252-0027

Einführung in die Programmierung

Übungen

Woche 6: References, Recursion, Precondition

Jonas Wetzel
Departement Informatik
ETH Zürich

Organisatorisches

- Mein Name: Jonas Wetzel
- Bei Fragen: jwetzel@ethz.ch
  - Mails bitte mit «[EProg24]» im Betreff
- 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

Plan für heute

- **kurze Nachbesprechung**
- **Theorie**
  - References, Rekursion, Weakest Precondition
- **Prüfungsaufgaben**
- (wenn Zeit Kahoot)
- **Vorbesprechung**

Euer Feedback

- **Einfache Kahoot Fragen skippen**
- **Mehr Tipps für schwierigere Aufgaben**
  - Werde ich einbauen
  - Wie man an die Probleme ran geht
  - Nützliche Methoden
- **Besprechung von Prüfungsaufgaben, Programmieraufgaben**

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.

```java
blic class Sieb {

    // Methode zum Initialisieren des Siebes
    public static boolean[] initSieb(int limit) {
        boolean[] sieb = new boolean[limit + 1];
        // Setze alle Elemente ab 2 auf true
        for (int i = 2; i < sieb.length; i++) {
            sieb[i] = true;
        }
        return sieb;
    }

    // Methode zur Berechnung der Primzahlen mit dem Sieb des Eratosthenes
    public static void berechnePrimzahlen(boolean[] sieb, int limit) {
        for (int i = 2; i < limit; i++) {
            if (sieb[i]) {
                // Setze alle Vielfachen von i auf false
                for (int vielfaches = 2 * i; vielfaches ⩽ limit; vielfaches += i) {
                    sieb[vielfaches] = false;
                }
            }
        }
    }

    // Methode zum Zählen der Primzahlen
    public static int zaehlePrimzahlen(boolean[] sieb) {
        int primzahlen = 0;
        // Zähle die verbleibenden Primzahlen
        for (int i = 2; i < sieb.length; i++) {
            if (sieb[i]) {
                primzahlen++;
            }
        }
        return primzahlen;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Geben Sie eine positive ganze Zahl ein: ");
        int limit = scanner.nextInt();

        if(limit ⩽ 0) {
            System.out.println("Keine positive ganze Zahl!");
        } else {
            boolean[] sieb = initSieb(limit);
            berechnePrimzahlen(sieb, limit);
            int primzahlen = zaehlePrimzahlen(sieb);
            System.out.println(primzahlen);
        }
    }
}
```

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 abändern und/oder mit Ihren eigenen Inputs erweitern.

```java
import java.util.Arrays;

public class ArrayUtil {

    public static int[] zeroInsert(int[] x) {
        if (x.length ≤ 1) { // Base case: Falls die Laenge des Arrays kleiner gleich 1 ist.
            return Arrays.copyOf(x, x.length);
        } else { // Rekursiver Fall.

            // Wir rufen zeroInsert mit dem Tail des Arrays auf.
            int[] tail = Arrays.copyOf(x, x.length - 1);
            int[] tailRes = zeroInsert(tail);

            int a = x[x.length - 1];
            int b = x[x.length - 2];

            int[] res;
            if ((a < 0 && b > 0) || (a > 0 && b < 0)) {
                // Falls die letzten beiden Zahlen unterschiedlich positiv oder negativ sind,
                // dann wird eine 0 hinzugefuegt.
                res = Arrays.copyOf(tailRes, tailRes.length + 2);
                res[res.length - 1] = a;
                res[res.length - 2] = 0;
                res[res.length - 3] = b;
            } else {
                // Ansonsten wird keine 0 hinzugefuegt.
                res = Arrays.copyOf(tailRes, tailRes.length + 1);
                res[res.length - 1] = a;
            }
            return res;
        }
    }
```

```java
public static boolean tenFollows(int[] x, int index) {
    if (x.length ≤ index) {
        return false;
    } else {
        // Wir speichern den Wert vom letzten Element in `last` um zu pruefen, ob das
        // aktuelle Element gleich das zehnfache des letzten Elements ist.
        // Wir fangen beim Index `index` an.
        int idx = index + 1;
        int last = x[index];

        while (idx < x.length) {
            if (x[idx] == last * 10) {
                return true;
            }
            last = x[idx];
            idx += 1;
        }

        return false;
    }
}
```

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 Eigentschaften 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 Klases `MatrixTest` bereits erstellt.

See Matrix.java

```
public class Matrix {

    /*
     * This method checks if the entries of m are all positive >1 and smaller or
     * equal n^2 it returns true if that is the case and false otherwise.
     */
    public static boolean checkDomain(int[][] m) {
        int n = m.length;
        int powN = n * n;

        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (m[i][j] <= 0 || m[i][j] > powN) { // Not positive
                    return false;
                }
            }
        }

        return true;
    }

    /*
     * This method checks if the matrix has only unique entries and returns true if
     * this is the case and false otherwise
     */
    public static boolean checkUniqueness(int[][] m) {
        int n = m.length;
        boolean[] contains = new boolean[n * n + 1];

        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                int entry = m[i][j];
                if (contains[entry]) {
                    return false;
                } else {
                    contains[entry] = true;
                }
            }
        }

        return true;
    }

    /*
     * This method checks if the row sums and column sums are equal for all rows and
     * columns. It returns true if that is the case and false otherwise.
     */
    public static boolean checkRowAndColSums(int[][] m) {
        int n = m.length;
        int sum = 0;

        // Check the first row first and set the sum
        for (int i = 0; i < n; ++i) {
            sum += m[0][i];
        }

        int currSumRow;
        int currSumCol;

        // Check row and columns simultaneously
        for (int i = 0; i < n; ++i) {
            currSumRow = 0;
            currSumCol = 0;

            for (int j = 0; j < n; ++j) {
                currSumCol += m[i][j];
                currSumRow += m[j][i];
            }

            if (currSumRow != sum || currSumCol != sum) {
                return false;
            }
        }

        return true;
    }

    /*
     * This method checks if the matrix is square including the edge cases
     */
    public static boolean checkDimensions(int[][] m) {
        if (m == null) {
            return false;
        }

        int rows = m.length;

        for (int i = 0; i < rows; ++i) {
            int cols = m[i].length;

            if (rows != cols) {
                return false;
            }
        }
        return true;
    }

    /*
     * This method get a n x n matrix m as input and checks if the matrix is
     * perfect. I, the matrix is square 2. All entries are positive 3. For all
     * entries m(i, j) we have 0 < m(i, j) <= n^2 4. All row sums and all column
     * sums are equal
     *
     * The order in which these methods are executed is very important, as they
     * depend on each others results.
     */
    public static boolean checkMatrix(int[][] m) {
        if (!checkDimensions(m) || !checkDomain(m) || !checkUniqueness(m) || !checkRowAndColSums(m)) {
            return false;
        }
        return true;
    }
}
```

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 $0 \leq \texttt{M} < 1000$ 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.

```java
public class Match {

    /**
     * Finds the position of the 3-digit number M in the number A.
     *
     * @param A is the number in which to search for the 3-digit number M.
     * @param M is a non-negative 3-digit number (M = M2 M1 M0).
     * @return The smallest index j such that A[j] = M0, A[j+1] = M1, and A[j+2] = M2,
     *         or -1 if no such j exists.
     */
    public static int matchNumber(long A, int M) {
        // Step 1: Convert M and the absolute value of A to strings.
        String mString = "" + M; // Convert M to string.
        String aString = "" + Math.abs(A); // Convert absolute A to string.

        // Step 2: Pad M with leading zeros to ensure it's 3 digits.
        while (mString.length() < 3) {
            mString = "0" + mString; // Add leading zeros if necessary.
        }

        // Step 3: Reverse the digits of M.
        String reversedM = "";
        for (int i = mString.length() - 1; i >= 0; i--) {
            reversedM += mString.charAt(i); // Reverse M.
        }

        // Step 4: Reverse the digits of A and add trailing zeros.
        String reversedA = "";
        for (int i = aString.length() - 1; i >= 0; i--) {
            reversedA += aString.charAt(i); // Reverse A.
        }
        reversedA += "000"; // Append zeros to prevent out-of-bounds errors.

        // Step 5: Search for reversedM in reversedA.
        int mIndex = 0; // Tracks matched digits of M.
        for (int i = 0; i < reversedA.length(); i++) {
            if (reversedA.charAt(i) == reversedM.charAt(mIndex)) {
                mIndex++; // Move to the next digit of M.
                if (mIndex == 3) {
                    return i - 2; // Return match position adjusted for reversal.
                }
            } else {
                mIndex = 0; // Reset index for M if no match.
                if (reversedA.charAt(i) == reversedM.charAt(mIndex)) {
                    mIndex++; // Start matching again.
                }
            }
        }

        // Step 6: Return -1 if no match was found.
        return -1;
    }
}
```

Theorie

References

Value vs Reference

- Variablen enthalten entweder einen **Wert** (Value) oder eine **Referenz** (Reference).
- Variablen enthalten **nie** Objekte.
- Referenzen zeigen auf das Objekt im Speicher.

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 “x2”

Scope “x1”

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 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};
10         setX(x);
11         String xStr = Arrays.toString(x);
12         System.out.println(xStr);
13     }
14 }
```

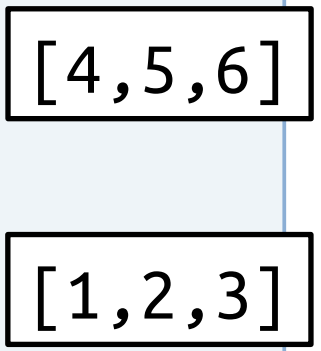
x

[4,5,6]

[1,2,3]

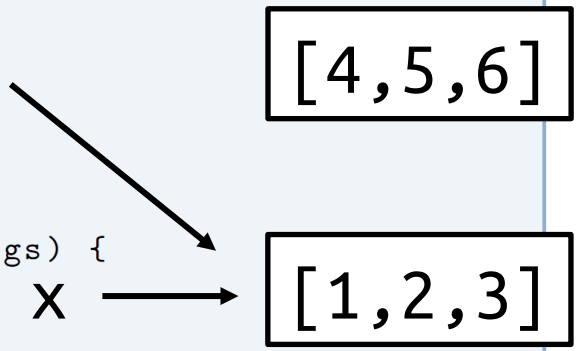
Value vs Reference - Beispiel

