0–11

Kompaktversion Tutorien

Von pinguinreduzierten Tutorien 0–11

Diese ***Kompaktversion*** ist dazu gedacht, die wichtigsten Kommentare und Lösungen zu sammeln. Sie ist allerdings ***ohne jede Garantie auf Vollständigkeit*** aufzufassen. Zusätzliche Inhalte finden sich in den Folien zu den einzelnen Tutorien.

Liebe Grüße, Flo

Blatt 0
Hello World

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 = `abababababa` ⟶ „Nein“ (Enthält ein b)
- S = `aaaaaaaaa` ⟶ „Ja“ (Hat a und kein b)
- S = `cdefg` ⟶ „Nein“ (Enthält kein a)
- S = `klmohna` ⟶ „Ja“ (Hat a und kein b)

Führen Sie die Schritte der Algorithmenentwicklung durch:

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

Problemspezifikation und -abstraktion

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

a) Spezifikation: Definiere alle relevante Begriffe unmissverständlich

- Eine „Zeichenkette“ S ist eine indizierte Liste $(s_1, \ldots, s_n)$ 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 $S = (s_1, \ldots, s_n)$

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

Algorithmenentwurf

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

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

Algorithmus 1 : Ein „Zeichenkettenfilter“

Korrektheitsnachweis, Verifikation

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

```
Z1: hatA ← Nein,    hatB ← Nein;
Z2: für i ← 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

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 ≤ 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.

Aufwandsanalyse

```
Z1: hatA ← Nein,    hatB ← Nein;
Z2: für i ← 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

- 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):

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

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

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 $\wedge$ ⟨?⟩“ 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.

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, …

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

*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`.

Zusammenfassend

- 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

Blatt 1
Algorithmenentwurf und -analyse

Präsenzaufgabe

2

It's true, isn't it?

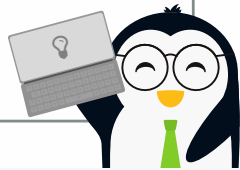
```
class BoolescheAusdruecke {
  public static void main(String[] args) {

  }
}
```

Erweitern Sie die Datei. Deklarieren und initialisieren sie benötigte Variablen mit „beliebigen" Werten.

Konstruieren Sie boolesche Ausdrücke, die folgendes abprüfen:

- Die Zimmertemperatur (`temperatur`) beträgt höchstens 22,5 Grad Celsius.
- Eine Person (`alter`) ist nicht zwischen 13 und 18 Jahre alt.

Geben Sie die Ergebnisse über `System.out.println(...)` aus.



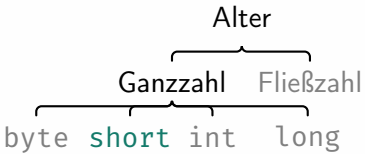
Die Datentypen finden

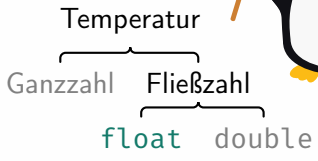
Zimmertemperatur höchstens 22,5 Grad Celsius, Alter nicht zwischen 13 und 18 Jahren.

```java
class BoolescheAusdruecke {
    public static void main(String[] args) {
        float temperatur = 18.0f;
        short alter = 15;
        // ...
    }
}
```

Einen generell korrekten Datentyp gibt es selten. Diverse Bedingungen hängen vom Kontext ab.

Die Zahlen sind erstmal beliebig.





Boolesche Ausdrücke konstruieren

Zimmertemperatur höchstens 22,5 Grad Celsius, Alter nicht zwischen 13 und 18 Jahren.

```
class BoolescheAusdruecke {
   public static void main(String[] args) {
      float temperatur = 18.0f;
      short alter = 15;
      System.out.println(temperatur <= 22.5f);
      System.out.println(!(13 <= alter && alter <= 18));
   }
}
```

BoolescheAusdruecke.java

Deklarieren, Initialisieren und Zuweisen

- Die Begriffe sind leicht definiert:
  - **Deklaration:** Reserviert einen Namen und setzt den Typ — `int x, y;`
  - **Initialisierung:** Die *erste* Zuweisung einer Variable — (Erst jetzt ist die Variable verwendbar)
  - **Zuweisung:** Setzen/Ändern des Wertes einer Variable — `x = 3;`

```
public static void main(String[] args) {
    int apfelsine;              Deklarierung einer Variable „apfelsine“ vom Typ int
    char zeichen = 'x';         Deklarierung mit Typ char und Initialisierung mit dem Wert 'x'
    apfelsine = 42;             Initialisierung mit dem Wert 42
    if(apfelsine % 2 == 0)
        zeichen = 'k';          Zuweisung des Wertes 'k'
}
```

Aufgabe 1: Aufwandsanalyse

In der Vorlesung haben wir uns mit der Aufwandsanalyse von Algorithmen beschäftigt und uns angeschaut, wie viele Operationen im schlechtesten Fall notwendig sind, um den Algorithmus durchzuführen. Im Folgenden wollen wir uns nun anschauen, wie sich solche Aussagen (vereinfacht) auf Maschinenoperationen übetragen lassen.
Bei einer Firma fällt im Produktionsprozess täglich ein Optimierungsproblem an, für dessen Lösung eine Stunde zur Verfügung steht. Der Algorithmus zur Lösung des Optimierungsproblems muss bei Eingabe eines Problems der Größe n *insgesamt* $100n^2$ *viele Operationen durchführen. Bisher verwendet die Firma zur Lösung einen Mikro-Rechner mit einem Prozessortakt von* 800 MHz. *Dieser kann in jeder Sekunde* 40 000 *der oben genannten Operationen berechnen.*

a) *Welche Problemgröße* n *dürfen die Probleme maximal haben, damit die Berechnung innerhalb der Frist durchgeführt werden kann?*

b) *Nun soll auf einen schnelleren Rechner mit* 3 200 MHz *Taktfrequenz umgestiegen werden. Nehmen Sie an, die höhere Taktfrequenz übersetze sich direkt in eine entsprechend höhere Rechengeschwindigkeit. Wie groß können nun die Optimierungsprobleme sein?*

c) *Anstelle einer CPU mit einem, soll eine CPU mit 16 Kernen (jeweils mit* 3 200 MHz*) eingesetzt werden. Glücklicherweise lässt sich das Optimierungsproblem leicht parallelisieren, so dass sich die Geschwindigkeit im Vergleich zu einem Kern verneunfacht. Wie groß können nun die Optimierungsprobleme sein?*

Aufwandsobergrenze

a) Was ist die maximale Problemgröße n um in der Frist zu bleiben?

In der Stunde haben wir $60 \cdot 60$ Sekunden und damit $60 \cdot 60 \cdot 40\,000$ Operationen. Bei Problemgröße n benötigen wir $100n^2$ Operationen:

$$
\begin{aligned}
100n^2 &\leq 60 \cdot 60 \cdot 40\,000 \\
100n^2 &\leq 144\,000\,000 && \mid \cdot 1/100 \quad \mid \sqrt{} \\
n &\leq \sqrt{1\,440\,000} \\
n &\leq 1200
\end{aligned}
$$

› Die Problemgröße n darf 1200 nicht übersteigen.

Die Macht der Taktfrequenz

b) *Angenommen, die höhere Taktfrequenz übersetze sich direkt in eine entsprechend höhere Rechengeschwindigkeit. Wie groß können die Optimierungsprobleme bei* 3 200 MHz *Taktfrequenz sein?*

Bisher schaffen wir 40 000 Operationen, nun schaffen wir $40\,000 \cdot {}^{3\,200\,\text{MHz}}/_{800\,\text{MHz}}$:

$$100n^2 \leq 60 \cdot 60 \cdot 40\,000 \cdot {}^{3\,200\,\text{MHz}}/_{800\,\text{MHz}}$$

$$100n^2 \leq 576\,000\,000 \qquad | \cdot {}^{1}/_{100} \quad | \sqrt{}$$

$$n \leq \sqrt{5\,760\,000}$$

$$n \leq 2400$$

› Die Problemgröße n darf nun 2400 nicht übersteigen.

Ein vervierfachen der Leistung verdoppelt die mögliche Problemgröße aufgrund des quadratischen Wachstums.

Parallelisierungskosten

c) Anstelle einer CPU mit einem, soll eine CPU mit 16 Kernen (jeweils mit 3 200 MHz) eingesetzt werden. Glücklicherweise lässt sich das Optimierungsproblem leicht parallelisieren, so dass sich die Geschwindigkeit im Vergleich zu einem Kern verneunfacht. Wie groß können nun die Optimierungsprobleme sein?

Die Anzahl der Operationen verneunfacht sich direkt:

$$
\begin{aligned}
100n^2 &\leq 60 \cdot 60 \cdot 40\,000 \cdot {}^{3\,200\,\mathrm{MHz}}/_{800\,\mathrm{MHz}} \\
100n^2 &\leq 5\,184\,000\,000 && \mid \cdot {}^{1}/_{100} \quad \mid \sqrt{} \\
n &\leq \sqrt{51\,840\,000} \\
n &\leq 7200
\end{aligned}
$$

› Die Problemgröße n darf nun 7200 nicht übersteigen.

Aufgabe 2: Korrektheit von Algorithmen

Betrachten Sie die Anweisungsfolge. Handelt es sich hierbei um einen Algorithmus? Prüfen Sie alle notwendigen Voraussetzungen.

```
1: Setze        x = 1
2: solange      x ≥ 0 wiederhole
3:              Verdopple x
4:              Verringere x um 1
5: ende
```

Ausführ- und Reproduzierbarkeit: Ja, alle Schritte sind klar und umsetzbar.

Von Prozessor schrittweise ausführbar: Ja, ein Mensch kann sie auf Papier durchführen.

Nur Elementaroperationen (oder in solche übersetzbar)**:** Ja, für alle per Definition.

Endliche Beschreibung: Ja, per der Beschreibungstext ist endlich.

Terminiert: Nein, es handelt sich um eine Endlosschleife. Die Werte von x verhalten sich $1 \to 2 \to 1 \to 2 \to \ldots$, damit ist nie $x \geq 0$.

Problemspezifikation

Tagespreis: positive reelle Zahl. Dargestellt mit zwei Nachkommastellen.

Tagespreisliste: chronologisch sortierte Liste an Tagespreisen $[p_1, \ldots, p_n]$ mit $n \in \mathbb{N}_1$.

Tag: natürlicher Index in Tagespreisliste.

Zwei aufeinanderfolgende Tage: zwei Listenindizes: i und i + 1 (wobei $i, i + 1 < n$).

Preisanstieg: positive Veränderung einer reellen Zahl (dem Tagespreis): $p_{i+1} - p_i > 0$.

Maximaler prozentualer Preisanstieg: größter relativer Preisanstieg: $\max_{1 \le i < n} \frac{p_{i+1} - p_i}{p_i} \cdot 100$.

Wichtig: Wir sind hier nicht in einer Programmiersprache: Typen wie „`int`“ gibt es nicht $\Longrightarrow$ mathematische Notation benutzen oder eigenen Typ definieren (wie Tupel, …)!

Problemabstraktion

Liste von Tagespreise

$(p_1, \ldots, p_n)_{n \in \mathbb{N}}$

▼

Max. % Preisanstieg

Δ_{max}

Gegeben: Chronologisch sortierte Liste $[p_1, \ldots, p_n]$ positiver reeller Zahlen. Annahme: $n > 1$ („letzte Monate“).

Gesucht: Positive reelle Zahl $m = 100 \cdot \max_{1 \leq i < n} \frac{p_{i+1} - p_i}{p_i}$ ($X\,\% \mathrel{\widehat{=}} \frac{X}{100}$).

Algorithmenentwurf

(1) Setze Δ_{max} auf $-\infty$.

(2) Setze i auf 1.

(3) Solange (i < n), wiederhole:

Setze test auf $(p_{i+1} - p_i)/p_i$.

Wenn (test > Δ_{max}): Setze Δ_{max} auf test.

Erhöhe i um 1.

(4) Ergebnis ist $100 \cdot \Delta_{max}$.

$\Delta_{max} = -\infty$ ist natürlich unpraktisch. Hier könnte man für $\Delta_{max} = 0$ argumentieren (da „Anstieg“). Wir können aber auch die erste Differenz als Maximum setzen! Das kommt später.

Korrektheitsnachweis, Verifikation

(1) $\Delta_{max} \leftarrow -\infty$.
(2) $i \leftarrow 1$.
(3) Solange $i < n$:
    $test \leftarrow (p_{i+1} - p_i)/p_i$.
    Wenn $test > \Delta_{max}$: $\Delta_{max} \leftarrow test$.
    $i \leftarrow i + 1$.
(4) Ergebnis ist $100 \cdot \Delta_{max}$.

1. Eine formale, vollständige Induktion, oder:
2. „Textbasiert“: i wächst streng monoton an, die Schleife wird damit genau $n - 1$ mal durchlaufen (n ist konstant). Alle mathematischen Berechnungen terminieren per Konstruktion, ebenso die Zuweisungen. Der Algorithmus *terminiert*.

   Zudem ist er *partiell korrekt*. Die maximale prozentuale Änderung ist immer die größte relative Differenz für alle $[p_1, \ldots, p_{i+1}]$ im i-ten Durchlauf. Nach $n - 1$ Durchläufen: $[p_1, \ldots, p_n]$. Basisfall mit $i = 2$, für Schritt $i \rightarrow i + 1$ (analog zum Maximumsalgorithmus).

Notizen

- Totale Korrektheit erfordert zwei Komponenten!

  Terminiertheit: Der Algorithmus terminiert für jede definierte Eingabe.

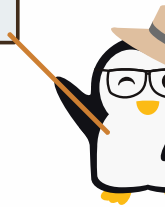
  Partielle Korrektheit: Wenn der Algorithmus für eine definierte Eingabe terminiert, ist das Ergebnis korrekt.

- Formulierungen wie folgt reichen nicht aus:
  - „Maximum ist sicher größer als alle betrachteten Alternativen“.
    Es muss dann zum Beispiel auch gezeigt werden, dass alle relevanten Alternativen in Betracht gezogen werden.
  - „Der Algorithmus findet das gesuchte Ergebnis“.
    Wir Beweisen zwar nicht formal, dennoch sollte ein ausreichendes Verständnis fürs Beweisen gezeigt werden.

Textbasierte Aufwandsanalyse

(1) Setze Δ_{max} auf $-\infty$.
(2) Setze i auf 1.
(3) Solange (i < n), wiederhole:
    Setze test auf $(p_{i+1} - p_i)/p_i$.
    Wenn (test > Δ_{max}): Setze Δ_{max} auf test.
    Erhöhe i um 1.
(4) Ergebnis ist $100 \cdot \Delta_{max}$.

Ob Division und Subtraktion als Elementaroperation zählen, kommt auf die Definition an.



- Die 1. Zuweisungen 1 & 2 sind 2 Elementaroperationen.
- Die äußere Schleife in 3 wird genau $n - 1$ mal durchlaufen.
  - Die 1. Schleifenanweisung hat 3 Elementaroperationen: Subtraktion, Division & Zuweisung.
  - 2. Schleifenanweisung ist 1 Vergleich und maximal 1 Zuweisung.
  - 3. Schleifenanweisung ist 1 Elementaroperation (je nach Definition auch 2).
- Das Endergebnis ist eine Multiplikation (und je nach Definition 1 Zuweisung).
- Im worst-case haben wir so: $2 + (n - 1) \cdot (3 + 2 + 1) + 2 = 6n - 2$ (also $\mathcal{O}(n)$).

Eine Alternative

- Wir können erst alle Preisanstiege berechnen und dann darin das Maximum suchen:

given days $1, \ldots, n$ **and** prices $(p_1, \ldots, p_n)$ **indexable by** p_i
when $n < 2$ **then output** *"Ein Preisanstieg braucht mind. 2 Tage"* **and stop**

Optional: Robust gegenüber zu wenig Tagen.

make list $L = (l_1, \ldots, l_{n-1})$ **of size** $n - 1$ **indexable by** l_i
for each d **from** 1 **to inclusive** $n - 1$ **in steps of** 1
- **let** l_d **be** $(p_{d+1} - p_d)/p_d \cdot 100$

Berechne den Preisanstieg an Tag d *für alle Tage.*

[Jetzt hält l_i den Preisanstieg an Tag i und hat $n - 1 \geq 1$ Elemente]
let max_diff **be** l_1
for each diff **from** 2 **to inclusive** $n - 1$ **in steps of** 1
- **when** diff > max_diff **then let** max_diff **be** diff

Normale Suche nach dem Maximum.

output *"Der Maximale Preisanstieg ist* diff*"*

Allgemeines

- Der *Algorithmenentwurf* ist nur ein Schritt der Algorithmenkonstruktion
- Annahmen festhalten
- Auch die Präzedenzregeln sind sicherlich hilfreich. Die wichtigsten:

$$
\begin{aligned}
&\texttt{a++},\ \texttt{a--} \longrightarrow \texttt{!a},\ \texttt{-a},\ \texttt{++a},\ \texttt{--a} \longrightarrow \texttt{a * b},\ \texttt{a / b},\ \texttt{a \% b}\\
&\qquad\qquad \longrightarrow \texttt{a + b},\ \texttt{a - b} \qquad\quad \longrightarrow \texttt{a == b},\ \texttt{a < b},\ \ldots\\
&\qquad\qquad \longrightarrow \texttt{a \^{} b} \qquad\qquad\ \longrightarrow \texttt{a \&\& b}\\
&\qquad\qquad \longrightarrow \texttt{a || b}
\end{aligned}
$$

Zusammenfassend

- Wir haben die Geburt von Variablen sowie ihre Wachstumsphase kennengelernt:
  - Die *Deklaration* (`int x`) reserviert einen Namen samt Charakteristika (Typ, …)
  - Die *Initialisierung* (`int x; x = 5`) beschreibt die erste Wertzuweisung
  - Eine *(Wert-)Zuweisung* (`x = 3`) ist jede weitere Änderung des gespeicherten Wertes

- In die Wahl des „richtigen™“ Datentyps fließen viele Informationen mit ein.

Blatt 2
Algorithmen, Datentypen, Boolesche Ausdrücke

Präsenzaufgabe

3 **Wenn… Ja wenn nur die Schleife nicht wär…**

Legen Sie eine Java Datei namens `ErsteSchleife.java` an (oder bearbeiten Sie die Aufgabe auf einem Blatt Papier). Lesen Sie in der main Methode eine `int` Variable namens n über die Kommandozeilenparameter ein und implementieren Sie eine for Schleife, die jede dritte Zahl von 1 bis einschließlich n ausgibt. Beispiel:

```
n = 13 // → 1 4 7 10 13
```

Ein Schleifenzähler

- Mit dem Vorwissen ist es an sich nur Einsetzen:

```java
public class ErsteSchleife {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        for(int i = 1; i <= n; i = i + 3) {
            System.out.println(i);
        }
    }
}
```

Was, wenn es keine Argumente gibt? Hier hilft eine if-Abfrage.

Das Beispiel der Aufgabe hatte keine neuen Zeilen! Dafür gibt es „print“ anstelle von „print **l**in**e**“.

