0

Mein Compiler und ich.

Tutorium Zerohero

Florian Sihler ○ KW 43

Präsenzaufgabe

1 **Schau mal, ich bin schon groß!**

Entwickeln Sie einen Algorithmus, der als Eingabe eine Liste von Zahlen erhält. Anschließend soll er für jedes Element der Liste bestimmen, ob nach diesem Element noch eine größere Zahl in der Liste folgt.

Beispiel mit der Liste „1, 0, 7, 5, 2“:

1. **ja** (da $1 < 7$)
2. **ja** (da $0 < 7$)
3. **nein** (da $7 > \max\{5, 2\}$)
4. **nein** (da $5 > 2$)
5. **nein** (da letztes Element)

Welche Laufzeit hat Ihr Algorithmus im schlechtesten Fall?

Präsenzaufgabe - Lösung

Input : Eingabe als Liste von n Zahlen: $a_1, \ldots, a_n$

```
1  for i = 1 to n do
2      gefunden ← falsch;
3      for j = i+1 to n do // Wird für j > n nicht betreten.
4          if a_j > a_i then
5              gefunden ← wahr;
6              break;
7          end
8      end
9      if gefunden then Gebe aus: „ja“ else Gebe aus: „nein“ ;
10 end
```

Algorithmus 1 : Naiver Ansatz

Nach kleinem Gauß ergibt sich die Laufzeit:

$$c \cdot \frac{n(n+1)}{2} \in O(n^2).$$

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($O(n)$):

Input : Eingabe als Liste von n Zahlen: $a_1, \ldots, a_n$

```
1 Erstelle Liste an n Wörtern: w_1, ..., w_n;
2 max ← −∞;
3 for i = n to 1 do
4 |   if max > a_i then w_i ← „ja“ else w_i ← „nein“ ;
5 |   if a_i > max then max ← a_i;
6 end
7 for i = 1 to n do
8 |   Gebe aus: w_i;
9 end
```

Algorithmus 2 : Bezüglich der Laufzeit besserer Ansatz

How-To Pseudocode

- **Konsistent bleiben**
- Menschenlesbare Notation (meist Programmiersprachen-Mathe-Gemisch)
- Solange *klar* und *eindeutig*, frei gestaltbar
- Beispiel:

```
// @author Florian, the lone pengu
```

Input : Eingabe als Liste von n Pingus:  $_1, \ldots$  $_n$

```
   // Jeder Pinguin ist klasse!
1  for i = 1 to n do
2  |  if  _i ist klasse! then return _i ;
3  end
4  return  ; // Wenn kein klasse Pingu gefunden.
```

Algorithmus 3 : Einen „klasse“ Pinguin finden

Pseudocode: do's and dont's

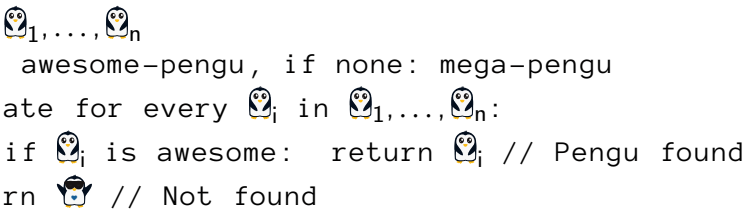
- *Do:* Gerne eine an Python oder C angelehnte Syntax verwenden.
- *Do:* mathematisch bleiben, also `n` $\in \mathbb{N}$, `n` $\in [0, \infty)$ oder „`Zeichenkette n`“.
- *Don't:* Spracheigene „syntactic-sugar“ Funktionen oder Definitionen verwenden. Also: *kein* `int`, `String` oder `double`.
- *Don't:* Einfach nur Java- oder C-Code.
- *Don't:* Zu allgemein Formulieren:

Input : Eingabe als Liste von `n` Pingus:  $_1, \ldots$  $_n$

```
1 sucheKlassePingu(1, ... n, );
```

Algorithmus 4 : Einen „klasse“ Pinguin finden

Pseudocode, Beispiele



Pseudocode, Beispiele II

```
Sei A die Liste an n Pingus, indexiert mit  .
Mache nun für jeden Pingu  aus A: [Durchsuche Pingus]
    Wenn  klasse ist:
            Gebe  zurück. [Pingu gefunden]
Wenn kein Pingu super war:
    Gebe Mega-Pingu (  ) zurück.
```

Übungsblatt 0 - Aufgabe 1

- In der Konsole: `java -version`:

```
openjdk version "11.0.17" 2022-10-18
OpenJDK Runtime Environment (build 11.0.17+8-alpine-r0)
OpenJDK 64-Bit Server VM (build 11.0.17+8-alpine-r0, mixed mode)
```

- Sowie: `javac -version`:

```
javac 11.0.17
```

Übungsblatt 0 - Aufgabe 2

- Tipp, tipp, tipp, …

```java
class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

- Sowie: `javac Hello.java`
- Und dann: `java Hello`:

