Woche 08

Rekursion, Rekursive Datenstrukturen, Bäume & Merge Sort

Rekursion

- ist ein Divide & Conquer Ansatz zum Lösen der Probleme.
- Wenn ein Problem in kleinere Teilprobleme zerlegt werden kann, die später kombiniert werden, kann man dies rekursiv lösen.

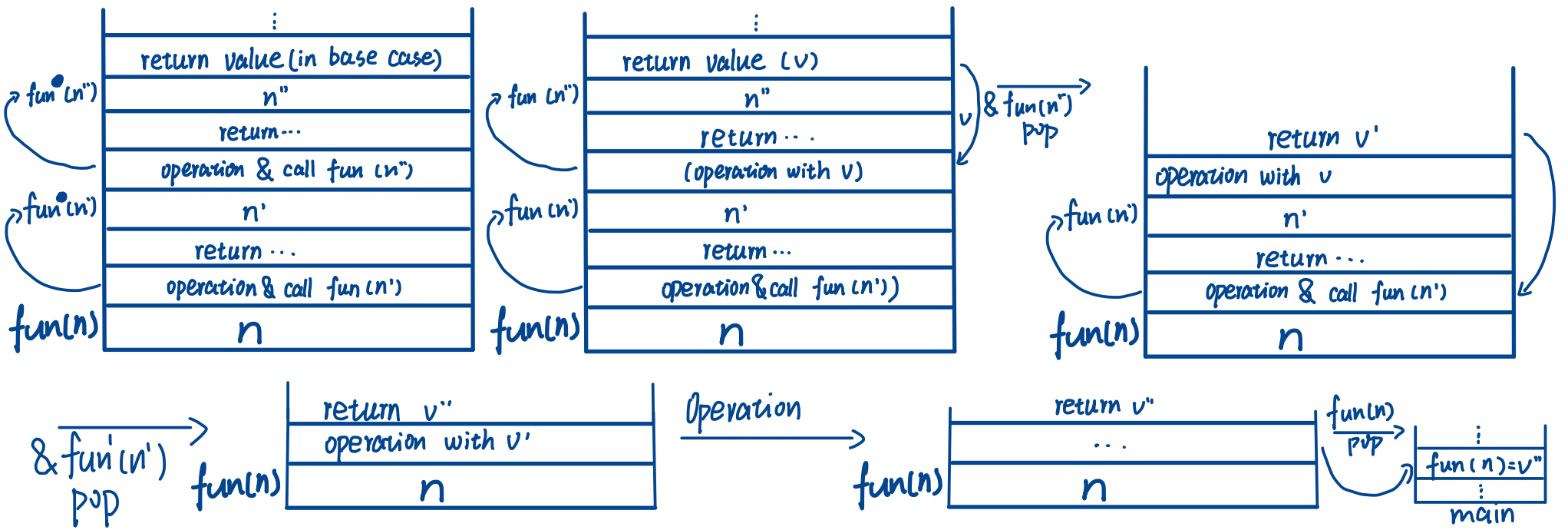
Methodenstruktur

```
recursiveFunc() {
    if (checkForBaseCase) { // base case
        Solve the problem directly without recursion;
    } else { // recursive case
        Divide the problem into smaller subproblems of the same type;
        Call recursiveFunc() on each subproblem;
        Combine the results from the subproblems;
    }
}
```

Rekursives Denken

- Jede Eingabe muss durch einen Fall abgedeckt sein
  - Basisfall oder Rekursiver Aufruf
- Es muss einen Basisfall geben, der keine rekursiven Aufrufe macht. vermeiden Stack-Overflow
- Der rekursive Fall muss das Problem vereinfachen und Fortschritte in Richtung des Basisfalls machen.

in Stack:



Rekursion

Implementierung der **iterativen** Fakultätsfunktion

```
1 public int fact (int n) {
2     int result = 1;
3     while (n > 0) {
4         result = result * n;
5         n--;
6     }
7     return result;
8 }
```

Ablauf:

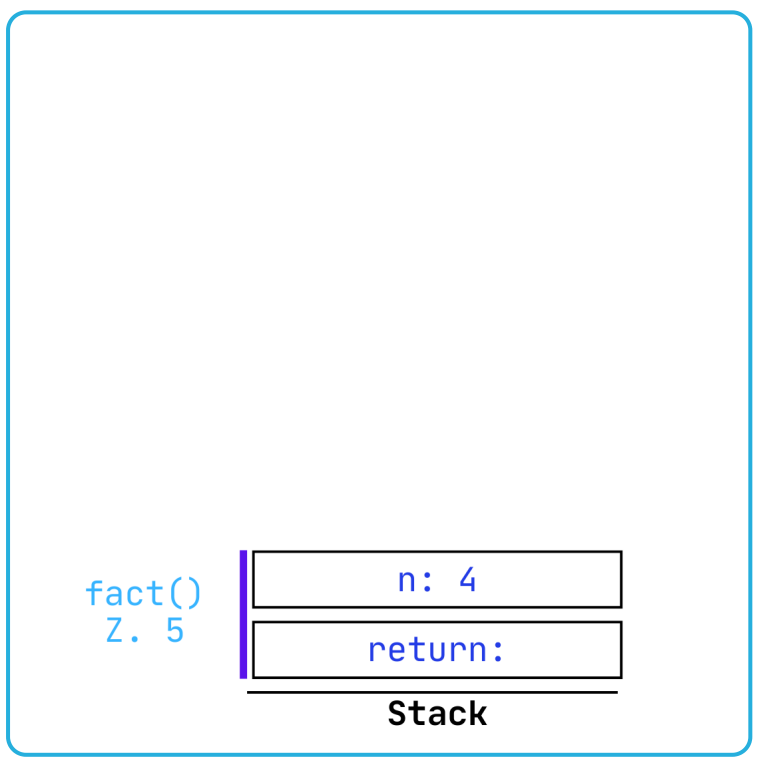
```
1 fact(1) = 1
2 fact(2) = 2 * 1
3 fact(3) = 3 * 2 * 1
4 fact(4) = 4 * 3 * 2 * 1
```