ErsteSchleife.java

Aufgabe 1: Algorithmen und Unteralgorithmen

Oft ist es zur besseren Strukturierung notwendig komplexere Algorithmen aufzuteilen. Dies ist beispielsweise der Fall, wenn Code-Elemente häufig an verschiedenen Stellen eingesetzt werden oder wenn es der Übersichtlichkeit dienen soll. Im Folgenden sollen Sie einen einfachen Sortieralgorithmus in zwei Teilalgorithmen umsetzen. Zusätzlich sollen Sie lernen die Eingaben (sog. Parameter oder Argumente) des Aufrufs auf Gültigkeit zu überprüfen. Für zukünftige Aufgaben sollen Sie dies selbstständig erkennen und umsetzen.

Indizes		*Liste*				*Vertauschen?*
i	*j*	l_1	l_2	l_3	l_4	
1	2	▸ ***5***	▸ ***8***	4	0	*Nein, denn* $5 \not> 8$
1	3	▸ ***5***	8	▸ ***5***	0	*Ja, denn* $5 > 4$
1	4	▸ ***4***	8	5	▸ ***0***	*Ja, denn* $4 > 0$
Runde vorbei, alle $j \in \{i+1, \ldots, N\}$ *betrachtet!*						
2	3	0	▸ ***8***	▸ ***5***	5	*Ja, denn* $8 > 5$
2	4	0	▸ ***5***	8	▸ ***4***	*Ja, denn* $5 > 4$
Runde vorbei, alle $j \in \{i+1, \ldots, N\}$ *betrachtet!*						
3	4	0	4	▸ ***8***	▸ ***5***	*Ja, denn* $8 > 5$
Fertig, alle $i \in \{1, \ldots, N-1\}$ *betrachtet!*						
→		0	4	5	8	

Gehen Sie zur Sortierung nach folgendem Schema vor.

Seien $l_1, \ldots, l_N$ *die Werte einer Liste* L *der Länge* N. *Für jedes* $i \in \{1, \ldots, N-1\}$ *betrachte der Reihe nach jedes* l_j *mit* $j \in \{i+1, \ldots, N\}$ *und vergleiche* l_i *mit* l_j. *Falls* $l_i > l_j$ *vertausche* l_i *und* l_j *und fahre mit dem nächsten* j *fort.*

So sortiert der beschriebene Algorithmus die Liste $(5, 8, 4, 0)$ *aufsteigend zu* $(0, 4, 5, 8)$. *Die beiden* ▸ ***Markierungen*** *je Zeile geben an, welche Elemente in dem jeweiligen Schritt verglichen werden.*

Elemente vertauschen

a) Entwerfen Sie einen Algorithmus, der gegeben einer Liste und zwei Indizes die entsprechenden Elemente vertauscht. Stellen Sie sicher, dass die Indizes i *und* j *gültig sind, d.h. dass sie* ***größer als*** 1 *und kleiner als die Länge der Liste sind. Stellen Sie zudem sicher, dass die Länge der Liste mindestens* 2 *beträgt. Der Algorithmus soll für den Fall der Ungültigkeit mit einer Fehlermeldung abbrechen.*

Input : List L = $(l_1, \ldots, l_N)$, Index i, and Index j

```
1 if N < 2 or i < 1 or j < 1 or i > N or j > N then
2     print „Eingabe ungültig!“ to output device;
3     stop program;
4 make new t have value l_i;
5 make l_i have value l_j;
6 make l_j have value t;
```

In der Aufgabe stand etwas von „größer als 1“, aber wie wir Listen indizieren ist nicht von uns standardisiert.

Sich den Tag schön-sortieren

b) Entwerfen Sie einen Algorithmus, der genau nach dem angegebenen Schema eine Liste sortiert. Stellen Sie auch hier sicher, dass die Liste mindestens zwei Elemente hat. Falls Sie Teilaufgabe a) nicht bearbeitet haben, können Sie annehmen, dass der Algorithmus `vertausche(L, i, j)` zur Verfügung steht.

Input : List L = $(l_1, \ldots, l_N)$

```
1 if N < 2 then
2     print „Eingabe ungültig!“ to output device and stop program;
3 Make new i have value 1;
4 while i < N do
5     Make new j have the value i + 1;
6     while j ≤ N do
7         if l_i > l_j then call vertausche(L, i, j);
8         Increment j by 1;
9     Increment i by 1;
```

Du kriegsts' nicht raus? Better Call Gauß!

c) Geben Sie eine worst-case Laufzeitabschätzung für Ihren Algorithmus aus Teilaufgabe b) an. Falls Sie Teilaufgabe a) nicht bearbeitet haben, können Sie annehmen, dass das Vertauschen (inkl. aller Überprüfungen höchstens) neun elementare Operationen benötigt. Begründen Sie Ihre Antwort.

„Wir" haben die a) bearbeitet. Also…Lets begin!

Input : List L = $(l_1, \ldots, l_N)$, Index i, and Index j

```
          1        1        1        1        1
1 if N < 2 or i < 1 or j < 1 or i > N or j > N then
2 |   print „Eingabe ungültig!" to output device;   1
3 |   stop program;   1
4 make new t have value l_i;   1
5 make l_i have value l_j;   1
6 make l_j have value t;   1
```

2 (Nur schlechtester Pfad)

3

→ $1E + 1E + 1E + 1E + 1E + 3E = 8E$

Du kriegsts' nicht raus? Better Call Gauß! The Sequel!

Input : List L = $(l_1, \ldots, l_N)$

```
1  if N < 2 then                                   1
2      print „Eingabe ungültig!“ to output device and stop program;   2
3  Make new i have value 1;                        1
4  while i < N do                                  N − 1
5      Make new j have the value i + 1;            2
6      while j ≤ N do                              (N²−N)/2
7          if l_i > l_j then                       1
8              call vertausche(L, i, j);           8
9          Increment j by 1;                       1
10     Increment i by 1;                           1
```

Angenommen N *sei* 5*, dann gilt für die Anzahl innerer Schleifendurchläufe:*

i	#
1	4
2	3
3	2
4	1

Wir summieren also von 1 *bis* N − 1*. Nach Gauß:* $\frac{N^2-N}{2}$ *Gesamtdurchläufe.*

$$\begin{aligned}
&\to 1\,E + 1\,E + (N-1)\cdot(2\,E + 1\,E) + \frac{N^2-N}{2}(1\,E + 8\,E + 1\,E) \\
&= -1\,E + 3\cdot N\,E + 5\cdot N^2\,E - 5\cdot N\,E \\
&= 5\cdot N^2\,E - 2\cdot N\,E - 1\,E \quad \in \mathcal{O}(N^2)
\end{aligned}$$

Aufgabe 2: Datentypen

Welchen Datentyp würden Sie wählen um folgende Daten zu speichern? ***Begründen Sie Ihre Antwort!***

a) *Die Email-Adresse einer Person*

Email-Adressen sind lange Zeichenketten welche auch Sonderzeichen enthalten. Daher ist `String` naheliegend.

b) *Den Akkustand eines Smartphones*

Ein Akkustand wird für gewöhnlich als Ganzzahl angegeben (→ `byte`, `short`, `int` und `long`). Der Akkustand reicht dabei meist von 0 % bis 100 % → `byte`.

c) *Den Notendurchschnitt einer Prüfung*

Die Noten einer Prüfung ist meist eine Fließkommazahl (→ `float`, `double`), deren Wertebereich klein mit geringer Auflösung an Nachkommastellen ist (z.B. 1–6 mit einer Nachkommastelle). Deswegen wählen wir hier `float`.

Aufgabe 2: Datentypen II

d) Ein Satzzeichen

Wir gehen hier von einem Satzzeichen aus, welches durch eine einzige UTF-16 Einheit dargestellt werden kann. In diesem Fall reicht ein `char` (sonst `String` das ist aber schon wieder Overkill).

- Eine eindeutig richtige Antwort gibt es nicht (Begründungen!).
- Begründungen wie „weniger Speicherplatz“ sind problematisch(er)
  - Java kann für ein `byte` auch 32 bit oder mehr (64 bit, …) reservieren
  - Analog wird `boolean` oft als `byte` umgesetzt (JVMS17 2.3.4)
  - Das hat mit Geschwindigkeit, Parallelisierung, Sicherheit, … zu tun
  - Für uns in EidI sind die Details nicht wichtig

Aufgabe 3: Boolesche Ausdrücke

Betrachten Sie den folgenden Java Code-Ausschnitt. Werten Sie die folgenden Ausdrücke in der Reihenfolge, in der sie ausgeführt werden (d.h. von oben nach unten), aus. Geben Sie dazu für jede Zeile (beginnend mit der zweiten Zeile) den neuen Wert der Variablen b an.

```
boolean b = true;
b = !(true || false) || b;
b = !(true && false);
b = (!(true && true) || (false || false));
b = !b;
```

true
false false true
true
b ist ***true***

true
b ist ***true***

true true
false false false
b ist ***true***

false
true
b ist ***true***

a++, a-- ⟶ !a, −a, ++a, --a ⟶ a ∗ b, a / b, a % b ⟶ a + b, a – b
⟶ a == b, a < b, . . . ⟶ a ^ b ⟶ a && b
⟶ a || b ⟶ . . .

Aufgabe 4: Varablen anlegen und verarbeiten

In der Vorlesung haben Sie gelernt, wie in Java Variablen deklariert und initialisiert werden. Nun sollen Sie in einem kleinen Beispiel diese Konzepte selbst umsetzen. Betrachten Sie die `.java`*-Datei namens* `Variablen.java` *mit folgendem Inhalt:*

```
class Variablen {
   public static void main(String[] args) {


   }
}
```

a) *Erweitern Sie den Code so, dass zwei int Variablen deklariert werden. Initialisieren Sie diese* ***anschließend*** *mit einem beliebigen gültigen Wert und addieren Sie sie. Das Ergebnis dieser Addition soll in eine neue dritte int Variable gespeichert werden. Geben Sie das Ergebnis dann über den Aufruf System.out.println(name) aus, wobei Sie name entsprechend ersetzen.*

b) *Geben Sie als Kommentar im Java Code aus Teilaufgabe a) jeweils an, ob es ich bei dieser Zeile um eine Deklaration, Initialisierung oder Zuweisung handelt.*

Simple Adds and Stuff

a) Erweitern Sie den Code (Deklarieren, Initialisieren, Addieren und Ausgeben)

b) Geben Sie an, wo Deklarationen, Initialisierungen und Zuweisungen stattfinden.

```
class Variablen {
   public static void main(String[] args) {
      int zahl1, zahl2, ergebnis;   Alle drei Variablen werden deklariert
      zahl1 = 40;   Eine Initialisierung mit Wert 40
      zahl2 = 2;   Eine Initialisierung mit Wert 2
      ergebnis = zahl1 + zahl2;   Eine Initialisierung mit dem Ergebnis der Summe
      System.out.println(ergebnis);   Eine Ausgabe (für die b irrelevant)
   }
}
```

Ist es wichtig, dass die letzte Initialisierung mit einer Summe ist? Nö. Es ist einfach nur der erste Wert, den `Ergebnis` erhält.

Ein kleiner Schwenk

- Wenn wir Java-Code möchten, steht das da (ab jetzt wird das der Standardfall sein).
- In Aufgabe 1 suchen wir *einen* booleschen Ausdruck der das Programm abbildet.
  Vergleiche hierzu die Präsenzaufgabe von letzter Woche.
- Für die Schleifen der 2. Aufgabe, kann man sich die Alternativen dieser Präsenzaufgabe nochmal ansehen.
- Allgemein gilt: Wenn Pseudocode oder ein Ansatz gegeben ist, soll der Algorithmus auch diesen Verwenden oder darauf aufbauen!

Aufgabe 3: Algorithmen in Java implementieren

- Eine abstrakte Notation wird konkret.
- Viele einzelne Punkte werden nun kollektiv Wichtig
  - Einhalten der Java-Syntax
  - Welcher Datentyp passt am besten?
  - Welche Art von Schleife, Fallunterscheidung, … eignet sich?
  - Was drückt der Pseudocode implizit aus, ist aber in Java explizit notwendig?
  - Und umgekehrt: was müssen wir in Java nicht schreiben?
  - …
- Kommentare empfehlen sich da, wo der Gedanke des Algorithmus hinter Java-Gebrabbel verschwindet. (If you are interested, consider looking up literate programming [Knu84])

Aufgabe 4: Iterative Wurzelberechnung

- Für die Fließkommazahl gibt es `Float.parseFloat` oder `Double.parseDouble`
- Verwendet das Heron-Verfahren wie angegeben (keine alternative Wurzelberechnung)
- Ihr dürft gerne eure Entscheidungen in den Kommentaren begründen.
- An welcher Stelle kommen Unteralgorithmen zum Einsatz?
  Und… Wo könnten sie noch hilfreich sein?

Zusammenfassend

[Knu84] Donald Ervin Knuth. „Literate programming“. 1984

- Kommandozeilenparameter landen im `args`-Array (als `String`)
- `Integer.parseInt("13")` liefert die vom `String` beschriebene Ganzzahl (*13*)
- Unteralgorithmen sind ein elementarer Programmiermechanismus
- Javas implizite Typkonvertierungen existieren nur für primitive Datentypen
  - `byte` → `short` → `int` → `long` → `double` und `char` → `int`
  - „Zahlen von klein nach groß, `char` zu `int`“

Blatt 3
Kontrollstrukturen

Präsenzaufgabe

4 Von doppelten Methoden

Mehrdimensionale Arrays, also solche Arrays, bei denen die Elemente selbst wiederum Arrays sein können, können wir als Matrizen interpretieren, beispielsweise ist `array[1][3]` das Element der zweiten Zeile in der vierten Spalte. Legen Sie eine Java Datei namens `PositiveEintraege.java` an und implementieren Sie folgende Teilaufgaben innerhalb dieser Datei (oder bearbeiten Sie die Aufgabe auf einem Blatt Papier).

1. Initialisieren Sie ein zweidimensionales Array mit 3×3 Elementen vom Typ `double` mit gültigen Werten in der `main`-Methode.
2. Implementieren Sie eine Methode `public static int anzahlPositive (double[][] matrix)`, die die Anzahl der positiven Eintrage (> 0) bestimmt und zurückgibt. Für ungültige Matrizen soll die Methode `-1.0` zurückgeben.
3. Testen Sie die Implementierung aus b) mit ihrem Array aus a).

An Array a day

- Wir basteln uns eine Klasse und Initialisieren froh umher.

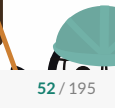
```
public class PositiveEintraege {
   public static void main(String[] args) {
      double[][] mat = {
           { -1.0,   21.0,   3.0 },   mat[0] → {-1.0, 21.0, 3.0}
           {  1.0,   42.0,  -3.0 },   mat[1][2] → -3.0
           {  1.0,  -84.0,   3.0 }
     };
    }
}
```

Hey guy's, i did some methods

*Implementieren Sie eine Methode `anzahlPositive(double[][])`, die die Anzahl der positiven Eintrage (> 0) bestimmt und zurückgibt. Für **ungültige** Matrizen soll die Methode `-1.0` zurückgeben.*

```
public class PositiveEintraege {
   public static void main(String[] args) { ... }
   public static int anzahlPositive(double[][] matrix) {
      int anzahl = 0;
      for(int row = 0; row < matrix.length; row++) {
         for(int col = 0; col < matrix[row].length; col++) {
            if(matrix[row][col] > 0) anzahl = anzahl + 1;
         }
      }
      return anzahl;
   }
}
```

Was heißt „ungültig?“ Das werden wir voerst zurückstellen!



Die Suche nach der positiven Anzahl

```java
public static int anzahlPositive(double[][] matrix) {
   if (matrix == null || matrix.length == 0) {
      System.out.println("Matrix ungültig!");
      return -1;
   }

   int anzahl = 0;
   for(int row = 0; row < matrix.length; row++) {
      for(int col = 0; col < matrix[row].length; col++) {
         if (matrix[row][col] > 0)
            anzahl = anzahl + 1;
      }
   }
   return anzahl;
}
```

Testen des Algorithmus

- Wir kehren in die `main`-Methode zurück:

```
public class PositiveEintraege {
    public static void main(String[] args) {
        double[][] mat = {{-1, 21, 3}, {1, 42, -3}, {1, -84, 3}};
        int anzahl = anzahlPositive(mat);
        System.out.println("erwartet: 6, erhalten: " + anzahl);
    }

    public static int anzahlPositive(double[][] matrix) { ... }
}
```

PositiveEintraege.java

Aufgabe 1: Bedingte Anweisungen und Boolesche Ausdrücke

Betrachten Sie den folgenden Java Code-Ausschnitt. Vereinfachen Sie die verschachtelten `if/else` Blöcke so, dass die korrekte Zuweisung der Variablen `z` ohne Kontrollstrukturen (`if`, `else`, `switch`, `while`, `do-while`, `for`, ternärer Operator) in einer Zeile verarbeitet wird, d.h. direkt über eine Zuweisung mithilfe eines Booleschen Ausdrucks. Gehen Sie davon aus, dass die Variablen `x` und `y` vom Typ `int` und `z` vom Typ `boolean` sind und alle Variablen gültige Werte besitzen.

```
if(x >= 0){ • •
    if(y <= 0) { • •
        z = true;
    } else {   •
        if(y == 1) { z = true; }
    }
} else {
    z = false;
}
```

z ist nur *true*, wenn

- `x >= 0 && y <= 0`

Oder:

- `x >= 0 && !(y <= 0) && y == 1`

Kombiniert kommen wir auf:

```
z = (x >= 0 && y <= 0) || (x >= 0 &&↩
     !(y <= 0) && y == 1);
```

Ausdrücke vereinfachen

- Wir hatten einen recht langen Ausdruck:

```
z = (x >= 0 && y <= 0) || (x >= 0 && !(y <= 0) && y == 1);
```

- Zunächst ist `!(y <= 0)` redundant, wenn `y == 1` gilt:

```
z = (x >= 0 && y <= 0) || (x >= 0 && y == 1);
```

- In beiden Fällen muss `x >= 0` gelten:

```
z = (x >= 0) && (y <= 0 || y == 1);
```

- Nun ist `y` ein `int`, zwischen `0` und `1` ist nichts:

```
z = (x >= 0) && (y <= 1);
```

Welche dieser Klammern sind optional?

Ein Mini-Kommentar über die Korrektheit

› Das hatte quasi jeder und die Mehrheit hat Recht!

```
if(x >= 0){
   if(y <= 0) {
      z = true;
   } else {
      if(y == 1) {
         z = true;
      }
   }
} else {
   z = false;
}
```

Sei $x \geq 0$ und $y > 1$. Welchen Wert hat z?

- Na ja, den Alten!

  ```
  z = x >= 0 && (y <= 1 || z);
  ```

- Das Problem? Was, wenn z davor nicht initialisiert wurde?
- Dann ist das ohne Kontrollstrukturen unmöglich
- Korrekt wäre: **die Aufgabe ist** (mit EidI-Wissen) **unmöglich**

Genau genommen kann man für eine spezifische Version und Programmstruktur. . .