```
Hello World!
```

Aussicht: Hornerschema, b zu Dezimal

- Am Beispiel von $1101_{(2)}$.

$$((1 \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1 = (3 \cdot 2 + 0) \cdot 2 + 1 = (6 \cdot 2) + 1 = 13_{(10)}.$$

- Tabellarisch:

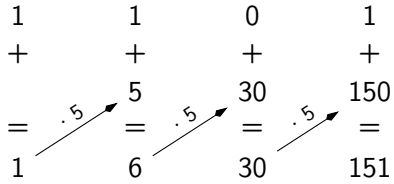
1		1		0		1
+		+		+		+
		2		6		12
=	$\cdot 2$ ↗	=	$\cdot 2$ ↗	=	$\cdot 2$ ↗	=
1		3		6		13

Aussicht: Hornerschema, b zu Dezimal (II)

- Andere Basis: $1101_{(5)}$.

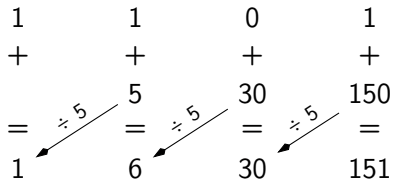
$$((1 \cdot 5 + 1) \cdot 5 + 0) \cdot 5 + 1 = (6 \cdot 5 + 0) \cdot 5 + 1 = (30 \cdot 5) + 1 = 151_{(10)}.$$

- Tabellarisch:



Aussicht: Hornerschema, Dezimal zu b

- 151_{10} zurück zur Basis 5.
- Tabellarisch:



Aussicht: Algorithmus, was ist das?

- *Eindeutige* Handlungsvorschrift zur Lösung eines Problems.
- Endlich viele, *wohldefinierte* (also nicht mehrdeutige) Einzelschritte.
- Mit der Zeit werden wir mehr Eigenschaften zur Charakterisierung betrachten.

Aussicht: Algorithmus, wie definiere ich das?

- Suche die größte ganze Zahl in einem beliebigen (ganzzahligen, nicht-leeren) Array.
- Information: Die Regeln zum Pseudocode gelten hier nach wie vor!
- Problemspezifikation:

  Ganzzahliges Array: Tupel der Größe $m \geq 1$ mit: $t = (t_1, t_2, \ldots, t_m) \in \mathbb{Z}^m$ ($t_i \in \mathbb{Z}$ für alle t_i)

  Beliebig: Die Elemente müssen nicht sortiert vorliegen.

  (Also gilt zum Beispiel nicht $t_1 \leq t_2 \leq \ldots \leq t_m$.)

  Größte: Die größte ganze Zahl $z \in t$ mit $z = \max(t)$.

  Ordentlich wäre rein mathematische/axiomatisch belegte Formalisierungen zu nutzen. In dieser Veranstaltung beschreiten wir einen Mittelweg.

Aussicht: Algorithmus, wie definiere ich das?

- Suche die größte ganze Zahl in einem beliebigen (ganzzahligen, nicht-leeren) Array.
- Problemabstraktion:

  Gegeben: Endliches unsortiertes Array t an ganzen Zahlen $n \in \mathbb{Z}$.
  Gesucht: Maximales ganzzahliges Element $x = \max(t)$.

  In diesem Fall ist die Abstaktion nahezu offensichtlich. Es dient der Veranschaulichung des Vorgehens 😄.

Aussicht: Algorithmus, wie definiere ich das?

- **Algorithmenentwurf:** (Annahme eines 0-indizierten Arrays mit Zugriff $a[i]$ für das i-te Element t_i.)
  1. Setze $max = a[0]$.
  2. Setze $i = 1$.
  3. Solange $i < m$:
     Wenn $(max < a[i])$:
     $max = a[i]$.
     Inkrementiere i um 1.
  4. Lösung ist max.
- **Korrektheitsnachweis:**

  Terminiert: Die Schleife aus 3. endet sicher, da i in jeder Iteration um 1 inkrementiert wird und damit streng monoton wächst, also sicher irgendwann $i < m$ mit $m \geq 1$ nicht mehr erfüllt.

  Partiell korrekt: Hier lässt sich leicht zeigen, dass für jeden Schritt in 3. gilt, dass $max \geq (t_1, \ldots, t_i)$. Für $m = 1$ ist weiter $max = a[0] = \max(t_0)$.

Aussicht: Algorithmus, wie definiere ich das?

- **Algorithmenentwurf:** (Annahme eines 0-indizierten Arrays mit Zugriff $a[i]$ für das i-te Element t_i.)
  1. Setze $max = a[0]$.
  2. Setze $i = 1$.
  3. Solange $i < m$:
     Wenn $(max < a[i])$:
     $max = a[i]$.
     Inkrementiere i um 1.
  4. Lösung ist max.
- **Aufwandsanalyse:** Wir machen dies als strukturierten Text:
  - 1. und 2. entsprechen jeweils einer Elementaroperation.
  - 3. wird genau $m - 1$ mal ausgeführt. Sie enthält sicher einen Vergleich und ein Inkrement. Eventuell eine Zuweisung. Für unseren Fall also drei Elementaroperationen.

Damit ist der Gesamtaufwand $1 + 1 + (m - 1) \cdot (3) = 2 + 3m - 3 = 3m - 1$.

Für komplexere Szenarien können sich die Analysen auch komplexer gestalten.