Mein Compiler und ich.

Tutorium Zerohero

Florian Sihler ◦ KW 43

Lithie

Präsenzaufgabe

1

Schau mal, ich bin schon groß!

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.

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“:

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 $1 < 7$)

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 $0 < 7$)

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 $7 > \max\{5, 2\}$)

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 $5 > 2$)

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)

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: $a_1, \ldots, a_n$

Präsenzaufgabe - Lösung

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

```
for i = 1 to n do
    gefunden ← falsch;
```

Präsenzaufgabe - Lösung

```
Input : Eingabe als Liste von n Zahlen: a1, ..., an
for i = 1 to n do
    gefunden ← falsch;
    for j = i+1 to n do // Wird für j > n nicht betreten.
```

Präsenzaufgabe - Lösung

```
Input : Eingabe als Liste von n Zahlen: a_1, ..., a_n
for i = 1 to n do
    gefunden ← falsch;
    for j = i+1 to n do // Wird für j > n nicht betreten.
        if a_j > a_i then
```

Präsenzaufgabe - Lösung

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

```
for i = 1 to n do
    gefunden ← falsch;
    for j = i+1 to n do // Wird für j > n nicht betreten.
        if a_j > a_i then
            gefunden ← wahr;
```

Präsenzaufgabe - Lösung

```
Input : Eingabe als Liste von n Zahlen: a_1, ..., a_n
for i = 1 to n do
    gefunden ← falsch;
    for j = i+1 to n do // Wird für j > n nicht betreten.
        if a_j > a_i then
            gefunden ← wahr;
            break;
        end
    end
```

Präsenzaufgabe - Lösung

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

```
for i = 1 to n do
    gefunden ← falsch;
    for j = i+1 to n do // Wird für j > n nicht betreten.
        if a_j > a_i then
            gefunden ← wahr;
            break;
        end
    end
    if gefunden then
```

Präsenzaufgabe - Lösung

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

```
for i = 1 to n do
    gefunden ← falsch;
    for j = i+1 to n do // Wird für j > n nicht betreten.
        if a_j > a_i then
            gefunden ← wahr;
            break;
        end
    end
    if gefunden then  Gebe aus: „ja“  else
```

Präsenzaufgabe - Lösung

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

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

Präsenzaufgabe - Lösung

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

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

Algorithmus 1 : Naiver Ansatz

Präsenzaufgabe - Lösung

```
Input : Eingabe als Liste von n Zahlen: a_1, ..., a_n
for i = 1 to n do
    gefunden ← falsch;
    for j = i+1 to n do // Wird für j > n nicht betreten.
        if a_j > a_i then
            gefunden ← wahr;
            break;
        end
    end
    if gefunden then  Gebe aus: „ja“  else  Gebe aus: „nein“ ;
end
```

Algorithmus 1 : Naiver Ansatz

Nach kleinem Gauß ergibt sich die Laufzeit:

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

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($\mathcal{O}(n)$):

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($\mathcal{O}(n)$):

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

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($\mathcal{O}(n)$):

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

Erstelle Liste an n Wörtern: $w_1, \dots, w_n$;

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($\mathcal{O}(n)$):

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

1 Erstelle Liste an n Wörtern: $w_1, \dots, w_n$;

2 $\max \leftarrow -\infty$;

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($\mathcal{O}(n)$):

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

Erstelle Liste an n Wörtern: $w_1, \ldots, w_n$;

max $\leftarrow -\infty$;

for $i = n$ **to** *1* **do**

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($\mathcal{O}(n)$):

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

Erstelle Liste an n Wörtern: $w_1, \ldots, w_n$;

$\mathrm{max} \leftarrow -\infty$;

for *i = n* **to** *1* **do**

  if $\mathrm{max} > a_i$ **then**

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($\mathcal{O}(n)$):

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

Erstelle Liste an n Wörtern: $w_1, \ldots, w_n$;