Rekursion

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

Stack Trace:

```
fact(4) = 4 * fact(3)
```

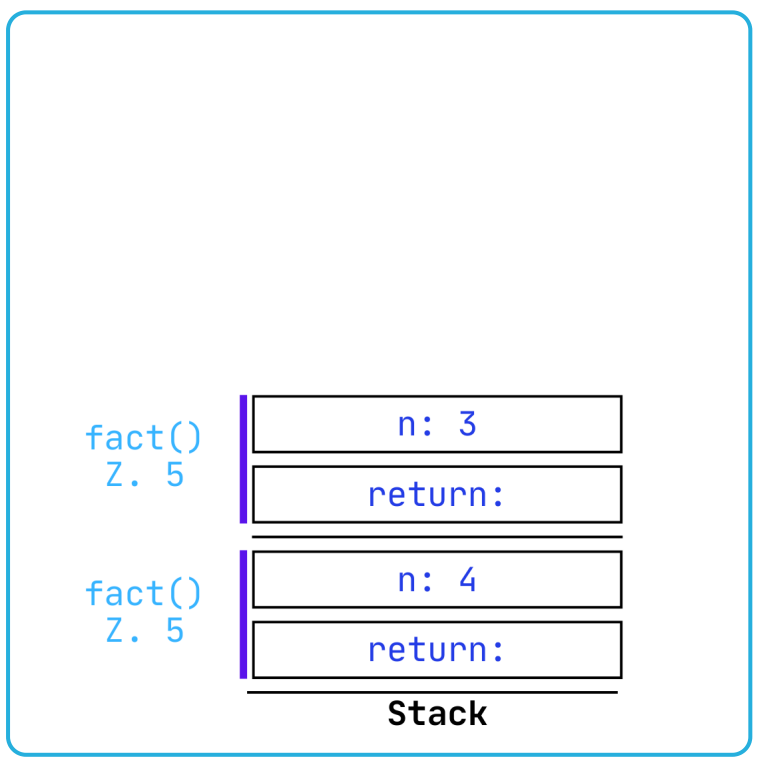


Rekursion

```java
public static int fact (int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact (n - 1);
}
```

Stack Trace:

```
fact(4) = 4 * fact(3)
fact(3) = 3 * fact(2)
```

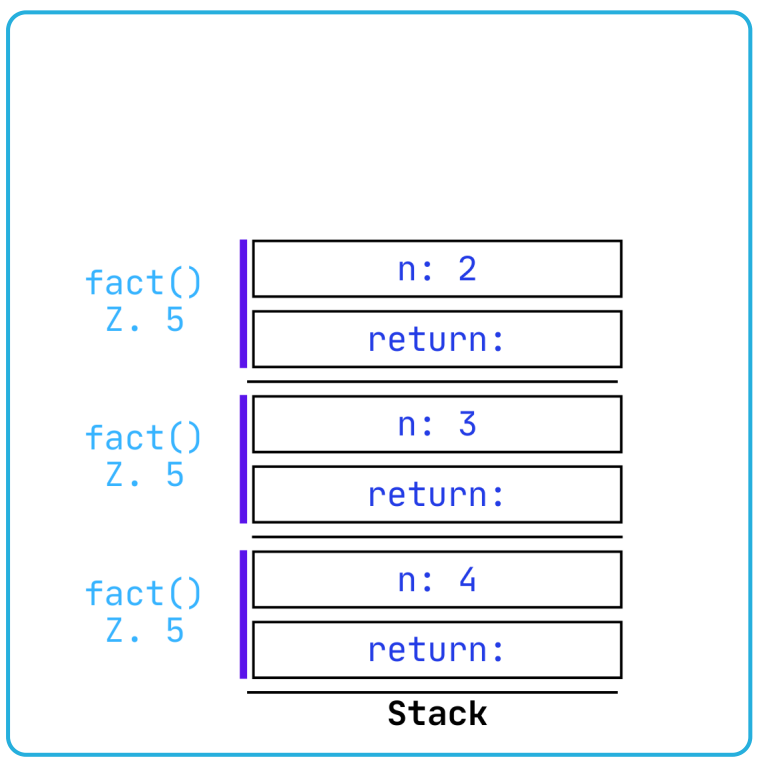


Rekursion

```java
public static int fact (int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact (n - 1);
}
```

