

252-0027

Einführung in die Programmierung

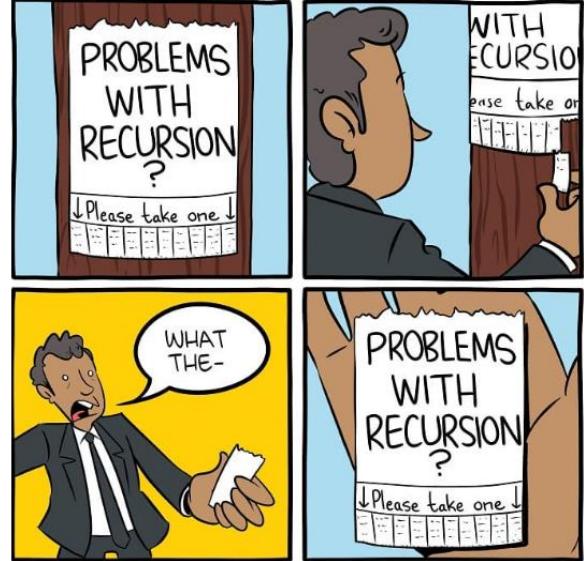
Übungen

Woche 6: References, Recursion, Precondition

Timo Baumberger

Departement Informatik

ETH Zürich



Organisatorisches

- Mein Name: Timo Baumberger
- Bei Fragen: tbaumberger@student.ethz.ch
(Discord: troxhi)
 - Mails bitte mit «[EProg25]» im Betreff
- Meine Website: timobaumberger.com
- Neue Aufgaben: **Dienstag Abend** (im Normalfall)
- Abgabe der Übungen bis **Dienstag Abend (23:59)** Folgewoche
 - Abgabe immer via Git
 - Lösungen in separatem Projekt auf Git



Programm

- **Bonusaufgabe von u05**
- **Java References**
- **Rekursion**
- **Weakest Precondition**
- **Vorbesprechung**
- **Nachbesprechung**

Bonusaufgabe von u05

```
public static boolean containsSubstringAt(String str, int position, String sub) { 17 usages new *
    return str.startsWith(sub, position);
}

public static int countSubstrings(String str, String sub) { 22 usages new *
    if (str.length() < sub.length()) {
        return 0;
    }
    int occurrences = 0;
    for (int i = 0; i < str.length(); i++) {
        if (containsSubstringAt(str, i, sub)) {
            occurrencesoccurrences;
}
```

```
public static int countDisjointSubstrings(String str, String sub) {  
    if (str.length() < sub.length()) {  
        return 0;  
    }  
  
    int occurrences = 0;  
    int i = 0;  
    while (i <= str.length() - sub.length()) {  
        if (containsSubstringAt(str, i, sub)) {  
            occurrences++;  
            i += sub.length();  
        } else {  
            i++;  
        }  
    }  
  
    return occurrences;  
}
```

Geht es besser als $O(n*m)$?

Es geht besser: In Zeit O(n)

- Z-Funktion kann verwendet werden
- Die Z-Funktion von einem String s ist ein int Array der Länge `s.length()` wobei der i'te Eintrag im Array die grösste Anzahl an Zeichen ist, sodass ab Index i in s die Zeichen mit dem Präfix von s übereinstimmen

Let us assume `str.length() == n` and `sub.length() == m`. We only continue if $n \geq m$. Thus we can conclude that computing the Z-function takes $n + m \leq n + n = 2n \leq O(n)$. The definition of the Z-function is as follows: $z[i] =$ The greatest number of characters starting from the position i that coincide with the first characters of s . Since our s is the concatenation of sub and str , we can check for every index $i \geq sub.length()$ (the first `sub.length()` chars are skipped) if the chars starting from position i coincide with the first `sub.length()` chars of s . This would mean that the substring at this index is equal to `sub`.

Params: `str`
`sub`

Returns:

```
21 usages new *
public static int countSubstrings(String str, String sub) {
    if (str.length() < sub.length()) {
        return 0;
    }
    String mergedString = sub + str;
    int[] z = calculateZFunction(mergedString);
    int occurrences = 0;
    for (int i = sub.length(); i < mergedString.length(); i++) {
        if (z[i] >= sub.length()) {
            occurrencesoccurrences;
}
```

References

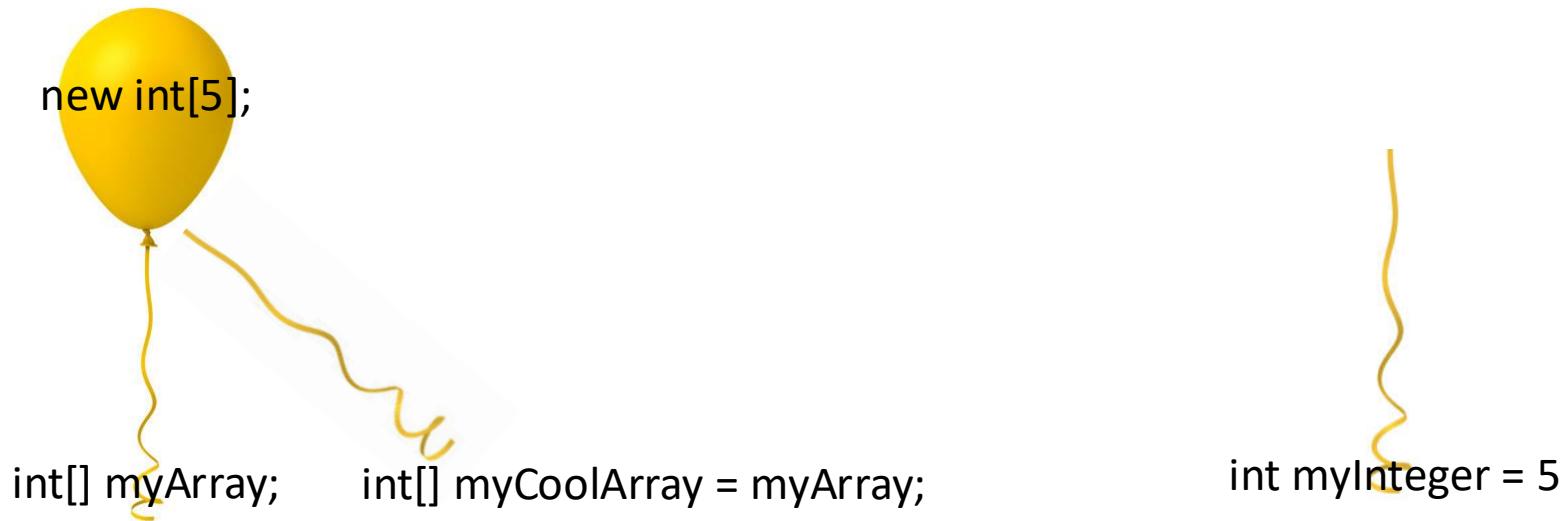
Value vs Reference

- Variablen und Parameter enthalten entweder einen **Wert** (Value) oder eine **Referenz** (Reference).
- Variablen enthalten **nie** Objekte.
- Referenzen zeigen auf das Objekt im Speicher.
- Der Typ einer Variable in Java ist entweder **Primitive Type** (int, boolean, long, double, char, etc.) oder **Reference Type**
- Jeder Reference Type Variable kann **null** zugewiesen werden

Value vs Reference Analogie



- Jedes Objekt (Strings, Arrays, etc.) ist ein Ballon
- Variablen / Parameter sind die Ballonschnur



Value vs Reference - Beispiel

```
1 public class Example {  
2     public static void setX(int x) {  
3         x = 2;  
4     }  
5  
6     public static void main(String[] args) {  
7         int x = 3;  
8         setX(x);  
9         System.out.println(x); → 3  
10    }  
11 }
```

Value vs Reference - Beispiel

```
1 public class Example {  
2     public static void setX(int x) {  
3         x = 2;  
4     }  
5  
6     public static void main(String[] args) {  
7         int x = 3;  
8         setX(x);  
9         System.out.println(x);  
10    }  
11 }
```

Scope setX

Scope main

Value vs Reference

- Referenzen befolgen auch Value Semantics:
 - In Java werden Variablen, die auf Objekte verweisen, tatsächlich als Referenzen behandelt.
 - Beim Zuweisen einer Referenz zu einer anderen wird nur die Speicheradresse (Referenz) kopiert, nicht das tatsächliche Objekt.
 - Jede Referenz speichert die Adresse des Objekts, auf das sie zeigt.
- Nur die Objekte selbst ermöglichen Reference Semantics:
 - Mehrere Variablen / Referenzen können auf dasselbe Objekt im Speicher zeigen (z.B. A obj1 = new A(); A obj2 = obj1;).
 - Referenzen in Java sind nicht selbstständige Kopien des Objekts, sondern Zeiger auf den gleichen Speicherort.

Value vs Reference - Beispiel

```
1 import java.util.Arrays;  
2  
3 public class Example {  
4     public static void setX(int[] x) {  
5         x = new int[] {4, 5, 6};  
6     }  
7  
8     public static void main(String[] args) {  
9         int[] x = new int[] {1, 2, 3}; X  
10        setX(x);  
11        String xStr = Arrays.toString(x);  
12        System.out.println(xStr);  
13    }  
14}
```

[4,5,6]

[1,2,3]

Value vs Reference - Beispiel

```
1 import java.util.Arrays;  
2  
3 public class Example {  
4     public static void setX(int[] x) {  
5         x = new int[] {4, 5, 6};  
6     }  
7  
8     public static void main(String[] args) {  
9         int[] x = new int[] {1, 2, 3};  
10        setX(x);  
11        String xStr = Arrays.toString(x);  
12        System.out.println(xStr);  
13    }  
14 }
```

[4,5,6]

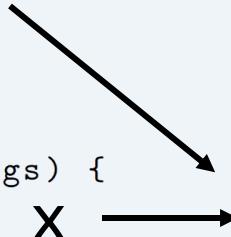
[1,2,3]

Value vs Reference - Beispiel

```
1 import java.util.Arrays;  
2  
3 public class Example {  
4     public static void setX(int[] x) {  
5         → x = new int[] {4, 5, 6};  
6     }  
7  
8     public static void main(String[] args) {  
9         int[] x = new int[] {1, 2, 3};  
10        → setX(x);  
11        String xStr = Arrays.toString(x);  
12        System.out.println(xStr);  
13    }  
14 }
```

[4,5,6]

[1,2,3]



Value vs Reference - Beispiel

```
1 import java.util.Arrays;  
2  
3 public class Example {  
4     public static void setX(int[] x) {  
5         → x = new int[] {4, 5, 6};  
6     }  
7  
8     public static void main(String[] args) {  
9         int[] x = new int[] {1, 2, 3};  
10        → setX(x);  
11        String xStr = Arrays.toString(x);  
12        System.out.println(xStr);  
13    }  
14 }
```

[4,5,6]

X → [1,2,3]

Value vs Reference - Beispiel

```
1 import java.util.Arrays;  
2  
3 public class Example {  
4     public static void setX(int[] x) {  
5         x = new int[] {4, 5, 6};  
6     }  
7  
8     public static void main(String[] args) {  
9         int[] x = new int[] {1, 2, 3};  
10        setX(x);  
11        String xStr = Arrays.toString(x);  
12        System.out.println(xStr);  
13    }  
14 }
```

[4,5,6]

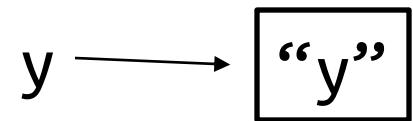
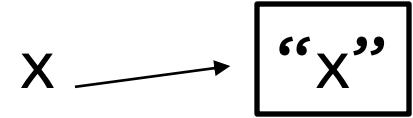
[1,2,3]

“[1,2,3]”



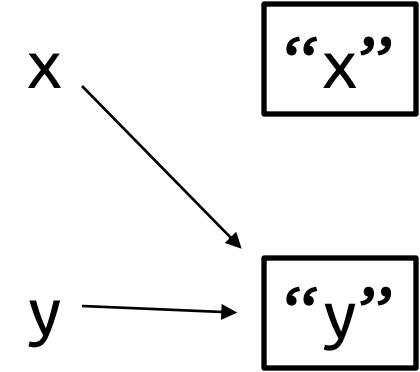
Value vs Reference - Beispiel

```
1 String x = "x";
2 String y = "y";
3
4 //Assignment
5 x = y;
```