```
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 }
```



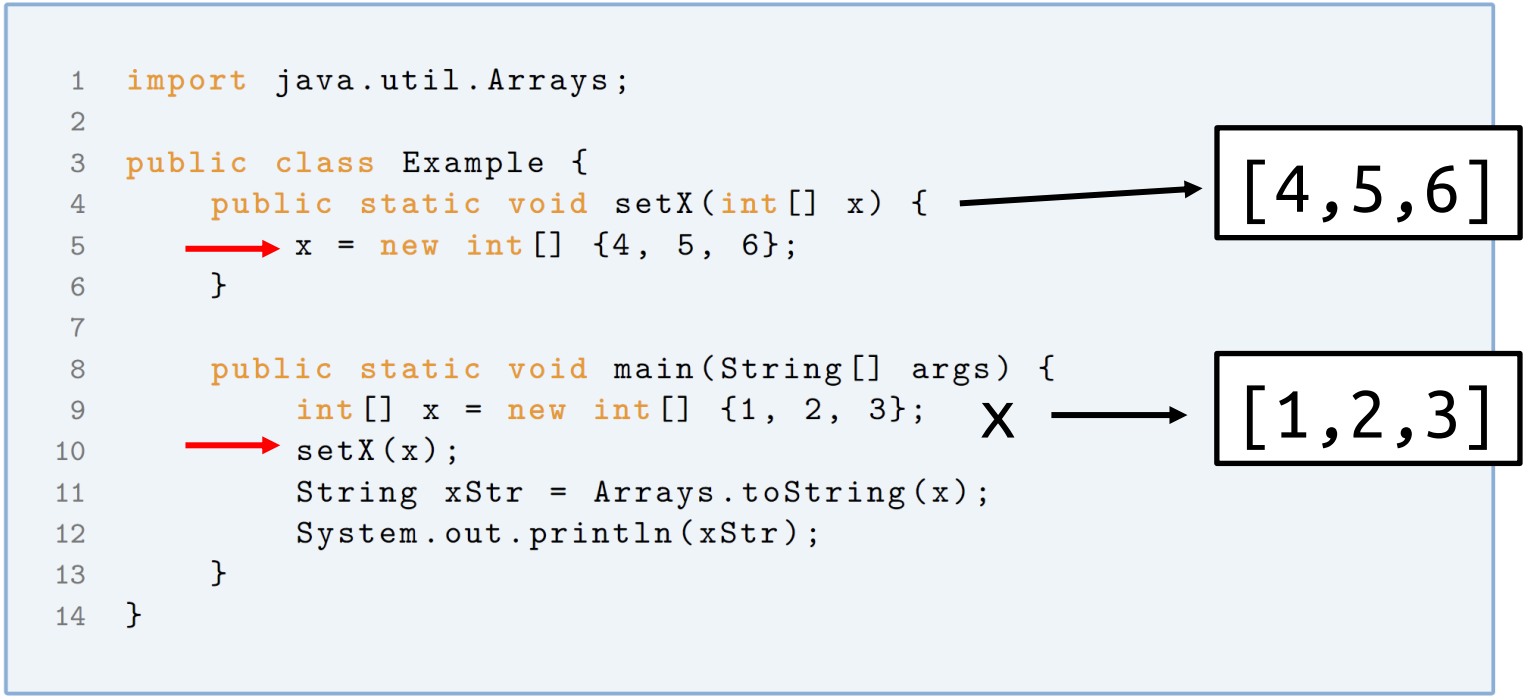
Value vs Reference - Beispiel

```
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 }
```



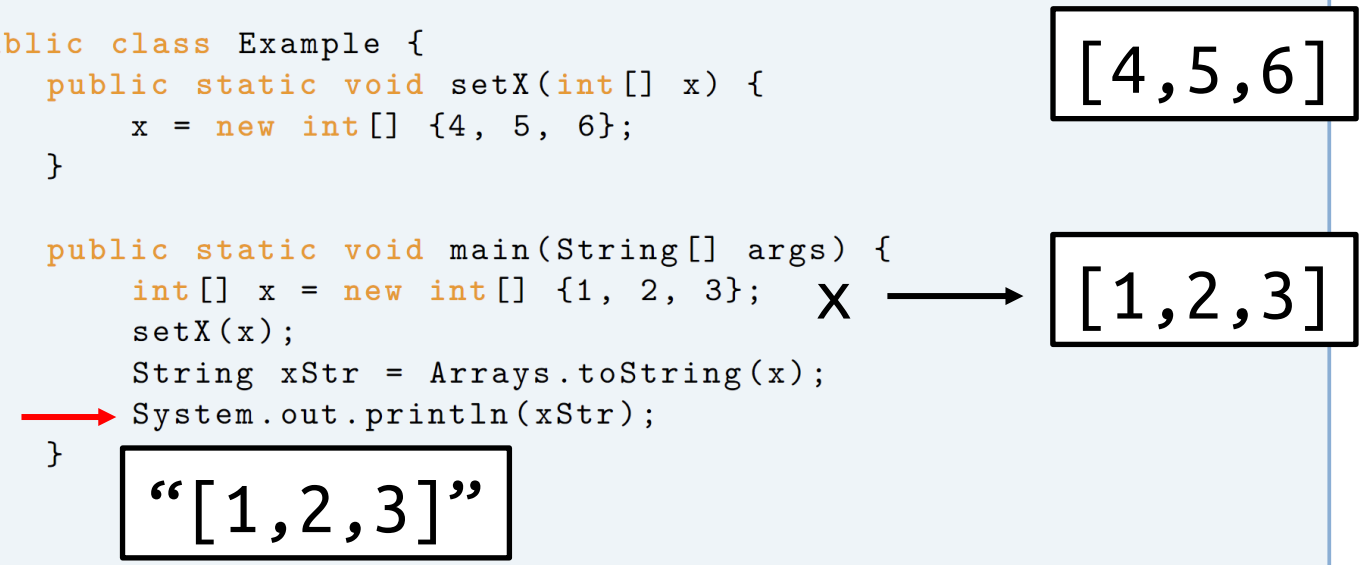
Value vs Reference - Beispiel

```
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 }
```



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 }
```



- Wenn wir

```
x = new int[] {4, 5, 6};
```

machen, erstellen wir einen neuen Array und ändern die Referenz von x auf diesen neuen Array

- Das Array auf das x davor gezeigt hat bleibt unverändert

- Wenn wir eine Referenz, also zum Beispiel x = new int[4] als Argument übergeben, so wird nicht der Array übergeben, sondern nur die Referenz, die uns sagt, wo der Array sich im Speicher befindet..
- x könnt ihr euch dann wie als int oder double ... vorstellen
- Wenn wir x in der Methode umändern, wird das eigentliche x nicht verändert

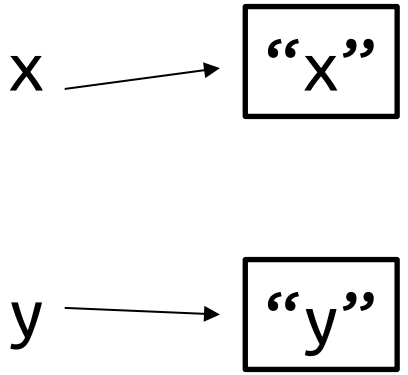
- wenn wir aber x[0] machen, dann sagen wir Java
  - “geh dahin, wo x hinzeigt und gib mir das element an der 0. Stelle”
- `x[2] = 4;`
  - “Geh dahin, wo x hinzeigt und schreibe 4 an die 2. Stelle”
- aber `x = new int[] {4, 5, 6};`
- Sagt Java, “lass x auf einen anderen Ort im Speicher zeigen“
- Und diese Änderung ist nur lokal, weil die Referenz als Value übergeben wird -> es gelten pass by value semantics, also genau wie int, double, etc.

Value vs Reference – Achtung!

```java
import java.util.Arrays;

public class Example {
    public static void setX(int[] x) {
        // Modify the existing array in place
        x[0] = 4;
        x[1] = 5;
        x[2] = 6;
    }

    public static void main(String[] args) {
        int[] x = new int[] {1, 2, 3};
        setX(x);
        String xStr = Arrays.toString(x);
        System.out.println(xStr);  // Output will now be "[4, 5, 6]"
    }
}
```