$\text{max} \leftarrow -\infty$;

for *i* = *n* **to** *1* **do**

  if $\text{max} > a_i$ **then** $w_i \leftarrow$ „ja“ **else**

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($\mathcal{O}(n)$):

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

Erstelle Liste an n Wörtern: $w_1, \ldots, w_n$;

$\max \leftarrow -\infty$;

for *i* = *n* **to** *1* **do**

  if $\max > a_i$ **then** $w_i \leftarrow$ „ja“ **else** $w_i \leftarrow$ „nein“ ;

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($\mathcal{O}(n)$):

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

Erstelle Liste an n Wörtern: $w_1, \ldots, w_n$;

$\max \leftarrow -\infty$;

for *i = n* **to** *1* **do**

  if $\max > a_i$ **then** $w_i \leftarrow$ „ja“ **else** $w_i \leftarrow$ „nein“ ;

  if $a_i > \max$ **then** $\max \leftarrow a_i$;

end

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($\mathcal{O}(n)$):

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

Erstelle Liste an n Wörtern: $w_1, \ldots, w_n$;
$\max \leftarrow -\infty$;
for *i* = *n* **to** *1* **do**
  if $\max > a_i$ **then** $w_i \leftarrow$ „ja“ **else** $w_i \leftarrow$ „nein“ ;
  if $a_i > \max$ **then** $\max \leftarrow a_i$;
end
for *i* = *1* **to** *n* **do**

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($\mathcal{O}(n)$):

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

```
Erstelle Liste an n Wörtern: w_1, ..., w_n;
max ← −∞;
for i = n to 1 do
    if max > a_i then w_i ← „ja“ else w_i ← „nein“ ;
    if a_i > max then max ← a_i;
end
for i = 1 to n do
    Gebe aus: w_i;
end
```

Präsenzaufgabe - Effizientere Lösung

Für eine optimierte/bessere Laufzeit ($\mathcal{O}(n)$):

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

Erstelle Liste an n Wörtern: $w_1, \dots, w_n$;
$\mathrm{max} \leftarrow -\infty$;
for $i = n$ **to** 1 **do**
    if $\mathrm{max} > a_i$ **then** $w_i \leftarrow$ „ja“ **else** $w_i \leftarrow$ „nein“ ;
    if $a_i > \mathrm{max}$ **then** $\mathrm{max} \leftarrow a_i$;
end
for $i = 1$ **to** n **do**
    Gebe aus: w_i;
end

Algorithmus 2 : Bezüglich der Laufzeit besserer Ansatz

How-To Pseudocode

How-To Pseudocode

- **Konsistent bleiben**

How-To Pseudocode

- **Konsistent bleiben**
- Menschenlesbare Notation (meist Programmiersprachen-Mathe-Gemisch)

How-To Pseudocode

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

How-To Pseudocode

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

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: $_1, \ldots$ $_n$

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!`

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!
```

for i = 1 **to** n **do**

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!`

for $i = 1$ **to** n **do**

  if $_i$ *ist klasse!* **then**

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, ..., 🐧_n
   // Jeder Pinguin ist klasse!
for i = 1 to n do
  |  if 🐧_i ist klasse! then  return 🐧_i ;
end
```

How-To Pseudocode

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

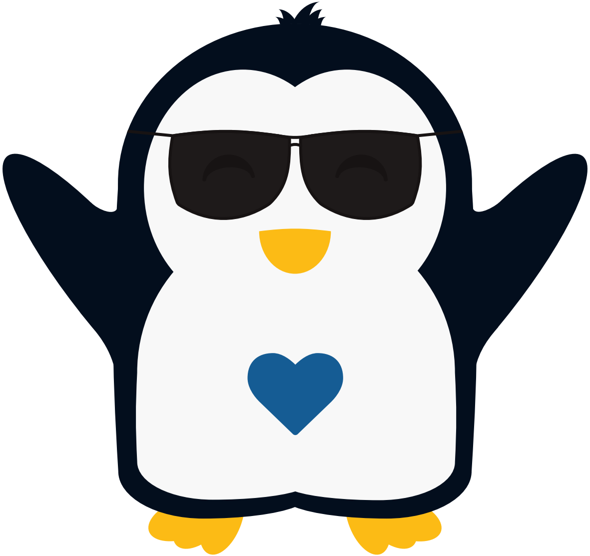
```
// @author Florian, the lone pengu
```