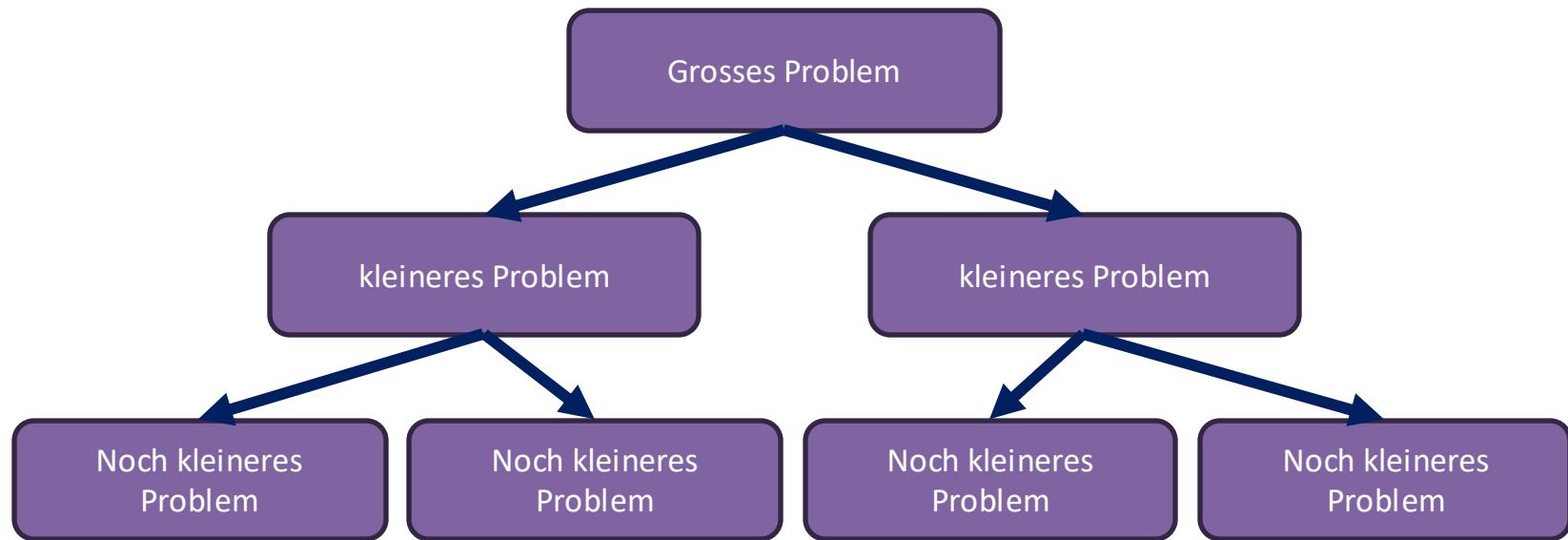
Value vs Reference - Beispiel

```
1 String x = "x";
2 String y = "y";
3
4 //Assignment
5 x = y;
```

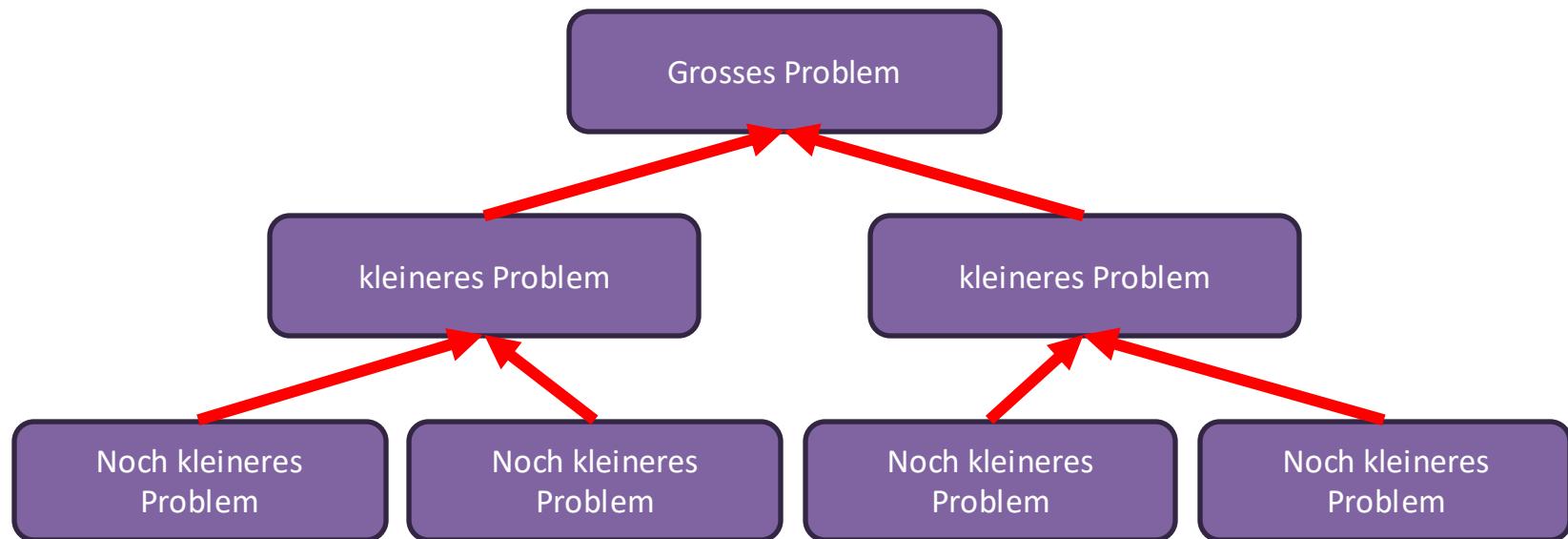


Rekursion

Rekusion – Grundprinzip Divide and Conquer



Rekusion – Grundprinzip Divide and Conquer



Decrease-and-Conquer - Fakultät

factorial(3)



```
public int factorial(int n) {  
    if (n==1) return 1;  
    return n*factorial(n-1);  
}
```

Decrease-and-Conquer - Fakultät



```
public int factorial(int n) {  
    if (n==1) return 1;  
    return n*factorial(n-1);  
}
```

3*factorial(2)

Decrease-and-Conquer - Fakultät



```
public int factorial(int n) {  
    if (n==1) return 1;  
    return n*factorial(n-1);  
}
```

3*factorial(2)

factorial(2)

Decrease-and-Conquer - Fakultät



```
public int factorial(int n) {  
    if (n==1) return 1;  
    return n*factorial(n-1);  
}
```

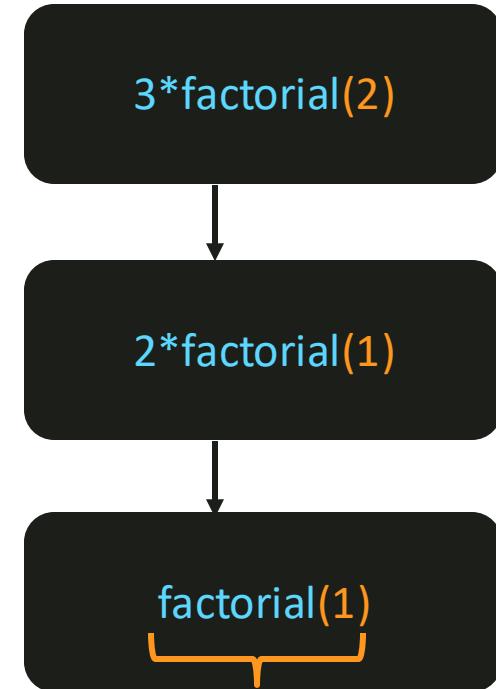
3*factorial(2)

2*factorial(1)

Decrease-and-Conquer - Fakultät



```
public int factorial(int n) {  
    if (n==1) return 1;  
    return n*factorial(n-1);  
}
```

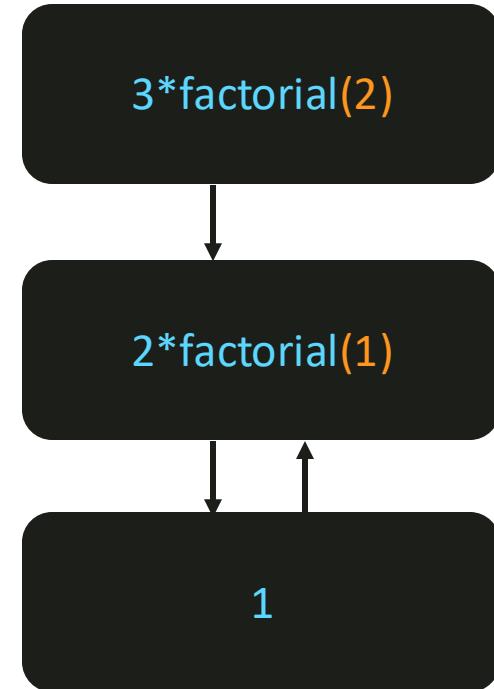


Base Case!

Decrease-and-Conquer - Fakultät



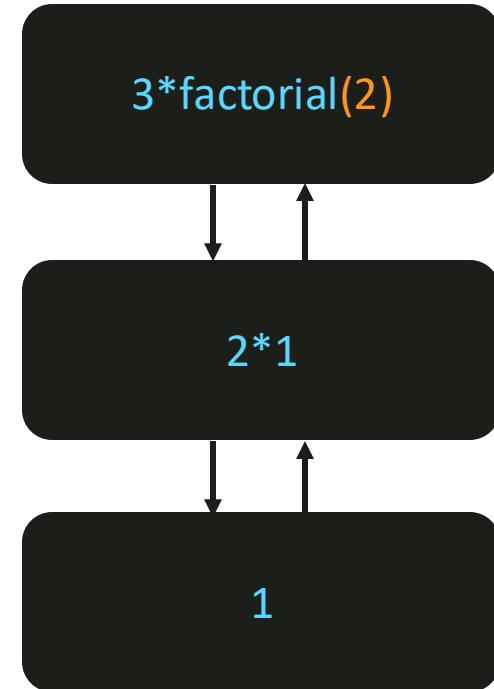
```
public int factorial(int n) {  
    if (n==1) return 1;  
    return n*factorial(n-1);  
}
```



Decrease-and-Conquer - Fakultät



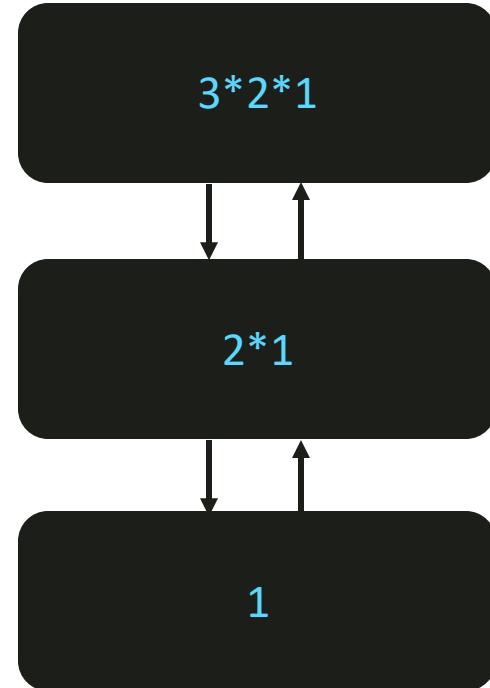
```
public int factorial(int n) {  
    if (n==1) return 1;  
    return n*factorial(n-1);  
}
```



Decrease-and-Conquer - Fakultät



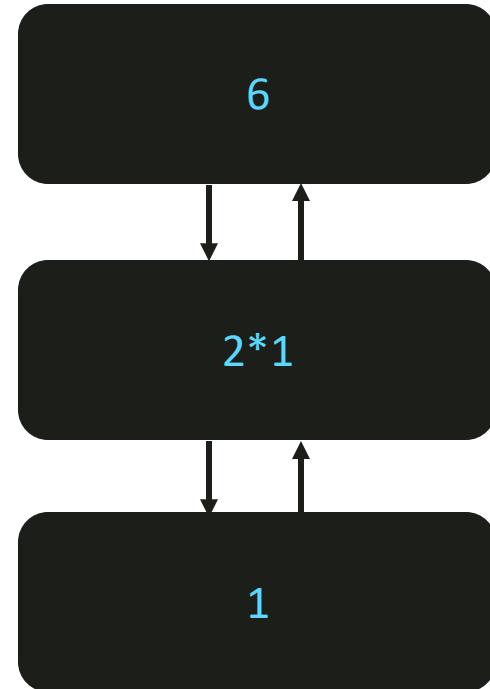
```
public int factorial(int n) {  
    if (n==1) return 1;  
    return n*factorial(n-1);  
}
```



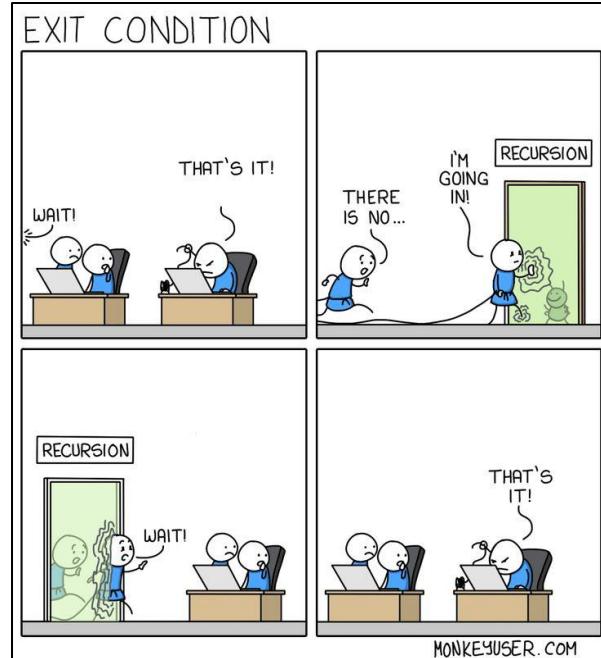
Decrease-and-Conquer - Fakultät



```
public int factorial(int n) {  
    if (n==1) return 1;  
    return n*factorial(n-1);  
}
```



Rekursion in Java – Base Case



Exception in thread "main"
java.lang.StackOverflowError

Rekursion – Step by Step

Problem, das wir lösen wollen

Aufgabe: Schreiben Sie eine Funktion $\text{sum}(n)$, die gegeben eine nichtnegative Zahl, alle nichtnegativen Zahlen bis einschliesslich n aufsummiert.