Stack Trace:

```
fact(4) = 4 * fact(3)
fact(3) = 3 * fact(2)
fact(2) = 2 * fact(1)
```

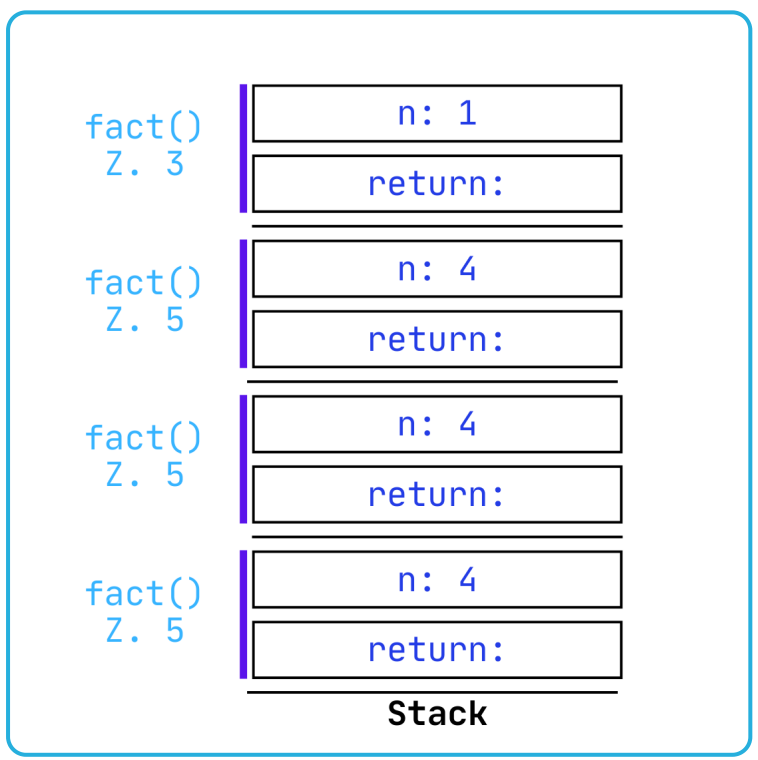


Rekursion

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

Stack Trace:

```
fact(4) = 4 * fact(3)
fact(3) = 3 * fact(2)
fact(2) = 2 * fact(1)
```

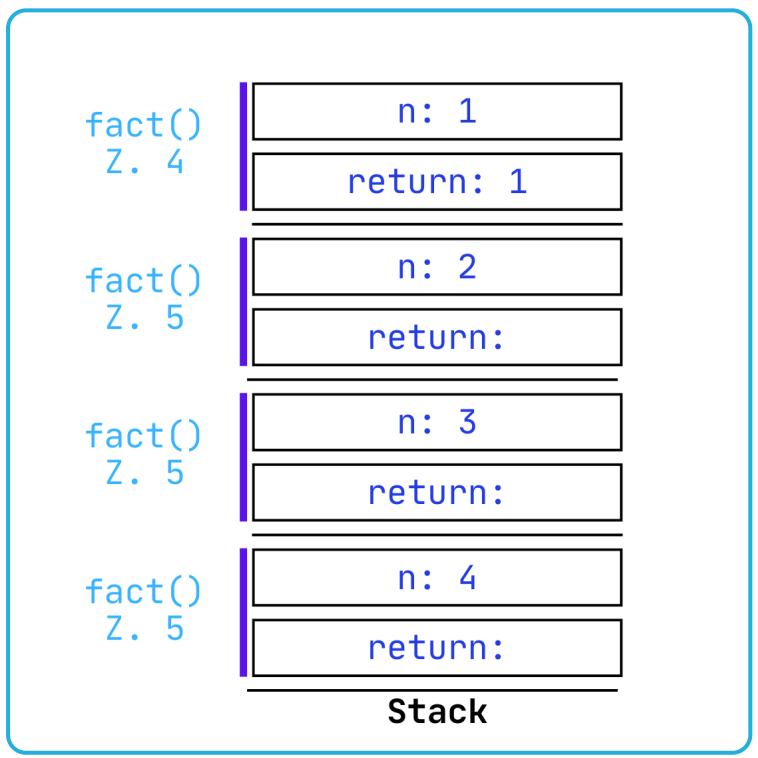


Rekursion

```java
public static int fact (int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact (n - 1);
}
```

Stack Trace:

```
fact(4) = 4 * fact(3)
fact(3) = 3 * fact(2)
fact(2) = 2 * fact(1)
fact(1) = 1
```

