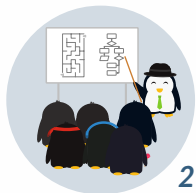


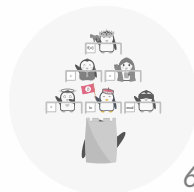
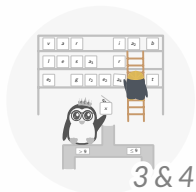
Mein Compiler und ich

Tutorium ZeroHero

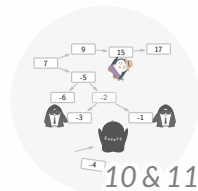
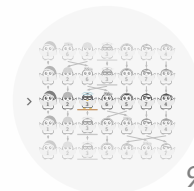
Theorie



Grundlagen



Vertiefungen



*„My dear Watson, try a little analysis yourself,” said he,
with a touch of impatience. „You know my methods.
Apply them, and it will be instructive to compare results.”
— Conan Doyle [Doy10, chp. VI]*

Präsenzaufgabe

1

Das habe ich schon gesehen!

Entwerfen Sie einen Algorithmus um herauszufinden, ob in einer Zeichenkette S das Zeichen a , aber nicht das Zeichen b vorkommt. Als Ergebnis soll der Algorithmus ausgeben, ob die Bedingung zutrifft („Ja“) oder nicht („Nein“). Ein paar Beispiele...

- | | | |
|--------------------------|----------|-------------------------|
| > $S = \text{ababababa}$ | → „Nein“ | (Enthält ein b) |
| > $S = \text{aaaaaaaaa}$ | → „Ja“ | (Hat a und kein b) |
| > $S = \text{cdefg}$ | → „Nein“ | (Enthält kein a) |
| > $S = \text{klmohna}$ | → „Ja“ | (Hat a und kein b) |

Führen Sie die Schritte der Algorithmenentwicklung durch:

- | | | |
|-------------------------|-------------------------|--------------------|
| a) Problemspezifikation | c) Algorithmenentwurf | e) Aufwandsanalyse |
| b) Problemabstraktion | d) Korrektheitsnachweis | |

Es gelte $i, j \in \{1, \dots, n\} \subseteq \mathbb{N}$

a) *Spezifikation: Definiere alle relevante Begriffe unmissverständlich*

- Eine „Zeichenkette“ S ist eine indizierte Liste (s_1, \dots, s_n) an Zeichen.
- Ein „Zeichen“ ist ein Symbol s_i aus einem Alphabet Σ .
- Mit „Zeichen x kommt in S vor“ meinen wir: $\exists i : s_i = x$

b) *Abstraktion: Definiere die Eingabe und die gewünschte Ausgabe*

Gegeben: Eine Zeichenkette $S = (s_1, \dots, s_n)$

Gesucht: Ausgabe „Ja“ genau dann, wenn $\exists i : s_i = a$ und $\forall j : s_j \neq b$, sonst „Nein“

Eingabe : Eingabe als Zeichenkette S mit Zeichen $S = (s_1, \dots, s_n)$

- 1 *enthältA* \leftarrow Nein;
- 2 *enthältB* \leftarrow Nein;
- 3 **für** $i \leftarrow 1$ **bis einschließlich** n **in Schritten von 1 tue:**
 - 4 **Wenn** s_i ist Buchstabe a **tue:** *enthältA* \leftarrow Ja;
 - 5 **Wenn** s_i ist Buchstabe b **tue:** *enthältB* \leftarrow Ja;
- 6 **Wenn** *enthältA* ist Ja aber *enthältB* ist Nein **tue:** Gebe aus: „Ja“;
- 7 **Sonst:** Gebe aus: „Nein“;

Algorithmus 1 : Ein „Zeichenkettenfilter“

d) *Verifikation: Zeige die totale Korrektheit (der Algorithmus terminiert und ist partiell Korrekt):*

Termination: Z1 und Z3–Z6 sind Elementaroperationen.
Diese Terminieren per Definition.

Die Schleife in Z2 terminiert, da i streng monoton ansteigt und damit in endlicher Zeit $i \leq n$ verletzt.

Partiell Korrekt: Wir merken uns mit *enthältA* und *enthältB*, ob der jeweilige Buchstabe gefunden wurde (Z2–Z4). Wir geben „Ja“ nur dann aus, wenn wir ein a, aber kein b finden (Z5). Sonst geben wir „Nein“ aus.

```
z1: hatA ← Nein,   hatB ← Nein;  
z2: für  $i \leftarrow 1$  bis  $n$  Schrittgröße 1:  
z3:   Wenn  $s_i = a$ : hatA ← Ja;  
z4:   Wenn  $s_i = b$ : hatB ← Ja;  
z5: Wenn hatA = Ja, hatB = Nein: „Ja“;  
z6: sonst: „Nein“;
```

Eine verkürzte Version

Verifikation – Eine kleine Vertiefung

- › Der Alternative, „weniger schwammige“ Beweis der partiellen Korrektheit:
 - Man überzeuge sich leicht, dass folgende Invariante für alle $i \in \{1, \dots, n\}$ gelte:

$$\text{enthält}A = \begin{cases} \text{Ja,} & \exists j \in \{1, \dots, i\} : s_j = a \\ \text{Nein,} & \text{sonst.} \end{cases} \quad \text{enthält}B = \begin{cases} \text{Ja,} & \exists j \in \{1, \dots, i\} : s_j = b \\ \text{Nein,} & \text{sonst.} \end{cases}$$

- Dies kann man beispielsweise mit vollständiger Induktion zeigen.
- Anschließend folgt der Beweis durch Abgleich der Bedingungen.