Beispiele:

$$\text{sum}(0) = 0$$

$$\text{sum}(1) = 1 = 1 + 0$$

$$\text{sum}(4) = 10 = 4 + 3 + 2 + 1 + 0$$

Wir wollen dies nun **rekursiv** lösen.

Schritte

1. Was ist die einfachste mögliche Eingabe?
2. Beispiele ausprobieren und visualisieren
3. Leite größere Beispiele von kleineren Beispielen ab
4. Verallgemeinere das Muster
5. Schreibe den Code

Schritte

1. Was ist die einfachste mögliche Eingabe?

→ Base Case

2. Beispiele ausprobieren und visualisieren

3. Leite größere Beispiele von kleineren Beispielen ab

4. Verallgemeinere das Muster

→ Rekursionsschritt

5. Schreibe den Code

1. Die einfachste Eingabe

- Hier ist es $n = \underline{\quad}$: $\text{sum}(n) = \underline{\quad}$
- Die einfachste Eingabe ist später oft unser *Base Case*.
- Der Base Case ist der einzige Fall einer rekursiven Funktion, in dem wir eine direkte Lösung angeben.
- Eine Funktion kann auch mehrere Base Cases haben.
- Alle anderen Lösungen bauen auf dem Base Case auf.

1. Die einfachste Eingabe

- Hier ist es $n = 0$: $\text{sum}(0) = 0$
- Die einfachste Eingabe ist später oft unser *Base Case*.
- Der Base Case ist der einzige Fall einer rekursiven Funktion, in dem wir eine direkte Lösung angeben.
- Eine Funktion kann auch mehrere Base Cases haben.
- Alle anderen Lösungen bauen auf dem Base Case auf.

Schritte

1. Was ist die einfachste mögliche Eingabe?

→ Base Case



2. Beispiele ausprobieren und visualisieren

3. Leite größere Beispiele von kleineren Beispielen ab

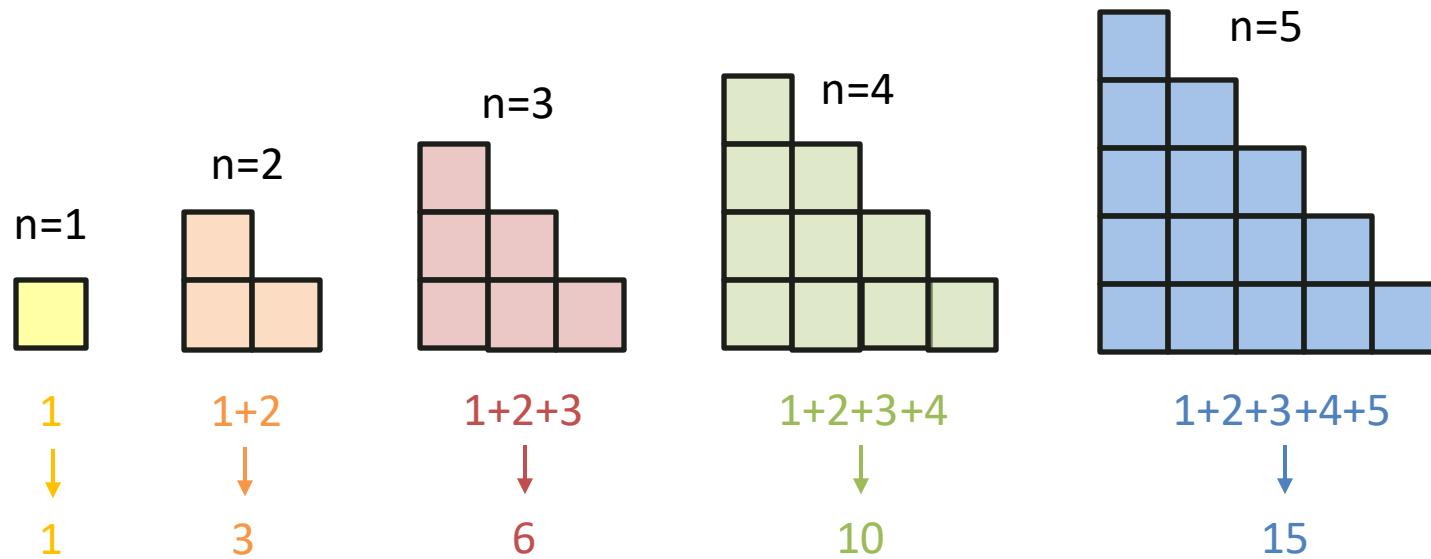
4. Verallgemeinere das Muster

→ Rekursionsschritt

5. Schreibe den Code

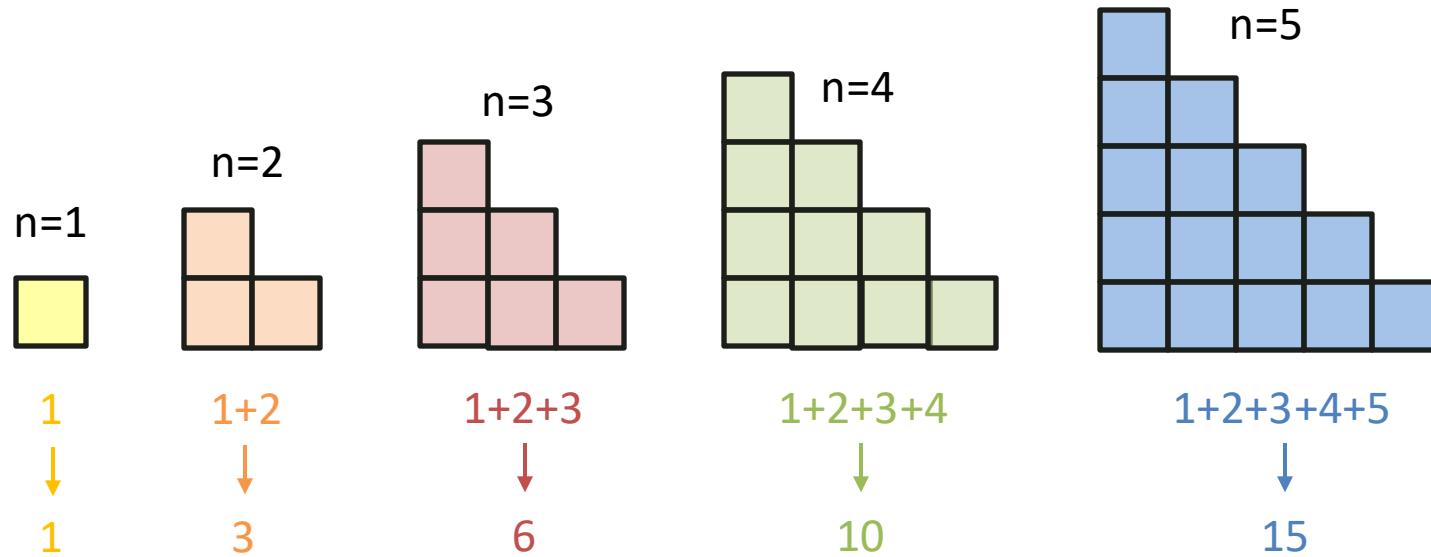
2. Die Beispiele

- Visualisiere, wie die Ein- und Ausgaben der Funktion miteinander zusammenhängen



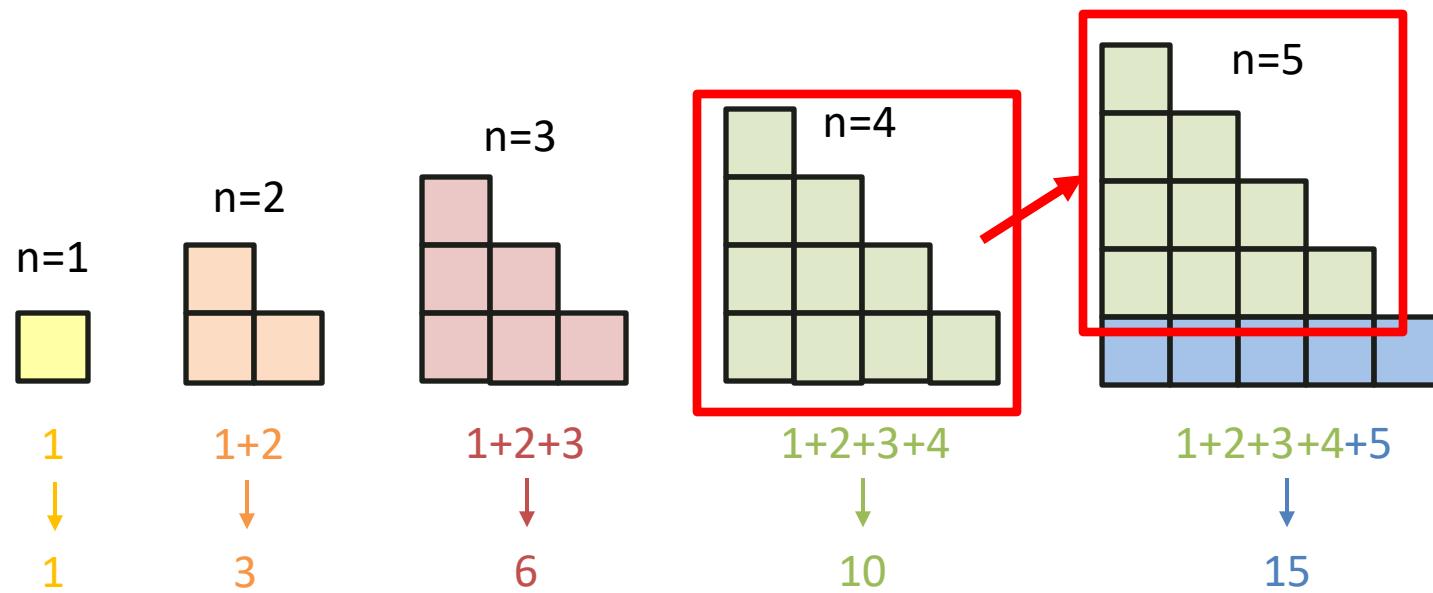
3. Erkenne den Zusammenhang

- „Wenn ich die Antwort für $n=4$ gegeben hätte, wie komme ich auf die Antwort für $n=5$? Für $n=3$ auf $n=4$? ...“



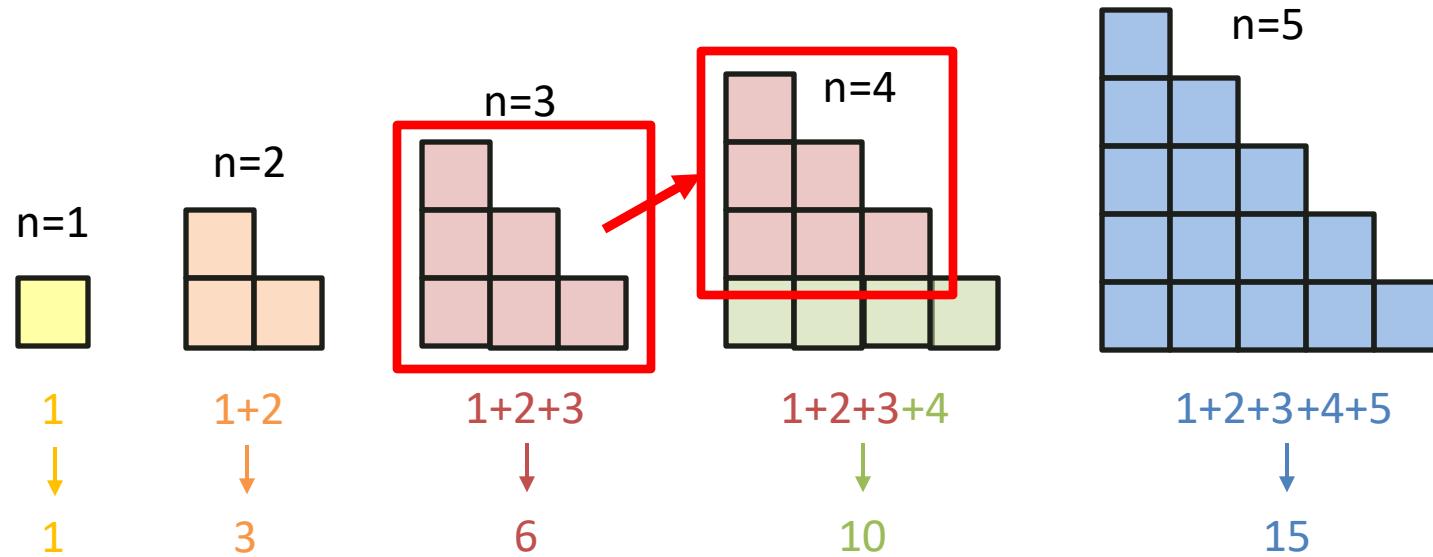
3. Erkenne den Zusammenhang

- „Wenn ich die Antwort für $n=4$ gegeben hätte, wie komme ich auf die Antwort für $n=5$? Für $n=3$ auf $n=4$? ...“