Aufgabe 2: Schleifen

Erstellen Sie die Java Datei `Schleifen.java` und implementieren Sie die folgenden Teilaufgaben innerhalb der `main` Methode (sog. Programmeinstiegspunkt) dieser Datei.

a) *Lesen Sie über die Kommandozeilenparameter eine Variable vom Typ `int` ein, die wir im Folgenden mit `n` bezeichnen. Implementieren Sie anschließend die folgenden Schleifen, deren Verhalten von `n` abhängen soll.*

b) *Implementieren Sie eine `for`-Schleife, die alle ganzen Zahlen von `n` bis `1` durchläuft und diese ausgibt.* Beispiel: *`n = 5` gibt `5 4 3 2 1` aus.*

c) *Implementieren Sie eine `do-while-Schleife`, die alle ungeraden ganzen Zahlen von `1` bis einschließlich `n` ausgibt.* Beispiel: *`n = 14` gibt `1 3 5 7 9 11 13` aus und `n = 5` gibt `1 3 5` aus.*

Making Loops

- Wir haben dies bereits letzte Woche analysiert, deswegen hier ein wenig schneller.

```java
int n = Integer.parseInt(args[0]);

for (int i = n; i >= 1; i--) {
   System.out.println(i); // oder System.out.print(...)
}

if(n >= 1) {
   int j = 1;
   do {
      if (j % 2 == 1) // ungerade
         System.out.println(j);
      j = j + 1;
   } while (j <= n);
}
```

Schleifen.java

Aufgabe 3: Algorithmen in Java implementieren

In dieser Aufgabe sollen Sie einen vorgegebenen Algorithmus in ein identisches Java Programm umwandeln. Achten Sie insbesondere auf eine passende Auswahl der Datentypen der Variablen und die Art der Schleife. Legen Sie dazu eine Java Datei namens `Quersumme.java` an und implementieren Sie den folgenden Algorithmus in der `main` Methode.

Eingabe: *Ganze Zahl* zahl

```
01: falls zahl < 0 :
02:     Gebe „Eingabe ungültig!“ aus
03:     beende Algorithmus vorzeitig
04: Setze quersumme = 0
05: Setze divRest = 0
06: solange zahl > 0 :
07:     divRest = zahl % 10
08:     quersumme = quersumme + divRest
09:     zahl = ganzzahligAbrunden(zahl / 10)
10: Gebe quersumme aus
11: ende
```

- Zunächst das Grundgerüst
- Doch welche Datentypen?
- Hier genügt `int` für alles.
- Abrunden? `int` is 'nuff

```
public static void main(String[] args) {
   int zahl = Integer.parseInt(args[0]);
}
```

Eine Implementation übernehmen

› Abseits der Datentypentscheidung ist die Java-Syntax das größte Hindernis:

```
   int zahl = Integer.parseInt(args[0]);
01: if (zahl < 0) {
02:     System.out.println("Eingabe ungültig!");
03:     return;
    }

04: int quersumme = 0;
05: int divRest = 0;
06: while (zahl > 0) {
07:     divRest = zahl % 10;
08:     quersumme = quersumme + divRest;
09:     zahl = zahl / 10;
    }
10: System.out.println(quersumme);
```

Alternativ auch `System.err.println`

Initialisierung notwendig? Ja, wegen „08“

Initialisierung notwendig? Nein, da „07“ überschreibt

Ganzzahldivision schneidet Nachkommastellen ab

```
Eingabe: Ganze Zahl zahl
falls zahl < 0 :
    Gebe „Eingabe ungültig!“ aus
    beende Algorithmus vorzeitig
Setze quersumme = 0
Setze divRest = 0
solange zahl > 0 :
    divRest = zahl % 10
    quersumme = quersumme + divRest
    zahl = ganzzahligAbrunden(zahl / 10)
Gebe quersumme aus
ende
```

Quersumme.java

Aufgabe 4: Iterative Wurzelberechnung

In dieser Aufgabe sollen Sie einen Algorithmus der als Berechnungsvorschrift gegeben ist implementieren. Im Gegensatz zu Aufgabe 3 ist der Algorithmus also nicht vollständig entwickelt sondern nur in der einfachsten Form angegeben. Das Heron-Verfahren ist ein numerisches Verfahren zur Approximation der k*-ten (*$k \in \mathbb{N}, k > 1$*) Wurzel einer Zahl* $\sqrt[k]{x} \in \mathbb{R}$, $x > 0$, *d.h. wir suchen* $y = \sqrt[k]{x}$.
Wir beginnen mit dem Startwert

$$y_0 = x$$

und fahren für $n \geq 1$ *nach der folgenden Iterationsvorschrift fort:*

$$y_{n+1} = \frac{(k-1)y_n^k + x}{k \cdot y_n^{k-1}}$$

wobei y_n *die Approximation nach* n *Berechnungsschritten sei. Da einige Wurzelzahlen (z.B.* $\sqrt{2}$*) nur näherungsweise dargestellt werden können, brechen wir ab, wenn* $|y_{n+1} - y_n| < 10^{-8}$ *erfüllt ist.*
Erstellen Sie eine Java Datei namens `Heron.java` *und implementieren Sie dieses Verfahren wie oben beschrieben. Lesen Sie die Werte für* k *und* x *über die Kommandozeilenparameter ein.*

Iterative Berechnung

```
public static void main(String[] args) {
   double x = Double.parseDouble(args[0]);
   int k = Integer.parseInt(args[1]);

   if (x <= 0 || k <= 1) {
      System.err.println("Eingaben ungültig!");
      return;
   }

   double y = x;
   ...
}
```

Eingabe: $k \in \mathbb{N}$, $k > 1$ *und* $x \in \mathbb{R}$, $x > 0$

Mit $y_0 = x$, $y_{n+1} = \frac{(k-1)y_n^k + x}{k \cdot y_n^{k-1}}$

Abbruch wenn $|y_{n+1} - y_n| < 10^{-8}$

Da wir für die Abbruchbedingung ein y_{n+1} *benötigen, eignet sich do-while*

Iterative Berechnung

Eingabe: $k \in \mathbb{N}$, $k > 1$ *und* $x \in \mathbb{R}$, $x > 0$

Mit $y_0 = x$, $y_{n+1} = \frac{(k-1)y_n^k + x}{k \cdot y_n^{k-1}}$

Abbruch wenn $|y_{n+1} - y_n| < 10^{-8}$

Da wir für die Abbruchbedingung ein y_{n+1} *benötigen, eignet sich do-while*

```
public static void main(String[] args) {
    double x; int k; ...
    double y = x;

    double oldy;
    do {
        oldY = y;
        y = ((k-1) * Math.pow(y, k) + x) / (k * Math.pow(y, k-1));
    } while (Math.abs(oldY - y) > 1E-8);

    System.out.println(y);
}
```

Zusammenfassend

- Arrays sind komplexe Datentypen, die auf dem Heap verwaltet werden.
  - Damit können sie `null` sein (keine Referenz auf den Heap)
  - Übergeben wir ein Array einer Methode, wird die Referenz (Stack) kopiert (→ selbes Array)
- Es gibt viele kleine Tricks und Kniffe, die man primär durch (selbst) ausprobieren lernt
- Eure Lösungen und Ansätze werden sich wahrscheinlich immer mehr unterscheiden
  - Je mehr Mühe ihr euch gebt, desto ausführlicher das individuelle Feedback
  - Formuliert gerne auch Fragen/Ideen/… und kommentiert fleißig

Blatt 4
Schleifen, Arrays, Methoden

Präsenzaufgabe

5

Flip-em till 'ya hit-em

Implementieren Sie beide Methoden, welche alle im übergebenen `char`-Array enthaltenen Klein- zu Großbuchstaben, und alle Groß- zu Kleinbuchstaben machen.

```
public class Praesenzaufgabe {
    public static void
        flipInPlace(char[] c) { /* TODO */ }
    public static char[]
        flipInCopy(char[] c) { /* TODO */ }
    public static void main(String[] args) { ... }
}
```

A–Z ≙ 65–90
a–z ≙ 97–122

`flipInPlace` soll diese Änderungen direkt im übergebenen Array vornehmen, `flipInCopy` soll auf einer Kopie des Arrays arbeiten und diese zurückgeben. Überlegen Sie sich, welche Vor- und Nachteile die jeweiligen Implementierungen haben.

Flip in Place with Java-Magic

```java
public static void flipInPlace(char[] flipBuchstaben) {
    int shift = 'a' - 'A';
    for(int i = 0; i < flipBuchstaben.length; i++) {
        if (flipBuchstaben[i] >= 'A' && flipBuchstaben[i] <= 'Z') {
            flipBuchstaben[i] += shift;
        } else if (flipBuchstaben[i] >= 'a' && flipBuchstaben[i] <= 'z') {
            flipBuchstaben[i] -= shift;
        }
    }
}
```

Implizite Typkonvertierung!
char → int

Flip in Copy — Custom Mode

Ich kannte `clone` gaaar niiischt. Unnu? Ist das nicht böse?

```
public static char[] flipInCopy(char[] flipBuchstaben) {
   char[] arrayKopie = new char[flipBuchstaben.length];
   for(int i = 0; i < arrayKopie.length; i++)
      arrayKopie[i] = flipBuchstaben[i];

   flipInPlace(arrayKopie);
   return arrayKopie;
}
```

Werte primitiver Datentypen werden bei Zuweisung kopiert. *Praesenzaufgabe.java*, *PraesenzaufgabeNaiv.java*

Allgemeines

- *Überlegen Sie sich, welche Vor- und Nachteile die jeweiligen Implementierungen haben.*
  - Vorteile einer Kopie (gegenüber in-place): Nebeneffekte von Methoden sind generell schlecht! „out-Parameter“ sollten vermieden werden (wo nicht unbedingt notwendig).
  - Nachteile einer Kopie (gegenüber in-place): Der benötigte Speicher wird verdoppelt (was, wenn das Array riesig ist?)

- Was ist nun besser?
  - In der Regel die Variante mit Kopie. Die unerwarteten/ungewollten Seiteneffekte können Kaskaden schwer zu findender Fehler verursachen.
  - Sollten wirklich große Arrays erwartet werden, sollte man sich unter Umständen ein grundlegend anderes System überlegen.

Alternativen?

- Hätte man hier auch „for-each“ für flip in place nutzen können?

```
for(int i = 0; i < flipBuchstaben.length; i++) {
   if ('A' <= flipBuchstaben[i] && flipBuchstaben[i] <= 'Z') {
      flipBuchstaben[i] += shift;
   } else if ('a' <= flipBuchstaben[i] && flipBuchstaben[i] <= 'z') {
      flipBuchstaben[i] -= shift;
   }
}
```

- Nein, so könnten wir nicht mehr `flipBuchstaben` an der Stelle `i` verändern.
- Mit `for(char c : flipBuchstaben)` ist c jeweils eine Kopie (primitiver Datentyp) und nicht mehr eine Referenz auf die Stelle im Array!


clone

Aufgabe 1: Referenzvariablen

Betrachten Sie den folgenden Java Code (Referenzvariablen.java):

```
1  public class Referenzvariablen {
2    public static void setzeElementVielleicht(
         int[] array, int idx, int ele) {
3      if (array == null || idx >= array.length
           || ele <= 0) {
4        System.out.println("Eingabe ungültig!");
5        return;
6      }
7      array[idx] = ele;
8      ele = 4;
9      array = null;
10   }
11
12   public static void main(String[] args) {
13     int ele = 1;
14     int idx = ele;
15     ele = 2;
16     int[] array = new int[3];
17     setzeElementVielleicht(array, idx, ele);
18     System.out.println(ele);
19     System.out.println(idx);
20     if (array != null) {
21       for (int wert : array) {
22         System.out.println(wert);
23       }
24     } else {
25       System.out.println("Array ungültig!");
26     }
27   }
28 }
```

a) *Welche Ausgaben erzeugen die Zeilen 18 und 19? Begründen Sie Ihre Antwort.*

b) *Welche Ausgaben erzeugen die Zeilen 20–26? Begründen Sie Ihre Antwort.*

Ein schneller Blick auf knappe Antworten

a) *Welche Ausgaben erzeugen die Zeilen 18 und 19?*

Z18 liefert 2, und *Z19* 1.

Zuerst die Ausgabe von `ele`, dieses wird in *Z13* mit 1 initialisiert und in *Z15* mit 2 überschrieben. Der Parameter `ele` in *Z2* besitzt einen anderen Sichtbarkeits- und Gültigkeitsbereich, zudem wird der Wert durch *Call-by-Value* kopiert. Damit hat das Unterprogramm keinen Einfluss auf den wert in `ele` aus `main`. Auf `ele` findet kein weiterer Zugriff statt, es ist 2.

Analog argumentiert es sich für `idx`, welches in *Z14* mit dem dortigen Wert von `ele` (ist 1) initialisiert wird (da es ein primitiver Datentyp ist, wird hier auch mit *Call-by-Value* der Wert kopiert). Anschließend findet kein verändernder Zugriff statt und `idx` ist 1.

b) *Welche Ausgaben erzeugen die Zeilen 20–26?*

Sie erzeugen „`0\n 2\n 0`“ (wobei \n ein neue Zeile darstellt)

a) *Welche Ausgaben erzeugen die Zeilen 18 und 19?*

Z18 liefert 2, und *Z19* 1.

b) *Welche Ausgaben erzeugen die Zeilen 20–26?*

Sie erzeugen „0\n2\n0“ (wobei \n ein neue Zeile darstellt)

Die Zielen befassen sich mit der Ausgabe von `array`. Dieses wird in *Z16* Java initialisiert hier direkt mit `{0, 0, 0}`. Das Unterprogramm erfüllt nun in *Z2* die Referenz zu `array` als Kopie (das ist *Call-by-Value*, verhält sich aber wie *Call-by-Reference*), das **`if`** in *Z3* scheitert, da `array` auf `{0, 0, 0}` verweist und `idx = 1`, `ele = 2` (aus vorheriger Antwort). *Z7* ändert damit das gemeinsame Array zu `{0, 2, 0}` (Seiteneffekt), die Neuzuweisung in *Z9* überschreibt lediglich die in den Parameter kopierte Referenz. Damit löst das **`if`** in *Z20* aus und es erfolgt besagte Ausgabe.

Aufgabe 2: Mehrdimensionale Arrays

Im Folgenden sollen Methoden zum Verarbeiten von mehrdimensionalen Arrays (Matrizen) implementiert werden. Legen Sie dazu eine Java Datei `Matrix.java` *an. Im Folgenden sei* n *die Anzahl der Zeilen,* m *die Anzahl der Spalten,* i *und* j *die jeweiligen Indizes, sowie* a_{ij} *das Element in der* i*-ten Zeile und* j*-ten Spalte. Die Zeilensummennorm ist in der Mathematik die von der Summennorm abgeleitete natürliche Matrixnorm. Die Zeilensummennorm* $\|A\|_\infty$ *einer Matrix* A *entspricht der maximalen Betragssumme aller ihrer Zeilen:*

$$\|A\|_\infty = \max_{i=1,\dots,n} \sum_{j=1}^{n} |a_{ij}|$$

Beispiel: *Die Zeilensummennorm der Matrix* $A = \begin{pmatrix} 1 & -2 & -3 \\ 2 & 4 & -1 \end{pmatrix}$ *is gegeben durch* $\|A\|_\infty = \max\{|1| + |-2| + |-3|,\ |2| + |4| + |-1|\} = \max\{6, 7\} = 7.$

a) *Legen Sie ein 2-D Array mit* 3×3 *Elementen vom Typ double an und füllen Sie es mit gültigen Werten.*

b) *Implementieren Sie eine Methode* `public static double berechneBetragssumme(double[] arr)`*, die die Betragsnorm von* `arr`*, d.h. die Summe der Beträge der Elemente, berechnet und zurückgibt. Für ungültige Eingaben (*`arr` *ist* `null` *oder hat die Länge* 0*) soll* `-1.0` *zurückgegeben werden.*

c) *Implementieren Sie eine Methode* `public static double berechneZeilensummennorm(double[][] mat)`*, die ein 2-D Array als Argument annimmt, die Zeilensummennorm berechnet und diese zurückgibt. Für ungültige Eingaben (mat ist null oder hat die Länge* 0*) soll* `-1.0` *zurückgegeben werden.*

d) *Rufen Sie die Methode aus Teilaufgabe 3 mit ihrer Matrix aus und geben Sie das Ergebnis auf die Konsole aus.*

a) Ein Array Anlegen

- Hier mal etwas beliebiges

```
double[][] matrix = { { 1.5, -2, 1 }, { -5, 1, 6.1 }, { -2, -4, 1 } };
```

- Hier haben wir übrigens — Java sei dank — stark abgekürzt:

```
double[][] matrix = new double[][] { new double[] { ... }, ... };
```

- Quiz! Sind diese Statements gültig? Wenn ja, welchen Wert hat `m1`, `m2`, …? (Warum?)

```
double[][] m1;
double[][] m2 = new double[][];
double[][] m3 = new double[2][];
int[][]    m4 = new int[1][3];
double[][] m5 = { null, {}, { null } };
```

✔ *`m1` ist noch nicht initialisiert*

↯ *Java* muss *das Array initialisieren → braucht dessen Größe*

✔ *`m3` ist `{ null, null }` (Default-Wert für `double[]`)*

✔ *`m4` ist `{ { 0, 0, 0 } }` (Default-Wert für `int`)*

↯ *`null` und `{}` (leer) sind valide `double[]`, `{ null }` aber nicht, da `double` nicht `null` sein kann.*

b) berechneBetragssumme(double[])

```
public static double berechneBetragssumme(double[] arr) {
   if (arr == null || arr.length == 0) {
      System.out.println("Eingabe ungültig!");
      return -1.0;
   }
   double norm = 0;
   for (double element : arr) {
      norm = norm + Math.abs(element);
   }
   return norm;
}
```

c) berechneZeilensummennorm(double[][])

```java
public static double berechneZeilensummennorm(double[][] mat) {
    if (mat == null || mat.length == 0) {
        System.out.println("Eingabe ungültig!");
        return -1.0;
    }
    double max = Double.MIN_VALUE;
    for (double[] array : mat) {
        double norm = berechneBetragssumme(array);
        if (norm == -1.0) return -1.0; // Fehler in Unteralgorithmus?
        else if (norm > max) max = norm;
    }
    return max;
}
```

d) Ausgabe

- Ein super kompliziertes Aufrufchen (für zwei Punkte!):
  `System.out.println(berechneZeilensummennorm(matrix));`

- Der gesamte Code findet sich in der Datei Matrix.java

Aufgabe 3: Numerische Integration

In dieser Aufgabe sollen Sie ein einfaches Verfahren zur numerischen Integration implementieren. Legen Sie dazu eine Java Datei namens `NumerischeIntegration.java` *an und implementieren Sie alle folgenden Teilaufgaben innerhalb dieser Datei. Die Funktion* $f(x) = \exp(-x^2)$ *besitzt keine Stammfunktion bestehend nur aus elementaren Funktionen, was eine Berechnung schwierig macht. Man ist jedoch an guten Approximationen interessiert. Wir beschränken uns auf das Intervall* $[0, 1]$*, suchen also möglichst genaue Approximationen für das folgende Integral:* $A_f = \int_0^1 \exp(-x^2)dx$

Eine bekannte Methode ist die Approximation des Integrals durch eine Summe von Trapezflächen. Dazu zerlegt man das Intervall $[0, 1]$ *in* $n \geq 1$ *Segmente* $[x_i, x_{i+1}]$ *mit* $0 = x_1 < x_2 < \ldots < x_{n-1} < x_n = 1$.