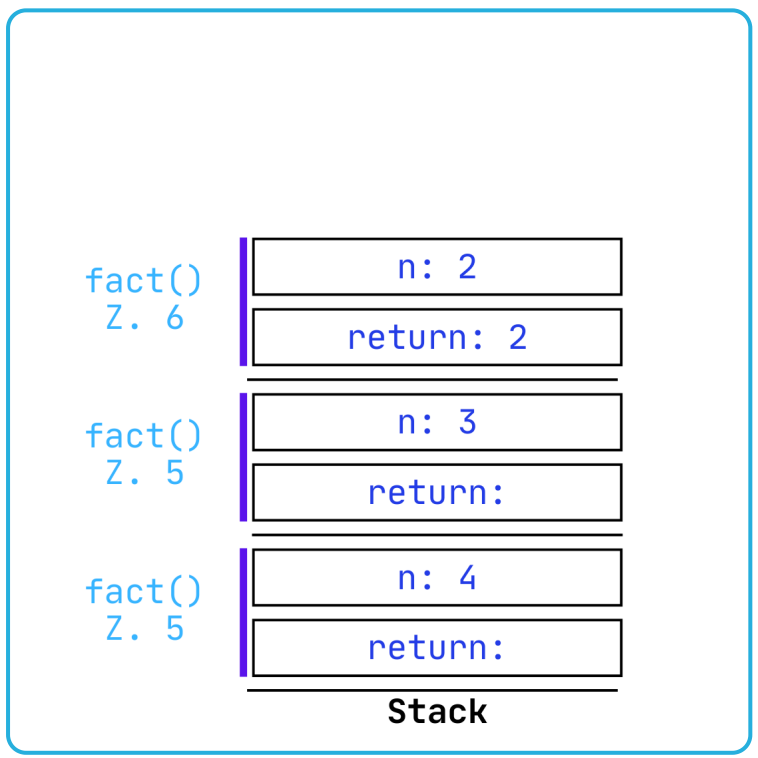


Rekursion

```java
public static int fact (int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact (n - 1);
}
```

Stack Trace:

```
fact(4) = 4 * fact(3)
fact(3) = 3 * fact(2)
fact(2) = 2 * fact(1) = 2
fact(1) = 1
```

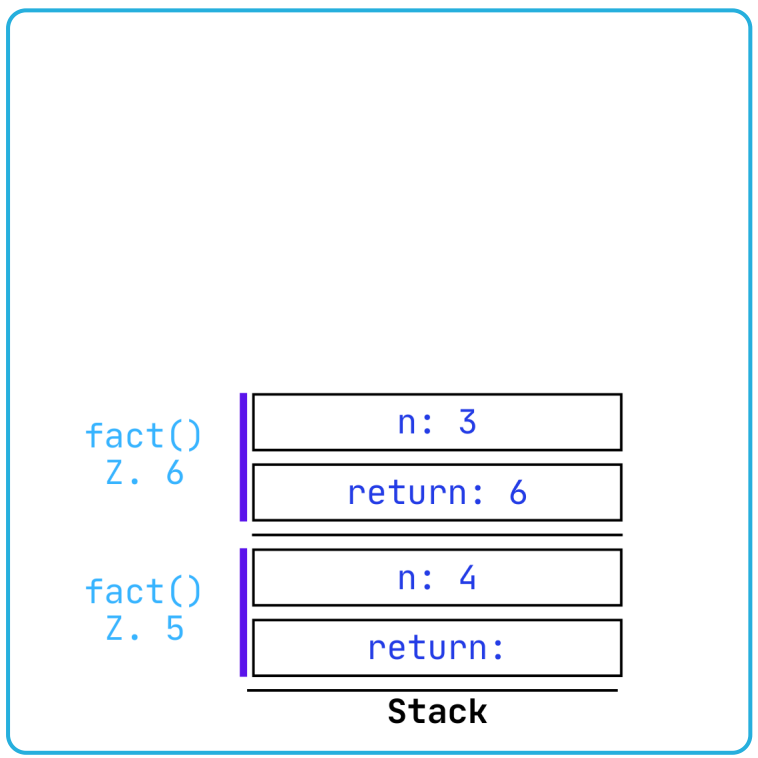


Rekursion

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

Stack Trace:

```
fact(4) = 4 * fact(3)
fact(3) = 3 * fact(2) = 6
fact(2) = 2 * fact(1) = 2
fact(1) = 1
```