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

```
// Jeder Pinguin ist klasse!
```

for *i = 1* **to** *n* **do**
    if $_i$ *ist klasse!* **then** **return** $_i$;
end
return ; `// Wenn kein klasse Pingu gefunden.`

Algorithmus 3 : Einen „klasse“ Pinguin finden



Pseudocode: do's and dont's

Pseudocode: do's and dont's

- *Do:* Gerne eine an Python oder C angelehnte Syntax verwenden.

Pseudocode: do’s and dont’s

- *Do:* Gerne eine an Python oder C angelehnte Syntax verwenden.
- *Do:* mathematisch bleiben, also $\mathtt{n} \in \mathbb{N}$, $\mathtt{n} \in [0, \infty)$ oder „`Zeichenkette n`“.

Pseudocode: do's and dont's

- *Do:* Gerne eine an Python oder C angelehnte Syntax verwenden.
- *Do:* mathematisch bleiben, also $\texttt{n} \in \mathbb{N}$, $\texttt{n} \in [0, \infty)$ oder „`Zeichenkette n`“.
- *Don't:* Spracheigene „syntactic-sugar“ Funktionen oder Definitionen verwenden.

Pseudocode: do's and dont's

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

Pseudocode: do's and dont's

- *Do:* Gerne eine an Python oder C angelehnte Syntax verwenden.
- *Do:* mathematisch bleiben, also $\mathtt{n} \in \mathbb{N}$, $\mathtt{n} \in [0, \infty)$ 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.

Pseudocode: do's and dont's

- *Do*: Gerne eine an Python oder C angelehnte Syntax verwenden.
- *Do*: mathematisch bleiben, also $\mathtt{n} \in \mathbb{N}$, $\mathtt{n} \in [0, \infty)$ 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: $_1, \ldots$ $_n$

Pseudocode: do's and dont's

- *Do:* Gerne eine an Python oder C angelehnte Syntax verwenden.
- *Do:* mathematisch bleiben, also $\mathtt{n} \in \mathbb{N}$, $\mathtt{n} \in [0, \infty)$ 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 $\mathtt{n}$ Pingus: $_1, \ldots$ $_n$

sucheKlassePingu($_1, \ldots$ $_n$, );

Algorithmus 4 : Einen „klasse“ Pinguin finden

Pseudocode, Beispiele I

Pseudocode, Beispiele I

In: $_1, \ldots,$ $_n$

Out: awesome-pengu, if none: mega-pengu

iterate for every $_i$ in $_1, \ldots,$ $_n$:

    if $_i$ is awesome: return $_i$ // Pengu found

return  // Not found

Pseudocode, Beispiele II

Pseudocode, Beispiele II

Sei A die Liste an n Pingus, indexiert mit $_i$.

Mache nun für jeden Pingu $_i$ aus A: [Durchsuche Pingus]

    Wenn $_i$ klasse ist:

        Gebe $_i$ zurück. [Pingu gefunden]

Wenn kein Pingu super war:

    Gebe Mega-Pingu () zurück.