Die Fläche auf dem Intervall $[x_i, x_{i+1}]$ *erhält man dann durch ein Trapez mit Eckpunkten* $a = (x_i, 0)$, $b = (x_{i+1}, 0)$, $c = (c_{i+x}, f(x_{i+1}))$ *und* $d = (x_i, f(x_i))$*. Die Fläche* A_i *eines derartigen Trapezes kann für* $x_{i+1} \geq x_i$ *durch* $A_i = (x_{i+1} - x_i)(f(x_i) + f(x_{i+1}))/2$ *berechnet werden. Die Approximation des Integrals ergibt sich dann durch Summierung der einzelnen Trapezflächen:* $A_f \approx A_f(n) = \sum_{i=1}^{n} A_i$.

a) Daten Einlesen und b) `berechneF(double)`

- Das Einlesen erfolgt hier leicht:
  ```
  int n = Integer.parseInt(args[0]);
  ```
- Ein Scanner ist hier theoretisch auch möglich
  - Den wollen wir dann aber auch immer schließen!
  - Uns reichen Kommandozeilenparameter
- Und nun noch berechnen ($f(x) = \exp(-x^2)$):
  ```
  public static double berechneF(double x) {
     return Math.exp(-(x*x));
  }
  ```
- Natürlich kann man hier auch `Math.pow(x, 2.0)` oder so benutzen

c) berechneTrapezFlaeche(double, double)

- Ein ungültiger Eckpunkt liegt vor, wenn $x_2 < x_1$:

```
public static double berechneTrapezFlaeche(double x1, double x2) {
   if(x2 < x1)
      return 0;
   else
      return (x2 - x1) * ((berechneF(x1) + berechneF(x2)) / 2.0);
}
```

- Auch hier handelt es sich um unspektakuläres übernehmen der Aufgabenstellung

d) `trapezVerfahren(int)`

- Wir können hier über die Schritte (i < n) oder über x terminieren
  Das Verhalten Umsetzung unterscheidet sich bei Rundungsfehlern!

```
public static double trapezVerfahren(int n) {
    if (n < 1) {
        System.err.println("Eingabe ungültig");
        return -1.0;
    }
    ...
}
```

d) trapezVerfahren(int)


```
public static double trapezVerfahren(int n) {
    ...
    double schrittweite = 1.0 / n;
    double x = 0.0;
    double A = 0.0;
    while (x <= 1.0 - schrittweite) {
        A = A + berechneTrapezFlaeche(x, x + schrittweite);
        x = x + schrittweite;
    }
    return A;
}
```

Ginge hier auch nur 1/n?

Nö, dann wäre das mit $n > 1$ immer 0 (da Ganzzahldivison).

Auch möglich wäre soetwas:

```
for(double x = 0; x <= 1 - schrittweite; x += schrittweite) {
    ...
}
```

e) Ausgabe

- Wieder eine fancy Ausgabe!
  ```
  System.out.println(trapezVerfahren(n));
  ```

- Der gesamte Code befindet sich hier: NumerischeIntegration.java

Zusammenfassend

- In manchen Fällen *muss* Java initialisieren (Arrays!)
  - Ganzzahlen werden `0`, Fließkommazahlen `0.0` (im jeweiligen Typ, also `0.0f`, …)
  - `boolean` wird `false`, `char` wird `'\0'` (Unicode-„Null“)
  - Referenzdatentypen (wie Arrays) werden **`null`**
- Java unterscheidet Methoden durch ihre Signatur (Name & Parametertypliste)
- Wir haben einen Blick auf Clone(s) geworfen (I've seen the dark side now)
- Auch wenn Heap und Stack nicht Klausurrelevant sind, helfen sie bei Vielem!
  - Call-by-Value und Call-by-Reference (Seiteneffekte, …)
  - Was genau schützt **`final`**, …

Blatt 5
Methoden und OOP

Präsenzaufgabe

6 — **Show me ya' potency**

1. Implementieren Sie eine Klasse `Potenz` für Potenzen. Die Klasse soll zwei private Attribute `basis` und `potenz` besitzen, sowie einen Konstruktor mit zwei Argumenten definieren, der den Attributen Anfangswerte zuweist. Zusätzlich soll es getter und setter für die Attribute geben. Die Klasse soll eine öffentliche Methode besitzen, die die Attribute der Instanz auf die Konsole ausgibt.
2. Legen Sie (im selben Ordner) eine zweite Java Datei namens `PotenzMain.java` an, die als Programmeinstiegspunkt dienen soll, d.h. hier ist die `main` Methode implementiert. Instanziieren Sie innerhalb der `main` Methode ein Objekt der Klasse `Potenz` und lassen Sie sich die Attribute des Objekts anzeigen.

```
javac Potenz.java PotenzMain.java
java PotenzMain
```

Ich bin Herbert Klassenzüchter!

Implementieren Sie eine **Klasse** `Potenz` **für Potenzen**. *Die Klasse soll* **zwei private Attribute** `basis` **und** `potenz` *besitzen, sowie einen* **Konstruktor mit zwei Argumenten** *definieren, der den* **Attributen Anfangswerte zuweist**.

Zusätzlich soll es **getter** *und* **setter** *für die Attribute geben. Die Klasse soll eine* **öffentliche Methode** *besitzen, die die Attribute der Instanz auf die Konsole ausgibt.*

Die Typen von `basis` und `potenz` sind hier sinnvoll, aber frei gewählt.

```java
public class Potenz {
  private double basis;
  private int potenz;

  public Potenz(double b, int e) {
    this.basis = b;
    this.potenz = e;
  }
  public double getBasis() { return this.basis; }
  public int getPotenz() { return this.potenz; }
  public void setBasis(double b) { this.basis = b; }
  public void setPotenz(int e) { this.potenz = e; }

  public void print() {
    System.out.println(basis + "^(" + potenz + ")");
  }
}
```

Potenz.java

Major Main

Legen Sie eine **zweite Java Datei** *namens* `PotenzMain.java` *an, die als Programmeinstiegspunkt dienen soll, d.h. hier ist die* `main` **Methode** *implementiert.*

Instanziieren Sie innerhalb der `main` *Methode* **ein Objekt der Klasse** `Potenz` *und lassen Sie sich die* **Attribute des Objekts anzeigen**.

Potenz.java

```
public class Potenz {
    public Potenz(double b, int e) { ... }
    ...
    public void print() { ... }
}
```

PotenzMain.java

```
public class PotenzMain {
    ▷ display in browser
    public static void main(String[] args) {
        Potenz p = new Potenz(2.0, 3);
        p.print(); // → 2.0^(3)
    }
}
```

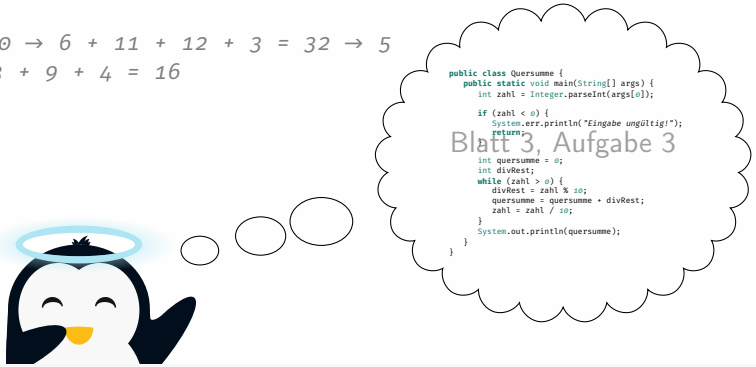
PotenzMain.java

Aufgabe 1: Methoden mit einer variablen Parameterzahl

Legen Sie eine Java Datei namens Quersummen.java und implementieren Sie die folgende Aufgabe innerhalb dieser Datei als eine öffentliche statische Methode namens `quersummeVonQuersummen`. Die Methode soll eine beliebige Anzahl an Parametern vom Typ `int` erwarten und für jede dieser Zahlen die Quersumme berechnen und diese aufsummieren. Zusätzlich soll eine weitere boolean Parameter angeben, ob aus der Summe wiederum die Quersumme berechnet werden soll. Testen Sie Ihre Implementierung mit den angebenden Beispielen.

Beispiele:

`Ja, 123, 92, 57, 30 → 6 + 11 + 12 + 3 = 32 → 5`
`Nein, 12, 9, 4 → 3 + 9 + 4 = 16`



Meta-Quersummen

› Wir sollen mehrere Quersummen berechnen. Wir haben schon ein Programm für eine:

```java
public static void main(String[] args) {
  int zahl = Integer.parseInt(args[0]);
  if (zahl < 0) {
    System.err.println("Eingabe doof!");
    return;
  }
  int quersumme = 0;
  int divRest;
  while (zahl > 0) {
    divRest = zahl % 10;
    quersumme += divRest;
    zahl = zahl / 10;
  }
  System.out.println(quersumme);
}
```

Ein allgemeinerer Eingabeparameter

Rückgabetyp!

Optionale Vereinfachung

Ordentliche Rückgabe

→

```java
public static int quersumme(int zahl) {
  if(zahl < 0){
    System.err.println("Eingabe doof!");
    return 0;
  }
  int quersumme = 0;
  while(zahl > 0){
    quersumme += zahl % 10;
    zahl = zahl / 10;
  }
  return quersumme;
}
```

It is always a great time to be a good boy or girl scout!

Beliebige Quersummen brummen

- Die beliebige Anzahl ints schaffen wir mit varargs:

```
public class Quersummen {
   public static int quersumme(int zahl) { ... }


   public static int quersummeVonQuersummen(boolean sum2, int... zahlen) {
      int quersummenSumme = 0;
      for(int zahl : zahlen)
         quersummenSumme += quersumme(zahl);

      return sum2 ? quersumme(quersummenSumme) : quersummenSumme;
   }
}
```

Hier nur ein kurzer Name, damit es auf die Folie passt

Signatur?
quersummeVonQuersummen(boolean, int...)

Ich bin die Biene!

- Jetzt fehlt noch die Methode zum Testen:

```java
public class Quersummen {
    public static int quersumme(int zahl) { ... }
    public static int quersummeVonQuersummen(boolean sum2, int... zahlen)
        { ... }


    public static void main(String[] args) {
        System.out.println(quersummeVonQuersummen(true, 123, 92, 57, 30));
        System.out.println(quersummeVonQuersummen(false, 12, 9, 4));
    }
}
```

Quersummen.java

Aufgabe 2: Globale Variablen

Legen Sie eine Java Datei namens `CharRotation.java` an. Innerhalb dieser Datei sollen Sie die folgende Aufgabe implementieren.

Implementieren Sie eine Methode namens `rotiereCharacterArray`, die ein char Array als Parameter erwartet und innerhalb dieses Arrays (in place) alle Klein- sowie Großbuchstaben um n Stellen zyklisch und alphabetisch verschiebt. Legen Sie n als statische globale Konstante an. Dabei sollen Klein- und Großbuchstaben erhalten bleiben. Testen Sie Ihre Implementierung mit mindestens einem Beispiel.

Beispiel:
`n = 3, {'a', 'Z'} → {'d', 'C'}`

Implizite Typkovertierung und Unterprogramme können hier sehr hilfreich sein.

Problemreduktion

- Wir reduzieren das Problem, alle Zeichen zu verschieben, zunächst auf ein Zeichen
- Bei Hilfsmethoden stellt sich die Frage, ob sie semantisch alleine sinnvoll sind
  - Ist es sinnvoll nur ein einziges Zeichen zu rotieren? Ja
  - Benötigen wir kontextabhängige Informationen? Nicht wirklich (n ist konstant)
  - Haben wir implizite Annahmen die gelten müssen? Auch nicht
  - In dem Fall empfiehlt sich **`public`**, sonst eher **`private`**

```
public static char rotiereCharacter(char c){
  // Denglish 4 Leben!
}
```

- Nun prüfen wir weiter ob es ein Großbuchstaben, ein Kleinbuchstaben oder ein sonstiges Zeichen ist.

Hilfsmethodenfreuden

```java
public static char rotiereCharacter(char c){
   if (isLowercase(c)) {
      return addCyclicOnBase('a', c);
   } else if (isUppercase(c)) {
      return addCyclicOnBase('A', c);
   } else {
      return c;
   }
}
```

Für Kleinbuchstaben ändert sich die Normalisierung

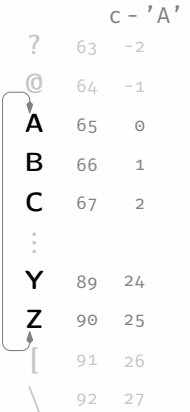
```java
private static char addCyclicOnBase(char base, char c) {
   return (char) (base + Math.floorMod(c - base + n, 26));
}
```

Javas '%' beachtet das Vorzeichen!

Wrap-Around mit Modulo!

Wir normalisieren, um nicht die 65 durch das Modulo zu verlieren.

Magic-Number (Alphabeeet)




```java
private static boolean isLowercase(char c) { return c >= 'a' && c <= 'z'; }
private static boolean isUppercase(char c) { return c >= 'A' && c <= 'Z'; }
```

Ein Beispiel zum Abschluss

- Nun verbleibt es jedes Zeichen im Array zu verschieben:

```
public static void rotiereCharacterArray(char[] arr) {
   for (int i = 0; i < arr.length; i++) {
      arr[i] = rotiereCharacter(arr[i]);
   }
}
```

- Nun gilt es noch aufzurufen:

```
public class CharRotation {
   public static void main(String[] args) {
      char[] arr = { 'a', 'Z' };
      rotiereCharacterArray(arr);
      System.out.println(arr);   System.out.println(char[])
   }
}
```

CharRotation.java

Aufgabe 3: Objektorientierung: Klassenentwurf

In der Vorlesung haben Sie die Grundlagen der Objektorientierung kennengelernt und anhand zweier Beispiele (Auto und Nachttischlampe auf Folie ~~13~~ 15, Kapitel ~~6~~ 7) einen Klassenentwurf gesehen. Entwerfen Sie nach dem in der Vorlesung vorgestellten Muster die folgenden Klassen (es ist noch keine Java Implementierung notwendig und bitte geben Sie auch keine für diese Aufgabe ab).

a) *Entwerfen Sie eine Klasse, die einen* `Artikel` *in einem Online Shop repräsentieren soll. Überlegen Sie sich hierzu, welche Eigenschaften (Attribute) und Funktionen (Methoden) diese Klasse besitzen soll. Geben Sie Datentypen der Attribute so an: „Attributname: Datentyp". Analog dazu können Sie die Rückgabetypen von Methoden angeben.*

b) *Geben Sie eine Instanz der Klasse* `Artikel` *an.*

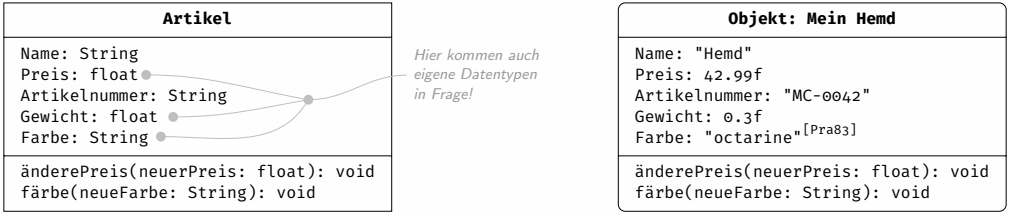
c) *Als nächstes soll nun die Klasse* `Warenkorb` *entworfen werden. Überlegen Sie sich hierzu insbesondere, wie die* `Artikel` *in einem* `Warenkorb` *modelliert werden können.*

d) *Geben Sie eine Instanz der Klasse* `Warenkorb` *an.*

Ein Artikel, der die Weld verändern wird!

a) *Entwerfen Sie eine Klasse, die einen* `Artikel` *in einem Online Shop repräsentieren soll. Überlegen Sie sich hierzu, welche Eigenschaften (Attribute) und Funktionen (Methoden) diese Klasse besitzen soll.*

b) *Geben Sie eine Instanz der Klasse* `Artikel` *an.*



Manchmal werden im Objektdiagramm auch keine Methoden wiederholt oder Methoden angepasst. Sichtbarkeiten haben wir hier vernachlässigt.

Shopping Spreeeeeee

c) *Als nächstes soll nun die Klasse `Warenkorb` entworfen werden. Überlegen Sie sich hierzu insbesondere, wie die `Artikel` in einem `Warenkorb` modelliert werden können.*

d) *Geben Sie eine Instanz der Klasse `Warenkorb` an.*

Warenkorb
`Kundennummer: int` `Artikelliste: Artikel[]`

`fügeArtikelHinzu(Artikel: Artikel): void`
`entferneArtikel(index: int): boolean`
`berechneGesamtpreis(): float`

Objekt: Mein Warenkorb
`Kundennummer: 744392` `Artikelliste: [`*`Mein Hemd`*`]`

`fügeArtikelHinzu(Artikel: Artikel): void`
`entferneArtikel(index: int): Artikel`
`berechneGesamtpreis(): float`

- Es ist nicht unbedingt notwendig, mit *Main Hemd*, ein Objekt zu referenzieren
  - Je nach Problem, reicht vielleicht auch einfach die Artikelnummer, da sie eindeutig ist.
  - Man kann das Objekt auch an Ort und Stelle „schreiben“
    Dann mit meist in JSON-(ähnlicher)-Syntax `{Name: "Hemd ", Preis: 42.99f, ...}`

Pinguineeee — Struktur

Penguin.java

Muss gleich heißen wie Datei (Groß- und Kleinschreibung beachten)!

```
public class Penguin {
  private final int age;
  final String name;

  public Penguin(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public int getAge() { return age; }
  public String toString() {
    return name + " watschelt seit " + age + " Jahr(en)";
  }
}
```

Die Pinguin-Klasse!

Attribute der Klasse. Sie definieren den ***Zustand***.

Muss gleich heißen wie die Klasse! Er ist damit auch keine Methode

Der Konstruktor liefert den ***initialen Zustand***

Die Parameter (können beliebig heißen)

`new Penguin("Piep", 3);` ✔ `new Penguin();` ϟ

Der leere default-Konstruktor existiert nur genau dann, wenn kein expliziter existiert. Hier existiert `Penguin(String, int)`.

Ein stinknormaler Getter

Methoden der Klasse. Sie definieren das ***Verhalten***.

`toString` ist in Java „besonders“ (durch `Object`). Es wird z.B. bei (String-)Konkatenation automatisch aufgerufen.

Pinguineeee — Gültigkeit & Sichtbarkeit

Penguin.java

```
public class Penguin {
  private final int age;
  final String name;

  public Penguin(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public int getAge() { return age; }
  public String toString() {
    return name + " watschelt seit " + age + " Jahr(en)";
  }
}
```

Objekte leben, wie mind. eine Referenz auf sie gültig ist

*Die Attribute besitzen die selbe **Gültigkeit**, wie das Objekt, dem sie gehören.*

*Die Parameter **überschatten** (JLS17 6.4) die gleichnamigen Attribute. Wir benötigen nun `this` um auf diese zuzugreifen.*

Das age-Attribut wird hier nicht überschattet.