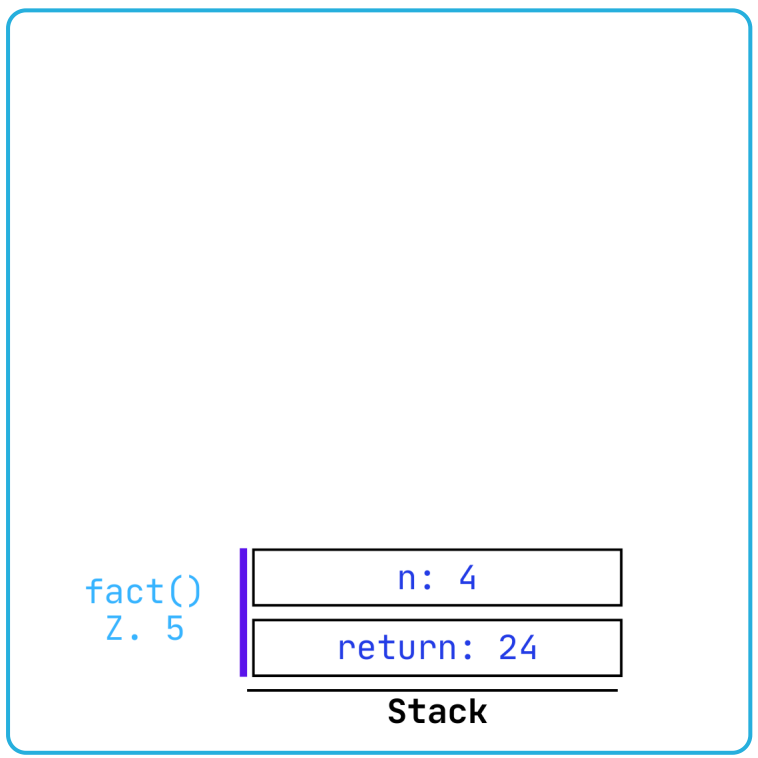


Rekursion

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

Stack Trace:

```
fact(4) = 4 * fact(3) = 24
fact(3) = 3 * fact(2) = 6
fact(2) = 2 * fact(1) = 2
fact(1) = 1
```



Endrekursion

Eine Methode ist endrekursiv (tail recursive), wenn der rekursiver Funktionsarufruf die letzte Aktion zur Berechnung von f ist. Accumulator als Argument

Beispiel Nr. 1 : Aufsummieren der Zahlen von 1 bis 10 **rekursiv**

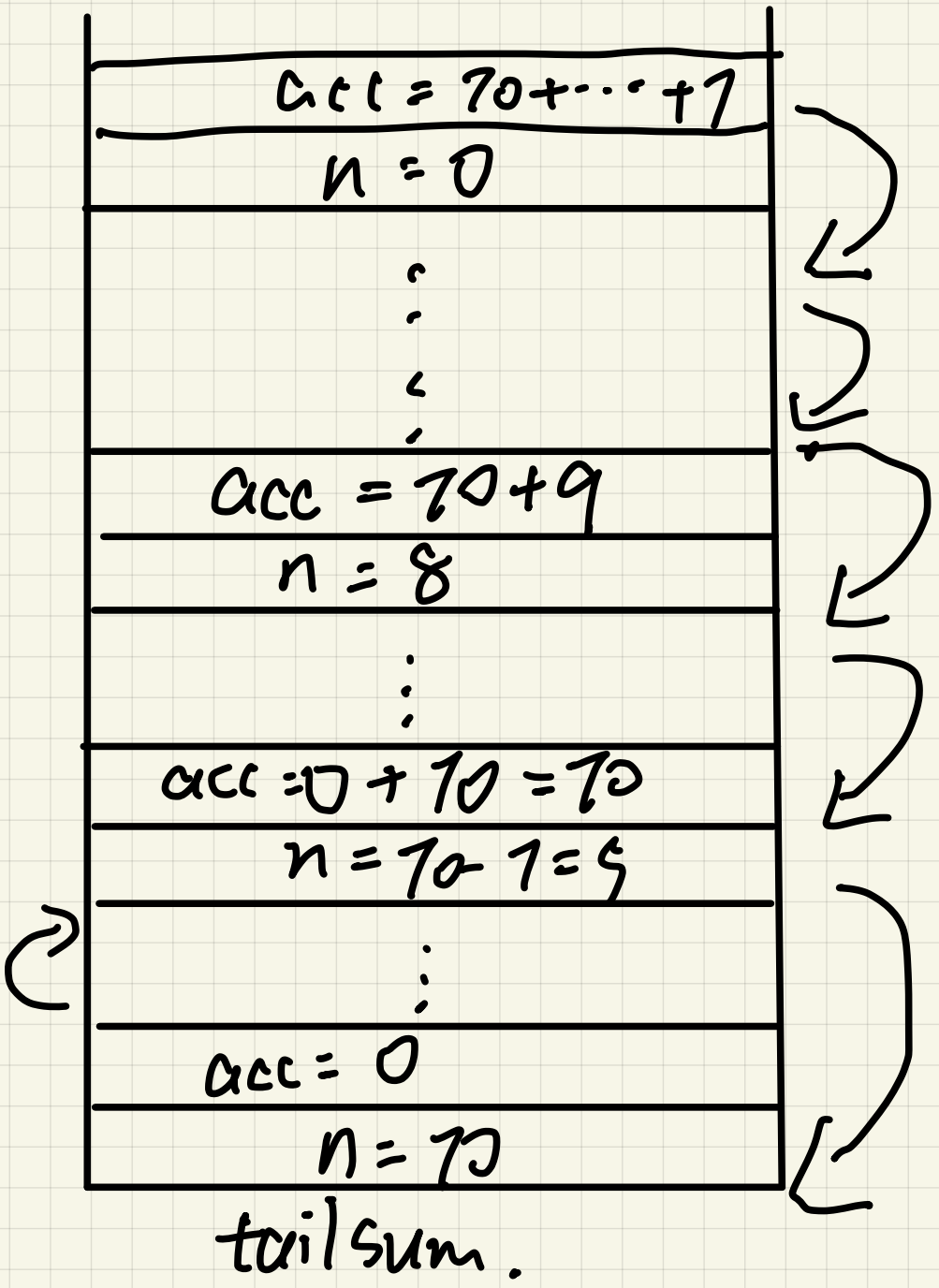
```
1  public static int sum (int n) {
2      if (n > 1) {                          //condition
3          return sum (n - 1) + n;           //recurse
4      }
5      return n;
6  }
```

Beispiel Nr. 2 : Aufsummieren der Zahlen von 1 bis 10 **endrekursiv**

```
1  public int tailSum(int accumulator, int n){
2      if (n < 1) {                          //condition
3          return accumulator;
4      }
5      return tailSum(accumulator + n, n-1);   //tail recurse as last item
6  }
```

Tail-Rekursion kann speichereffizienter als normale Rekursion durchgeführt werden, da keine zusätzlichen Stack-Frames für verschachtelte Aufrufe benötigt werden.