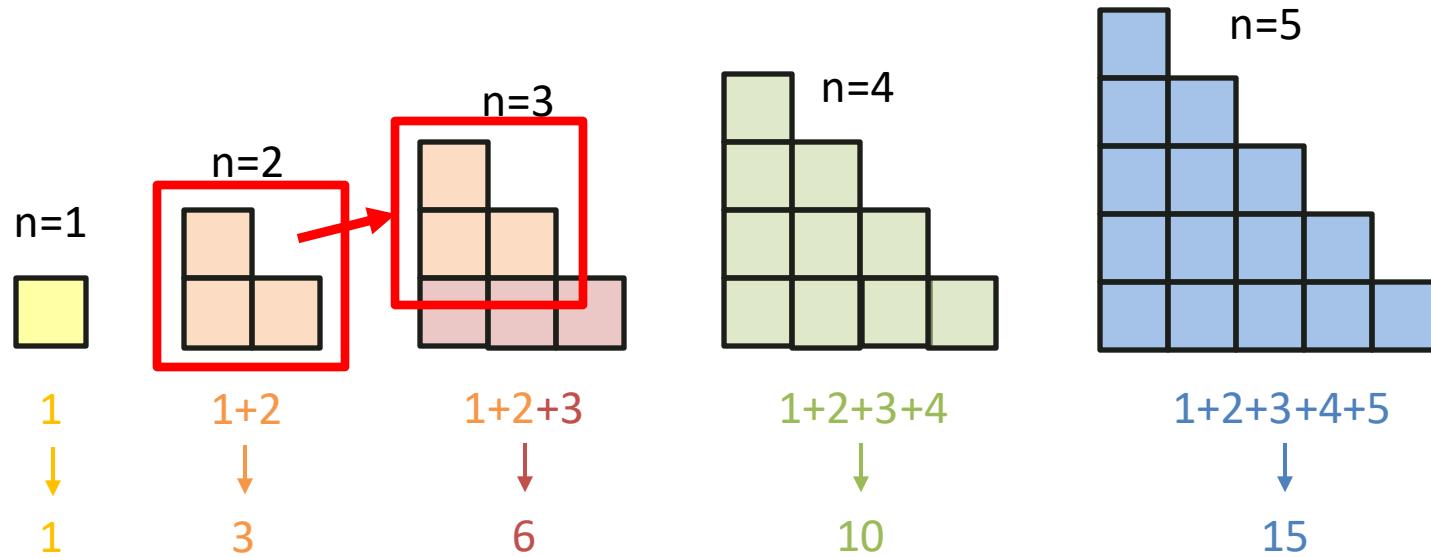
3. Erkenne den Zusammenhang

- „Wenn ich die Antwort für $n=4$ gegeben hätte, wie komme ich auf die Antwort für $n=5$? Für $n=3$ auf $n=4$? ...“



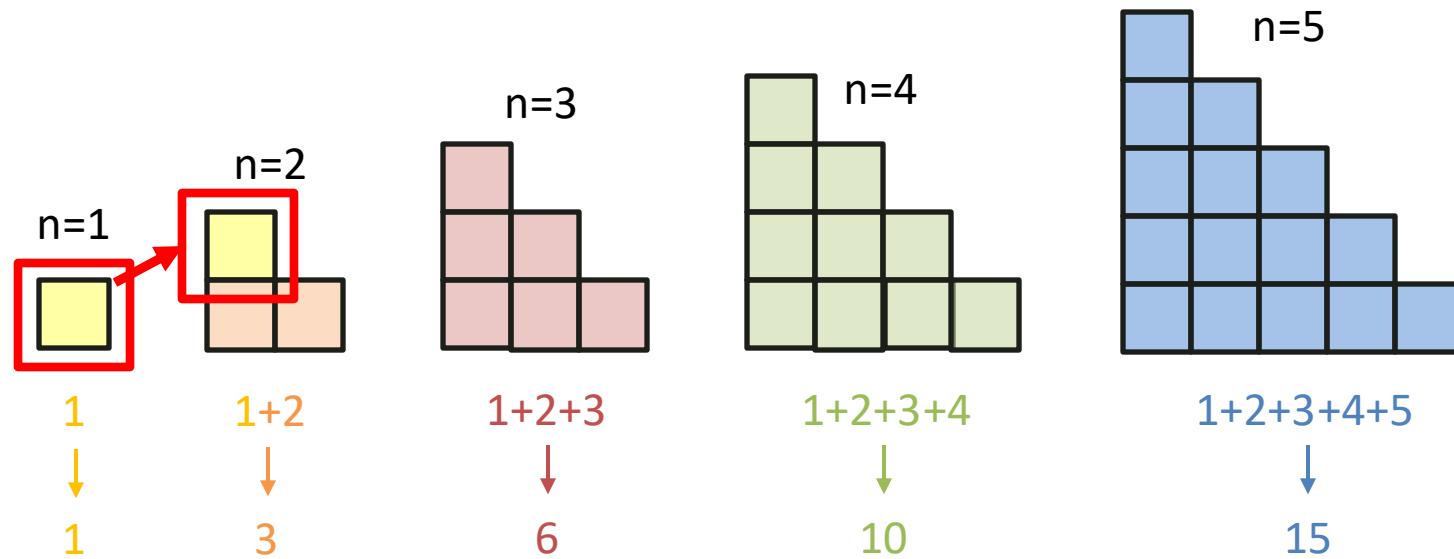
3. Erkenne den Zusammenhang

- „Wenn ich die Antwort für $n=4$ gegeben hätte, wie komme ich auf die Antwort für $n=5$? Für $n=3$ auf $n=4$? ...“

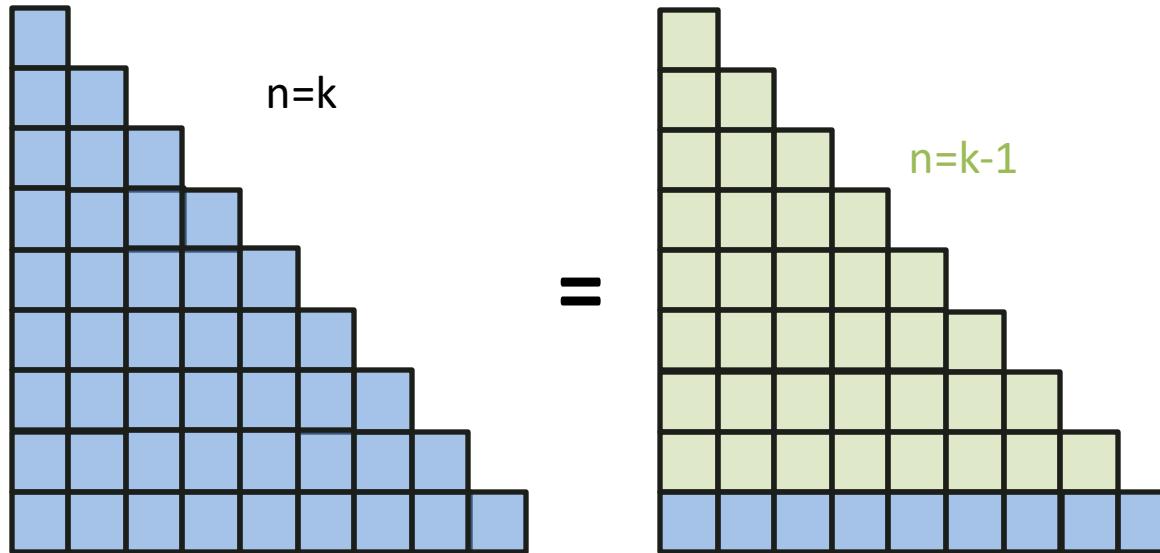


3. Erkenne den Zusammenhang

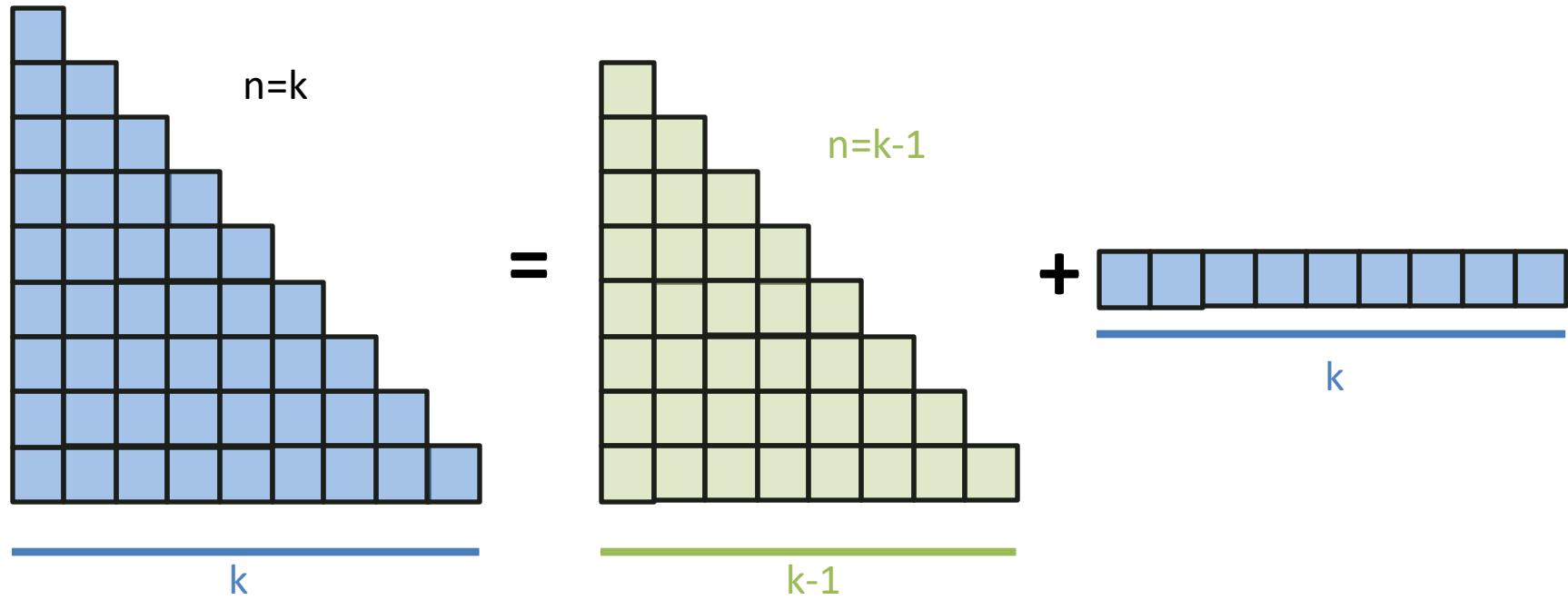
- „Wenn ich die Antwort für $n=4$ gegeben hätte, wie komme ich auf die Antwort für $n=5$? Für $n=3$ auf $n=4$? ...“



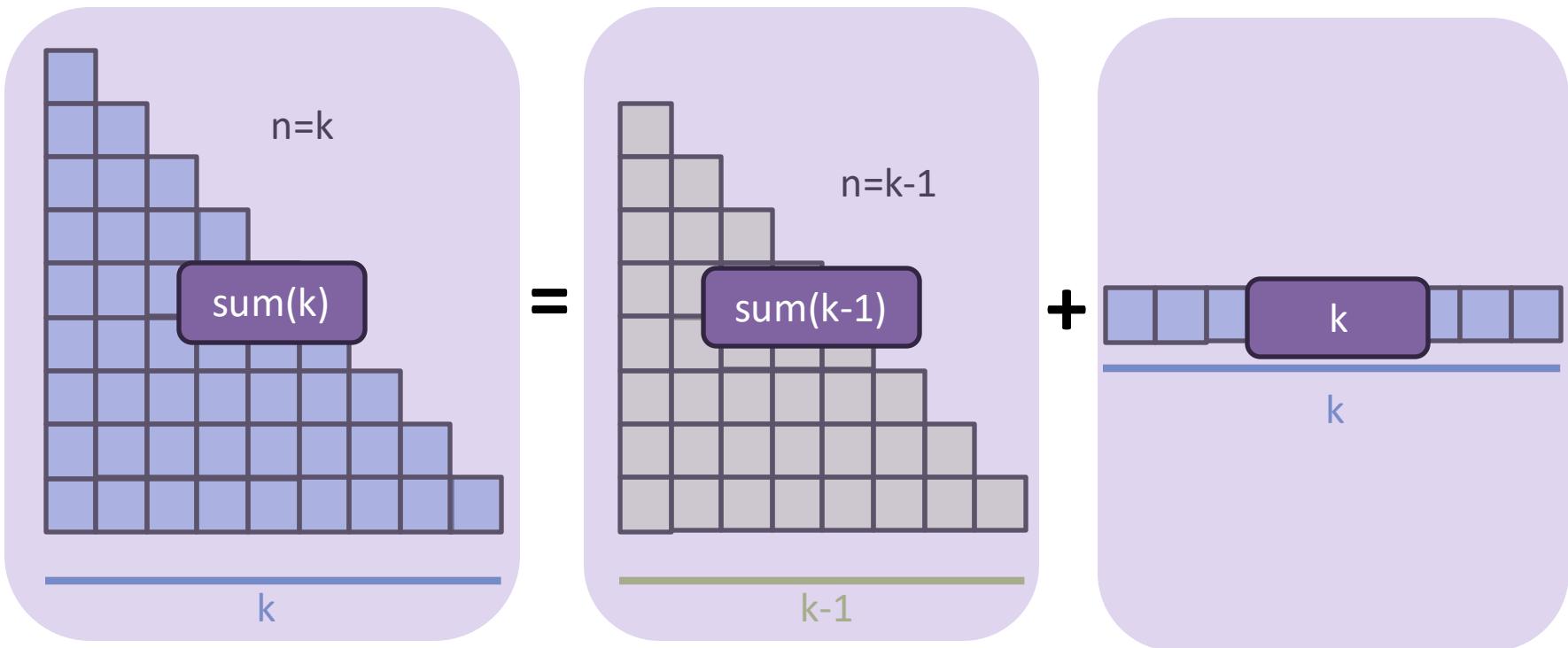
4. Das Muster



4. Das Muster



4. Das Muster



Schritte

1. Was ist die einfachste mögliche Eingabe?

→ Base Case



2. Beispiele ausprobieren und visualisieren

3. Leite größere Beispiele von kleineren Beispielen ab

4. Verallgemeinere das Muster

→ Rekursionsschritt



5. Schreibe den Code

5. Der Code

Was haben wir herausgefunden?