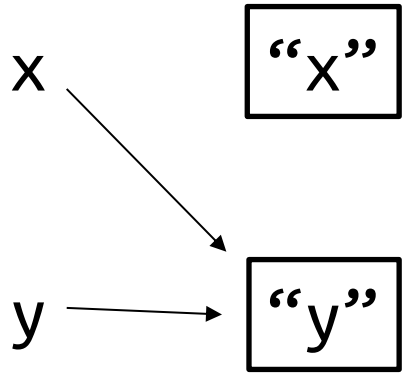
Value vs Reference - Beispiel

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



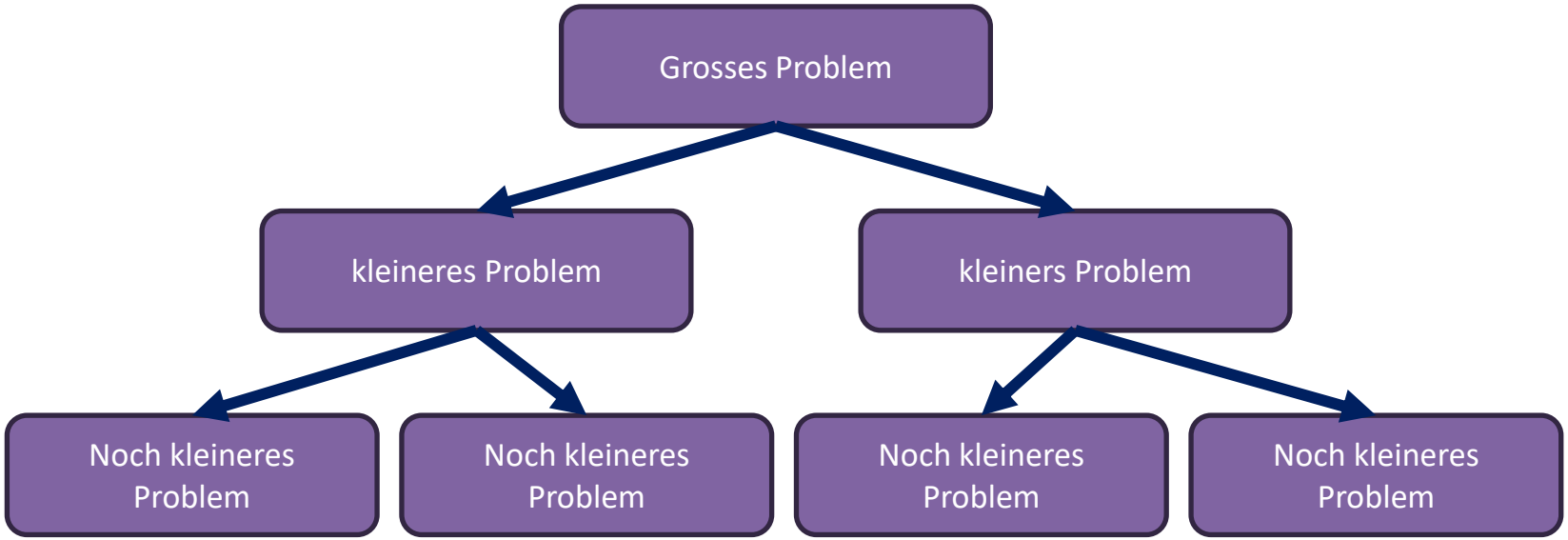
Value vs Reference - Beispiel

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

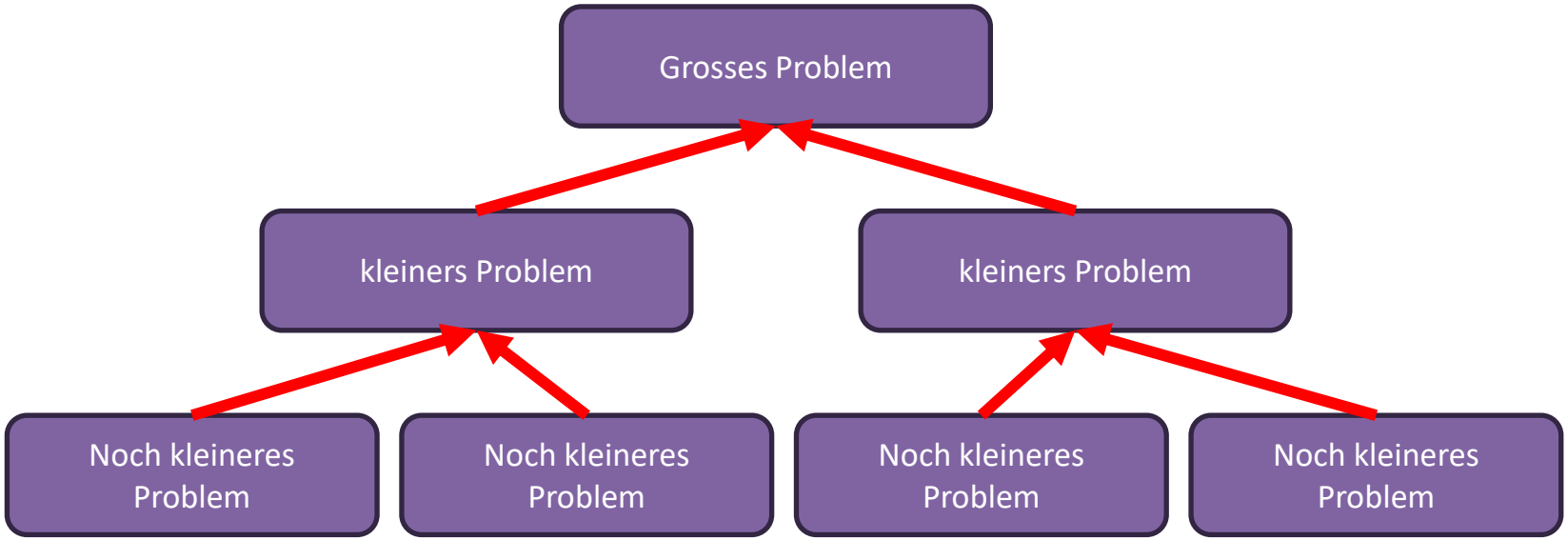


Rekursion

Rekusion – Grundprinzip Divide and Conquer



Rekusion – Grundprinzip Divide and Conquer



Decrease-and-Conquer - Fakultät

factorial(3)

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

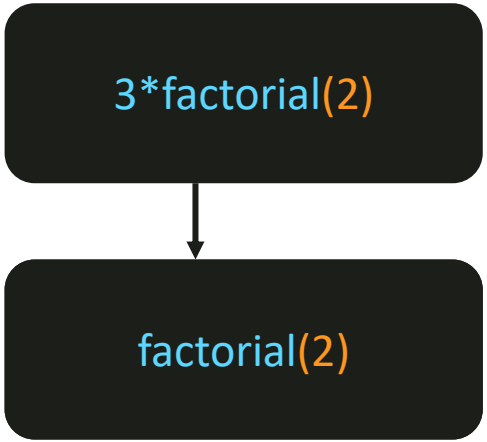
Decrease-and-Conquer - Fakultät

3*factorial(2)

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

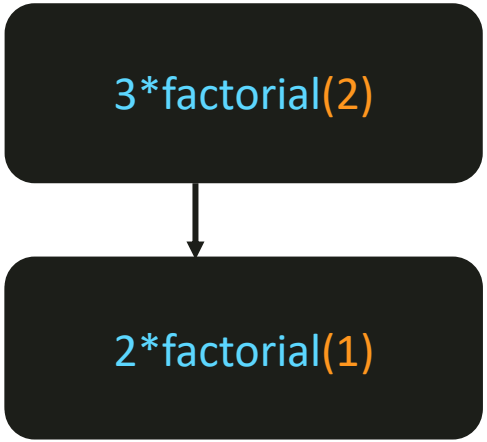
Decrease-and-Conquer - Fakultät

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



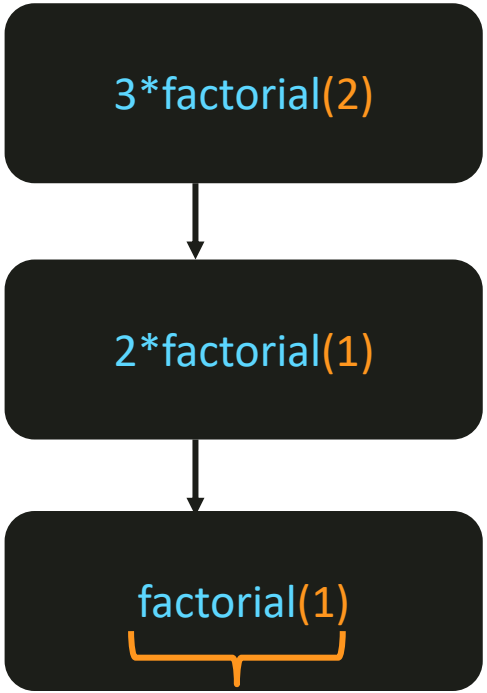
Decrease-and-Conquer - Fakultät

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



Decrease-and-Conquer - Fakultät

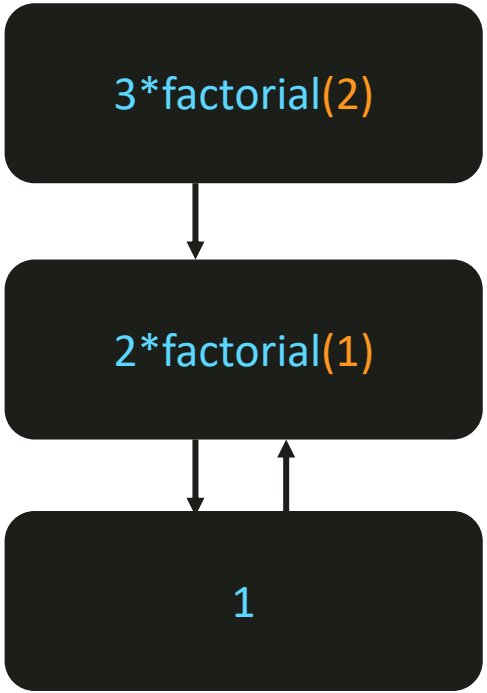
```java
public int factorial(int n) {
    if (n==1)return 1;
    return n*factorial(n-1);
}
```



Base Case!