$10+9+8+2+6+5+4+3+2+1$.



tailsum.

W08P01 - Rekursion

Bearbeite nun die W08P01 auf Artemis.

W08P02 - Potenzmenge

Bearbeite nun die W08P02 auf Artemis.

* nur den Basisfall und den letzte rekursive Aufruf berücksichtigen
  Funktion mehrmals rekursive aufrufen, Parameter nach Bedarf ändern

Rekursives Sortieren

Merge Sort:
- funktioniert nach dem Divide & Conquer Prinzip.
- teilt die Eingabe in kleinere Teile, sortiert diese und fügt die wieder zusammen.

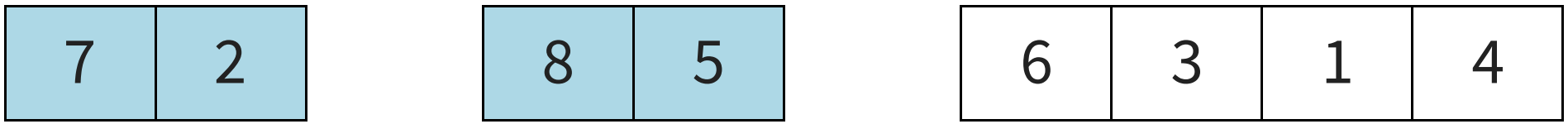
- Kann das Array nicht in zwei gleich lange Teile aufgeteilt werden, sollte die erste Hälfte um ein Element länger sein als die zweite.

Merge Sort - 1

Beispiel mit Eingabe: [7, 2, 8, 5, 6, 3, 1, 4]

Teile das Array in die Hälfte:

Teile nochmal in die Hälfte:

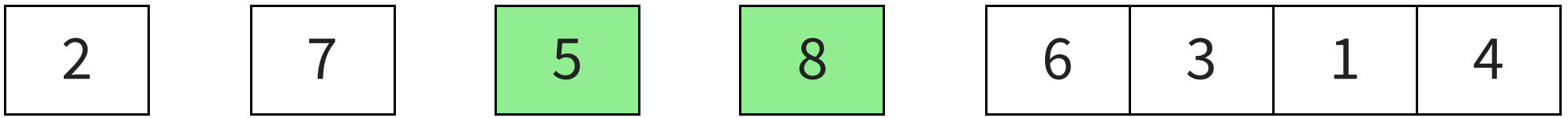


Teile nochmal in die Hälfte um Einzelelemente zu als Teilarrays zu erhalten:

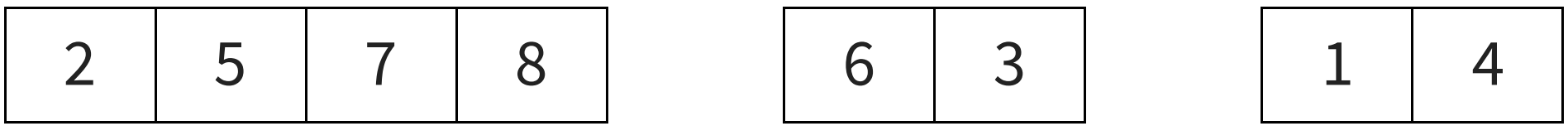
Sortiere den ersten Teil.

Merge Sort - 2

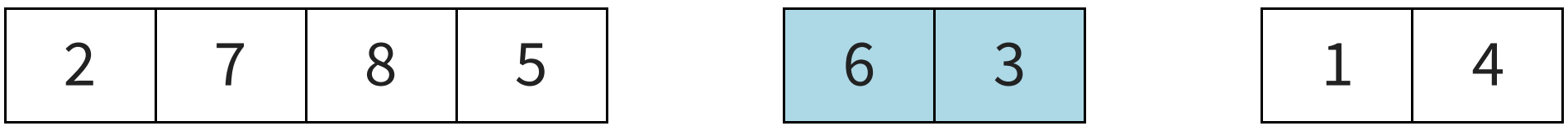
Sortiere die Einzelelemente im zweiten Teil.



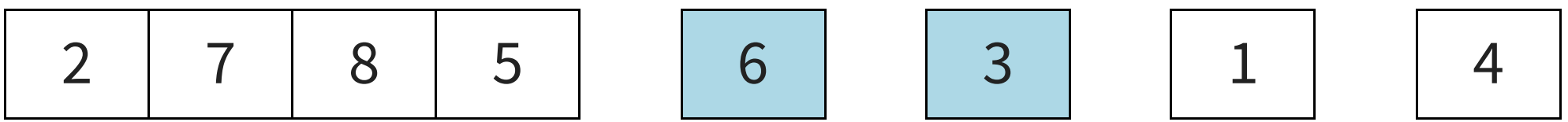
Berechne nun die Sortierung von [7, 2, 8, 5] aus Sortierungen der kleineren Felder [7, 2] & [8, 5]. (Merge)



Der erste Teil ist nun aufsteigend sortiert. Jetzt muss der zweite Teil genauso sortiert werden. Teile das linkere Array in die Hälfte.