Im 1. Schritt haben wir den *Base Case* gefunden:

Falls $n=0$, dann $\text{sum}(n) = 0$

Danach haben wir das *Rekursionsmuster* gefunden:

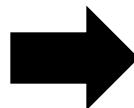
$\text{sum}(n) = \text{sum}(n-1)+n$

→ Dies können wir jetzt in Code umschreiben

5. Base Case + Muster → Code

Falls $n=0$: $\text{sum}(n) = 0$

Sonst: $\text{sum}(n) = \text{sum}(n-1)+n$

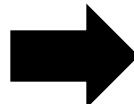


```
public static int sum(int n) {  
    }  
}
```

5. Base Case + Muster → Code

Falls $n=0$: $\text{sum}(n) = 0$

Sonst: $\text{sum}(n) = \text{sum}(n-1)+n$

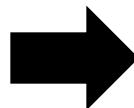


```
public static int sum(int n) {  
    if (n == 0) {  
        return 0;  
    }  
}
```

5. Base Case + Muster → Code

Falls $n=0$: $\text{sum}(n) = 0$

Sonst: $\text{sum}(n) = \text{sum}(n-1)+n$

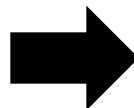


```
public static int sum(int n) {  
    if (n == 0) {  
        return 0;  
    }  
}
```

5. Base Case + Muster → Code

Falls $n=0$: $\text{sum}(n) = 0$

Sonst: $\text{sum}(n) = \text{sum}(n-1)+n$



```
public static int sum(int n) {  
    if (n == 0) {  
        return 0;  
    }  
    return sum(n-1) + n;  
}
```

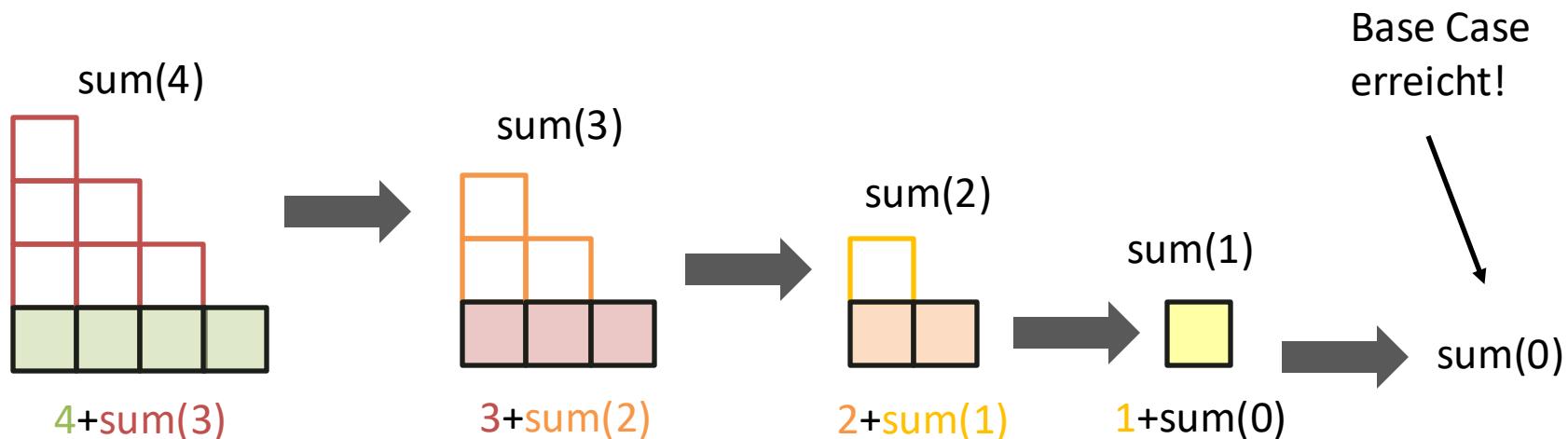
5. Base Case + Muster → Code

```
public static int sum(int n) {  
    if (n == 0) {  
        return 0;  
    }  
    return sum(n-1) + n;  
}
```

Fragen?

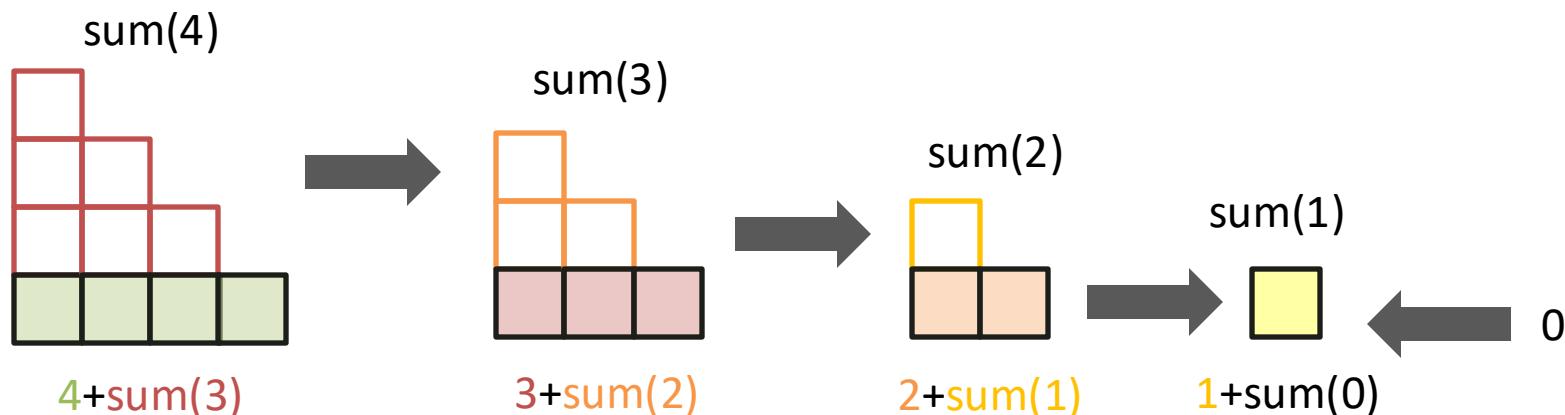
Was passiert wenn ich diesen Code ausführe?

1. Rekursive Aufrufe



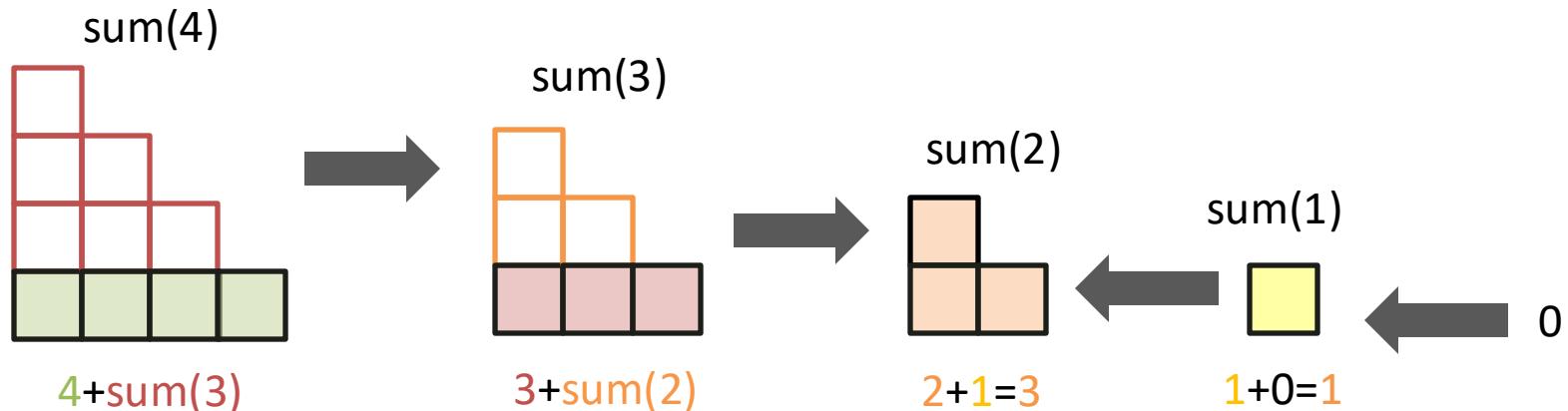
Was passiert wenn ich diesen Code ausführe?

2. Auflösen



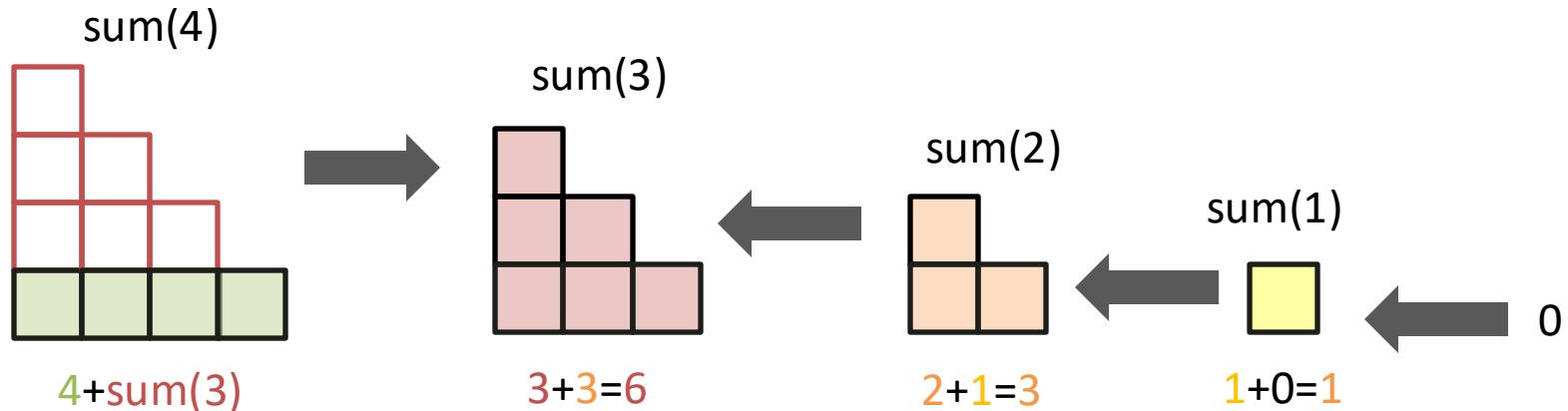
Was passiert wenn ich diesen Code ausführe?