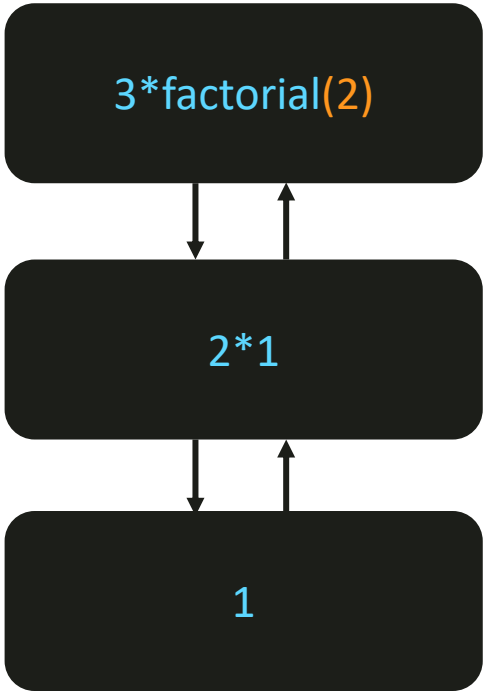
Decrease-and-Conquer - Fakultät

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



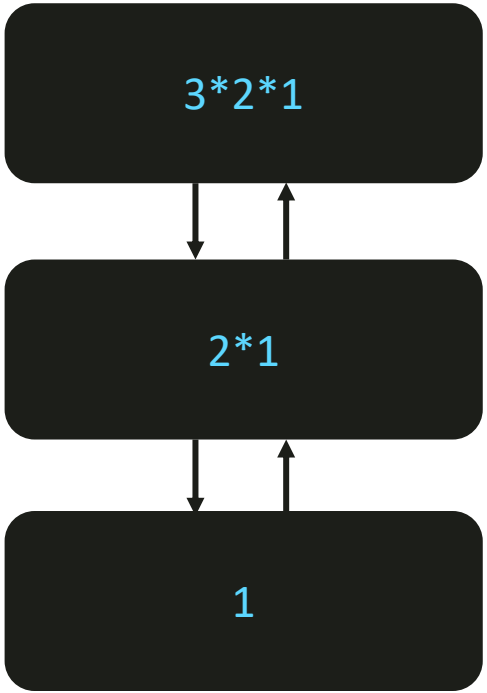
Decrease-and-Conquer - Fakultät

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



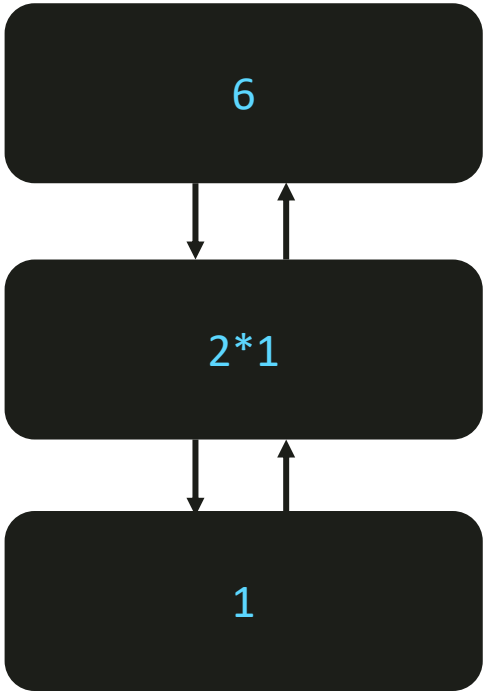
Decrease-and-Conquer - Fakultät

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

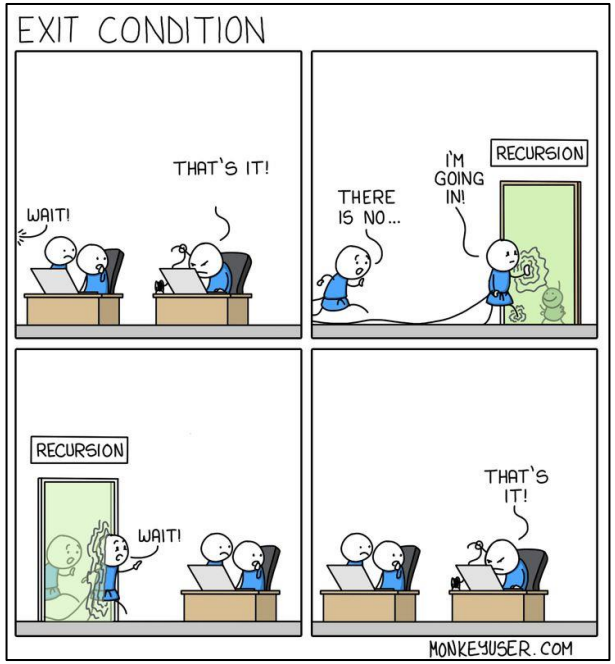


Decrease-and-Conquer - Fakultät

```java
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 `sum(n)`, die gegeben eine nichtnegative Zahl, alle nichtnegativen Zahlen bis einschliesslich n aufsummiert.

Beispiele:

sum(0) = 0

sum(1) = 1 = 1 + 0

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 = _ : sum (n) = __**
- 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 andere Lösungen bauen auf dem Base Case auf.