Beide Attribute werden nicht überschattet!

Gültigkeit

	Selbe Klasse	Selbes Paket	Unterklasse	Überall sonst	Beispiel
public	✔	✔	✔	✔	age
protected	✔	✔	✔		—
⟨nichts⟩	✔	✔			name
private	✔				Penguin

Sichtbarkeitsmodifikatoren

Defaultkonstruktoren und -Werte

- Java erzeugt genau dann einen default-Konstruktor, wenn kein expliziter existiert
- Konstruktoren sind keine Methoden, können aber z.B. überladen werden
  Überladen heißt: gleicher Name, aber andere Signatur (Name & Parametertypliste)
  - Ein Konstruktor kann nur mit `new` aufgerufen werden
  - Er hat keinen Rückgabetyp
  - Er muss genau so heißen wie die Klasse
  - Er kann andere Konstruktoren über `this` aufrufen (erstes Statement)
- Wie bei Arrays, weißt Java Attributen default-Werte zu, wenn wir dies nicht tun:

```
public class Waddler {
    int age;
    float speed;
    char[] directions;
    boolean[][] canWaddleOn;
}
```

```
Waddler w = new Waddler();
w.age          → 0
w.speed        → 0.0f
w.directions   → null
w.canWaddleOn  → null (kein new!)
```

Die Summierung

Hier ist Platz für Heap und Stack.
Oh, schon 2 Uhr.
Das wird eine Tafelnummer

PenguSum.java

```java
public class PenguSum {
  static long sum(Penguin[] ps) { sum(Penguin[])
    long sum = 0;
    for (int i = 0; i < ps.length; i++)
      sum += ps[i].getAge();
    return sum;
  }

  public static void main(String[] args) {
    Penguin[] pengus = { new Penguin("Hugo", 3),
      new Penguin("Sophie", 7) };
    System.out.println(pengus[0]);
    System.out.println(sum(pengus));
  }
}
```

Zusammenfassend

[Pra83] Terry Pratchett. *The Colour of Magic*. 1983

- Javas' Methodensignaturen bestehen aus dem Namen und der Parametertypliste
- Java unterscheidet vier verschiedene Sichtbarkeitsbereiche (von denen drei bisher relevant sind)
  - Java verwendet Pakete (Ordner) zur hierarchischen Verwaltung
  - `public`: überall, nichts: „package-private", `private`: innerhalb der Klasse
- Klassen besitzen einen Zustand (Attribute) und ein Verhalten (Methoden)
  - Der Konstruktor legt den initialen Zustand fest
  - Es gibt nur dann einen default-Konstruktor, wenn kein expliziter vorliegt
  - (nicht-final) Attribute werden von Java mit default-Werten initialisiert

Blatt 6
Grundlagen der OOP

Präsenzaufgabe

7 **Meine Kleine, du hast Palindrom!**

In dieser Aufgabe sollen Sie eine rekursive Methode implementieren, die überprüfen soll, ob es sich bei dem als Parameter übergebenen String um ein Palindrom handelt. Ein Palindrom ist ein Wort, das vorwärts und rückwärts gelesen identisch ist. Die Überprüfung soll nicht case-sensitive sein, d.h. das Wort „Kajak“ soll zum Beispiel ein gültiges Palindrom sein.

Beantworten Sie zudem folgende Fragen bezüglich Ihrer Implementierung:

a) Ist Ihre Implementierung ein linear-rekursiver Algorithmus? Warum?

b) Ist Ihre Implementierung ein end- oder kopf-rekursiver Algorithmus? Warum?

Eine erste Idee

In dieser Aufgabe sollen Sie eine rekursive Methode implementieren, die überprüfen soll, ob es sich bei dem als Parameter übergebenen String um ein Palindrom handelt. Ein Palindrom ist ein Wort, das vorwärts und rückwärts gelesen identisch ist. Die Überprüfung soll nicht case-sensitive sein, d.h. das Wort „Kajak“ soll zum Beispiel ein gültiges Palindrom sein.

Idee:

- Prüfe für i = 0 bis i = $\lfloor s.\texttt{length}/2 \rfloor$, ob „`s[i] == s[s.length - i - 1]`“.
- `String::toLowerCase()` entweder für jeden Vergleich oder einmal per Hilfsmethode.
- Noch einfacher: Anstelle `i` zu inkrementieren, löschen wir das erste und letzte Zeichen nach dem Vergleich (via `String::substring(int, int)`—das Ende ist exklusiv)!

Von Palindromen

Palindrome.java

```java
public static boolean isPalindrome(String s) {  die „Hilfsmethode“
   return isPalindromeRecursive(s.toLowerCase());
}


private static boolean isPalindromeRecursive(String s) {
   if(s.length() < 2)  Basisfall: weniger als zwei Zeichen
      return true;
   else if(s.charAt(0) != s.charAt(s.length() - 1))  Basisfall: kein Palindrom!
      return false;
   else  Rekursionsfall
      return isPalindromeRecursive(s.substring(1, s.length() - 1));
}
```

Ja was isses nun?

a) *Ist Ihre Implementierung ein linear-rekursiver Algorithmus? Warum?*

Unser Verfahren ruft sich *höchstens ein mal selbst auf*, damit ist er linear-rekursiv!

b) *Ist Ihre Implementierung ein end- oder kopf-rekursiver Algorithmus? Warum?*

Der Algorithmus ist End-Rekursiv, da die letzte im Rekursionsfall ausgeführte Operation die Rekursion selbst ist. Damit geschieht im Aufstieg nichts mehr.

Aufgabe 1: Klassenentwurf in Java

- *Erstellen Sie eine Klasse Kreis. Diese soll folgende Eigenschaften und Methoden implementieren:*
  - *Privaten Instanzvariablen für den Radius, Flächeninhalt und Umfang.*
  - *einen öffentlichen Konstruktor, der den Radius des Kreises als Parameter übernimmt, die restlichen Eigenschaften daraus ableitet, und die entsprechenden Variablen initialisiert.*
  - *get- und set-Methoden für die Instanzvariablen. Dabei soll es nur für den Radius eine set-Methode geben und Flächeninhalt und Umfang sollen daraus abgeleitet werden. Stellen Sie sicher, dass nur gültige Werte für den Radius ($r \geq 0$) akzeptiert werden.*
  - *Eine Instanzmethode um die Eigenschaften des Kreises auf der Kommandozeile auszugeben*
  - *Eine Instanzmethode die einen Kreisradius als Parameter übernimmt, den entsprechenden Flächeninhalt berechnet, das zugehörige Attribut aktualisiert und dieses zurückgibt.*
  - *Eine Instanzmethode die einen Kreisradius als Parameter übernimmt, den entsprechenden Umfang berechnet, das zugehörige Attribut aktualisiert und dieses zurückgibt.*
- *Erstellen Sie eine weitere Klasse `KreisMain`, die den Programmeinstiegspunkt implementiert. Lesen Sie über die Kommandozeilenparameter einen Radius ein und instanziieren Sie innerhalb der `main` Methode ein Objekt der Klasse `Kreis` mit dem eingelesenen Radius. Lassen Sie sich die Eigenschaften dieser Instanz anzeigen.*

Oh halt! Die `KreisMain.java`!

KreisMain.java

```java
public class KreisMain {
  public static void main(String[] args) {
    double radius = Double.parseDouble(args[0]);
    Kreis k = new Kreis(radius);
    k.print();
  }
}
```

Aufgabe 2: Gültigkeitsbereiche von Variablen

a) *Betrachten Sie folgende Klassendefinition:*

```
public class DemoA {
   static int x = 0;
   int y = 1;
   public static int methode(int z){
      return this.y + z;
   }
}
```

this bezieht sich auf das Objekt auf dem die Methode aufgerufen wird. Statische Methoden werden aber auf der Klasse aufgerufen! Das gilt natürlich auch ohne das this, da es sich dann immernoch auf eine Instanzvariable bezieht.

Handelt es sich hierbei um eine gültige Java Klassendefinition? Begründen Sie Ihre Antwort kurz.

Nein eine statische Methode kann nicht auf Instanzattribute (hier: y) zugreifen.

Und nun Ausgaben!

b) *Betrachten Sie folgende Klassendefinition:*

```
public class DemoB {
    int x = 0;
    public static void methode(int x){
        System.out.println(x);
    }
    public static void methode2(){
        methode(1);
    }
    public static void main(String[] args){
        methode2();
    }
}
```

Welche Ausgabe erzeugt das Programm? Begründen Sie Ihre Antwort kurz.

Ausgabe: „1\n“. Die `methode2()` ruft `methode(int)` auf, wobei der Parameter `x = 1` die Instanzvariable `x = 0` überschattet.

Um an die Instanzvariable zu kommen, bräuchten wir in `methode(int)` den Ausdruck `this.x`!

Und nochmal Ausgaben!

c) *Betrachten Sie folgende Klassendefinition:*

```
public class DemoC {
   static int i;
   public static void methode(){
      for(; i <= 3; i++){
         System.out.println(i);
      }
   }
   public static void main(String[] args){
      methode();
      methode();
   }
}
```

Welche Ausgabe erzeugt das Programm? Begründen Sie Ihre Antwort kurz.

Ausgabe: „0\n1\n2\n3“. Die Klassenvariable erhält den Default-Wert `0`. Der erste Aufruf von `methode()` erzeugt „0\n1\n2\n3“. Da `i` statisch ist, verbleibt `i = 4` als Seiteneffekt. Die for-Schleife iteriert kein weiteres mal.

Aufgabe 3: Seiteneffekte

Person.java

```
class Person {
  private String name;
  private ValuePair v;
  public Person(String name, int x, int y) {
    this.name = name;
    v = new ValuePair(x,y);
  }
  public String getName() {
    return name;
  }
  public ValuePair getValuePair() {
    return v;
  }
}
```

ValuePair.java

```
class ValuePair {
  int x; int y;
  public ValuePair(int x, int y) {
    this.x = x; this.y = y;
  }
}
```

TestPerson.java

```
class TestPerson {
  ▷ display in browser
  public static void main(String[] args) {
    Person p = new Person("Heinz", 11, 22);
    String n1 = p.getName();
    ValuePair xy1 = p.getValuePair();
    System.out.println(n1 + " " + xy1.x + " " + xy1.y);
    n1 = "Hugo";
    String n2 = p.getName();
    xy1.x = 33;
    xy1.y = 44;
    ValuePair xy2 = p.getValuePair();
    System.out.println(n2 + " " + xy2.x + " " + xy2.y);
  }
}
```

Die Animationen finden sich im Animationsfoliensatz des Tutoriums!

Und wer-wo-wie-was ist das jetzt?

a) Beschreiben Sie kurz, warum der Seiteneffekt auftritt.

Die Methode `Person::getValuePair` gibt nur eine Referenz auf das `ValuePair` v-Attribut zurück. Änderungen an dieser Referenz betreffen das selbe Objekt wie das, auf welches das v zeigt.

Und dafür hat der *Depp* von Autor über sechs Stunden lang nur an der einen Animationsfolie gearbeitet??

Ja!

Wie kriegen wir das weg?

b) *In einigen Fällen sind Seiteneffekte erwünscht oder zumindest toleriert. Hier gehen wir davon aus, dass dies nicht der Fall ist und Sie sollen die Dateien `Person.java`, `ValuePair.java` und `TestPerson.java` so anpassen, dass der Seiteneffekt aus a) nicht mehr auftritt. Schreiben Sie die Begründung für die vorgenommene Änderung als Kommentar in die Datei. Achten Sie darauf, dass die Datei kompilierbar bleibt und auch zur Laufzeit nicht abstürzt, sowie dass die Funktionalität der Klassen erhalten bleibt.*

Zunächst: es reicht nicht einfach die `main`-Methode zu ändern. So wird das Auftreten des prinzipiell Seiteneffekts ja nicht verhindert sondern nur kaschiert. Wir können aber einfach `Person::getValuePair` eine Kopie zurückgeben lassen:

Person.java

```
class Person {
  ...
  public ValuePair getValuePair() {
    return v;
    return new ValuePair(v.x, v.y);
  }
}
```

Jetzt liefert `Person::getValuePair` eine neue Referenz auf ein neues Objekt zurück. Änderungen lassen das Attribut unberührt.

Zusammenfassend

- Rekursive Methoden rufen sich selbst (direkt oder indirekt) wieder auf
  - Geschieht dies maximal ein mal, so ist die Rekursion linear
  - Ist der rekursive Aufruf das letzte Statement, haben wir eine Tail-Rekursion
- Variablen haben Gültigkeits- und Sichtbarkeitsbereiche
  - Die Gültigkeit hängt an der Deklaration (Überschattung, …)
  - Die Sichtbarkeit hängt an `public`, `private`, …
- Seiteneffekte sind potentiell ein großes Problem!

Blatt 7
Grundlagen der Rekursion

Präsenzaufgabe

8 Irgendwo sind meine Socken doch

In der Vorlesung wurde der Binary Algorithmus zur Suche in einer geordneten Struktur vorgestellt.

a) *Geben Sie an, wie das folgende Array vom Suchalgorithmus Binary Search nach dem Schlüssel „4“ durchsucht wird.*

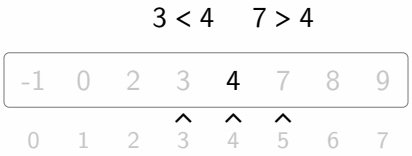
-1	0	2	3	4	7	8	9

b) *Begründen Sie kurz, weshalb der Binary Search Algorithmus eine worst-case Laufzeit von* $T(n)_{max} \in \mathcal{O}(\log n)$ *hat.*

Ja wie suchen wir denn die 4?

$\frac{0+7}{2} = 3.5 \rightarrow 3$

$\frac{4+7}{2} = 5.5 \rightarrow 5$



0. Initial ist min = 0 und max = 7. Wir springen in die Mitte: mid = $\lfloor (\text{min} + \text{max})/2 \rfloor$.
   Hierbei runden wir im Gleichheitsfall ab. Das ist an sich willkürlich, hier aber von uns festgesetzt.
1. Das betrachtete Element ist kleiner als der Schlüssel, wir springen in die rechte Hälfte.
   Das heißt wir verschieben min = mid + 1 und wiederholen die Berechnung von mid.
2. Das betrachtete Element ist größer als der Schlüssel, wir springen in die linke Hälfte.
   Das heißt wir setzen max = mid – 1 und wiederholen die Berechnung von mid.
3. Das betrachtete Element entspricht dem gesuchten, liefere Index: 4 zurück.

Und warum ist das schnell?

- Mit jedem Vergleich sind wir entweder fertig, oder wir halbieren den Suchraum
- Bei n Elementen können wir maximal $\log_2(n)$ oft halbieren
  Ist n = 8 so beispielsweise $\log_2(8) = 3$ mal.
- So benötigen wir maximal $\log_2(n)$ viele Vergleichsschritte
- Bei der $\mathcal{O}$-Notation ist die Basis des Logarithmus egal
  So ist für $\log_a(x)$ der Unterschied zu $\log_b(x)$ nur ein Faktor, der sich durch den Basiswechsel ergibt. Es ist: $\log_a(x) = \frac{\log_b(x)}{\log_b(a)}$ dabei ist $\log_b(a)$ eine Konstante die von der $\mathcal{O}$-Notation verschluckt wird.
- Alle anderen Operationen benötigen konstante Zeit (sie sind nicht von n abhängig)
- So kommen wir auf $\mathcal{O}(\log n)$

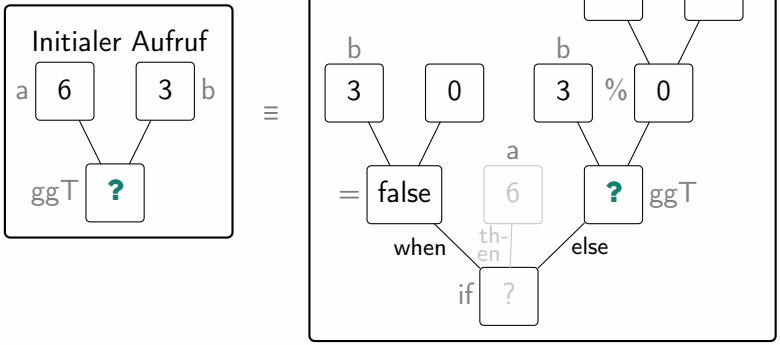
Aufgabe 1: Ablauf eines rekursiven Programms

Für einen rekursiv definierten Algorithmus kann man ein „Berechnungsformular“ aufstellen, das für jeden rekursiven Aufruf (Inkarnation) vervielfältigt wird. Das Ergebnis der Berechnung wird dann durch Ein- und Rückübertragung von (Teil-) Lösungen bestimmt, wobei die Termination durch die Reduktion des Arguments ausgelöst wird. In dieser Aufgabe sollen Sie eine derartige Formularmaschine selbst angeben.

Betrachten Sie das folgende rekursive Programm und geben Sie das Formularblatt, sowie die Inkarnationen für den Aufruf `ggt(6, 3)` an. Folgen Sie dabei der Gestaltung und Ausführung des Beispiels aus der Vorlesung (Kap. 8, Folien 15ff.)

```
public int ggT(int a, int b) {
   if(b == 0) {
      return a;
   } else {
      return ggt(b, a % b);
   }
}
```

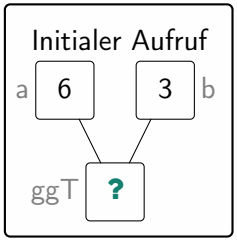
Formulargefahr



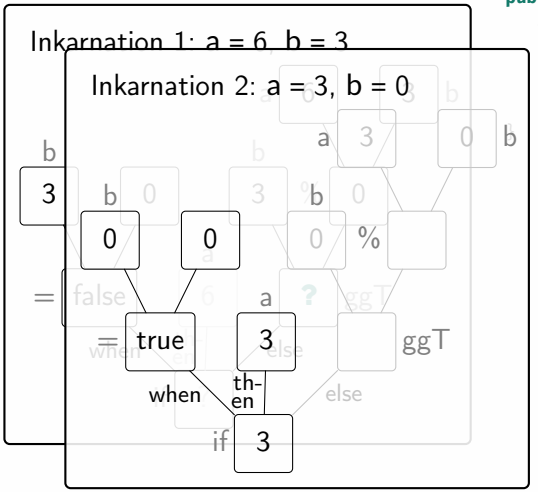

```
public int ggT(int a, int b) {
   if(b == 0) {
      return a;
   } else {
      return ggt(b, a % b);
   }
}
```

Das Abschneiden von Ästen hier durch ausgrauen!