2. Auflösen



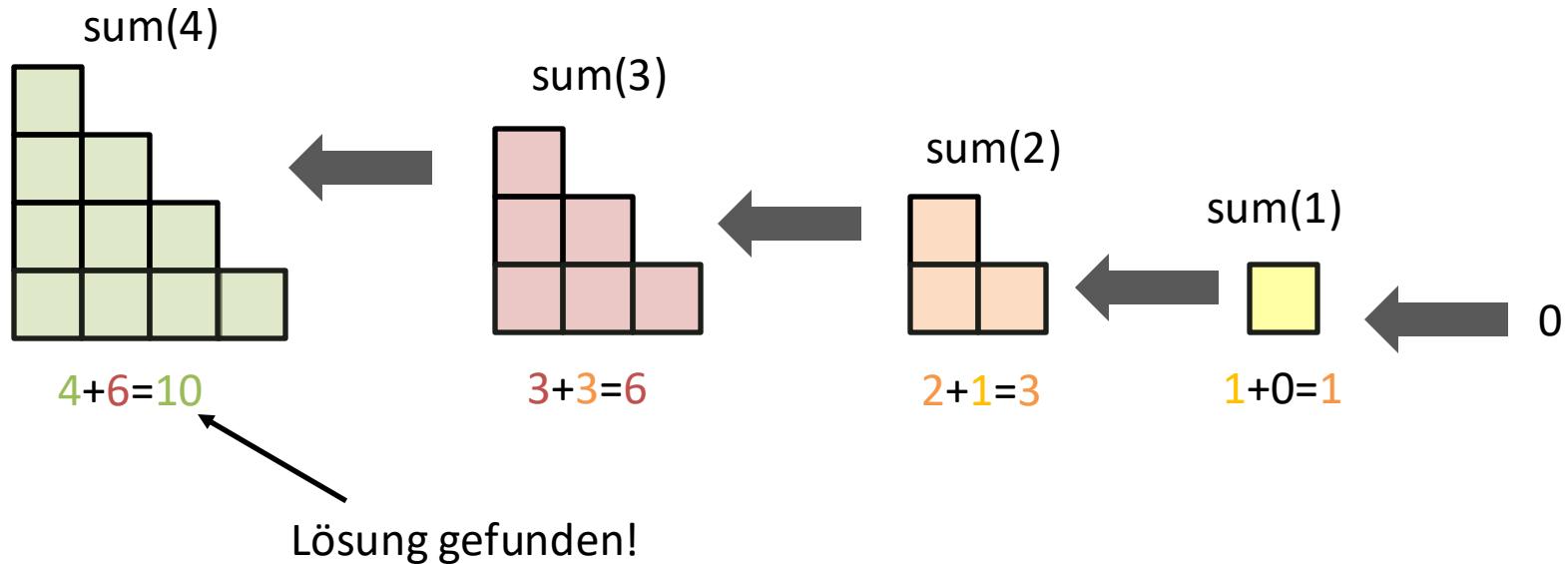
Was passiert wenn ich diesen Code ausführe?

2. Auflösen



Was passiert wenn ich diesen Code ausführe?

2. Auflösen



Rekursion in Java – Aufgabe

Erstelle eine Klasse `CheckIsPalindromRecursive`.

Implementiere eine **statische, rekursive Methode** `IsPalindrome`, die:

- Einen String akzeptiert
- Einen **boolean** zurückgibt, ob der gesamte String ein Palindrom ist.

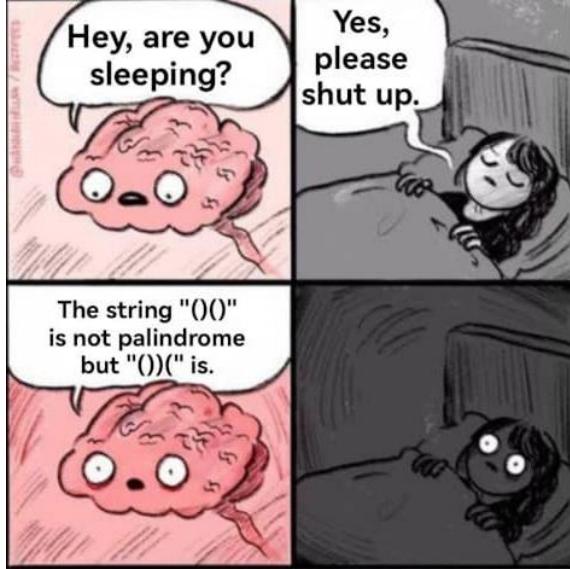
Basisfall der Rekursion:

- Wenn die String-Länge 0 oder 1 beträgt, gib `true` zurück
- Bei einer Länge von 2: Prüfe, ob beide Zeichen gleich sind, und gib entsprechend `true` oder `false` zurück.

Rekursiver Fall:

- Nimm das **erste** und **letzte** Zeichen des Strings.
- Überprüfe, ob sie **gleich** sind.
- Verknüpfe das Ergebnis logisch mit einem Aufruf von `IsPalindrome` für den Teilstring ohne das erste und letzte Zeichen.

Tipp: Benutze `str.charAt(i)`, `str.substring(i, j)`



Rekursion in Java – Lösung



```
1 public class Palindrome {  
2     public static boolean isPalindrome(String word) {  
3         if (word.length() <= 1)  
4             return true;  
5         boolean firstAndLast = word.charAt(0) == word.charAt(word.length() - 1);  
6         String withoutFirstAndLast = word.substring(1, word.length() - 1);  
7         return firstAndLast && isPalindrome(withoutFirstAndLast);  
8     }  
9 }
```

Rekursion mit Debugger

The screenshot shows a Java code editor and a debugger interface. The code in the editor is:

```
1 public class CheckIsPalindrome {
2
3     public static void main(String[] args) {    args: []
4         boolean result = isPalindrome(word: "wow");
5         System.out.println(result);
6     }
7
8     public static boolean isPalindrome(String word) {  2 usages
9
10        int wordLength = word.length();
11
12        if (wordLength <= 1) {
13            return true;
14        }
15
16        boolean firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);
17
18        String withoutFirstAndLast = word.substring(1, wordLength - 1);
19        return firstAndLast && isPalindrome(withoutFirstAndLast);
20    }
21 }
```

The debugger interface shows the current state of the application. The thread is labeled "main" and is running. The stack trace shows the method call: "main:4, checkIsPalindrome". The arguments are listed as "args = {String[0]@825} []".

- Wir setzen den Breakpoint dort, wo die Methode **isPalindrome** aufgerufen wird. Anschließend gehen wir mit 'Step Into' in die Methode, wobei das Argument **word** = "**wow**" ist.

```
1 public class checkIsPalindrome {  
2     public static void main(String[] args) {  
3         boolean result = isPalindrome(word: "wow");  
4         System.out.println(result);  
5     }  
6     public static boolean isPalindrome(String word) { 2 usages      word: "wow"  
7  
8         int wordLength = word.length();  wordLength: 3  
9  
10        if (wordLength <= 1) {  
11            return true;  
12        }  
13  
14        boolean firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);  firstAndLast: true  
15  
16        String withoutFirstAndLast = word.substring(1, wordLength - 1);  withoutFirstAndLast: "o"  
17  
18        return firstAndLast = true && isPalindrome(withoutFirstAndLast);  firstAndLast: true  withoutFirstAndLast: "o"  
19    }  
20 }
```

- Wir gehen mit 'Step Over' durch die Methode, und die Variablen **word**, **wordLength**, **firstAndLast** erscheinen sowohl unter „Threads & Variables“ als auch in der Zeile , nachdem sie deklariert wurden.
- In Zeile 19 rufen wir die Methode **isPalindrome** erneut auf, diesmal innerhalb der Methode **isPalindrome** (wir gehen in die Rekursion). Das Argument in diesem Aufruf der Methode ist **withoutFirstAndLast** (= "o").

The screenshot shows a Java code editor with a dark theme. A code review interface is overlaid on the editor. The code in the editor is:

```
1 public class checkIsPalindrome {  
2     public static void main(String[] args) {  
3         boolean result = isPalindrome(word: "wow");  
4         System.out.println(result);  
5     }  
6  
8     @  
9     public static boolean isPalindrome(String word) { 2 usages      word: "o"  
10  
11         int wordLength = word.length();  word: "o"  
12  
13         if (wordLength <= 1 = true ) {  
14             return true;  
15         }  
16  
17         boolean firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);  
18  
19         String withoutFirstAndLast = word.substring(1, wordLength - 1);  
20         return firstAndLast && isPalindrome(withoutFirstAndLast);  
21     }  
}
```

The code review interface includes a toolbar with icons for file operations and a 'Threads & Variables' tab. The 'Threads & Variables' tab shows the current thread state:

- "main" @1 in...in: RUNNING
- isPalindrome:10, checkIsPalindrome
- isPalindrome:19, checkIsPalindrome
- main:4, checkIsPalindrome

It also shows variable values:

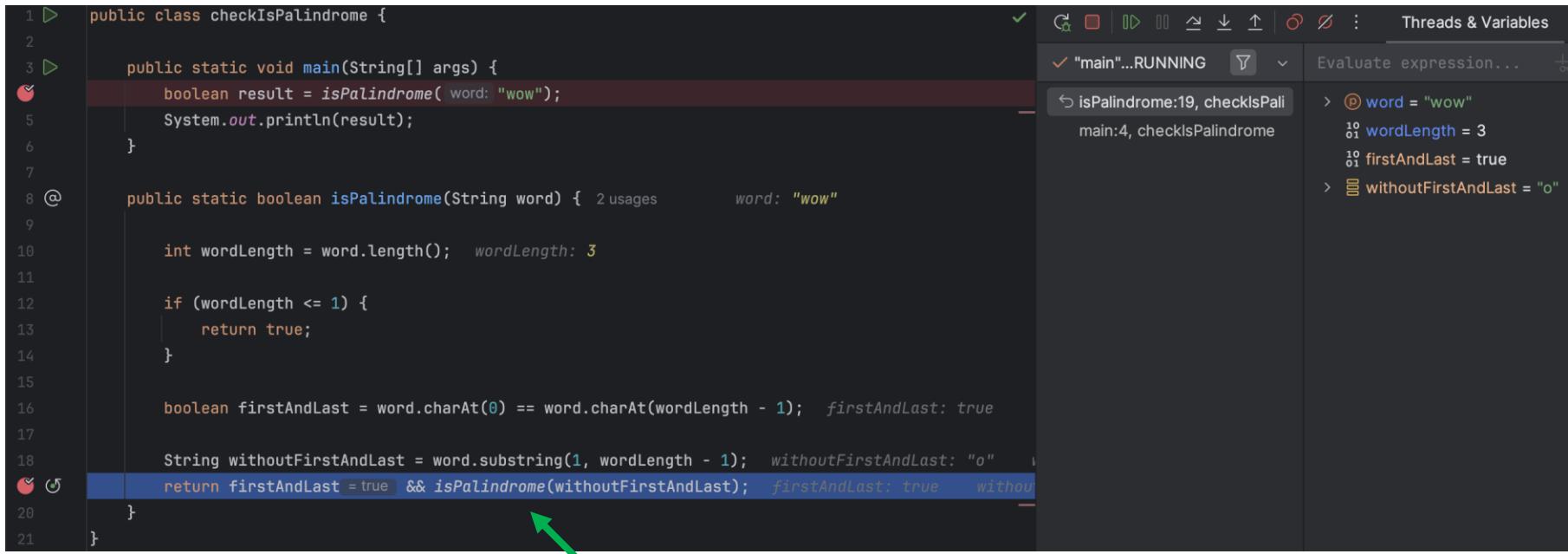
- word = "o"

- Wir sind wieder in die Methode eingetreten, aber dieses Mal mit **word = "o"**.
- Beachte, dass die Werte, die im vorherigen Methodenaufruf von `isPalindrome` sichtbar waren, nicht mehr im Scope sind, und daher auch nicht in unter „Threads & Variables“ .

```
1 D public class checkIsPalindrome {  
2  
3 D     public static void main(String[] args) {  
4         boolean result = isPalindrome(word: "wow");  
5         System.out.println(result);  
6     }  
7  
8 @     public static boolean isPalindrome(String word) { 2 usages      word: "o"  
9  
10    int wordLength = word.length();  wordLength: 1      word: "o"  
11  
12    if (wordLength <= 1) {  wordLength: 1  
13        return true;  
14    }  
15  
16    boolean firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);  
17  
18    String withoutFirstAndLast = word.substring(1, wordLength - 1);  
19    return firstAndLast && isPalindrome(withoutFirstAndLast);  
20 }  
21 }
```

The screenshot shows a Java code editor with a debugger interface. The code is a palindrome checker. A break point is set at the line `return true;` (line 13). The debugger shows the current stack trace: `main@1 in...in: RUNNING` with frames `isPalindrome:13, checkIsPalindrome`, `isPalindrome:19, checkIsPalindrome`, and `main:4, checkIsPalindrome`. The variable `word` is set to "o" and its length `wordLength` is 1.

- "Da **wordLength = 1** ist, ist der Base Case erfüllt. Somit betreten wir das **if-Statement** und es wird **true** zurückgegeben.
- Aber wohin wird es zurückgegeben?



```

1 public class checkIsPalindrome {
2
3     public static void main(String[] args) {
4         boolean result = isPalindrome(word: "wow");
5         System.out.println(result);
6     }
7
8     public static boolean isPalindrome(String word) { 2 usages      word: "wow"
9
10        int wordLength = word.length();    wordLength: 3
11
12        if (wordLength <= 1) {
13            return true;
14        }
15
16        boolean firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);    firstAndLast: true
17
18        String withoutFirstAndLast = word.substring(1, wordLength - 1);    withoutFirstAndLast: "o"  ↴
19        return firstAndLast = true && isPalindrome(withoutFirstAndLast);    firstAndLast: true    withoutFirstAndLast: "o"
20    }
21 }

```

The screenshot shows a Java code editor with a debugger interface. The code defines a class `checkIsPalindrome` with a `main` method that calls `isPalindrome` with the argument "wow". The `isPalindrome` method checks if a word is a palindrome by comparing its first and last characters. If they are equal, it then checks the substring from index 1 to `wordLength - 1`. A green arrow points to the line `return firstAndLast = true && isPalindrome(withoutFirstAndLast);` in the `isPalindrome` method. The debugger sidebar shows variable values: `word = "wow"`, `wordLength = 3`, `firstAndLast = true`, and `withoutFirstAndLast = "o"`. The status bar indicates the thread is running.