1. Die einfachste Eingabe

- Hier ist es **n = 0: 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 andere Lösungen bauen auf dem Base Case auf.

Schritte

1. Was ist die einfachste mögliche Eingabe?

   → Base Case

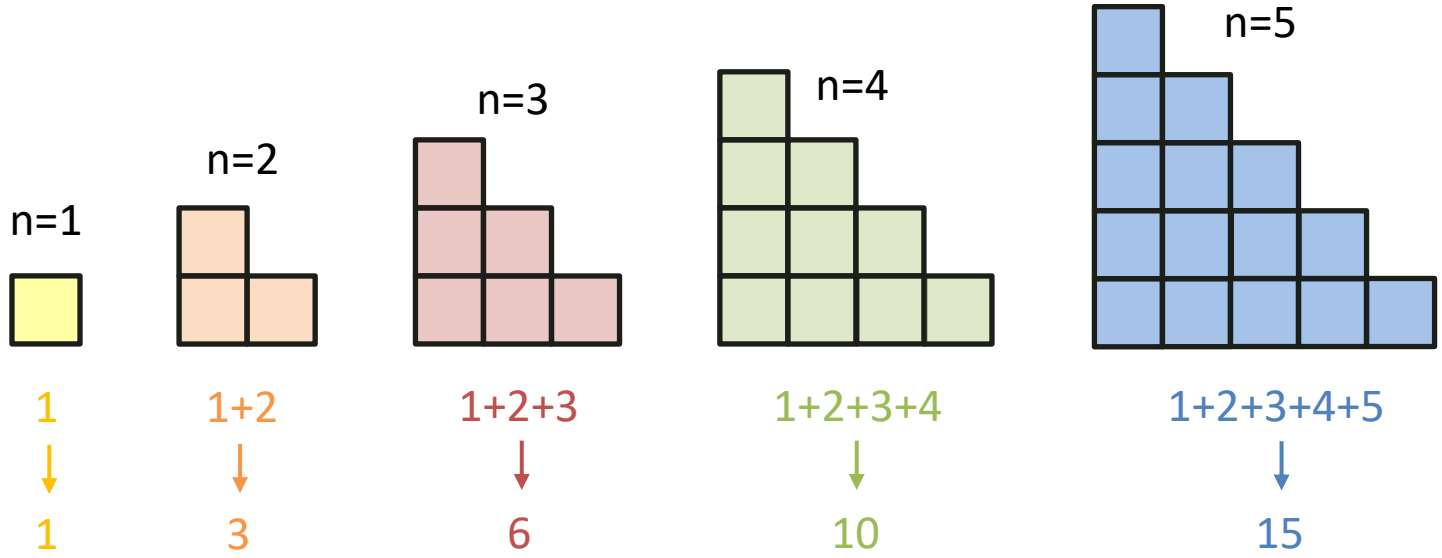
2. Beispiele ausprobieren und visualisieren
3. Leite größere Beispiele von kleineren Beispielen ab
4. Verallgemeinere das Muster

   → Rekursionsschritt

5. Schreibe den Code

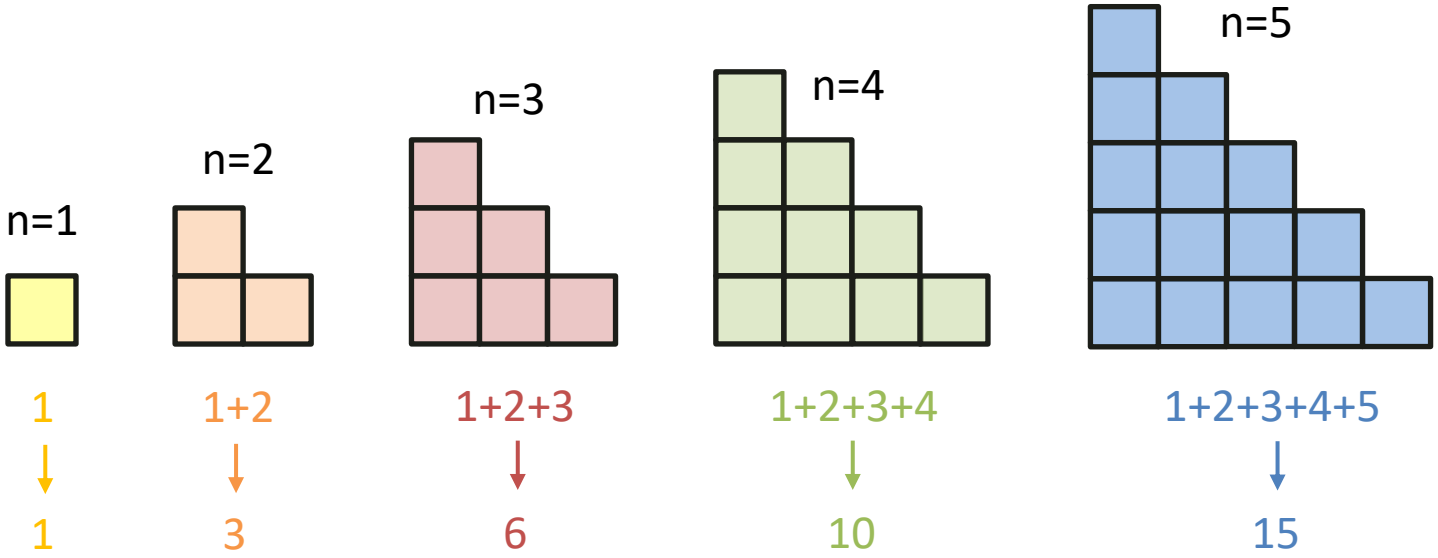
2. Die Beispiele

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



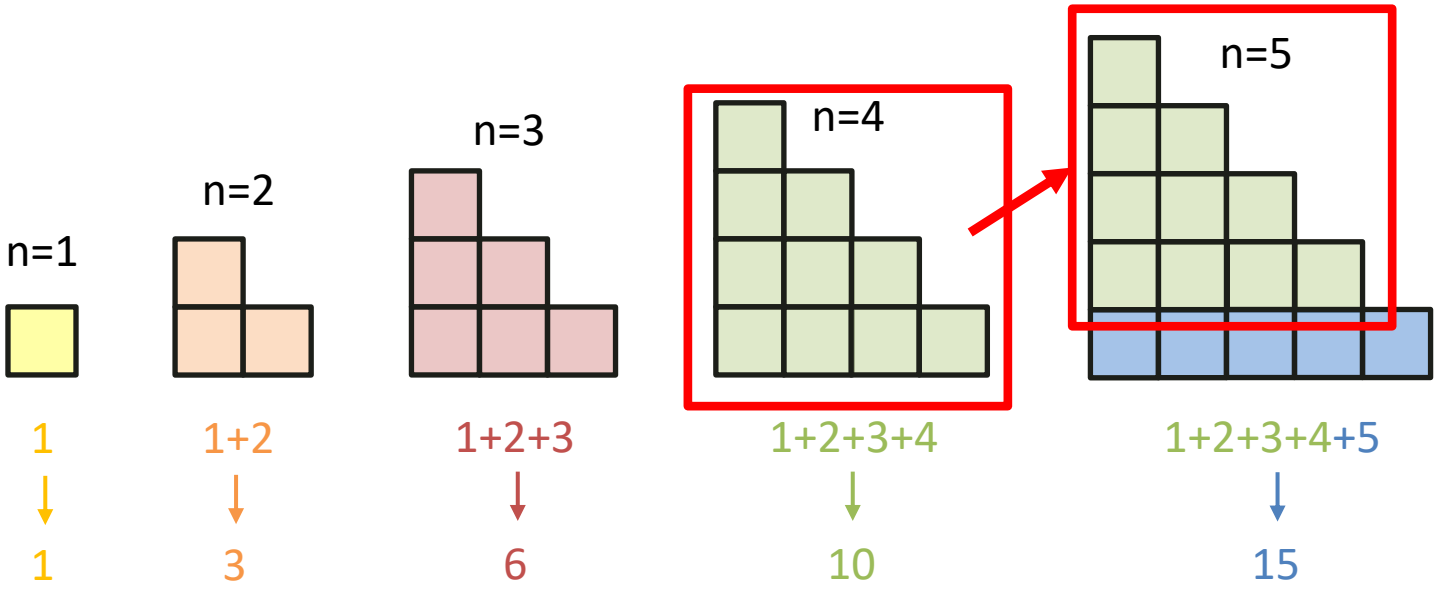
3. Erkenne den Zusammenhang

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



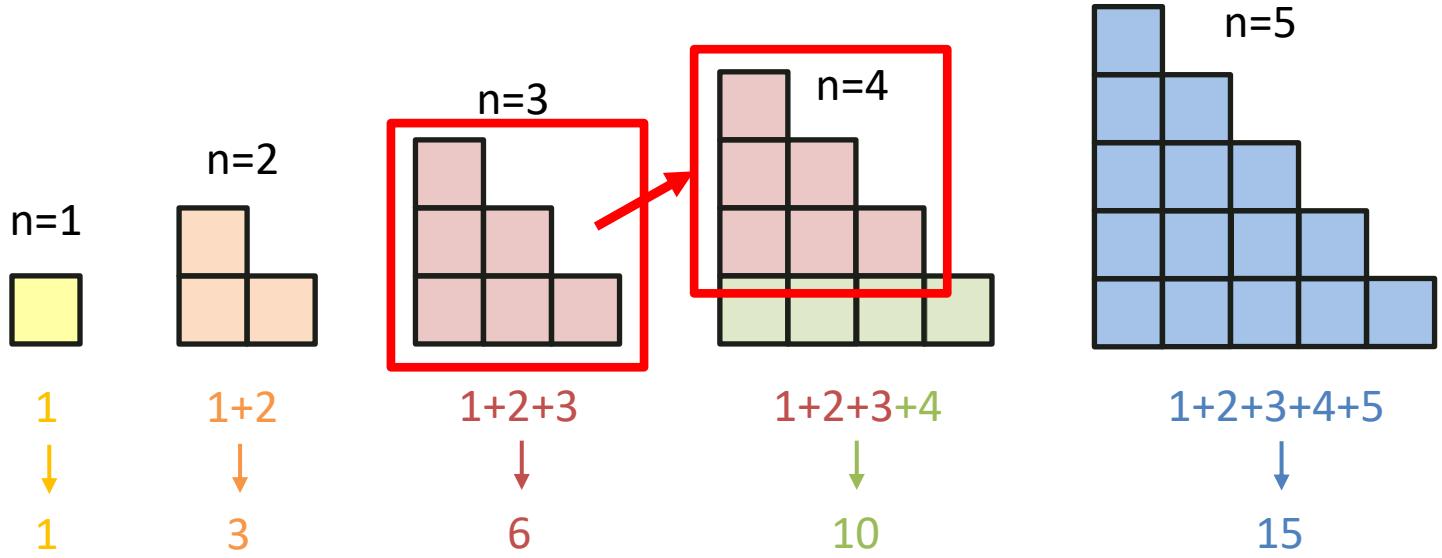
3. Erkenne den Zusammenhang

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



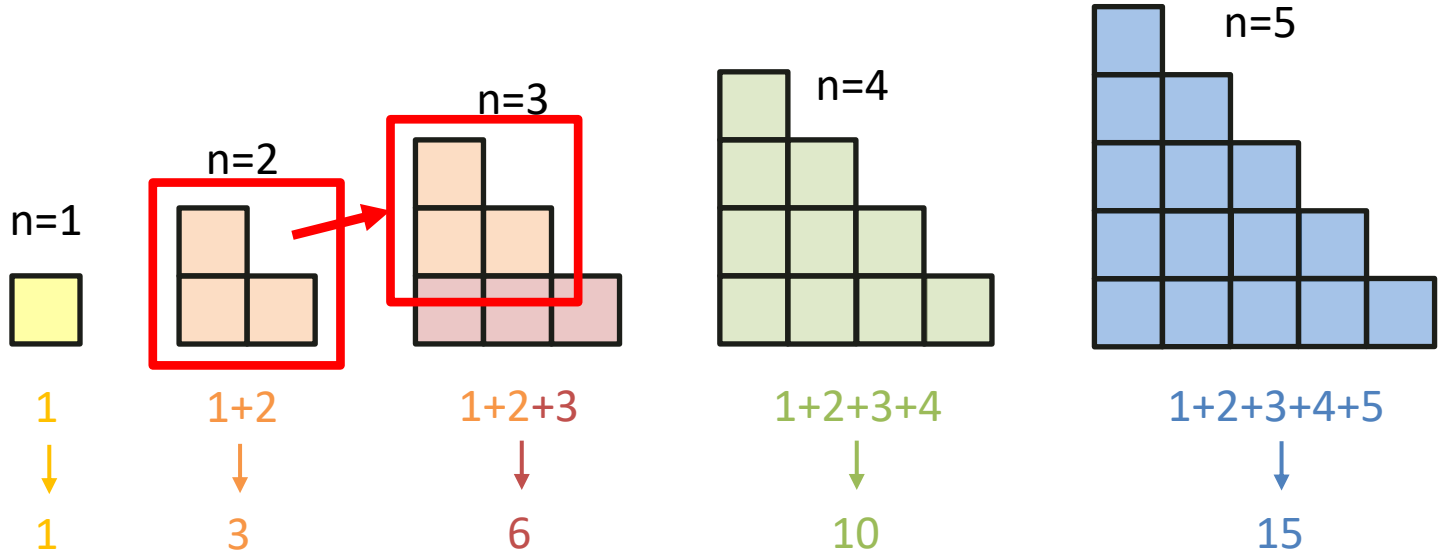