Übungsblatt 0 - Aufgabe 1

Übungsblatt 0 - Aufgabe 1

- In der Konsole: `java -version`:

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)
```

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`:

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

Übungsblatt 0 - Aufgabe 2

- Tipp, tipp, tipp, …

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

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`

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`:

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

Aussicht: Hornerschema, b zu Dezimal

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

Aussicht: Hornerschema, b zu Dezimal

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

$$((1 \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1$$

Aussicht: Hornerschema, b zu Dezimal

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

$$((1 \cdot 2 + 1) \cdot 2 + 0) \cdot 2 + 1 = (3 \cdot 2 + 0) \cdot 2 + 1$$

Aussicht: Hornerschema, b zu Dezimal

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

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

Aussicht: Hornerschema, b zu Dezimal

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

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

Aussicht: Hornerschema, b zu Dezimal

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

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

Aussicht: Hornerschema, b zu Dezimal

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

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

Aussicht: Hornerschema, b zu Dezimal

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

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

Aussicht: Hornerschema, b zu Dezimal

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

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

Aussicht: Hornerschema, b zu Dezimal

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

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

Aussicht: Hornerschema, b zu Dezimal

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

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

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:

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:

$$\begin{array}{c} 1 \\ + \\ \\ = \\ 1 \end{array}$$

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:

$$\begin{tikzcd}
1 & 1 \\
+ & + \\
 & 2 \\
= & = \\
1 \arrow[uur, "\cdot 2"] & 3
\end{tikzcd}$$

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:

$$\begin{array}{ccccc} 1 & & 1 & & 0 \\ + & & + & & + \\ & & 2 & & 6 \\ = & \overset{\cdot 2}{\nearrow} & = & \overset{\cdot 2}{\nearrow} & = \\ 1 & & 3 & & 6 \end{array}$$

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:

$$\begin{array}{cccc}
1 & 1 & 0 & 1 \\
+ & + & + & + \\
 & 2 & 6 & 12 \\
= & = & = & = \\
1 & 3 & 6 & 13
\end{array}$$

Aussicht: Hornerschema, b zu Dezimal (II)

Aussicht: Hornerschema, b zu Dezimal (II)

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

Aussicht: Hornerschema, b zu Dezimal (II)

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

$$((1 \cdot 5 + 1) \cdot 5 + 0) \cdot 5 + 1$$

Aussicht: Hornerschema, b zu Dezimal (II)

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

$$((1 \cdot 5 + 1) \cdot 5 + 0) \cdot 5 + 1 = (6 \cdot 5 + 0) \cdot 5 + 1$$

Aussicht: Hornerschema, b zu Dezimal (II)

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

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

Aussicht: Hornerschema, b zu Dezimal (II)

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

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

Aussicht: Hornerschema, b zu Dezimal (II)

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

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

Aussicht: Hornerschema, b zu Dezimal (II)

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

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

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, 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:

$$\begin{array}{c} 1 \\ + \\ \\ = \\ 1 \end{array}$$

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:

$$\begin{array}{ccc}
1 & & 1 \\