- **True** wird an die Stelle zurückgegeben, an der die Methode zuletzt aufgerufen wurde. Die Methode wurde zuletzt mit 'o' aufgerufen. Nun zeigt auch „Threads & Variables“ wieder die Werte der Variablen, wie sie beim vorherigen Methodenaufruf waren.
- Jetzt wird zuerst **firstAndLast && True** (Rückgabewert) ausgewertet. Das ergibt **True**, was weiter zurückgegeben wird, an die Stelle, an der `isPalindrome` mit `word = "wow"` aufgerufen wurde.

The screenshot shows a Java code editor with a dark theme. On the left, the code for `checkIsPalindrome` is displayed:

```
1 public class checkIsPalindrome {  
2     public static void main(String[] args) {    args: []  
3         boolean result = isPalindrome( word: "wow");  result: true  
4         System.out.println(result = true );   result: true  
5     }  
6     public static boolean isPalindrome(String word) {  2 usages  
7         int wordLength = word.length();  
8         if (wordLength <= 1) {  
9             return true;  
10        }  
11        boolean firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);  
12        String withoutFirstAndLast = word.substring(1, wordLength - 1);  
13        return firstAndLast && isPalindrome(withoutFirstAndLast);  
14    }  
15 }  
16 }
```

The code editor highlights several lines in red, indicating errors or warnings. The right side of the interface shows the debugger's state:

- A green checkmark icon is present.
- The status bar shows "Threads & Variables".
- The stack trace indicates "main:5, checkIsPalindrome" is running.
- The variable pane shows `args = {String[0]@825} []` and `result = true`.

- **isPalindrome** wurde mit dem Parameter "wow" in der **main**-Methode aufgerufen. **True** wird also hier zurückgegeben, und anschließend wird **System.out.println(true)** ausgeführt.

```
1 ▶ public class checkIsPalindrome { ✓
2
3 ▶     public static void main(String[] args) {
4         boolean result = isPalindrome(word: "WOW");
5         System.out.println(result);
6     }
7
8 @    public static boolean isPalindrome(String word) { 2 usages
9
10        int wordLength = word.length();
11
12        if (wordLength <= 1) {
13            return true;
14        }
15
16        boolean firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);
17
18        String withoutFirstAndLast = word.substring(1, wordLength - 1);
19        return firstAndLast && isPalindrome(withoutFirstAndLast);
20    }
21 }
```

Threads & Variables

/Users/henrikpaetzold/Library/Java/JavaVirtualMachines/
Connected to the target VM, address: '127.0.0.1:50525'
true
Disconnected from the target VM, address: '127.0.0.1:5600'
Process finished with exit code 0

Was wird hier passieren?

```
1 public class checkIsPalindrome {  
2     public static void main(String[] args) {  
3         boolean result = isPalindrome("baab");  
4         System.out.println(result);  
5     }  
6  
7     public static boolean isPalindrome(String word) {  
8         int wordLength = word.length();  
9         boolean firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);  
10        String withoutFirstAndLast = word.substring(1, wordLength - 1);  
11        return isPalindrome(withoutFirstAndLast) && firstAndLast;  
12    }  
13 }
```

Error? Endlose Rekursion? Korrektes ausführen? Sonstiges?

Was wird hier passieren?



```
1 public class checkIsPalindrome {
2     public static void main(String[] args) {
3         boolean result = isPalindrome("wow");
4         System.out.println(result);
5     }
6
7     public static boolean isPalindrome(String word) {
8         int wordLength = word.length();
9         boolean firstAndLast = false;
10        String withoutFirstAndLast = "";
11        if (wordLength != 1 && wordLength != 0) {
12            withoutFirstAndLast = word.substring(1, wordLength - 1);
13            firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);
14        }
15        return isPalindrome(withoutFirstAndLast) && firstAndLast;
16    }
17 }
```

Was wird hier passieren?



```
1 public class checkIsPalindrome {
2     public static void main(String[] args) {
3         boolean result = isPalindrome("wow");
4         System.out.println(result);
5     }
6
7     public static boolean isPalindrome(String word) {
8         int wordLength = word.length();
9         if (wordLength < 1) {
10             return true;
11         }
12         boolean firstAndLast = false;
13         String withoutFirstAndLast = "";
14         if (wordLength != 1 && wordLength != 0) {
15             withoutFirstAndLast = word.substring(1, wordLength - 1);
16             firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);
17         }
18         return isPalindrome(withoutFirstAndLast) && firstAndLast;
19     }
20 }
```

Logisches Schliessen I

Stärkere und schwächere Aussagen

- Wir können *stärkere* und *schwächere* Aussagen unterscheiden
 - Wenn $P_1 \Rightarrow P_2$ gilt, dann ist P_1 stärker als P_2 (und P_2 schwächer als P_1)
- Die stärkste Aussage ist `false`, da `false` alles impliziert
- Die schwächste Aussage ist `true`, da `true` nur `true` impliziert

Rückwärtsschliessen: Vorgehen

The diagram illustrates the process of backward reasoning. It features a vertical sequence of code statements enclosed in a light gray box. On the left, three orange curved arrows point upwards from the **{Postcondition}** at the bottom to the **{Assertion_{n-1}}** and **{Assertion_{n-2}}** sections, and finally to the **{Precondition}** at the top. The code itself consists of:
 {Precondition}
 Statement₁;
 ...
 {Assertion_{n-2}}
 Statement_{n-1};
 {Assertion_{n-1}}
 Statement_n;
 {Postcondition}

- **Start:** Wählen (wissen, raten) einer sinnvollen *Nachbedingung*
- **Schrittweise:** Herleiten der vorherigen Aussage (*Assertion_{i-1}*) durch Einbeziehen des *Effekts* der vorherigen Anweisung (*Statement_i*)
- **Ziel:** Herleiten einer *notwendigen und hinreichenden Vorbedingung*
- Rückwärts = «Welche Vorbedingung braucht mein Code, damit er die gewählten Garantien (Nachbedingung) geben kann?»

Weakest Precondition

- Die **schwächste Vorbedingung (weakest precondition)** ist die schwächste Vorbedingung, die die Postcondition impliziert.
 - Falls die Postcondition { true } ist, so ist { true } die schwächste Vorbedingung. Alles impliziert die Postcondition, also insbesondere auch die schwächste Bedingung true.
 - Falls die Postcondition { false } ist, so ist { false } die schwächste Vorbedingung. Nur { false } impliziert die Postcondition, dementsprechend ist es die schwächste (und einzige) Vorbedingung.
- Die vorgestellten Regeln fürs Rückwärtsschliessen ergeben direkt die schwächsten Vorbedingungen.

Weakest Precondition – Einfaches Beispiel

{ }

$x = y^*y$ —————

{ $x > 4$ }

Weakest Precondition – Einfaches Beispiel

{ }

$x = y^*y$ ————— { $x > 4$ }

Weakest Precondition – Einfaches Beispiel

{ }

$x = y * y$ ————— { $y * y > 4$ }

Weakest Precondition – Einfaches Beispiel

$$\{y^*y > 4\}$$

$$x = y^*y \text{ ——— }$$

Weakest Precondition – Einfaches Beispiel

$$\{|y| > 2\}$$

$$x = y^*y \text{ ——— }$$

Weakest Precondition – Zweites Beispiel

{ }

$y = x + 3$ —————

$z = y + 1$ —————

$\{z > 4\}$

Weakest Precondition – Zweites Beispiel

{ }

$y = x + 3$ —————

$z = y + 1$ ————— $\{z > 4\}$

Weakest Precondition – Zweites Beispiel

{ }

$y = x + 3$ —————

$z = y + 1$ ————— $\{y + 1 > 4\}$

Weakest Precondition – Zweites Beispiel

{ }

$y = x + 3$ ————— $\{y + 1 > 4\}$

$z = y + 1$ —————

Weakest Precondition – Zweites Beispiel

{ }

$y = x+3$ ————— $\{x+3+1 > 4\}$

$z = y+1$ —————

Weakest Precondition – Zweites Beispiel

{ }

$y = x + 3$ ————— $\{x > 0\}$

$z = y + 1$ —————

Weakest Precondition – Zweites Beispiel

$\{x > 0\}$

$y = x + 3$ —————

$z = y + 1$ —————

Weakest Precondition – Prüfungsbeispiel

{

}

$p = 3 * q$

$p = p + 1;$

$\{p > 15\}$



Vorbesprechung

Aufgabe 1: Präfixkonstruktion

Gegeben seien zwei Strings s und t und ein Integer n mit $n \geq 0$. Schreiben Sie ein Programm, das zurückgibt, ob s eine Konkatenation von maximal n vielen Präfixen von t ist.

Beispiele:

- $s = "abcababc"$, $t = "abc"$, $n = 4$: Das Programm sollte true zurückgeben, da "abc" und "ab" Präfixe von t sind und s eine Konkatenation von "abc", "ab", "abc" ist.
- $s = "abcbcabc"$, $t = "abc"$, $n = 4$: Das Programm sollte false zurückgeben, da "bc" kein Präfix von t ist.
- $s = "abab"$, $t = "abac"$, $n = 2$: Das Programm sollte true zurückgeben, da "ab" ein Präfix von t ist und s eine Konkatenation von "ab", "ab" ist.

Implementieren Sie die Methode `isPrefixConstruction(String s, String t, int n)` in der Klasse `PrefixConstruction`. Die Methode hat drei Argumente: die beiden Strings s und t und der Integer n . Sie dürfen davon ausgehen, dass der Integer grösser oder gleich 0 ist. In der Datei "PrefixConstructionTest.java" finden Sie Tests.

Tipp: Lösen Sie die Aufgabe rekursiv.

Aufgabe 2: Weakest Precondition

Bitte geben Sie für die folgenden Programmsegmente die schwächste Vorbedingung (weakest precondition) an. Bitte verwenden Sie Java-Syntax. Alle Anweisungen sind Teil einer Java Methode. Alle Variablen sind vom Typ int und es gibt keinen Overflow.

1.

```
P: { ?? }
S:   m = n * 4;  k = m - 2;
Q: { n > 0 && k > 5 }
```

2.

```
P: { ?? }
S:   m = n * n;  k = m * 2;
Q: { k > 0 }
```

3.

```
P: { ?? }
S:   y = x + 3; z = y + 1;
Q: { z > 4 }
```

4.

```
P: { ?? }
S:   y = x + 1; z = y - 3;
Q: { z == 10 }
```

Beispiel

```
P: { ?? }
S:   a = b * 3; c = a + 1;
Q: { a > 0 && c < 5 }
```

Aufgabe 3: Wörter Raten

Das Programm "WoerterRaten.java" enthält Fragmente eines Rate-Spiels, welches Sie vervollständigen sollen. In dem Spiel wählt der Computer zufällig ein Wort w aus einer Liste aus und der Mensch muss versuchen, das Wort zu erraten. In jeder Runde kann der Mensch eine Zeichenfolge z (welche einen oder mehrere Buchstaben enthält) eingeben und der Computer gibt einen Hinweis dazu. Folgende Hinweise sind möglich:

1. w beginnt mit z
2. w endet mit z
3. w enthält z
4. w enthält nicht z

Tipp? **e**
Das Wort enthält nicht "e"!
Tipp? **a**
Das Wort endet mit "a"!
Tipp? **j**
Das Wort beginnt mit "j"!
Tipp? **v**
Das Wort enthält "v"!
Tipp? **java**
Das Wort ist "java"!
Glückwunsch, du hast nur 5 Versuche benötigt!

Aus Textdatei lesen

woerter.txt

8
wort
blasinstrument
computer
schlange
java
programmieren
welt
sugus

Code zum Einlesen der Wörter:

```
Scanner scanner = new Scanner(new File("woerter.txt"));
String[] woerter = new String[scanner.nextInt()];
for(int i = 0; i < woerter.length; i++) {
    woerter[i] = scanner.next();
}
```

Anzahl Wörter = 8

Liest das nächste Wort

Absoluter Pfad

- beginnt mit `c:\...` unter Windows oder `/...` unter Linux/macOS

Relativer Pfad

- ist relativ zum aktuellen Verzeichnis (“working directory”) des Programms (bei uns der Projektordner)

Aufgabe 4: Datenanalyse

In dieser Aufgabe werden Sie die Kelchblatlänge von Iris Blumen, welche auch Schwertlilien genannt werden, analysieren. Dazu verwenden Sie ein öffentlich zugängliches Dataset ¹ welches die Längen vom Kelchblatt (Sepal Length) von jeweils 150 verschiedenen Iris Blumen enthält. Es gibt sehr viele Arten von Iris Blumen, insgesamt sind 285 Arten bekannt. Diese zu unterscheiden ist ein komplexer Prozess. Wir werden jedoch versuchen mittels der Länge des Kelchblattes drei Arten von Iris Blumen zu unterscheiden, in dem wir die gegebenen Daten analysieren. Wir werden uns auf die folgenden drei Iris Blumen konzentrieren:

- Iris setosa, auch Borsten-Schwertlilie genannt.
- Iris versicolor, auch Verschiedenfarbige Schwertlilie genannt.
- Iris virginica, auch blaue Sumpfschwertlilie genannt.