3. Erkenne den Zusammenhang

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



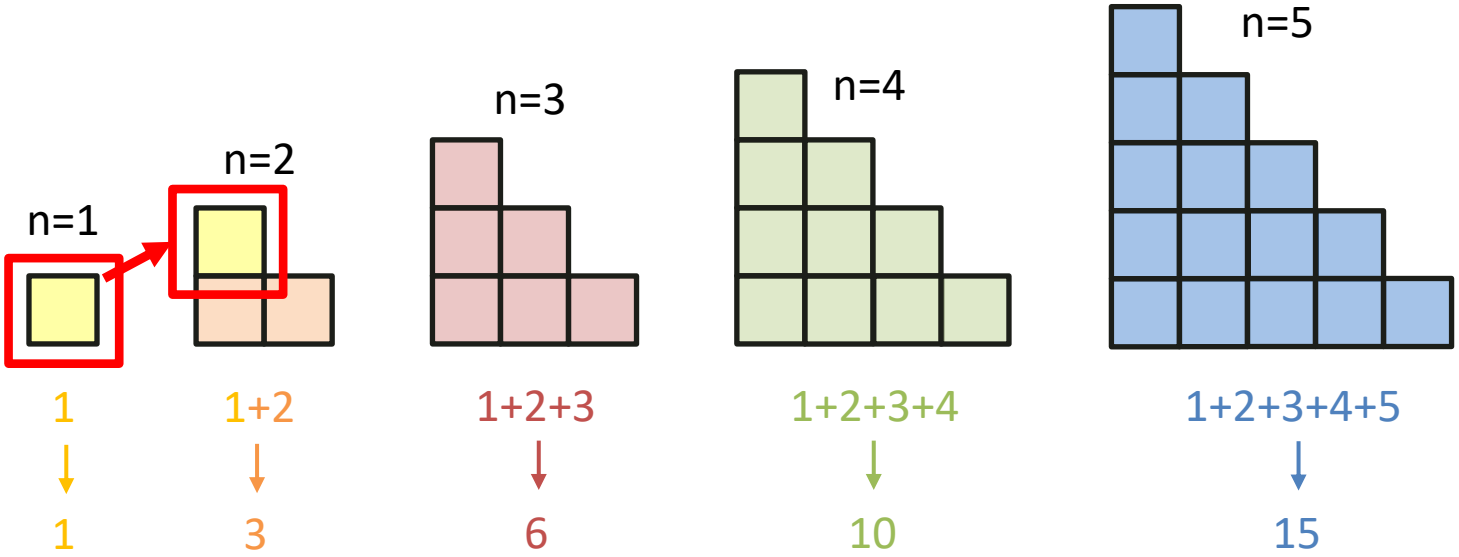
3. Erkenne den Zusammenhang

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

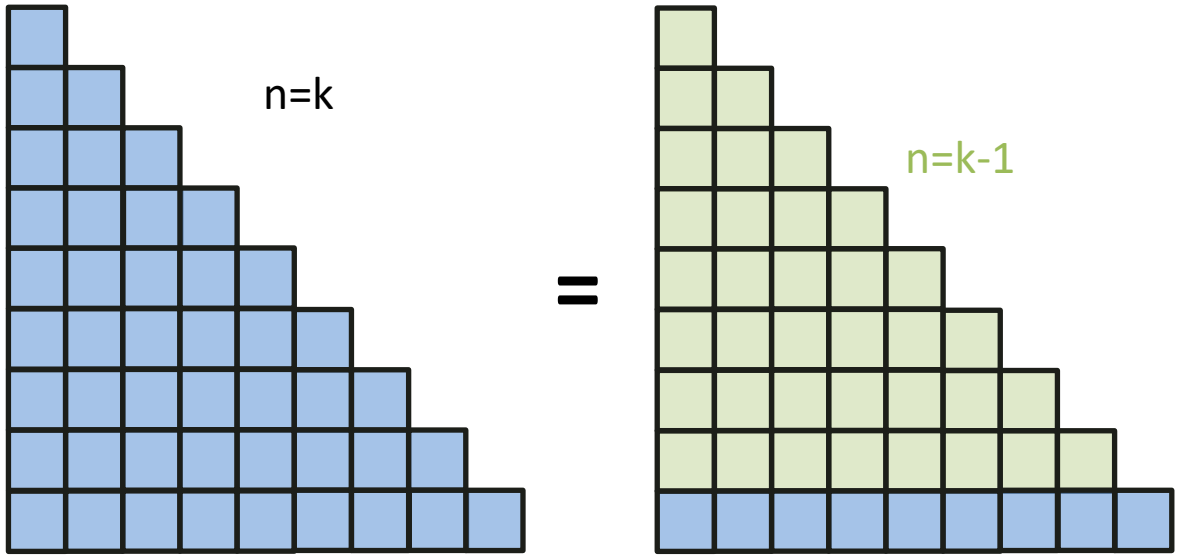


3. Erkenne den Zusammenhang

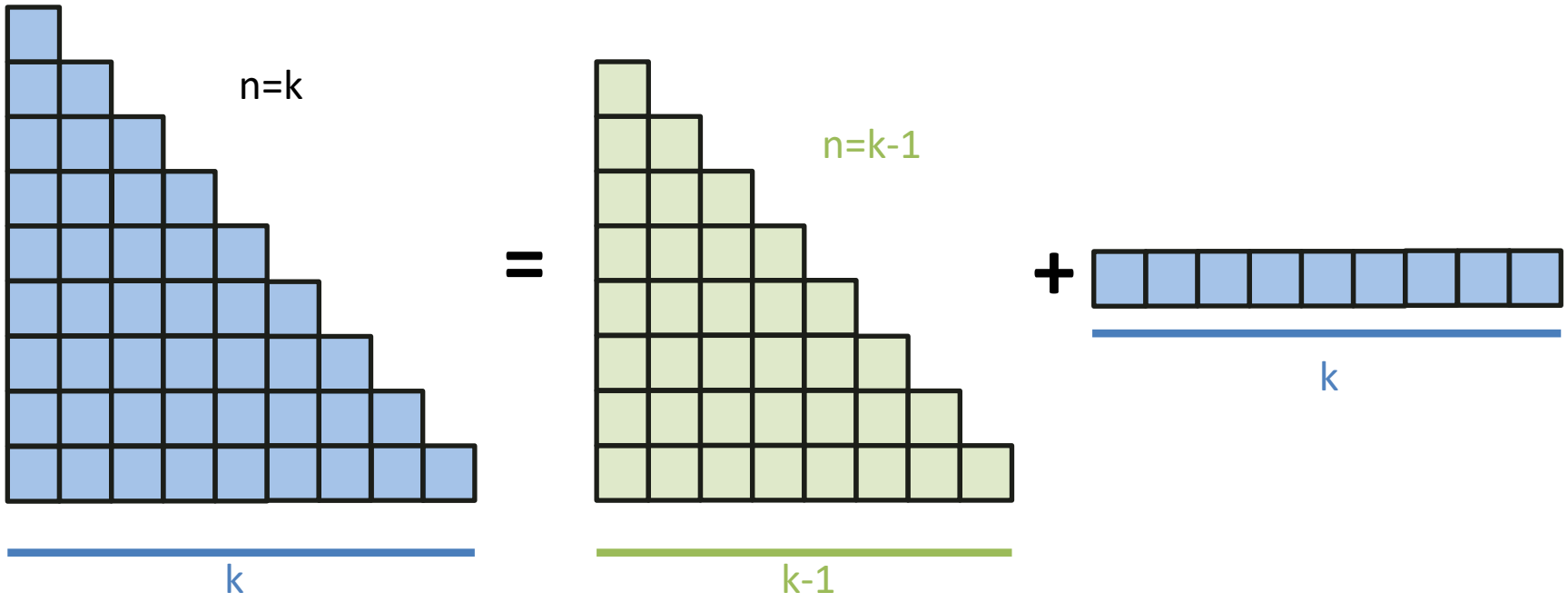
- „Wenn ich die Antwort für n=4 gegeben hätte, wie komme ich auf die Antwort für n=5? Für n=3 auf n=4? …“



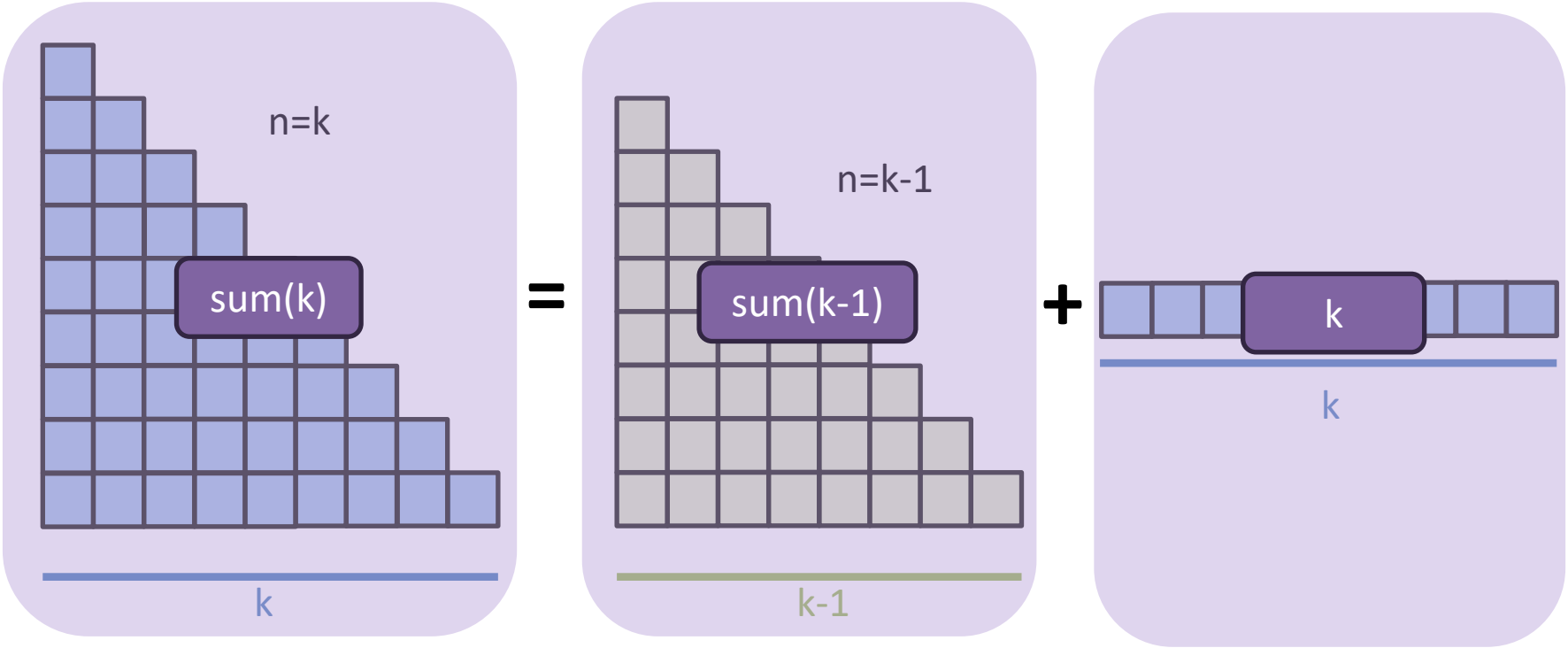
4. Das Muster



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 sum(n) = 0

Danach haben wir das *Rekursionsmuster* gefunden:

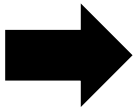
sum(n) = sum(n-1)+n

→ Dies können wir jetzt in Code umschreiben

5. Base Case + Muster → Code

Falls n=0: sum(n) = 0

Sonst: sum(n) = sum(n-1)+n

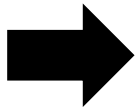