+ & & + \\
 & & 5 \\
= & \cdot 5 & = \\
1 & & 6
\end{array}$$

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:

$$\begin{tikzcd}[row sep=small]
1 & 1 & 0 \\
+ & + & + \\
 & 5 & 30 \\
= & = & = \\
1 \arrow[uur, "\cdot 5"] & 6 \arrow[uur, "\cdot 5"] & 30
\end{tikzcd}$$

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:

$$\begin{array}{ccccccc}
1 & & 1 & & 0 & & 1 \\
+ & & + & & + & & + \\
 & & 5 & & 30 & & 150 \\
= & \cdot 5 & = & \cdot 5 & = & \cdot 5 & = \\
1 & & 6 & & 30 & & 151
\end{array}$$

Aussicht: Hornerschema, Dezimal zu b

Aussicht: Hornerschema, Dezimal zu b

- 151_{10} zurück zur Basis 5.

Aussicht: Hornerschema, Dezimal zu b

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

Aussicht: Hornerschema, Dezimal zu b

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

$$\begin{array}{c} 1 \\ + \\ 150 \\ = \\ 151 \end{array}$$

Aussicht: Hornerschema, Dezimal zu b

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

$$
\begin{array}{ccc}
0 & & 1 \\
+ & & + \\
30 & & 150 \\
= & \swarrow_{\div 5} & = \\
30 & & 151
\end{array}
$$

Aussicht: Hornerschema, Dezimal zu b

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

$$
\begin{tikzcd}
1 & 0 & 1 \\
+ & + & + \\
5 & 30 \arrow[ddl, "\div 5"'] & 150 \arrow[ddl, "\div 5"'] \\
= & = & = \\
6 & 30 & 151
\end{tikzcd}
$$

Aussicht: Hornerschema, Dezimal zu b

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

$$
\begin{array}{ccccccc}
1 & & 1 & & 0 & & 1 \\
+ & & + & & + & & + \\
 & & 5 & & 30 & & 150 \\
= & \overset{\div 5}{\swarrow} & = & \overset{\div 5}{\swarrow} & = & \overset{\div 5}{\swarrow} & = \\
1 & & 6 & & 30 & & 151
\end{array}
$$

Aussicht: Algorithmus, was ist das?

Aussicht: Algorithmus, was ist das?

- *Eindeutige* Handlungsvorschrift zur Lösung eines Problems.

Aussicht: Algorithmus, was ist das?

- *Eindeutige* Handlungsvorschrift zur Lösung eines Problems.
- Endlich viele, *wohldefinierte* (also nicht mehrdeutige) Einzelschritte.

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?

Aussicht: Algorithmus, wie definiere ich das?

Vorbedingung $\longrightarrow$ $\boxed{\text{Algorithmus}}$ $\longrightarrow$ Nachbedingung

Aussicht: Algorithmus, wie definiere ich das?

Vorbedingung $\longrightarrow$ Algorithmus $\longrightarrow$ Nachbedingung

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

Aussicht: Algorithmus, wie definiere ich das?

$$\text{Vorbedingung} \longrightarrow \boxed{\text{Algorithmus}} \longrightarrow \text{Nachbedingung}$$

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

Aussicht: Algorithmus, wie definiere ich das?

Vorbedingung → Algorithmus → Nachbedingung

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

Aussicht: Algorithmus, wie definiere ich das?

$$\text{Vorbedingung} \longrightarrow \boxed{\text{Algorithmus}} \longrightarrow \text{Nachbedingung}$$

- 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)

Aussicht: Algorithmus, wie definiere ich das?

Vorbedingung $\longrightarrow$ **Algorithmus** $\longrightarrow$ Nachbedingung

- 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 $t_1 \leq t_2 \leq \ldots \leq t_m$.)

Aussicht: Algorithmus, wie definiere ich das?

$$\text{Vorbedingung} \longrightarrow \boxed{\text{Algorithmus}} \longrightarrow \text{Nachbedingung}$$

- 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 $z = \max(t)$.

Aussicht: Algorithmus, wie definiere ich das?

Vorbedingung $\longrightarrow$ Algorithmus $\longrightarrow$ Nachbedingung

- 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.

Aussicht: Algorithmus, wie definiere ich das?

Vorbedingung $\longrightarrow$ **Algorithmus** $\longrightarrow$ Nachbedingung

- 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?

Vorbedingung $\longrightarrow$ $\boxed{\text{Algorithmus}}$ $\longrightarrow$ Nachbedingung

Aussicht: Algorithmus, wie definiere ich das?

Vorbedingung → Algorithmus → Nachbedingung

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

Aussicht: Algorithmus, wie definiere ich das?

Vorbedingung → **Algorithmus** → Nachbedingung

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

Aussicht: Algorithmus, wie definiere ich das?

Vorbedingung $\longrightarrow$ $\boxed{\text{Algorithmus}}$ $\longrightarrow$ Nachbedingung

