

- **Schleifeninvariante** *«Keine Zahl* `> 1, < j` *teilt* `n`*, bei Schleifenterminierung teilt* `j n`*»*

```
isPrime(n)
  j=2
  while j < n
    if n% j == 0
      break
    j= j+1

  return !(j < n)
```

- … und jetzt?
- Benutzen der SI
  - Nach der Schleife gilt Schleifeninvariante!
  - Also testet `!(j < n)` ob es keinen Teiler kleiner `n` gibt, also ob `n` eine Primzahl ist.

Schleifeninvarianten selber verwenden

- Typische Schritte:
  1. **Formulierung** *I(Zustand)* = *...*
  2. **Beweis** (Initialisierung, Fortsetzung)
  3. **Terminierung** Das Verlassen der Schleife untersuchen und Schleifeninvariante benutzen
- Beweis der Eigenschaften entspricht Induktion:
  - Initialisierung ~ Induktionsverankerung
  - Fortsetzung ~ Induktionsschritt

Aufgabe 2: Theoretische Aufgabe

UNIVERSITÄT BERN

> Tipp zu Teilaufgabe 3 (Suchproblem)
  - Einfacher Algorithmus, ~4-Zeilen Pseudocode

> Bemerkung zu Teilaufgabe 4 (binäre Suche)
  - **Iteratives** binäres Suchen

> Tipp zu Schleifeninvarianten
  - Schleifeninvarianten des Typs *„Das gesuchte Element ist/ist nicht im Bereich ...“* werden nützlich sein

Aufgabe 3 (Praktische Aufgabe)

UNIVERSITÄT BERN

- Experimentelle Evaluation der Laufzeiten von Merge-Sort und Insertion-Sort
  - Sortieralgorithmen sind bereits implementiert
    `Sorting.java` (Ilias)
  - Klasse zur Zeitmessung ist bereits implementiert
    `Timer.java` (Ilias)
  - Verwendungsbeispiel der vorgegebenen Klassen
    `SortTester.java` (Ilias)

Aufgabe 3 (Praktische Aufgabe)

- **Aufgabe** Arrays von Zufallszahlen erzeugen, beide Algorithmen aufrufen und Zeit messen.
  — 30'000 / 300’000 Zahlen

- Laufzeit von 10'000'000 Zahlen abschätzen:
  — Beispiel: Insertion-Sort $O(n^2)$, d.h. für die Laufzeit L gilt ungefähr

$$L = c \cdot n^2$$

  — also kann man c aus den gemessenen Laufzeiten l_i für n_i Elemente schätzen:

$$c \approx \frac{l_i}{n_i^2}$$

- Poolstunde im ExWi Pool
  - Montag 17:00 – 18:00
- Postet & beantwortet viele Fragen im Forum
- Arbeitet in Teams

?