```
public static int sum(int n) {



}
```

5. Base Case + Muster → Code

Falls n=0: sum(n) = 0

Sonst: sum(n) = sum(n-1)+n

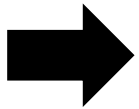


```
public static int sum(int n) {
    if (n == 0) {
        return 0;
    }

}
```

5. Base Case + Muster → Code

Falls n=0: sum(n) = 0

Sonst: sum(n) = sum(n-1)+n

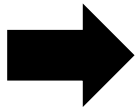


```
public static int sum(int n) {
    if (n == 0) {
        return 0;
    }

}
```

5. Base Case + Muster → Code

Falls n=0: sum(n) = 0

Sonst: sum(n) = sum(n-1)+n



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

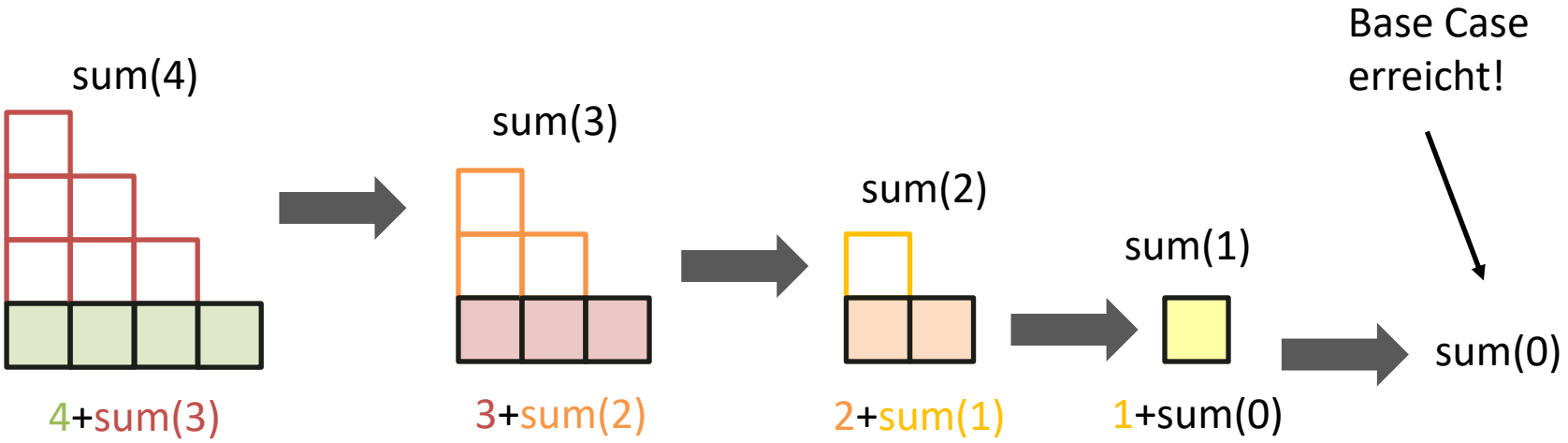
5. Base Case + Muster → Code

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

Fragen?

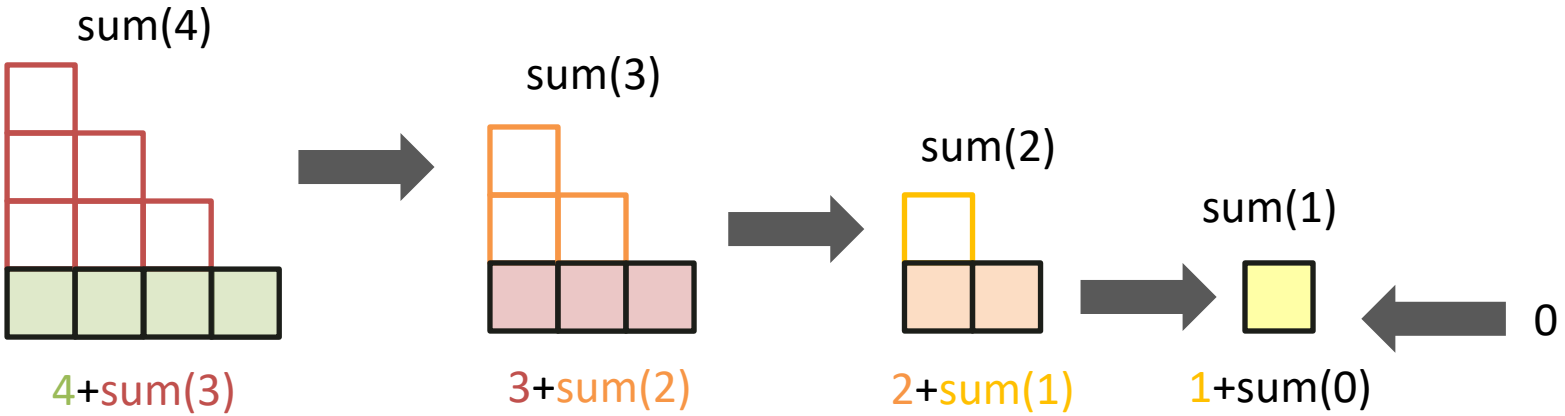
Was passiert wenn ich diesen Code ausführe?

1. Rekursive Aufrufe



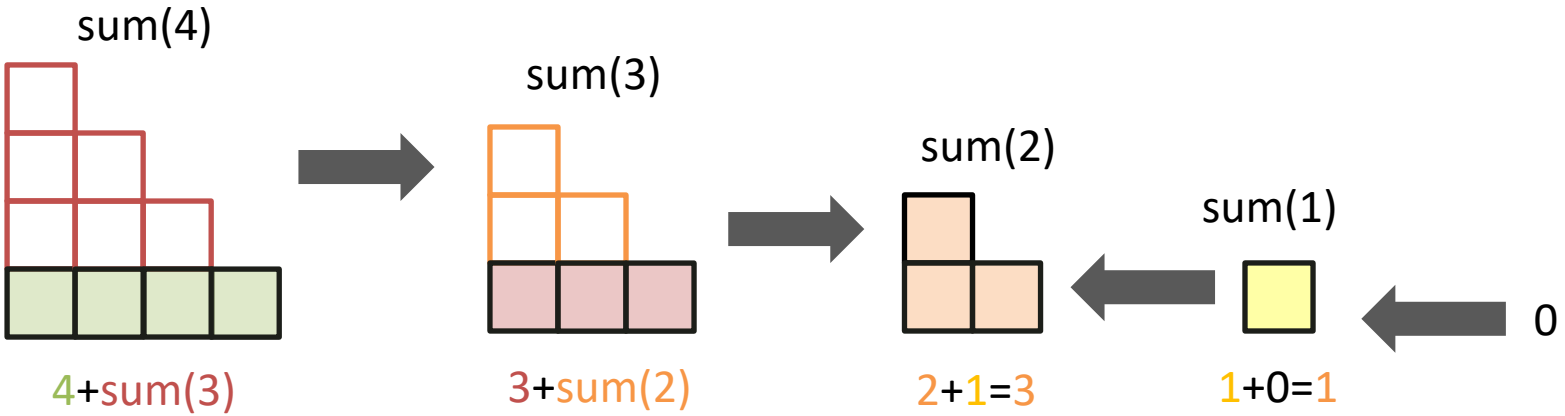
Was passiert wenn ich diesen Code ausführe?

2. Auflösen



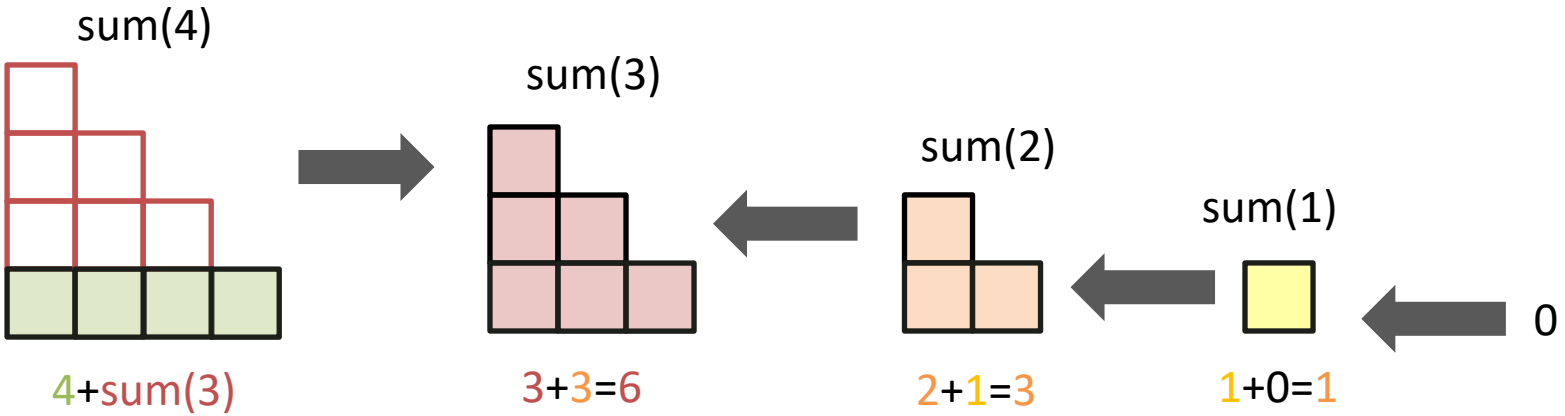
Was passiert wenn ich diesen Code ausführe?

2. Auflösen



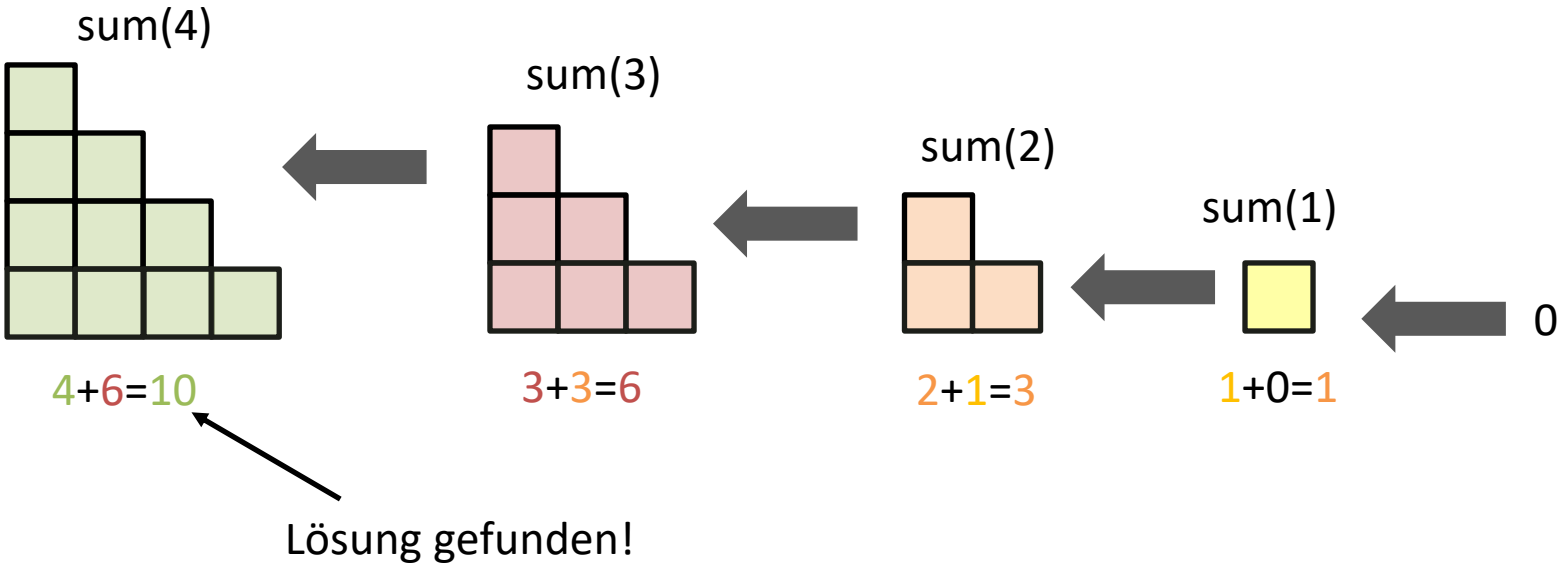
Was passiert wenn ich diesen Code ausführe?

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

```java
public class Palindrome {
    public static boolean isPalindrome(String word) {
        if (word.length() <= 1)
            return true;
        boolean firstAndLast = word.charAt(0) == word.charAt(word.length() - 1);
        String withoutFirstAndLast = word.substring(1, word.length() - 1);
        return firstAndLast && isPalindrome(withoutFirstAndLast);
    }
}
```

Rekursion mit Debugger

- 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.

- Wir gehen mit 'Step Over' durch die Methode, und die Variablen **`word`**, **`wordLength`**, **`firstAndLast`** erscheinen in der Variablentabelle, nachdem sie deklariert wurden.

- In Zeile 22 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").

- 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 der Variables Tabelle.

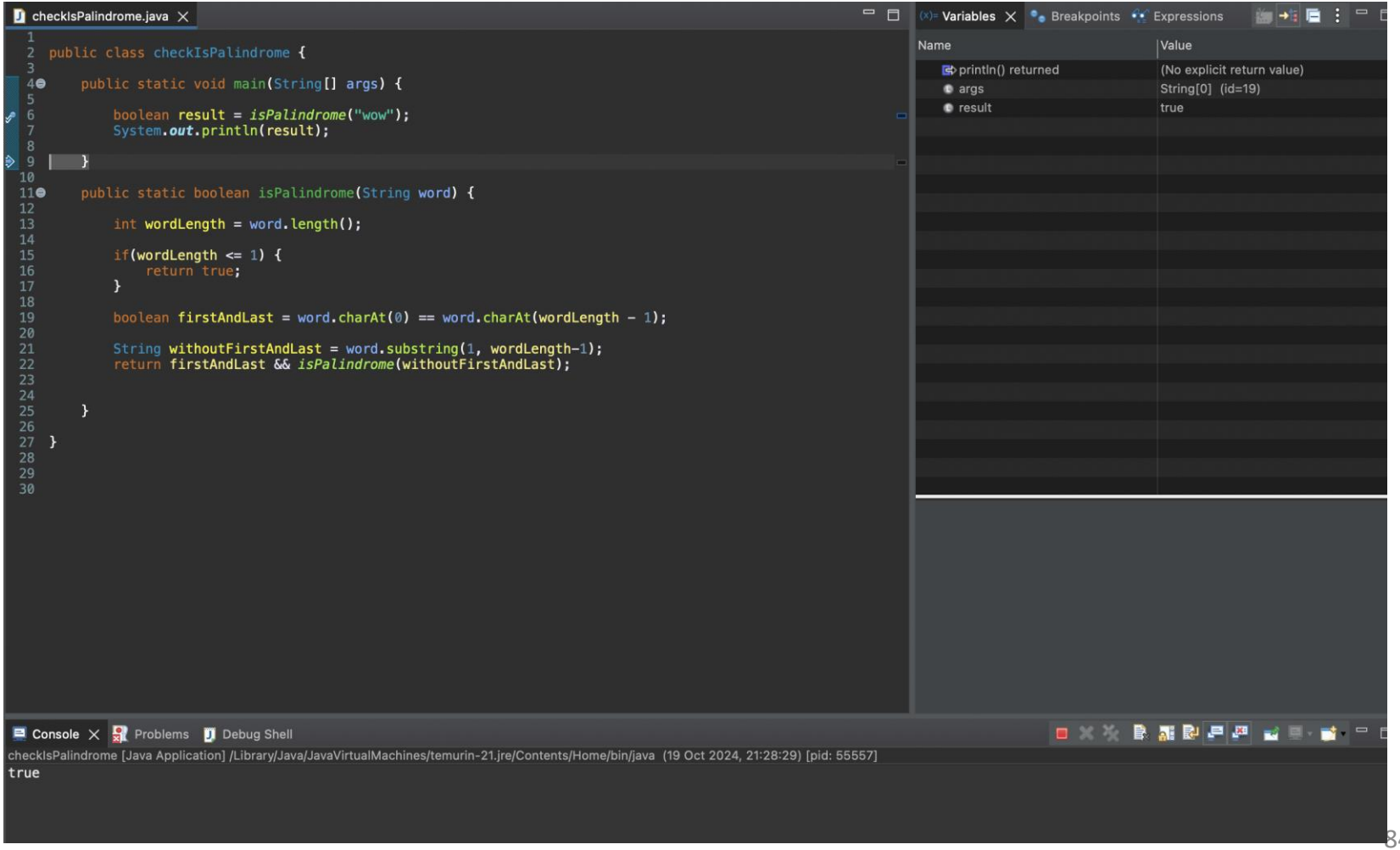
- "Da **`wordLength = 1`** ist, ist der Base Case erfüllt. Somit betreten wir das **if-Statement** und es wird **`True`** zurückgegeben.

- Aber wohin wird es zurückgegeben?

- **`True`** wird an die Stelle zurückgegeben, an der die Methode zuletzt aufgerufen wurde. Die Methode wurde zuletzt mit **`'o'`** aufgerufen. Nun zeigt auch die Variablentabelle 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.

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


checkIsPalindrome.java
public class checkIsPalindrome {
public static void main(String[] args) {
boolean result = isPalindrome("wow");
System.out.println(result);
}
public static boolean isPalindrome(String word) {
int wordLength = word.length();
if(wordLength <= 1) {
return true;
}
boolean firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);
String withoutFirstAndLast = word.substring(1, wordLength-1);
return firstAndLast && isPalindrome(withoutFirstAndLast);
}
}
Variables
Breakpoints
Expressions
Name
Value
println() returned
(No explicit return value)
args
String[0] (id=19)
result
true
Console
Problems
Debug Shell
checkIsPalindrome [Java Application] /Library/Java/JavaVirtualMachines/temurin-21.jre/Contents/Home/bin/java (19 Oct 2024, 21:28:29) [pid: 55557]
true

Was wird hier passieren?