Formulargefahr



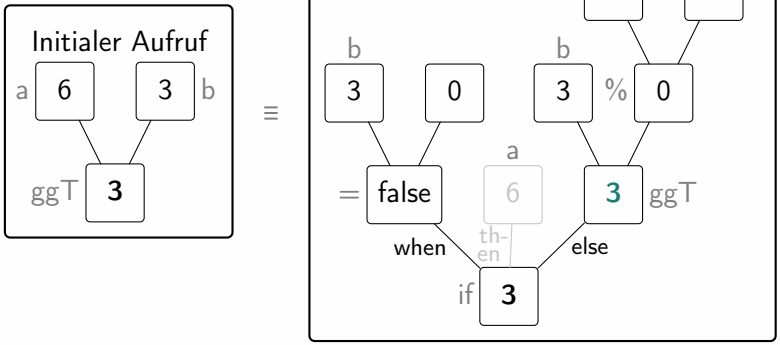
≡




```
public int ggT(int a, int b) {
   if(b == 0) {
      return a;
} else {
      return ggt(b, a % b);
}
```

Das Abschneiden von Ästen hier durch ausgrauen!

Formulargefahr




```java
public int ggT(int a, int b) {
   if(b == 0) {
      return a;
   } else {
      return ggt(b, a % b);
   }
}
```

Das Abschneiden von Ästen hier durch ausgrauen!

Aufgabe 2: Rekursive Algorithmen implementieren

Unter dem Integral $R \coloneqq \int_a^b f(x)\,dx$ *einer Funktion* f *versteht man auch die Fläche zwischen der* x*-Achse und dem Graphen der Funktion. Im Folgenden sollen Sie eine rekursive Approximation* $A(a, b)$ *für das Integral der Funktion* $f(x) = x^2$ *im Intervall* $[a, b]$ *implementieren. Dazu sollen Sie das Intervall solange in Teilintervalle halbieren, bs diese für eine direkte Berechnung hinreichend klein sind. Anschließend werden alle Teilflächen addiert. Es ergibt sich also folgende Rekursionsgleichung:*

$$A^t(a, b) \coloneqq \begin{cases} (b - a) \cdot f(a) & \textit{falls } b - a \leq 0.01 \\ A^{t+1}\big(a, (a + b)/2\big) + A^{t+2}\big((a + b)/2, b\big), & \textit{sonst} \end{cases}$$

Implementieren Sie eine Methode, die das Integral wie beschreiben rekursiv approximiert und das Ergebnis als Resultat zurückgibt. Testen Sie Ihre Implementierung für das Intervall $[0, 1]$.

Mathematische Definitionen übernehmen

Rekursion.java

```
public class Rekursion {
  public static double f(double x) {
    return x * x;
  }

  public static double A(double a, double b) {
    if(b - a <= 0.01)
      return (b - a) * f(a);
    else
      return A(a, (a + b)/2) + A((a + b)/2, b);
  }

  public static void main(String[] args) {
    System.out.println("∫[0,1] x^2 = " + A(0,1));
  }
}
```

$$f(x) = x^2$$

$$A^t(a,b) = \begin{cases} (b-a) \cdot f(a) & \text{falls } b - a \leq 0.01 \\ A^{t+1}(a,(a+b)/2) + A^{t+2}((a+b)/2,b), & \text{sonst} \end{cases}$$

Aufgabe 3: Nicht-monotone Rekursion

Die bisher in der Vorlesung betrachteten rekursiven Algorithmen haben alle eine monoton in der Argumentgröße steigende Laufzeit. Es gibt aber rekursive Funktionen, bei denen diese Annahme nicht zutrifft. Ein Beispiel hierfür ist die Collatz Funktion, die für $n \geq 1$ *wie folgt definiert sei:*

$$C(n) := \begin{cases} 1 & \text{falls } n = 1 \\ C(n/2) & \text{falls } n \text{ gerade} \\ C(3n + 1) & \text{sonst} \end{cases}$$

a) *Implementieren Sie die Collatz Funktion in Java, messen Sie jeweils die Ausführungszeit in Nanosekunden (s.* `System.nanoTime()`*) für die Eingaben von* 1 *bis* 1000 *und lassen Sie sich diese ausgeben.*

b) *Was können Sie über den Zusammenhang von* n *und der Ausführungszeit sagen? Schreiben Sie Ihre Antwort als Kommentar in die Java Datei.*

Ein wenig mehr von Collatz

$$C(n) = \begin{cases} 1 & \text{falls } n = 1 \\ C(n/2) & \text{falls } n \text{ gerade} \\ C(3n+1) & \text{sonst} \end{cases}$$

Collatz.java

```
public class Collatz {
  public static int C(int n) {
    if (n == 1)
      return 1;
    else if (n % 2 == 0)
      return C(n / 2);
    else
      return C(3 * n + 1);
  }

}
```

```
public static void main(String[] args) {
  for (int i = 1; i <= 1_000; i++) {
    long start = System.nanoTime();
    C(i);
    System.out.println("C(" + i + ") = "
      + (System.nanoTime() - start));
  }
}
```

Ausführungszeitzusammenhänge

- Zwischen n und der Ausführungszeit existiert kein offensichtlicher Zusammenhang
- Ganz allgemein zeigt die Collatz-Funktion wie eine einfache Formulierung ein quasi unmöglich vorhersehbares Verhalten an den Tag legen kann [Van05], [AM98]
  - C(870) benötigt beispielsweise 28 Schritte
  - C(871) benötigt hingegen 178 Schritte
  - C(872) benötigt wieder 116 Schritte
  - C(876) benötigt nur 54 Schritte

Zusammenfassend

[AM98] Ştefan Andrei und Cristian Masalagiu. „About the Collatz conjecture“. 1998

[Van05] Jean Paul Van Bendegem. „The Collatz conjecture. A case study in mathematical problem solving“. 2005

- Rekursion ist neben Iteration ein mächtiger Abstraktionsmechanismus
  - Ein gleichmächtiger, um genau zu sein
  - Es erlaubt beispielsweise die Umsetzung von Divide-And-Conquer-Verfahren
- Neben der Formularmaschine gibt es viele andere Visualisierungen für Rekursion
- Selbst einfach aussehende Algorithmen können sehr schwer nachvollziehbar sein!
- Bei der Komplexitätsbetrachtung hilft die $\mathcal{O}$-Notation zum Vergleich
  - $T(n) \in \mathcal{O}(f(n)) \iff \exists n_0 \in \mathbb{N}\, c \in \mathbb{R}^+\, \forall n \geq n_0 : T(n) \leq c \cdot f(n)$
  - „Der am stärksten wachsende Ausdruck“

Blatt 8
Suchen und Sortieren

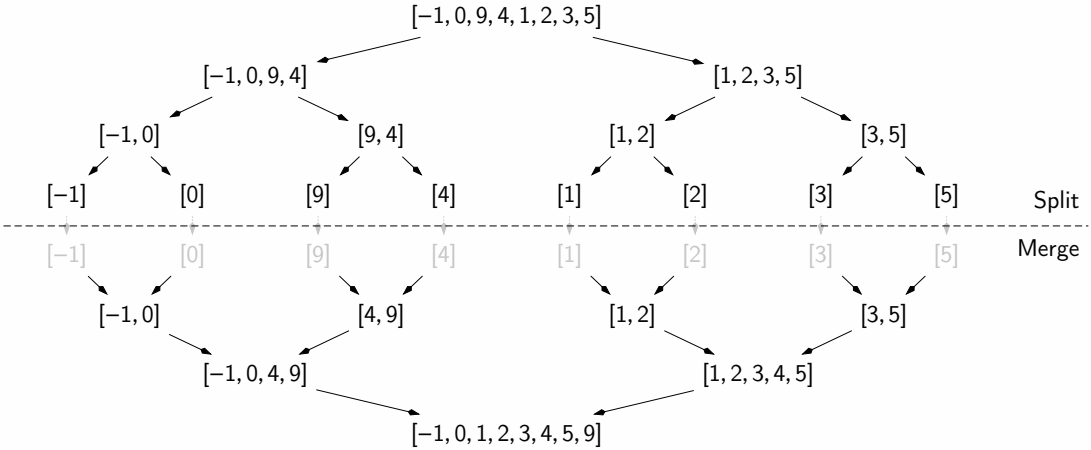
Präsenzaufgabe

9 **Mit vereinter Kraft!**

Sortieren Sie das folgende Array händisch absteigend mit dem Merge Sort Algorithmus. Geben Sie die Split- und die Mergephase an und begründen Sie, anhand dieser informell die worst case Laufzeit von $\mathcal{O}(n \log n)$ *für den Mergesort Algorithmus.*

$$[-1, 0, 9, 4, 1, 2, 3, 5]$$

Ein wenig Mergesort



Eigenschaften unter der Lupe

- Mergesort ist „easy split, hard join“, bezeichne nun n die Eingabelänge
- Das Aufteilen benötigt immer $\mathcal{O}(\log n)$ viele Schritte (wie bei der binären Suche)
  Genau genommen kommt das auf die Implementation an, nutzen wir Arrays und müssen diese jedes mal kopieren landen wir schon eher bei $\mathcal{O}(n \log n)$,
- Merge hat auch $\mathcal{O}(\log n)$ Schritte und muss pro Schritt $\mathcal{O}(n)$ Elemente vergleichen
  Bei Mergesort vergleichen wir ja jeweils nur die vordersten Elemente beider Arrays, jeder Vergleich übernimmt ein Element in das Zielarray des Schrittes.
- Alle anderen Operationen (Kopie der Zahl ins Ziel-Array, …) sind konstant
  Selbst, wenn sie nicht konstant wären, sind Array-Erzeugungen und Kopien alle auch maximal in $\mathcal{O}(n)$ machbar.
- Wir erhalten: $\mathcal{O}(\log n) + \mathcal{O}(n) \cdot \mathcal{O}(\log n) = \mathcal{O}(\max\{\log n, n \cdot \log n\}) = \mathcal{O}(n \log n)$

Aufgabe 1: Naive Sortieralgorithmen

Teil von Übungsblatt 2 war der Entwurf eines naiven Sortieralgorithmus, welcher durch iteratives Vertauschen eine gegebene Liste aufsteigend sortiert. Hier finden Sie die Algorithmen als Pseudocode nochmals aufgeführt:

Input : List L = ($l_1, \ldots, l_N$), Indizes i und j

```
1 make new N have value length(L);
2 if N < 2 or i < 1 or j < 1 or i > N or j > N
    then
3     print out „Eingabe ungültig!“;
4     stop program;
5 make new t have value l_i;
6 make l_i have value l_j;
7 make l_j have value t;
```

Input : List L = ($l_1, \ldots, l_N$)

```
1 make new N have value length(L);
2 if N < 2 then print out „Eingabe ungültig!“ and stop program ;
3 make new i have value 1;
4 while i < N do
5     make new j have the value i + 1;
6     while j ≤ N do
7         if l_i > l_j then call vertausche(L, i, j);
8         Increment j by 1;
9     Increment i by 1;
```

a) *Implementieren Sie dieses Verfahren nun in Java. Halten Sie sich dabei genau an die Angaben aus den gegeben Algorithmen und testen Sie Ihre Implementierung an einem Beispiel.*

b) *Für die Pseudocodevariante des Sortierverfahrens haben wir bereits eine Laufzeitabschätzung von:*

$$T^{sort}_{max}(n) = 3+3n+12\cdot\frac{n^2-n}{3}$$

Elementaroperationen (s. Übungsblatt 2) angegeben. Geben Sie nun hierfür die Laufzeit in der $\mathcal{O}$-Notation an und begründen Sie ihre Antwort kurz.

Swappsies

Von Zahlen war nicht die Rede. Von Arrays auch nicht! Das sind (willkürlich sinnvolle) Annahmen.

```
public static void vertausche(int[] arr, int i, int j) {
   int n = arr.length;
   if(n < 2 || i < 0 || j < 0 || i >= n || j >= n) {
      System.err.println("Eingabe ungültig!");
      return;
   }
   int t = arr[i];
   arr[i] = arr[j];
   arr[j] = t;
}
```

Mathematisch: $\{1, \ldots, n\}$, *Java:* $\{0, \ldots, n-1\}$

Ich habe mir hier die Freiheit genommen, L und N umzubenennen.

Input : List L = $(l_1, \ldots, l_N)$, Indizes i und j

1 make new N have value length(L);
2 **if** N < 2 **or** i < 1 **or** j < 1 **or** i > N **or** j > N **then**
3 print *out* „Eingabe ungültig!“;
4 stop program;
5 make new t have value l_i;
6 make l_i have value l_j;
7 make l_j have value t;

Sort it!

```java
public static void sortiere(int[] arr) {
   int n = arr.length;
   if (n < 2) {
      System.err.println("Eingabe ungültig!");
      return;
   }
   int i = 0;
   while(i < arr.length - 1) {
      int j = i + 1;
      while(j < arr.length) {
         if (arr[i] > arr[j]) {
            vertausche(arr, i, j);
         }
         j++;
      }
      i++;
   }
}
```

Input : List L = $(l_1, \ldots, l_N)$

1 make new N have value length(L);
2 **if** N < 2 **then** print *out* „Eingabe ungültig!“ and stop program ;
3 make new i have value 1;
4 **while** i < N **do**
5 make new j have the value i + 1;
6 **while** j ≤ N **do**
7 **if** $l_i > l_j$ **then** call `vertausche(L, i, j)`;
8 Increment j by 1;
9 Increment i by 1;

Testen an einem Beispiel

NaiveSort.java

```
public class NaiveSort {
  public static void vertausche(int[] arr, int i, int j) { ... }
  public static void sortiere(int[] arr) { ... }

  ▷ display in browser
  public static void main(String[] args) {
    int[] arr = { 5, 8, 1, -1 };
    sortiere(arr);
    for (int i : arr)
      System.out.println(i);
  }
}
```

Komplexitätsabschätzung

- Nun fehlt noch die Laufzeit in $\mathcal{O}$-Notation für:

$$T^{sort}_{max}(n) = 3 + 3n + 12 \cdot \frac{n^2 - n}{3}$$

- Vereinfachen wir zunächst:

$$T^{sort}_{max}(n) = 3 + 3n + 4 \cdot n^2 - 4 \cdot n = 3 - 1 \cdot n + 4 \cdot n^2$$

- Was Wachstumsverhalten des Ausdrucks wird von n^2 dominiert, der Leitkoeffizient entfällt wie der Rest nach den Rechenregeln.
- Damit ist: $T^{sort}_{max}(n) \in \mathcal{O}(n^2)$

Aufgabe 2: Nicht ganz so naive Sortieralgorithmen

Betrachten Sie folgende Implementierung von Insertion Sort zur aufsteigenden Sortierung:

```
public static void insertionSort(int[] arr) {
   int pos, key;
   for (int i = 1; i < arr.length; i++) {
      pos = i - 1;
      key = arr[pos + 1];
      while (pos >= 0 && arr[pos] > key) {
         arr[pos + 1] = arr[pos];
         pos--;
      }
      arr[pos + 1] = key;
   }
}
```

a) *Unter welchen Bedingungen tritt im obigen Sortieralgorithmus der **best case** ein? Geben Sie die Laufzeit hierfür in $\mathcal{O}$-Notation an.*

b) *Unter welchen Bedingungen tritt im obigen Sortieralgorithmus der **worst case** ein? Geben Sie die Laufzeit hierfür in $\mathcal{O}$-Notation an.*

Case-Analyse

a) *Unter welchen Bedingungen tritt im obigen Sortieralgorithmus der **best case** ein? Geben Sie die Laufzeit hierfür in $\mathcal{O}$-Notation an.*

Ist das Array bereits *aufsteigend* sortiert, läuft die äußere Schleife n – 1-mal und die innere nie, da nie „`arr[pos] > arr[pos + 1]`“ eintritt.

Dies ist der best case mit $\mathcal{O}(n)$ (der Rest ist $\mathcal{O}(1)$).

b) *Unter welchen Bedingungen tritt im obigen Sortieralgorithmus der **worst case** ein? Geben Sie die Laufzeit hierfür in $\mathcal{O}$-Notation an.*

Ist das Array *absteigend* sortiert, läuft die äußere Schleife n – 1 mal, und die innere über den kleinen Gauß insgesamt $\mathcal{O}(n^2)$ mal, da stets „`arr[pos] > arr[pos + 1]`“ gilt.

Somit haben wir einen worst case mit einer Laufzeitkomplexität von $\mathcal{O}(n^2)$.

Aufgabe 3: Rekursives Suchen

In dieser Aufgabe sollen Sie den iterativen Binary Search aus der Vorlesung in eine rekursive Variante umprogrammieren. Beantworten Sie die Fragen als Kommentar in der entsprechenden Java Datei.

a) *Implementieren Sie eine rekursive Version des Binary Search Algorithmus. Für den Fall, dass der gesuchte Wer nicht gefunden werden konnte, geben Sie eine Meldung auf die Konsole aus sowie den Index -1 zurück. Testen Sie Ihre Implementierung an einem Beispiel. Nehmen Sie an, dass die Eingaben immer sortiert sind.*

b) *Ist Ihre Implementierung aus Teilaufgabe a) Kopf- oder End-rekursiv? Begründen Sie Ihre Antwort kurz.*

c) *Geben Sie an, unter welchen Bedingungen für Ihre Implementierung der worst-case bzgl. der Laufzeit eintritt. Begründen Sie Ihre Antwort kurz.*

d) *Geben Sie die bestmögliche Laufzeitabschätzung für den worst-case in $\mathcal{O}$-Notation an. Begründen Sie Ihre Antwort kurz.*

Save me, Sonic!

```java
public static int binSearchRec(int[] arr, int key, int left, int right) {
    if (left > right) {
        System.out.println("Wert konnte nicht gefunden werden!");
        return -1;
    }
    int mid = (left + right) / 2;
    if (arr[mid] == key) {
        return mid;
    } else if (arr[mid] > key) {
        return binSearchRec(arr, key, left, mid - 1);
    } else {
        return binSearchRec(arr, key, mid + 1, right);
    }
}
```

Eine wunderschöne Tail(s)-Rekursion!

Kunckles! Echidnas want to test that!

BinarySearchRecursive.java

```java
public class BinarySearchRecursive {
  public static int binSearchRec(int[] arr, int k, int l, int r) { ... }

  public static void main(String[] ars) {
    int[] arr = { 1, 2, 4, 8, 16, 32, 64, 128, 256 };
    System.out.println("Pos: " + binSearchRec(arr, 7, 0, arr.length - 1));
  }
}
```

Kopflose Analysen

b) *Ist Ihre Implementierung aus Teilaufgabe a) Kopf- oder End-rekursiv? Begründen Sie Ihre Antwort kurz.*

Wie bereits angemerkt liegt eine Endrekursion vor: alle Berechnungen passieren im Abstieg, die rekursiven Aufrufe sind für alle rekursiven Ausführungspfade das letzte Statement.

c) *Geben Sie an, unter welchen Bedingungen für Ihre Implementierung der worst-case bzgl. der Laufzeit eintritt. Begründen Sie Ihre Antwort kurz.*

Wenn der gesuchte Wert nicht im Array enthalten ist, müssen wir die meisten Vergleiche durchführen.

Nachdem wir das letzte Element betrachtet haben, gehen wir noch einmal in die Rekursion um dann festzustellen, dass es nichts mehr zu durchsuchen gibt — das Suchfenster ist leer.

Komplexity? Analyse!

d) Geben Sie die bestmögliche Laufzeitabschätzung für den worst-case in $\mathcal{O}$-Notation an. Begründen Sie Ihre Antwort kurz.