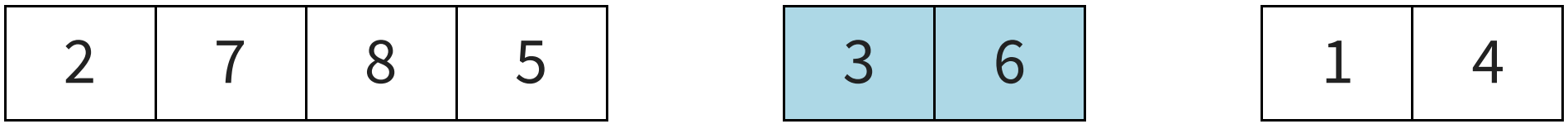


Teile das Array nochmal in die Hälfte.



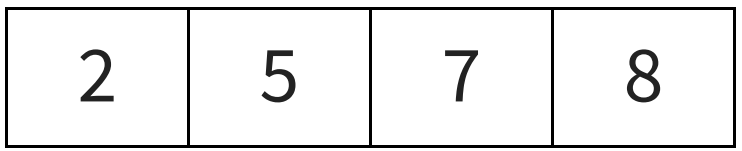
MergeSort - 3

Sortiere die Einzelelemente und Merge.

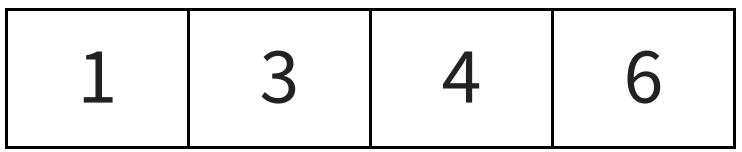


Jetzt sind beide Hälften von dem Eingabearray sortiert und müssen noch zusammengeführt werden.

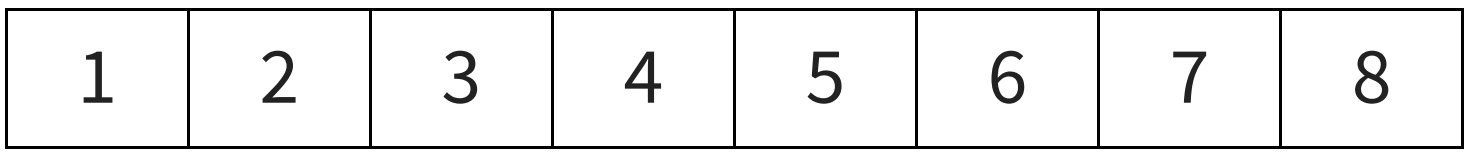
Bereits berechnete Sorierung von [7, 2, 8, 5]:



Bereits berechnete Sorierung von [6, 3, 1, 4]:



Merge beide Teile für die Gesamtsortierung:



W08P03 - Rekursives Sortieren

Bearbeite nun die W08P03 auf Artemis.

Nach deiner rekursiven Implementierung für Merge Sort bzw. Stooge Sort führe die Klasse SortingComparison anhand des bereitgestellten Templates aus. Die Ausgabe deines Programms sollte sich wie folgt verhalten:

```
> Task :SortingComparison.main()

Testing with the large list (20,000 elements):

Bubble Sort Time: 1703 ms
Merge Sort Time: 19 ms
```

Nun kannst du die Laufzeitkomplexitäten von Sortierverfahren vergleichen.

Binäre Suche

Ausgehend von einem aufsteigend sortierten Array: Gesucht: Ist 9 in diesem Array vorhanden?

1	3	4	7	9	14	15	17	20	21	22	42	44

Ist die gesuchte 9 größer oder kleiner als der mittlere Eintrag des Arrays?

1	3	4	7	9	14	**15**	17	20	21	22	42	44

Kleiner. D.h, die 9 kann nicht im rechten Teil sein.

1	3	4	7	9	14	15	17	20	21	22	42	44

Ist die gesuchte 9 größer oder kleiner als der mittlere Eintrag des neuen Arrays?

1	3	4	**7**	9	14	15	17	20	21	22	42	44

Größer. D.h, 9 kann nicht im linkeren Teil sein.

1	3	4	7	9	14	15	17	20	21	22	42	44

Binäre Suche

Ist die gesuchte 9 größer oder kleiner als der mittlere Eintrag des neuen Arrays?

1	3	4	7	9	14	15	17	20	21	22	42	44

Kleiner. D.h, kann nicht im rechten Teil sein.