```
1
2  public class checkIsPalindrome {
3
4      public static void main(String[] args) {
5
6          boolean result = isPalindrome("baab");
7          System.out.println(result);
8
9      }
10
11     public static boolean isPalindrome(String word) {
12
13         int wordLength = word.length();
14
15         boolean firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);
16
17         String withoutFirstAndLast = word.substring(1, wordLength - 1);
18
19         return isPalindrome(withoutFirstAndLast) && firstAndLast;
20
21     }
22
23 }
24
```

Error?,Endlose Rekursion?, Korrektes ausführen?, …

Was wird hier passieren?

```
1
2  public class checkIsPalindrome {
3
4      public static void main(String[] args) {
5
6          boolean result = isPalindrome("wow");
7          System.out.println(result);
8
9      }
10
11     public static boolean isPalindrome(String word) {
12
13         int wordLength = word.length();
14
15         boolean firstAndLast = false;
16
17         String withoutFirstAndLast = "";
18
19         if (wordLength != 1 && wordLength != 0) {
20             withoutFirstAndLast = word.substring(1, wordLength - 1);
21             firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);
22         }
23
24         return isPalindrome(withoutFirstAndLast) && firstAndLast;
25
26     }
27
28 }
29
```

Was wird hier passieren?

```java
public class checkIsPalindrome {

    public static void main(String[] args) {

        boolean result = isPalindrome("wow");
        System.out.println(result);

    }

    public static boolean isPalindrome(String word) {

        int wordLength = word.length();

        if(wordLength < 1) {
            return true;
        }

        boolean firstAndLast = false;

        String withoutFirstAndLast = "";

        if (wordLength != 1 && wordLength != 0) {
            withoutFirstAndLast = word.substring(1, wordLength - 1);
            firstAndLast = word.charAt(0) == word.charAt(wordLength - 1);
        }

        return isPalindrome(withoutFirstAndLast) && firstAndLast;

    }

}
```

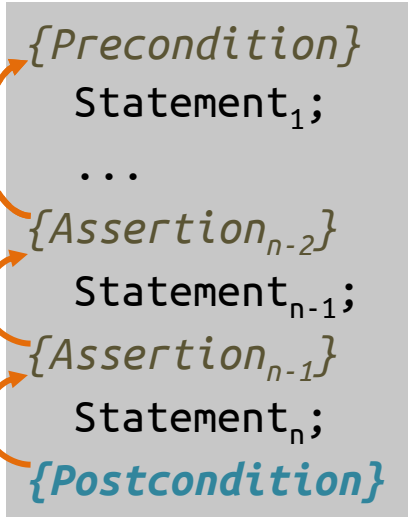
Logisches Schliessen I

Stärkere und schwächere Aussagen

- Wir können *stärkere* und *schwächere* Aussagen unterscheiden
  - Wenn $P_1 \Longrightarrow P_2$ 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



- **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, demensprechend 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 ————

Weakest Precondition – Einfaches Beispiel

{|y|>2}

x = y*y ————

Weakest Precondition – Zweites Beispiel

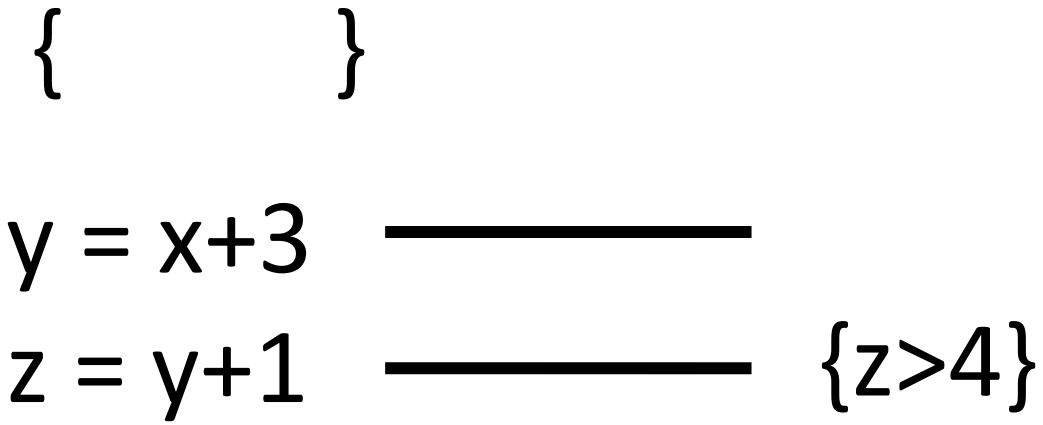
{ }

$y = x+3$ ———

$z = y+1$ ———

{z>4}

Weakest Precondition – Zweites Beispiel



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}

$z = y+1$ ———

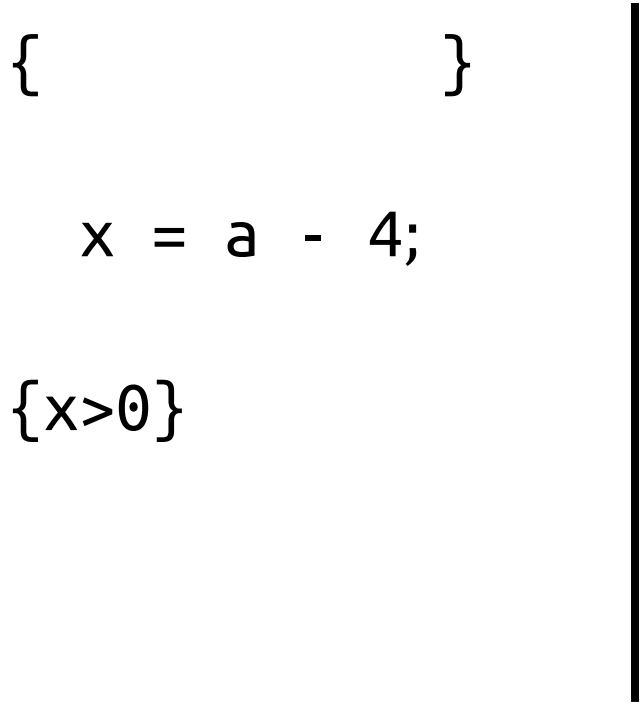
Weakest Precondition – Zweites Beispiel

$\{x > 0\}$

$y = x+3$ ——————

$z = y+1$ ——————

Schwächste Vorbedinung - Beispiel



Schwächste Vorbedinung – Beispiel

Alle Wege zur Postcondition müssen funktionieren mit eurer Precondition

```
{                }

  if (x >= y){
    y = x
  }

{y >= x}
```

Schwächste Vorbedinung - Beispiel

```
{                  }

  x = y + z;

  y = y - 5;

{y < 0}
```

Prüfungsbeispiele

Schwächste Vorbedinung – Prüfungsbeispiel

```
{                    }

  w = a

  x = w + b;

  y = x * 2;

{y > 0 && b > 10}
```

Schwächste Vorbedinung – Prüfungsbeispiel

```
{                    }

 p = 3 * q

 p = p + 1;

{p > 15}
```

Schwächste Vorbedinung – Prüfungsbeispiel

```
{                    }

if (x > y) {
  z = x - y
} else {
  z = y - x;
}

{z > 0}
```

(schwer)

2. WP: {

```
if (a > 0) {
  c = 2 * a;
  b = c + 1;
}

Q: { b > 5 }
```

}

Lösung

2. WP: {

```
if (a > 0) {
  c = 2 * a;
  b = c + 1;
}

Q: { b > 5 }
```

}

$$a > 2 \,||\, (b > 5 \,\&\&\, a <= 0)$$

2. WP: { }

```
p = 2 * q + 1;
t = p + 1;

Q: { t > 15 }
```

Lösung

2. WP: {

```
p = 2 * q + 1;
t = p + 1;

Q: { t > 15 }
```

{q >= 7}

}

1. WP: { }

```
x = z;
if (x > 0) {
  a = x * x + 1;
} else {
  a = x * (x + 1);
}

Q: { a > 0 }
```

1. WP: {

```
x = z;
if (x > 0) {
  a = x * x + 1;
} else {
  a = x * (x + 1);
}

Q: { a > 0 }
```

}

{z != 0 && z != -1}

Kahoot / Vorbesprechung

Kahoot?

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.

- **Base case überlegen**
- **Rekursion überlegen**
- **Wie können wir das Problem stückweise reduzieren**
  - Hier wäre eine Idee zum Beispiel immer die Substrings zu verkleinern

Aufgabe 2: Weakest Precondition

Bitte geben Sie für die folgenden Programmsegmente die schwächste Vorbedingung (weakest precondition) an. Bitte verwenden Sie Java Syntax (die Klammern { und } können Sie weglassen). 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!
```

Nützliche Methoden

Für Strings: .endsWith() .startsWith() .contains()   .equals()

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 Kelchblattlänge 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 gennannt.
- 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.

- **Um OutOfBoundsExceptions zu vermeiden, eignet es sich immer einen check zu machen**
- **z.B. i>=0, oder i < array.length.**
- **Überlegt euch wie ihr über eine Matrix iteriert**