Die Analyse bleibt zur iterativen Variante weitgehend unverändert. Wir können das Array $\log_2 n$-mal teilen und haben somit $\mathcal{O}(\log_2 n)$ rekursive Aufrufe. Da alle anderen Operationen konstant sind, landen wir somit bei:

$$T_{max}^{\text{rec. bin-search}} \in \mathcal{O}(\log n)$$

Zusammenfassend

- Such- und Sortierverfahren sind mit die „Musterkinder“ in der Informatik
  - Suchverfahren können durch eine vorhergehende Sortierung beschleunigt werden
  - Verschiedene Verfahren haben verschiedene Eigenschaften
  - Selectionsort kann beispielsweise jederzeit abgebrochen werden und liefert immer noch eine Teil-Sortierung (Mergesort beispielsweise nicht!)
- Es ist wichtig, ein Grundverständnis für alle Such- und Sortierverfahren zu entwickeln
- Rekursion wird uns nicht nur beim Suchen und Sortieren begegnen
  Uuuuh, dynamische-Datenstruktur mich! Weitere Verfahren folgen nächste Woche.

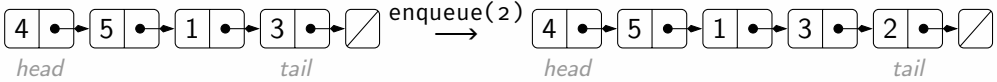
Blatt 9
Suchen und Sortieren II

Präsenzaufgabe

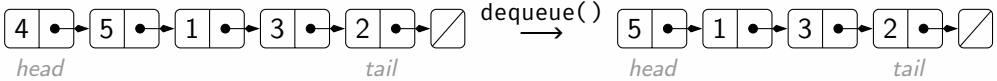
10 **FIFO-Freuden**

Implementieren Sie eine Queue, die `int` Werte speichert. Verwenden Sie dazu `Element.java` und `Queue.java` als Grundgerüst und implementieren Sie:

- `void enqueue(int value)`, welche den Wert ans Ende der Queue einreiht



- `boolean dequeue()`, welche den vordersten Wert entfernt



Verwenden Sie keine Arrays oder vorgefertigte dynamische Datenstrukturen, sondern eine eigene, (einfach) verkettete Liste.

Adding something to a Queue

```
public class Queue {
   private Element first;
   private Element last;
   private int length;

   public void enqueue(int value) {
      Element next = new Element(value);
      if(length == 0)   Spezialfall beim Einfügen: Isse leer?
         this.first = next;
      else
         this.last.setNextElement(next);
      this.last = next;   In jedem Fall: Verschiebe den Tail
      length++;
   }
}
```

Moving out

Queue.java

```
public class Queue {
   private Element first;
   private Element last;
   private int length;
   public void enqueue(int value) { ... }

   public boolean dequeue() {
      if(length > 0) {
         this.first = this.first.getNextElement();
         length--;
         return true;    Es gab etwas zu Entfernen!
      }
      return false;
   }
}
```

Aufgabe 1: Heap Sort

In dieser Aufgabe sollen Sie das folgende Array händisch aufsteigend mit dem Heap Sort Algorithmus sortieren:

$$[0, -1, 9, 4]$$

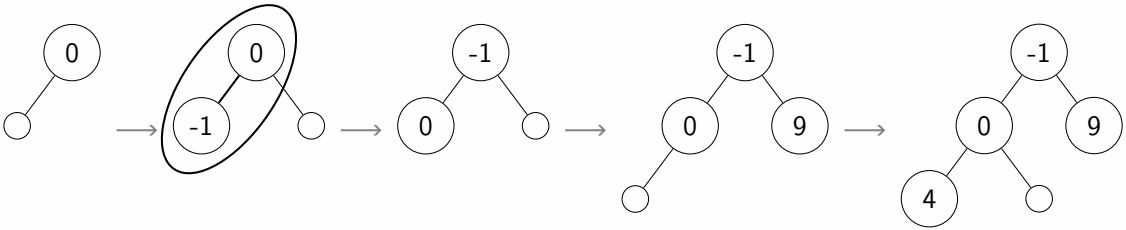
a) *Phase 1: Geben Sie den schrittweise den entsprechenden Heap an und markieren Sie jeweils wo die Heap-Eigenschaft verletzt ist. Orientieren Sie sich dabei an der Darstellung aus der Vorlesung.*

b) *Phase 2: Wandeln Sie den Heap aus Phase 1 schrittweise in ein sortiertes Array um. Markieren Sie jeweils wo nach dem Herausnehmen des Head die Heap-Eigenschaft verletzt wurde und das entsprechende Resultat der Heapify Operation.*

Der Aufbau eines Heaps

a) Geben Sie den schrittweise den Heap an und markieren Sie jeweils wo die Heap-Eigenschaft verletzt ist.

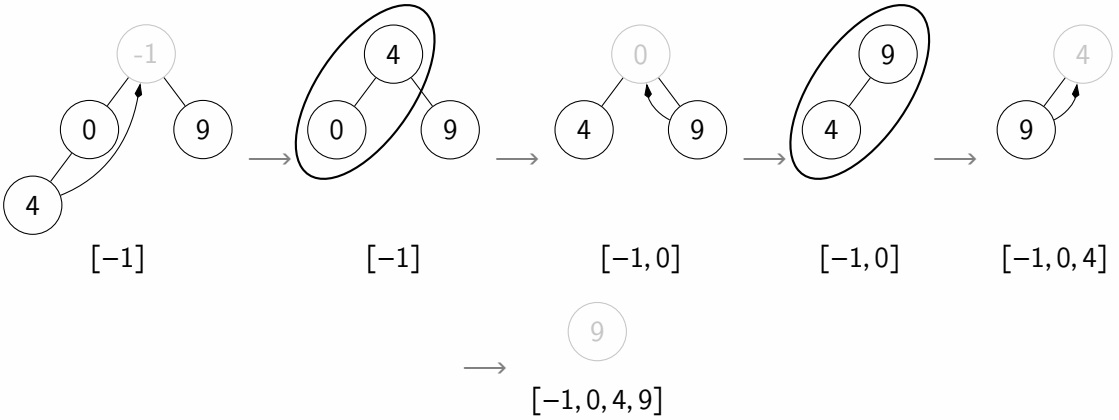
$[0, -1, 9, 4]$



Hier haben wir einen Min-Heap, ein Max-Heap würde genau so funktionieren!

Der Abbau eines Miep

b) Wandeln Sie den Heap aus Phase 1 schrittweise in ein sortiertes Array um.



Aufgabe 2: Rekursive Sortieralgorithmen: Shaker Sort

In dieser Aufgabe sollen Sie einen rekursiven Sortieralgorithmus selbst implementieren. Wir betrachten eine Variante des Bubble-Sort Algorithmus, bei dem das Array in jedem Sortierschritt alternierend von vorne und hinten durchlaufen wird. Nach jedem Durchlaufen werden die bereits sortieren Elemente vorne und hinten nicht mehr betrachtet werden. Implementieren Sie Shaker Sort in Java und testen Sie Ihre Lösung an mindestens einem Beispiel.

Helping Swap

```
public class ShakerSort {
   private static void swap(int[] arr, int i, int j) {
      int tmp = arr[i];
      arr[i] = arr[j];
      arr[j] = tmp;
    }
}
```

Recursive Bubble-Up

```
public class ShakerSort {
   private static void swap(int[] arr, int i, int j) { ... }

   static boolean moveUp(int[] arr, int i, int end, boolean swapped) {
      if (i >= end)
         return swapped;
      if (arr[i] < arr[i - 1]) {
         swap(arr, i, i - 1);
         swapped = true;
      }
      return moveUp(arr, i+1, end, swapped);
   }
}
```

Was müsste man anpassen um immer mit `i + 1` zu vergleichen?

Recursive Bubble-Down

```
public class ShakerSort {
   private static void swap(int[] arr, int i, int j) { ... }
   static boolean moveUp(int[] arr, int i, int end, boolean swapped)
      { ... }

   static boolean moveDown(int[] arr, int i, int end, boolean swapped) {
      if (i <= end)
         return swapped;
      if (arr[i] < arr[i - 1]) {
         swap(arr, i, i - 1);
         swapped = true;
      }
      return moveDown(arr, i-1, end, swapped);
   }
}
```

I shake it up, I shake it down!

```
public class ShakerSort {
   private static void swap(int[] arr, int i, int j) { ... }
   static boolean moveUp(int[] arr, int i, int end, boolean s) { ... }
   static boolean moveDown(int[] arr, int i, int end, boolean s) { ... }

   public static void shakerSort(int[] array, int n) {
      int offset = array.length - n;
      if (n <= array.length / 2)
         return;
      else if (!moveUp(array, offset + 1, n, false))
         return;
      else if (!moveDown(array, n - 1, offset, false))
         return;
      shakerSort(array, n - 1);
   }
}
```

Some sample Code

ShakerSort.java

```
public class ShakerSort {
    public static void shakerSort(int[] array, int n) { ... }

    public static void main(String[] args) {
        Random random = new Random();
        int length = random.nextInt(11) + 10;
        int[] array = new int[length];
        for (int i = 0; i < length; i++)
            array[i] = random.nextInt(100);
        System.out.println("Unsortiert: " + Arrays.toString(array));
        shakerSort(array, length);
        System.out.println("Sortiert: " + Arrays.toString(array));
    }
}
```

Aufgabe 3: Laplacescher Entwicklungssatz

In dieser Aufgabe sollen Sie nochmal die Implementierung rekursiver Funktionen üben. Dazu betrachten wir die rekursive Berechnung der Determinante einer quadratischen $n \times n$ *Matrix* M*. Die Determinante gibt an, wie sich das Volumen bei der durch die Matrix beschriebenen linearen Abbildung ändert, und ist ein Hilfsmittel bei der Lösung linearer Gleichungssysteme. Sie kann für kleine Matrizen* ($1 \leq n \leq 10$) *rekursiv über den Laplaceschen Entwicklungssatz nach der ersten Spalte berechnet werden:*

$$\det(M) = \begin{cases} m_{1,1} & \text{falls } n = 1 \\ \sum_{i=1}^{n} (-1)^{i+1} m_{i,1} \det(M_{-i1}) & \text{für } n \geq 2 \end{cases}$$

wobei $n \times n$ *die Dimensionalität der Matrix* M, $M_{-i,1}$ *die Matrix, die aus* M *entsteht, wenn die erste Spalte und die* i*-te Zeile entfernt werden (samt anschließendem Zusammenschieben) und* $m_{i,j}$ *das element in der* i*-ten Zeile und* j*-ten Spalte seien.*
In den Materialien zu diesem Übungsblatt finden Sie die Datei `Determinante.java`*, die bereits alle notwendigen Hilfsmethoden enthält. Sie sollen die vorgegebene Methode* `int det(int[][] mat)` *implementieren.*

Translating a formula… 1:1 (caring for zeros)

```
private static int det(int[][] mat) {
   if(mat.length == 1) {
      return mat[0][0];
   } else {
      int result = 0;
      for(int i = 0; i < mat.length; i++) {
         result += (int)Math.pow(-1, i+1) * mat[i][0]
                   * det(reduceMatrix(mat, i));
      }
      return result;
   }
}
```

$$\det(M) = \begin{cases} m_{1,1} \\ \sum_{i=1}^{n} (-1)^{i+1} m_{i,1} \det(M_{-i1}) \end{cases}$$

Zusammenfassend

- Rekursion wird uns so schnell nicht verlassen
  - Viele Datenstrukturen sind rekursiv, was wir ausnutzen können
  - Für die verschiedenen Tiefen-Traversierungen beispielsweise:

```
void preorder(Node n) {
   if(n == null) return;
   // process n
   preorder(n.left);
   preorder(n.right);
}
```

```
void inorder(Node n) {
   if(n == null) return;
   inorder(n.left);
   // process n
   inorder(n.right);
}
```

```
void postorder(Node n) {
   if(n == null) return;
   postorder(n.left);
   postorder(n.right);
   // process n
}
```

  - Auch lassen sich Regeln für Heaps so beschreiben
- Listen (als spezielle Bäume) und Bäume sind vielseitig und elementar
  - Auch bei ihrer Handhabung hilft Übung!

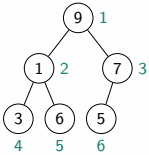
Blatt 10
Dynamische Datenstrukturen

Präsenzaufgabe

11 Traverse My-Tree

Sie sollen binären Bäume mit einem Breitendurchlauf traversieren. Verwenden Sie neben den gegebenen (BinaryTree.java, BinaryTreeMain.java, Queue.java, IntegerNode.java) keine vorgefertigten dynamischen Datenstrukturen (sie entsprechen der Vorlesung). Implementieren Sie die Methode `public void breadthFirstTraversal()`, welche den Baum im Breitendurchlauf durchläuft und die Werte der einzelnen Knoten ausgibt. Hierfür werden Sie `Queue` benötigen.

Zahlenbeispiel Im Breitendurchlauf traversiert ergibt sich für diesen Baum folgende Reihenfolge:



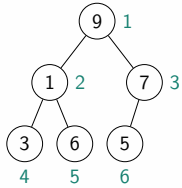

```
public class BinaryTree {
  private IntegerNode root;
  public BinaryTree(int[] items) { ... }
  public void breadthFirstTraversal() { ★ }
}
public class IntegerNode {
  public IntegerNode(int value) { ... }
  public void setLeftChild(IntegerNode l) { ... }
  public IntegerNode getLeftChild() { ... }
  public void setValue(int value) { ... }
  public int getValue() { ... }
  ...
}
```

```
public class Queue {
  class Element {
    public Element(IntegerNode node) { ... }
    public void setNextElement(Element n) { ... }
    public Element getNextElement() { ... }
    public IntegerNode getNode() { ... }
  }
  public Queue() { ... }
  public void enqueue(IntegerNode node) { ... }
  public IntegerNode dequeue() { ... }
  public int getLength() { ... }
  public boolean isEmpty() { ... }
}
```

Let it be code

BinaryTree.java, BinaryTreeMain.java, IntegerNode.java, Queue.java

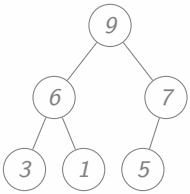
```
public void breadthFirstTraversal() {
    Queue queue = new Queue();
    queue.enqueue(root);

    while (!queue.isEmpty()) {
        IntegerNode node = queue.dequeue();
        if (node.getLeftChild() != null)
            queue.enqueue(node.getLeftChild());
        if (node.getRightChild() != null)
            queue.enqueue(node.getRightChild());
        System.out.println(node.getValue());
    }
}
```



Es lässt sich auch mit **if**(node == **null**) beim Entnehmen prüfen.

Aufgabe 1: Bäume

Betrachten Sie den folgenden Binärbaum:



a) *Welchen Verzweigungsgrad hat der Baum? Begründen Sie Ihre Antwort kurz.*

b) *Welche Tiefe hat der Baum?*

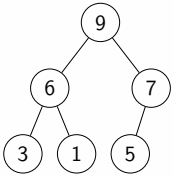
c) *Wie viele Knoten hat der Baum und wie viele davon sind Blätter?*

d) *Formt der Baum einen Max-Heap? Begründen Sie Ihre Antwort kurz.*

e) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Tiefendurchlauf traversiert wird?*

f) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Breitendurchlauf traversiert wird?*

Grundlagen Fragen

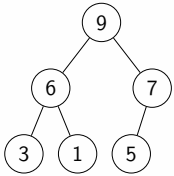


a) *Welchen Verzweigungsgrad hat der Baum? Begründen Sie Ihre Antwort kurz.*
2, alle Knoten haben *höchstens* zwei Kinder.

b) *Welche Tiefe hat der Baum?*
3, der längste Pfad von einem Knoten zur Wurzel ist 2 lang.

c) *Wie viele Knoten hat der Baum und wie viele davon sind Blätter?*
6 Knoten ($\{3, 6, 9, 1, 7, 5\}$), 3 davon sind Blätter (Knoten ohne Kinder, $\{3, 1, 5\}$)

d) *Formt der Baum einen Max-Heap? Begründen Sie Ihre Antwort kurz.*
Ja, die Elternknoten simd immer kleiner als alle ihre Kinder.



e) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Tiefendurchlauf traversiert wird?*

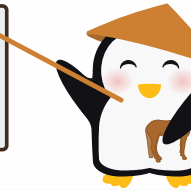
Prä-Order: 9, 6, 3, 1, 7, 5
In-Order: 3, 6, 1, 9, 5, 7
Post-Order: 3, 1, 6, 5, 7, 9

Unter Angabe der Variante (Prä-, In- oder Post-Order) reicht hier auch nur eine Version.

f) *In welcher Reihenfolge werden die Knoten besucht, wenn der Baum im Breitendurchlauf traversiert wird?*

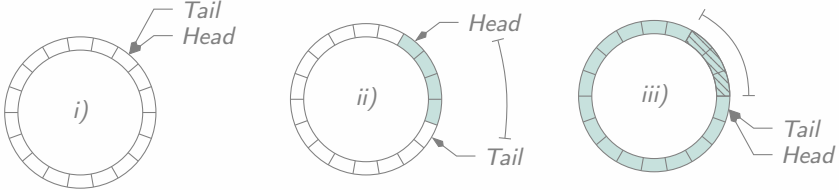
9, 6, 7, 3, 1, 5.

Neben Pre-, In- und Post-Order gibt es für spezifische Bäume (wie Binärbäume) auch noch weitere Versionen, die uns aber erstmal egal sind.