Iris Versicolor



Iris Setosa



Iris Virginica

Aufgabe 4: Datenanalyse

1. Das Programm soll als erstes die Kelchblattgrößen für die drei Arten von Iris Blumen aus den Dateien "sepal_length_setosa.txt", "sepal_length_versicolor.txt", und "sepal_length_virginica.txt" im Projekt-Verzeichnis in ein Array einlesen. Die Dateien haben ein ähnliches Format wie die "woerter.txt"-Datei der letzten Aufgabe. Die Kelchblattgrößen liegen als ganze Zahlen in Millimetern [mm] vor. Die erste Zahl gibt die Anzahl Daten im File an. Implementieren Sie dazu die Methode `liesLaengen()`, indem Sie die nötigen Daten aus dem gegebenen Scanner auslesen. Falls Sie Schwierigkeiten dabei haben, können Sie sich an der `WoerterRaten.liesWoerter()`-Methode orientieren. Sie können Ihre Methode anhand eines der drei Files testen um zu schauen, ob diese funktioniert.

Verwenden Sie den Test `testLiesLaengen` in der Datei "DatenAnalyseTest.java", um Ihren Code zu testen.

Aufgabe 4: Datenanalyse

2. Führen Sie als nächstes eine einfache Analyse der Daten durch, indem Sie das Minimum, das Maximum, den Durchschnitt und ausserdem die Anzahl der Kelchblattgrössen ausgeben. Füllen Sie dazu die Methode `einfacheAnalyse()` aus. Beachten Sie folgende Methoden: `Math.min()` und `Math.max()`.

Beispiel:

Anzahl Daten: 50

Minimum: 43 mm

Maximum: 58 mm

Durchschnitt: 50 mm

Aufgabe 4: Datenanalyse

3. Die drei Werte, die Sie in 2. berechnet haben, sagen nicht viel über die Daten aus. Um die Daten besser zu verstehen, soll Ihr Programm ein **Histogramm** berechnen und auf der Konsole ausgeben. Die Textausgabe könnte ungefähr so aussehen:

```
[40,43)  
[43,46)  
[46,49)  
[49,52) |  
[52,55)  
[55,58) ||  
[58,61) |||||  
[61,64) |||||||||  
[64,67) |||||||||  
[67,70) |||||||||  
[70,73) |||||  
[73,76) ||  
[76,79) |||||  
[79,82) |  
[82,85)
```

Implementieren Sie die Methode `histogrammAnalyse()`. Diese Methode soll zuerst den Benutzer nach der Anzahl der Histogramm-Klassen fragen, dann das Histogramm berechnen und schliesslich ausgeben. Zwei (leere) Methoden sind schon vorgegeben: `erstelleHistogramm()` und `klassenBreite()`. Für diese beiden Methoden existieren Tests in "DatenAnalyseTest.java" und Kommentare, welche Ihnen beim Schreiben des Programms helfen. Überlegen Sie sich, wie Sie das Problem weiter aufteilen möchten, und erstellen Sie entsprechende Methoden.

Bonusaufgabe

Aufgabe 6: Matrix (Bonus!)

Achtung: Diese Aufgabe gibt Bonuspunkte (siehe "Leistungskontrolle" im www.vvz.ethz.ch). Die Aufgabe muss eigenhändig und alleine gelöst werden. Die Abgabe erfolgt wie gewohnt per Push in Ihr Git-Repository auf dem ETH-Server. Verbindlich ist der letzte Push vor dem Abgabetermin. Auch wenn Sie vor der Deadline committen, aber nach der Deadline pushen, gilt dies als eine zu späte Abgabe. Bitte lesen Sie zusätzlich [die allgemeinen Regeln](#).

Nachbesprechung

Aufgabe 1: Sieb des Eratosthenes

Schreiben Sie ein Programm "Sieb.java", das eine Zahl $limit$ einliest und die Anzahl der Primzahlen, die grösser als 1 und kleiner oder gleich dem $limit$ sind, ausgibt. Dazu ermitteln Sie in einem ersten Schritt alle Primzahlen, die kleiner oder gleich $limit$ sind. Dieses Teilproblem können Sie mit dem [Sieb des Eratosthenes](#) lösen. Das Sieb des Eratosthenes findet Primzahlen bis n . Man betrachtet alle Zahlen von 2 bis n und streicht zuerst alle Vielfachen der ersten Zahl (2). Dann geht man zur nächsten ungestrichenen Zahl (3) und wiederholt das Streichen ihrer Vielfachen. Das macht man, bis man dies für alle Zahlen gemacht hat. Sie können ein Boolean-Array verwenden, um zu speichern, welche Zahlen Primzahlen sind und welche nicht. Übrig bleiben die Primzahlen. Danach können Sie die Anzahl der gefundenen Primzahlen anhand dieses Arrays bestimmen.

Beispiel: Für $limit = 13$ sollte Ihr Programm 6 ausgeben (Primzahlen: 2, 3, 5, 7, 11, 13).

Hinweis: Es ist nicht zwingend nötig von 2 bis n zu gehen. Von 2 bis \sqrt{n} zu gehen reicht bereits aus, da eine Zahl $\leq n$ nicht einen Teiler grösser als \sqrt{n} ausser sich selbst haben kann.

Aufgabe 2: Arrays

1. Implementieren Sie die Methode `ArrayUtil.zeroInsert(int[] x)` in der Datei "ArrayUtil.java". Die Methode nimmt einen Array `x` als Argument und gibt einen Array zurück. Der zurückgegebene Array soll die gleichen Werte wie `x` haben, ausser: Wenn eine positive Zahl direkt auf eine negative Zahl folgt oder wenn eine negative Zahl direkt auf eine positive Zahl folgt, dann wird dazwischen eine 0 eingefügt.

Beispiele:

- Wenn `x` gleich `[3, 4, 5]` ist, dann wird `[3, 4, 5]` zurückgegeben.
- Wenn `x` gleich `[3, 0, -5]` ist, dann wird `[3, 0, -5]` zurückgegeben.
- Wenn `x` gleich `[-3, 4, 6, 9, -8]` ist, dann wird `[-3, 0, 4, 6, 9, 0, -8]` zurückgegeben.

Versuchen Sie, die Methode rekursiv zu implementieren.

Aufgabe 2: Arrays

2. Implementieren Sie die Methode `ArrayUtil.tenFollows(int[] x, int index)`. Die Methode gibt einen Boolean zurück. Die Methode soll `true` zurückgeben, wenn im Array `x` ab Index `index` der zehnfache Wert einer Zahl `n` direkt der Zahl `n` folgt. Dies muss nur für das erste Auftreten der Zahl `n` ab Index `index` im Array `x` geprüft werden. Ansonsten soll die Methode `false` zurückgeben.

Beispiele:

- `tenFollows([1, 2, 20], 0)` gibt `true` zurück.
- `tenFollows([1, 2, 7, 20], 0)` gibt `false` zurück.
- `tenFollows([3, 30], 0)` gibt `true` zurück.
- `tenFollows([3], 0)` gibt `false` zurück.
- `tenFollows([1, 2, 20, 5], 1)` gibt `true` zurück.
- `tenFollows([1, 2, 20, 5], 2)` gibt `false` zurück.

Die `main` Methode in `ArrayUtil` gibt die oben genannten Beispielaufrufe sowie das entsprechende Ergebnis der jeweiligen Methode aus. Hiermit können Sie überprüfen, ob Ihre Implementierungen die richtigen Ergebnisse zurückliefern. In "ArrayUtilTest.java" im Ordner "test" in der Übungsvorlage finden Sie zusätzlich einige Unit-Tests für beide Methoden (für eine detaillierte Beschreibung zu automatisiertem Testen und der Ausführung solcher Tests siehe Aufgabe 3). Sie können die `main` Methode und die Tests beliebig ändern und/oder mit Ihren eigenen Inputs erweitern.

Aufgabe 3: 2D Arrays

Gegeben einer Matrix M , prüfen Sie zuerst ob diese eine $n \times n$ Matrix ist, deren Elemente positive ganze Zahlen sind. Danach prüfen Sie ob zusätzlich alle Zahlen kleiner gleich n^2 sind. Somit gilt nun $0 < m_{i,j} \leq n^2$. Prüfen Sie ebenfalls, ob die Elemente der Matrix jeweils genau einmal vorkommen, sprich ob $m_{x,y} = m_{p,q} \Rightarrow (x = p) \wedge (y = q)$ gilt. Wir sagen, dass die Matrix M *perfekt* ist, wenn zusätzlich alle Zeilensummen und Spaltensummen gleich sind (also $\sum_{k=0}^{k=n-1} m_{i,k} = \sum_{k=0}^{k=n-1} m_{j,k}$ für alle i, j und $\sum_{k=0}^{k=n-1} m_{k,i} = \sum_{k=0}^{k=n-1} m_{k,j}$ für alle i, j mit $0 \leq i, j < n$).

Vervollständigen Sie die Methode `boolean checkMatrix(int[][] m)` von der Klasse `Matrix`, so dass diese Methode `true` zurückgibt wenn die Input Matrix *perfekt* ist, und `false` sonst. Sie können davon ausgehen, dass der Parameter `m` nicht `null` ist. Alle anderen Eigenschaften müssen Sie selber testen. Eine Matrix ist nur perfekt, wenn alle genannten Eigenschaften gelten.

Testen Sie Ihr Programm ausgiebig - am besten mit JUnit - und pushen Sie die Lösung vor dem Abgabetermin. Wir haben Ihnen einen JUnit Test in der Klasse `MatrixTest` bereits erstellt.

Aufgabe 4: Testen mit JUnit

- **Zweck des Programms:**
 - Wochentag eines Datums (nach 01.01.1900) ausgeben
 - Beispiel: 13.10.2017 -> Friday
 - Gibt fälschlicherweise aber "The 13.10.2017 is a Sunday" aus.
 - Berücksichtigt Schaltjahre ("Leap Year")
- **Funktionsweise:**
 1. Überprüft, ob das Datum OK ist
 2. Zählt die Tage ab 01.01.1900 bis zum eingegebenen Datum
 3. Wochentag = Tage % 7

Aufgabe 5: Matching Numbers

Implementieren Sie die Methode `Match.matchNumber(long A, int M)`. Die Methode soll für eine Zahl A und eine nicht-negative drei-stellige Zahl M die Position von M in A zurückgeben. Sei M eine Zahl mit den Ziffern $M_2M_1M_0$ (das heisst, es gilt $M = M_0 + 10 \cdot M_1 + 100 \cdot M_2$), wobei jede Ziffer 0 sein kann. Zusätzlich sei A eine Zahl, sodass A_i die i -te Ziffer von A ist (das heisst, es gilt $|A| = \sum_i 10^i \cdot A_i$), wobei A unendlich viele führende Nullen hat. Die Position von M in A ist die kleinste Zahl j , sodass $A_j = M_0$ und $A_{j+1} = M_1$ und $A_{j+2} = M_2$ gilt. Die Methode soll -1 zurückgeben, falls es kein solches j gibt.

Beispiele:

`matchNumber(32857890, 789)` soll 1 zurückgeben.

`matchNumber(37897890, 789)` soll 1 zurückgeben.

`matchNumber(1800765, 7)` soll 2 zurückgeben.

`matchNumber(1800765, 8)` soll -1 zurückgeben (die drei Ziffern von 8 sind 008).

`matchNumber(75, 7)` soll 1 zurückgeben (da 007 and Position 1 von 0075 ist).

Aufgabe 5: Matching Numbers

Implementieren Sie die Berechnung in der Methode `int matchNumber(long A, int M)`, welche sich in der Klasse Match befindet. Die Deklaration der Methode ist bereits vorgegeben. Sie können davon ausgehen, dass $0 \leq M < 1000$ gilt.

In der main Methode der Klasse Match finden Sie die oberen Beispiele als kleine Tests, welche Beispiel-Aufrufe zur `matchNumber`-Methode machen und welche Sie als Grundlage für weitere Tests verwenden können. In der Datei `MatchTest.java` geben wir die gleichen Tests zusätzlich auch als JUnit Test zur Verfügung. Sie können diese ebenfalls nach belieben ändern. Es wird *nicht* erwartet, dass Sie für diese Aufgabe den JUnit Test verwenden.

Tipp: Die Methode `Integer.toString(int i)` wandelt einen Integer in einen String um.

Kahoot

<https://create.kahoot.it/details/4446eed8-9c6f-4c9c-98e8-ba531d535207>