Invarianten mögen vielleicht gruselig klingen, sind aber nur Aussagen, die zum Beispiel im Rahmen von Schleifen „immer“ (also in jedem Durchlauf) gelten.



➤ Mögliche ist eine tabellarische oder textuelle Erfassung.

➤ Wir machen es textuell:

- Z1 enthält zwei Zuweisungen, die Bedingung in Z5 enthält zwei Vergleiche und in jedem Fall eine Ausgabe.
- Die Schleife in Z2 enthält zwei Vergleiche und im schlechtesten Fall eine Zuweisung (so ist für $a \neq b$ höchstens eine beider Bedingungen trifft zu).
- Die Schleife wird immer genau n -mal durchlaufen.

➤ So erhalten wir im best- (keine a's und b's) und worst-case (nur a's und b's):

$$\text{best-case} = 2 + n \cdot (2 + 0) + 2 + 1 = 2n + 5 \in \mathcal{O}(n)$$

$$\text{worst-case} = 2 + n \cdot (2 + 1) + 2 + 1 = 3n + 5 \in \mathcal{O}(n)$$

```
z1: hatA ← Nein,   hatB ← Nein;
z2: für i ← 1 bis n Schrittgröße 1:
z3:   Wenn si = a: hatA ← Ja;
z4:   Wenn si = b: hatB ← Ja;
z5: Wenn hatA = Ja, hatB = Nein: „Ja“;
z6: sonst: „Nein“;
```

Eine verkürzte Version

Schweigsame Annahmen

Für das Programm „Zeichenkettenfilter“ und seine Analyse gelten die Annahmen:

1. die Länge n von S sei endlich und (in endlicher Zeit) berechenbar
2. wir können die Zeichen s_i aus Σ vergleichen.

Ob Z5 wirklich immer zwei Vergleiche hat, hängt davon ab, wie das logische „und“ funktioniert. Wertet es für „falsch \wedge $\langle ? \rangle$ “ den zweiten Operand nicht mehr aus, haben wir Falle eines S ohne a 's, einen Aufwand von $2n + 4$.

Noch werden wir uns solchen Annahmen zärtlich nähern. Mit der Zeit wird es aber immer mehr auch darum gehen, was man eigentlich annehmen darf.



Input : String S of chars S[i] and length N

```
1 i = 0;
2 Answer = false;
3 B = false;

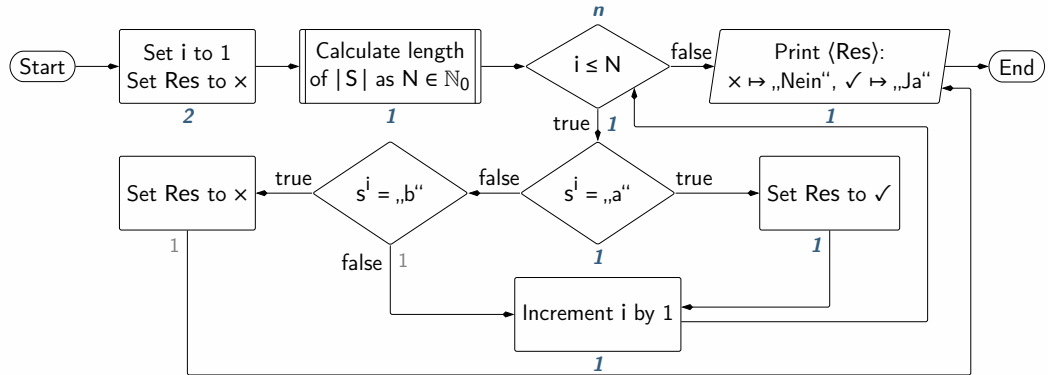
4 while i ≤ N do
5   | if S[i] == 'a' and B == false then Answer = true;
6   | if S[i] == 'b' then B = true; Answer = false; Exit while early;
7   | Erhöhe i um 1;
8 end

9 if Answer == true then Output: „Ja“;
10 else Output: „Nein“;
```

Die Laufzeit beläuft sich auf:

$$\begin{aligned} & 3 + n \cdot (2 + \max(2 + 1 + 1, 2 + 1 + 2)) + 1 \\ & = 4 + n \cdot 7 \in \mathcal{O}(n) \end{aligned}$$

Given the String $S = s^0 s^1 s^2 \dots$ and let $i \in \mathbb{N}$, $\text{Res} \in \{x, \checkmark\}$



$$2 + 1 + n \cdot (2 + 2) + 1 = 4 + n \cdot 4 \in \mathcal{O}(n)$$

- › Ob wir das Inkrement mitzählen oder nicht, hängt von der Definition ab.
- › Analog könnte das Berechnen der Länge auch in $\mathcal{O}(n)$ oder vergleichbar sein.
- › Deswegen wichtig: Annahmen in Textform festhalten!