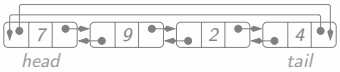


Aufgabe 2: Self-Overriding Ring Buffer

In dieser Aufgabe sollen Sie einen Ring Buffer implementieren. Hierbei handelt es sich um eine dynamische Datenstruktur, ähnlich zu einer `DoublyLinkedList` *aus der Vorlesung, jedoch verweist hier jedes Element auf seinen Vorgänger und Nachfolger, wodurch sie einen Ring formen. Zusätzlich hat ein Ring Buffer eine fest vorgegebene Kapazität. Hierfür werden intern zwei Referenzen* `head` *und* `tail` *verwendet, die so Beginn und Ende des belegten Bereichs markieren. Alle Positionen bis zum* `tail` *sind beginnend beim* `head` *Zeiger belegt. Wir betrachten ein modifiziertes Ring Buffer, bei welchem alte Elemente überschrieben werden, falls der Buffer voll ist. Der Ring Buffer soll nach dem First–In–First–Out Prinzip arbeiten: neue Elemente werden an* `tail` *angefügt und es werden vom* `head` *aus Elemente abgearbeitet. Ein Beispiel finden Sie im Folgenden.*



i) Der `RingBuffer` *ist leer und* `head`*, sowie* `tail` *zeigen auf dasselbe Element. ii) Teilweise belegt, nachdem vier Elemente eingefügt wurden und* `head`*, sowie* `tail` *nun auf verschiedenen Elemente zeigen. iii) Der Buffer ist voll (*`tail` *hat* `head`*„überrundet“) und wird nun überschreiben. Die Elemente sind wie eine* `DoublyLinkedList` *verbunden und haben zusätzlich eine Verknüpfung er beiden äußeren Elemente (unten rechts).*



Aufgabe 2: Self-Overriding Ring Buffer

a) *Nehmen Sie die Klassendefinition für Element, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen* `int` *halten.*

b) *Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse* `RingBuffer`*. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden.* `head` *und* `tail` *zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine* `NegativeArraySizeException` *ausgelöst werden.*

c) *Die Methode* `public int peek()` *der Klasse* `RingBuffer` *soll den Wert an der Position zurückliefern, auf die* `head` *zeigt. Sollte der Buffer leer sein, soll eine* `NoSuchElementException` *ausgelöst werden.*

d) *Die Methode* `public void put(int value)` *der Klasse* `RingBuffer` *soll den übergebenen Wert an der Position im Ring Buffer speichern, auf die* `tail` *zeigt und* `tail` *auf den Nachfolger verschoben werden (obige Abb. ii). Sollte der Buffer komplett gefüllt sein, soll wieder von vorne begonnen werden und alte Elemente überschrieben werden (obige Abb. iii). In diesem Fall soll auch* `head` *auf den Nachfolger verschoben, damit* `head` *immer auf das älteste Element im Buffer zeigt und so die FIFO Eigenschaft gewahrt bleibt.*

e) *Die Methode* `public int remove()` *der Klasse* `RingBuffer` *soll den Wert an der Position zurückliefern, auf die* `head` *zeigt und* `head` *um eine Stelle zurückverschoben werden. Diese Methode simuliert also ein Abarbeiten der Elemente im Buffer. Sollte der Buffer leer sein, soll eine NoSuchElementException ausgelöst werden.*

Extend the Element

a) Nehmen Sie die Klassendefinition für `Element`, die Sie bereits aus der Vorlesung kennen. Erweitern Sie diese um eine private Referenz für den Vorgänger sowie passende getter und setter. Die Elemente sollen `int` halten.

Element.java

```
class Element {
   private int value; private Element next;
   private Element previous;
   public Element() { this.next = null; this.previous = null; }
   public void setNextElement(Element n) { this.next = n; }
   public Element getNextElement() { return this.next; }
   public void setPreviousElement(Element p) {
      this.previous = p;
   }
   public Element getPreviousElement() {
      return this.previous;
   }
   public void setValue(int value) { this.value = value; }
   public int getValue() { return this.value; }
}
```

Start des RingBuffers

b) Um den Ring Buffer zu implementieren, definieren Sie zunächst die entsprechende Klasse `RingBuffer`. Der überladene Konstruktor soll die Kapazität als Parameter übernehmen und die Datenstruktur anlegen, indem der Kapazität entsprechend viele Elemente (siehe a)) angelegt und verknüpft werden. `head` und `tail` zeigen nun auf das erste Element (obige Abb. i). Wird als Kapazität ein negativer Wert übergeben, soll eine `NegativeArraySizeException` ausgelöst werden

RingBuffer.java

```
public class RingBuffer {
    private Element head;
    private Element tail;
    private boolean isEmpty;

    public RingBuffer(int capacity) {
        ...
    }
}
```

Der Konstruktor

```
public class RingBuffer {
   private Element head, tail; private boolean isEmpty;
   RingBuffer(int capacity) {
      if (capacity == 0) { this.isEmpty = false; return; }
      else if (capacity < 0) { throw new NegativeArraySizeException(); }

      Element current = new Element();
      this.head = this.tail = current;
      this.isEmpty = true;
      current = fillBuffer(capacity, current);
      this.head.setPreviousElement(current);
      current.setNextElement(head);
   }
}
```

Der Konstruktor II

```
public class RingBuffer {
   private Element head, tail; private boolean isEmpty;
   RingBuffer(int capacity) { ... }

   private Element fillBuffer(int capacity, Element current) {
      for (int i = 1; i < capacity; i++) {
         Element next = new Element();
         current.setNextElement(next);
         next.setPreviousElement(current);
         if (i == capacity - 1)
            next.setNextElement(head);
         current = next;
      }
      return current;
   }
}
```

Modern Peek

c) *Die Methode `public int peek()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt. Sollte der Buffer leer sein, soll eine `NoSuchElementException` ausgelöst werden.*

```
public class RingBuffer {
   private Element head, tail; private boolean isEmpty;
   RingBuffer(int capacity) { ... }

   public int peek() {
      if (this.isEmpty)
         throw new NoSuchElementException();
      else
         return head.getValue();
   }
}
```

Put Put Put

d) Die Methode `public void put(int value)` der Klasse `RingBuffer` soll den übergebenen Wert an der Position im Ring Buffer speichern, auf die `tail` zeigt und `tail` auf den Nachfolger verschoben werden (obige Abb. ii). Sollte der Buffer komplett gefüllt sein, soll wieder von vorne begonnen werden und alte Elemente überschrieben werden (obige Abb. iii). In diesem Fall soll auch `head` auf den Nachfolger verschoben, damit `head` immer auf das älteste Element im Buffer zeigt und so die FIFO Eigenschaft gewahrt bleibt.

```
public class RingBuffer {
   private Element head, tail; private boolean isEmpty;
   RingBuffer(int capacity) { ... }
   public void put(int value) {
      if (this.tail == this.head && !this.isEmpty)
         this.head = this.head.getNextElement();
      this.tail.setValue(value);
      this.tail = this.tail.getNextElement();
      this.isEmpty = false;
   }
}
```

Operation Removal

e) *Die Methode `public int remove()` der Klasse `RingBuffer` soll den Wert an der Position zurückliefern, auf die `head` zeigt und `head` um eine Stelle zurückverschoben werden. Diese Methode simuliert also ein Abarbeiten der Elemente im Buffer. Sollte der Buffer leer sein, soll eine NoSuchElementException ausgelöst werden.*

```
public class RingBuffer {
   private Element head, tail; private boolean isEmpty;
   RingBuffer(int capacity) { ... }
   public int remove() {
      if (this.isEmpty)
         throw new NoSuchElementException();
      int value = this.head.getValue();
      this.head = this.head.getNextElement();
      this.isEmpty = this.head == tail;
      return value;
   }
}
```

Zusammenfassend

- Meldet euch für die Übungsleistung an
  - Dies geht in campusonline
  - Die Prüfung heißt „10850 Praktische Informatik - Übung“
- Meldet euch auch für die Prüfung an
- Feedback bezüglich des Tutoriums wird immer gern gesehen
  - Ist hier eine anonyme Umfrage erwünscht?
- Die Betrachtung von Algorithmen und Datenstrukturen ist elementar!

Blatt 11
Dynamische Datenstrukturen II

Präsenzaufgabe

12

Dir streichen wir das Erbe!

RegularPolygon.java

```java
public abstract class RegularPolygon {
    protected final int numSides;
    protected final double sideLength;
    public RegularPolygon(int numSides, double sideLength) {
        this.numSides = numSides;
        this.sideLength = sideLength;
    }
    public int getNumSides() { return numSides; }
    public double getCircumference() { return numSides * sideLength; }
    public abstract double getArea();
}
```

Erstellen Sie eine neue Klasse `EquilateralTriangle`, welche ein gleichseitiges Dreieck repräsentieren und von `RegularPolygon` erben soll. Erzeugen Sie einen Konstruktor, welcher die Seitenlänge a als Parameter verarbeitet ($A = 1/4 \cdot \sqrt{3} \cdot a^2$).

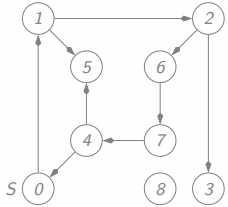
How to legacy

EquilateralTriangle.java

```java
public class EquilateralTriangle extends RegularPolygon {
  public EquilateralTriangle(double sideLength) {
    super(3, sideLength);
  }

  @Override
  public double getArea() {
    return 0.25 * Math.sqrt(2) * sideLength * sideLength;
  }
}
```

Aufgabe 1: Graphen

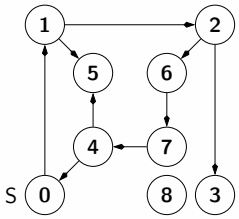
Geben sei der folgende Graph G:



a) *Geben Sie die Adjazenzliste für G an.*
b) *Geben Sie die Adjazenzmatrix für g an.*
c) *In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Tiefendurchlaufs“ traversiert wird?*
d) *In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Breitendurchlaufs“ traversiert wird?*

Eine Adjazenzliste

a) Geben Sie die Adjazenzliste für G an.



0 → 1

1 → 2 → 5

2 → 3 → 6

3

4 → 0 → 5

5

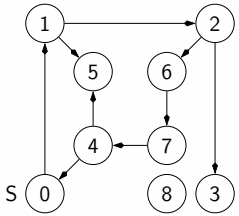
6 → 7

7 → 4

8

Eine Adjazenzmatrix

b) Geben Sie die Adjazenzmatrix für G an.

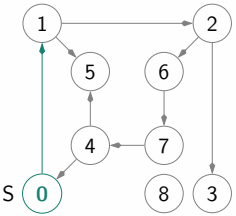


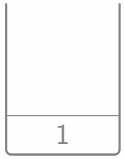
	0	1	2	3	4	5	6	7	8
0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	0	0
2	0	0	0	1	0	0	1	0	0
3	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	1	0
7	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0

(Nullen können mit einem Kommentar, dass alle freien Einträge automatisch „0“ sind, auch weggelassen werden.)

Eine Tiefendurchlauf

c) In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Tiefendurchlaufs“ traversiert wird?





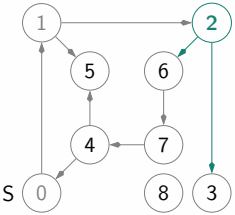
Stack

→ 0

Die Reihenfolge in der mehrere, noch nicht besuchte Nachbarn, auf den Stack gelegt werden, ist willkürlich und hängt von der Implementation ab.

Eine Tiefendurchlauf

c) In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Tiefendurchlaufs“ traversiert wird?

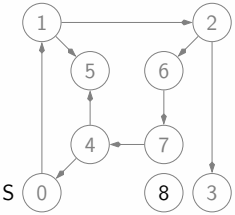


Stack

→ 0, 1, 2

Eine Tiefendurchlauf

c) In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Tiefendurchlaufs“ traversiert wird?

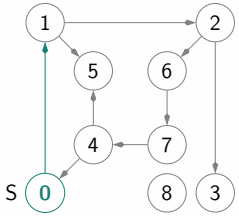


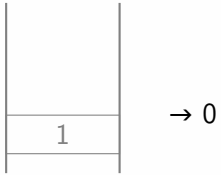
Stack

→ 0, 1, 2, 3, 6, 7, 4, 5

Eine Breitendurchlauf

d) In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Breitendurchlaufs“ traversiert wird?

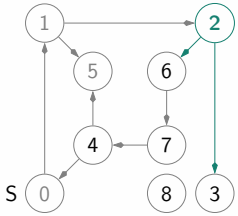


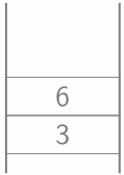


Erneut ist die Reihenfolge bei mehreren Nachbarn willkürlich.

Eine Breitendurchlauf

d) In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Breitendurchlaufs“ traversiert wird?



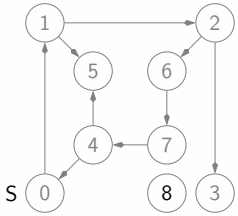


Queue

→ 0, 1, 5, 2

Eine Breitendurchlauf

d) In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Breitendurchlaufs“ traversiert wird?



Queue

→ 0, 1, 5, 2, 3, 6, 7, 4

Aufgabe 2: Stacks

In dieser Aufgabe sollen Sie einen `Stack` implementieren, welcher nach dem `LIFO`-Prinzip arbeitet (Last In First Out) und `int` Werte verwaltet. Nutzen Sie hierfür wieder die Element.java Klasse. Legen Sie die Klasse `Stack.java` an und implementieren Sie die folgenden Methoden:

a) *Implementieren Sie einen Konstruktor, der den Stack erstellt. Der Stack soll eine private Referenz auf das oberste Element, sowie eine private Instanzvariable, die die Größe des Stacks darstellt, besitzen.*

b) *Die Methode `public void push(int value)` soll den übergebenen Wert oben auf dem Stack platzieren.*

c) *Die Methode `public int pop()` soll den obersten Wert vom Stack entfernen und zurückgeben. Beachten Sie, dass der Stack unter Umständen leer sein kann. In diesem Fall soll eine `NoSuchElementException` ausgelöst werden.*

d) *Testen Sie Ihre Implementierungen, indem Sie einige Werte auf den Stack legen und wieder entfernen.*

Ein Stack für die Masse

Stack.java

```
1  public class Stack {
2     private Element top;
3     private int size;
4     public Stack() {    Person(String,int,int)
5        this.top = null;
6        this.size = 0;
7     }
8  ›  public void push(int value) {
9        Element newTop = new Element();
10       newTop.setValue(value);
11       newTop.setNextElement(this.top);
12       this.top = newTop;
13       size += 1;
14    }
15
16 }
```

```
15 public int pop() {
16    if (size <= 0)
17       throw new NoSuchElementException();
18    int value = top.getValue();    Person(Strin
19    top = top.getNextElement();
20    size -= 1;
21    return value;
22 }
```

▷ display in browser

```
23 public static void main(String[] args) {
24    Stack stack = new Stack();
25    stack.push(3);    Person(String,int,int)
26    stack.push(4);
27    System.out.println(stack.pop());
28 }
```

top

Ein Stack für die Masse

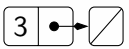
Stack.java

```
1  public class Stack {
2     private Element top;
3     private int size;
4     public Stack() {    Person(String,int,int)
5        this.top = null;
6        this.size = 0;
7     }
8     public void push(int value) {
9        Element newTop = new Element();
10       newTop.setValue(value);    value=3
11       newTop.setNextElement(this.top);
12       this.top = newTop;
13 >     size += 1;    size=1
14    }
15
16 }
```

```
15 public int pop() {
16    if (size <= 0)
17       throw new NoSuchElementException();
18    int value = top.getValue();    Person(Strin
19    top = top.getNextElement();
20    size -= 1;
21    return value;
22 }
```

▷ display in browser

```
23 public static void main(String[] args) {
24    Stack stack = new Stack();
25    stack.push(3);    Person(String,int,int)
26    stack.push(4);
27    System.out.println(stack.pop());
28 }
```



newTop, top

Ein Stack für die Masse

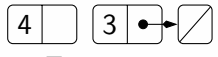
Stack.java

```
1  public class Stack {
2     private Element top;
3     private int size;
4     public Stack() {
5        this.top = null;
6        this.size = 0;
7     }
8     public void push(int value) {
9        Element newTop = new Element();
10 >     newTop.setValue(value);   value=4
11       newTop.setNextElement(this.top);
12       this.top = newTop;
13       size += 1;
14    }
15
16 }
```

```
15 public int pop() {
16    if (size <= 0)
17       throw new NoSuchElementException();
18    int value = top.getValue();   Person(Strin
19    top = top.getNextElement();
20    size -= 1;
21    return value;
22 }
```

▷ display in browser

```
23 public static void main(String[] args) {
24    Stack stack = new Stack();
25    stack.push(3);   Person(String,int,int)
26    stack.push(4);
27    System.out.println(stack.pop());
28 }
```



Ein Stack für die Masse

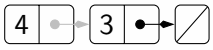
Stack.java

```java
public class Stack {
   private Element top;
   private int size;
   public Stack() {
      this.top = null;
      this.size = 0;
   }
   public void push(int value) {
      Element newTop = new Element();
      newTop.setValue(value);
      newTop.setNextElement(this.top);
      this.top = newTop;
      size += 1;
   }

}
```

```java
public int pop() {
   if (size <= 0)
      throw new NoSuchElementException();
   int value = top.getValue();
   top = top.getNextElement();
   size -= 1;
   return value;
}
```

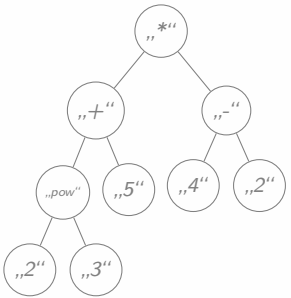
▷ display in browser

```java
public static void main(String[] args) {
   Stack stack = new Stack();
   stack.push(3);
   stack.push(4);
   System.out.println(stack.pop());   4
}
```



Aufgabe 3: Post-Order Traversierung von Binärbäumen

Ein `BinaryExpressionTree` repräsentiert einen Ausdruck, bei dem die Operatoren höchstens zwei Operanden verarbeiten können. So an die Berechnungen $(2^3 + 5) * (4 - 2)$ durch den folgenden Baum ausgedrückt werden:



Die Darstellungsweise erlaubt rekursives abarbeiten des Ausdrucks. Dazu wird jeweils ein Knoten evaluiert: handelt es sich um einen Operator (kein Blatt und somit einen Rekursionsfall), wird dieser wieder evaluiert. Handelt es sich um einen Operanden (Blatt und somit ein Basisfall), kann das Ergebnis berechnet werden.

Wir beschränken uns hier einfachheitshalber nur auf balancierte Binärbäume, das heißt, es können nicht beliebige Ausdrücke dargestellt werden. Sie finden in den Unterlagen (zu diesem Übungsblatt) die Klasse StringNode.java, die die Knoten darstellen soll, sowie die Klasse BinaryExpressionTree.java, die den Baum selbst repräsentiert, sowie BinaryExpressionTreeMain.java als Programmeinstiegspunkt.

Implementieren Sie die Methode `public double traverse(StringNode node)`, die den Binärbaum wie beschrieben rekursiv im post-Order Verfahren traversiert.

Ihre Implementierung soll mindestens die Operatoren `+`, `-`, `*`, `/`, sowie `pow` unterstützten. Enthält ein Knoten einen unbekannten Operator, soll eine `InvalidParameterException` ausgelöst werden. Testen Sie Ihre Implementierung an dem obigen Beispiel. Die Methode `private void insertNodes(...)` der Klasse `BinaryExpressionTree` erstellt einen Binärbaum aus einem gegebenen String-Array.

Evaluate the World

BinaryExpressionTree.java, StringNode.java

```java
public class BinaryExpressionTree {
   public double traverse(StringNode node) {
      if (node == null) throw new IllegalStateException();
      if (node.isLeaf()) return Double.parseDouble(node.getItem());

      double valueLeft = traverse(node.getLeftChild());
      double valueRight = traverse(node.getRightChild());
      switch (node.getItem()) {
         case "+": return valueLeft + valueRight;
         case "-": return valueLeft - valueRight;
         case "*": return valueLeft * valueRight;
         case "/": return valueLeft / valueRight;
         case "pow": return Math.pow(valueLeft, valueRight);
         default: throw new InvalidParameterException();
      }
   }
}
```

Everything on Main!

BinaryExpressionTreeMain.java

```java
public class BinaryExpressionTreeMain {
    ▷ display in browser
    public static void main(String[] args) {
        String[] items = {"*", "+", "-", "pow", "5", "4", "2", "2", "3"};
        BinaryExpressionTree tree = new BinaryExpressionTree(items);

        double result = tree.evaluate();
        System.out.println("Ergebnis: " + result);
    }
}
```