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

  Gegeben: Endliches unsortiertes Array t an ganzen Zahlen $n \in \mathbb{Z}$.

Aussicht: Algorithmus, wie definiere ich das?

Vorbedingung → Algorithmus → Nachbedingung

- 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 $x = \max(t)$.

Aussicht: Algorithmus, wie definiere ich das?

$$\text{Vorbedingung} \longrightarrow \boxed{\text{Algorithmus}} \longrightarrow \text{Nachbedingung}$$

- 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.

Aussicht: Algorithmus, wie definiere ich das?

Vorbedingung → **Algorithmus** → Nachbedingung

- 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?

Aussicht: Algorithmus, wie definiere ich das?

- **Algorithmenentwurf:** (Annahme eines 0-indizierten Arrays mit Zugriff `a[i]` für das i-te Element t_i.)

Aussicht: Algorithmus, wie definiere ich das?

- **Algorithmenentwurf:** (Annahme eines 0-indizierten Arrays mit Zugriff $\mathtt{a[i]}$ für das i-te Element t_i.)
  1. Setze $\mathtt{max} = \mathtt{a[0]}$.

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 $\mathsf{max} = \mathtt{a[0]}$.
  2. Setze $i = 1$.

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 $i < m$:

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 $\texttt{max} = \texttt{a[0]}$.
  2. Setze $\texttt{i} = 1$.
  3. Solange $\texttt{i} < m$:

     Wenn ($\texttt{max} < \texttt{a[i]}$):

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 $\mathtt{max} = \mathtt{a[0]}$.
  2. Setze $i = 1$.
  3. Solange $i < m$:

     Wenn ($\mathtt{max} < \mathtt{a[i]}$):

     $\mathtt{max} = \mathtt{a[i]}$.

Aussicht: Algorithmus, wie definiere ich das?

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

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 $\mathtt{max} = \mathtt{a[0]}$.
  2. Setze $i = 1$.
  3. Solange $i < m$:
     Wenn ($\mathtt{max} < \mathtt{a[i]}$):
        $\mathtt{max} = \mathtt{a[i]}$.
     Inkrementiere i um 1.
  4. Lösung ist $\mathtt{max}$.

Aussicht: Algorithmus, wie definiere ich das?

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

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,

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 $\mathtt{max} = \mathtt{a[0]}$.
  2. Setze $i = 1$.
  3. Solange $i < m$:
     Wenn ($\mathtt{max} < \mathtt{a[i]}$):
       $\mathtt{max} = \mathtt{a[i]}$.
     Inkrementiere i um 1.
  4. Lösung ist $\mathtt{max}$.
- **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.

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 $\max \geq (t_1, \ldots, t_i)$.

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 $\mathrm{max} \geq (t_1, \ldots, t_i)$. 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 $\mathtt{max} = \mathtt{a[0]}$.
  2. Setze $i = 1$.
  3. Solange $i < m$:
     Wenn ($\mathtt{max} < \mathtt{a[i]}$):
       $\mathtt{max} = \mathtt{a[i]}$.
     Inkrementiere i um 1.
  4. Lösung ist $\mathtt{max}$.

Aussicht: Algorithmus, wie definiere ich das?

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

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 $\mathsf{max} = \mathtt{a[0]}$.
  2. Setze $i = 1$.
  3. Solange $i < m$:
     Wenn ($\mathsf{max} < \mathtt{a[i]}$):
     $\mathsf{max} = \mathtt{a[i]}$.
     Inkrementiere i um 1.
  4. Lösung ist max.
- **Aufwandsanalyse:** Wir machen dies als strukturierten Text:

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.

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.

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 $\texttt{max} = \texttt{a[0]}$.
  2. Setze $i = 1$.
  3. Solange $i < m$:
     Wenn ($\texttt{max} < \texttt{a[i]}$):
     $\texttt{max} = \texttt{a[i]}$.
     Inkrementiere i um 1.
  4. Lösung ist $\texttt{max}$.
- **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.

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 $1 + 1 + (m - 1) \cdot (3) = 2 + 3m - 3 = 3m - 1$.

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 $\mathtt{max} = \mathtt{a[0]}$.
  2. Setze $i = 1$.
  3. Solange $i < m$:
     Wenn ($\mathtt{max} < \mathtt{a[i]}$):
       $\mathtt{max} = \mathtt{a[i]}$.
     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.