- › **Spielt das überhaupt eine Rolle?**

Ja und nein. Im Allgemeinen Fall nein, weil es uns mehr darum geht ob die Algorithmen konstant ($\mathcal{O}(1)$), polynomiell ($\mathcal{O}(n^2)$) oder schlimmer sind ($\mathcal{O}(2^n)$, $\mathcal{O}(n!)$, ...). Dennoch kann es sinnvoll sein um Algorithmen der gleichen Wachstumsklasse zu vergleichen oder um Randentscheidungen zu treffen (vergleiche 2^n vs. n^{4000} für $n = 10$). Die Nützlichkeit der Definitionen hängt vom konkreten Fall ab.

- › **Wir präferieren strukturiert-textuelle Beschreibungen gegenüber Programmablaufplänen.**

*Avoid syntactic elements from the target
programming language*
— Steve McConnell, [McC93, p. 54]

Bonus: Pseudocode

How-To Pseudocode

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

Input : Eingabe als Liste von n Pingus: 🐧₁, ..., 🐧_n

// Jeder Pinguin ist klasse!

1 $i \leftarrow 1$;

2 **while** $i \leq n$ **do**

3 **if** 🐧_i *ist klasse!* **then** **return** 🐧_i ;


4 $i \leftarrow i + 1$;


5 **return** 🐧; // Wenn kein klasse Pingu gefunden.











Pseudocode: do's and don't'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, ... _n

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

Algorithmus 4 : Einen „klasse“ Pinguin finden

```
In: 1, ..., n  
Out: awesome-pengu, if none: mega-pengu  
iterate for every i in 1, ..., n:  
    if i is awesome: return i // Pengu found  
return  // Not found
```

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. [Standardwert: Nichts gefunden]

Übungsblatt 0

Aufgabe 1: Java Compiler und Laufzeitumgebung

Installieren Sie die für Ihr Betriebssystem aktuelle Version des Java Development Kits. Diese werden Sie auch noch im Lauf der Veranstaltung für die Bearbeitung der Programmieranteile der Übungsblätter benötigen. Bestimmen und notieren Sie anschließend die Versionsnummern.

› 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
```

Aufgabe 2: Erste Schritte in Java

Speichern Sie den Code in einer Textdatei namens `HelloWorld.java` ab.

> Tipp, tipp, tipp, ...

```
1 class HelloWorld {  
2     public static void main(String[] args) {  
3         System.out.println("Hello World!");  
4     }  
5 }
```

Übersetzen Sie das Programm mittels **javac** und führen Sie den erzeugten Java-Bytecode mit **java** aus.

> Wir tun wie geheißen: **javac** HelloWorld.java

> Und dann auch für die Ausgabe: **java** HelloWorld:

```
Hello World!
```

Aussicht: Übungsblatt 1

Ein „Algorithmus“, was ist das?

- › *Eindeutige* Handlungsvorschrift zur Lösung eines Problems.
Die ausführbar und reproduzierbar ist.
- › Endlich viele, *wohldefinierte* (also nicht mehrdeutige) und elementare Einzelschritte.
- › Der Algorithmus muss nach einer endlichen Zeit zum Ende kommen („terminieren“).

Mit der Zeit werden wir mehr Eigenschaften zur Charakterisierung betrachten.





- › 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 $t = (t_1, t_2, \dots, t_m) \in \mathbb{Z}^m$ der Größe $m \geq 1$ mit ($t_i \in \mathbb{Z}$ für alle t_i)

Beliebig: Die Elemente können unsortiert vorliegen (z.B. $t_1 \leq t_2 > t_3$).

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.



> 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 Abstraktion nahezu offensichtlich. Es dient der Veranschaulichung des Vorgehens 😊.

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

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

› **Korrektheitsnachweis:**

Terminiert: Die Schleife aus 3. terminiert, da i pro Iteration um 1 inkrementiert wird und damit streng monoton wächst.

Es ist sicher irgendwann $i \geq m$, da Länge m konstant.

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

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

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

*He who chooses the beginning of a road chooses the
place it leads to. It is the means that determine the end.*
— Harry Emerson Fosdick, [Fos41, p. 111]

Abschließendes

- [Doy10] Arthur Conan Doyle. *The sign of four*. 2010
- [Fos41] Harry Emerson Fosdick. *Living Under Tension: Sermons on Christianity Today*. 1941
- [McC93] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. 1993

> Für Pseudocode wünschen wir uns

- eine konsistente und menschenlesbare Notation,
- mathematische Definitionen,
- aber nichts sprachspezifisches (`int`) oder zu allgemeines („löst Problem“).

> Algorithmen und deren Konstruktion:

- Eine eindeutige, endliche Beschreibung wohldefinierter Elementaroperationen, deren schrittweise Ausführung durch einen Prozessor möglich und endlich ist.
- Spezifikation > Abstraktion > Entwurf > Verifikation > Aufwandsanalyse