1	3	4	7	9	14	15	17	20	21	22	42	44

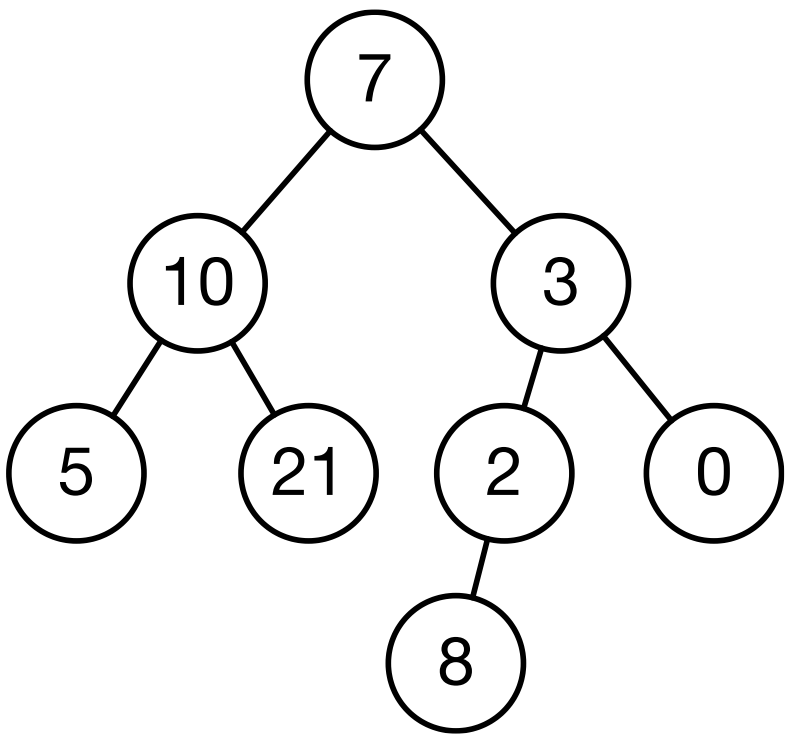
Ist die gesuchte 9 größer oder kleiner als der mittlere Eintrag des neuen Arrays? –> Gleich. Gefunden!

1	3	4	7	9	14	15	17	20	21	22	42	44

Die binäre Suche funktioniert auf sortierten Arrays/Listen. Dies kann man auf einem binären Suchbaum modellieren.

Rekursive DS - Bäume

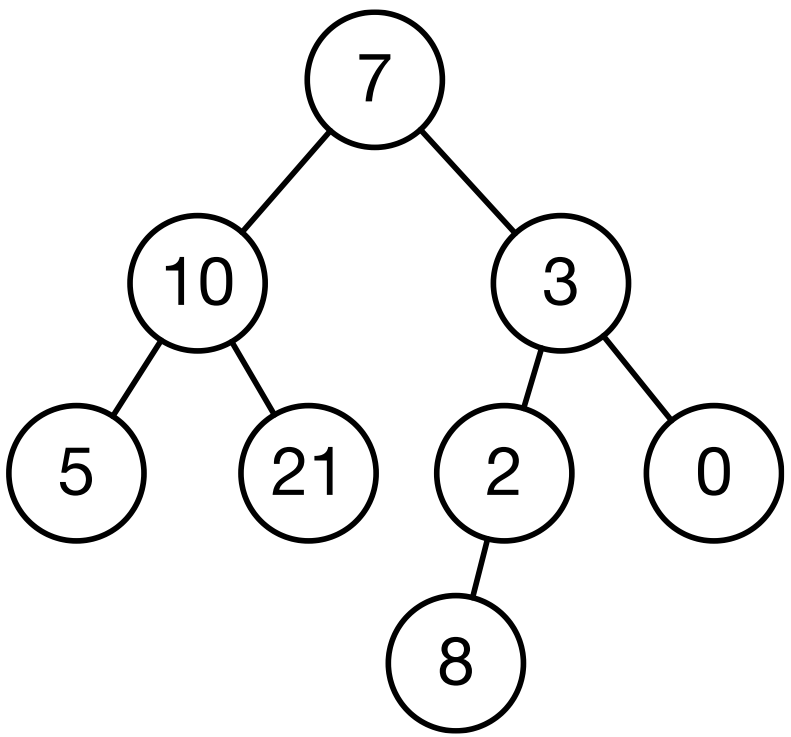
- **Dynamische Datenstruktur**: Bäume ermöglichen das Einfügen und Löschen von Daten effizient, ohne die gesamte Datenstruktur umzuordnen.
- Die Daten müssen nach Einfügen oder Löschen eines Elements nicht nochmal neu sortiert werden.



- **Blatt**: Ein Knoten ohne Nachfolger. Blätter sind die Enden des Baums.
- **Knoten**: Ein Element des Baums, das Daten enthält und mit anderen Knoten verbunden ist.
- **Wurzel**: Der oberste Knoten des Baums.
- **Höhe**: Die Länge des längsten Pfades von der Wurzel zu einem Blatt.
- **Size**: Die Gesamtzahl der Knoten im gesamten Baum.

Bäume Traversieren

In-Order: Linker Teilbaum - Wurzel - Rechter Teilbaum
Pre-Order: Wurzel - Linker Teilbaum - Rechter Teilbaum
Post-Order: Linker Teilbaum - Rechter Teilbaum - Wurzel



In-Order:

5	10	21	**7**	8	2	3	0

Pre-Order:

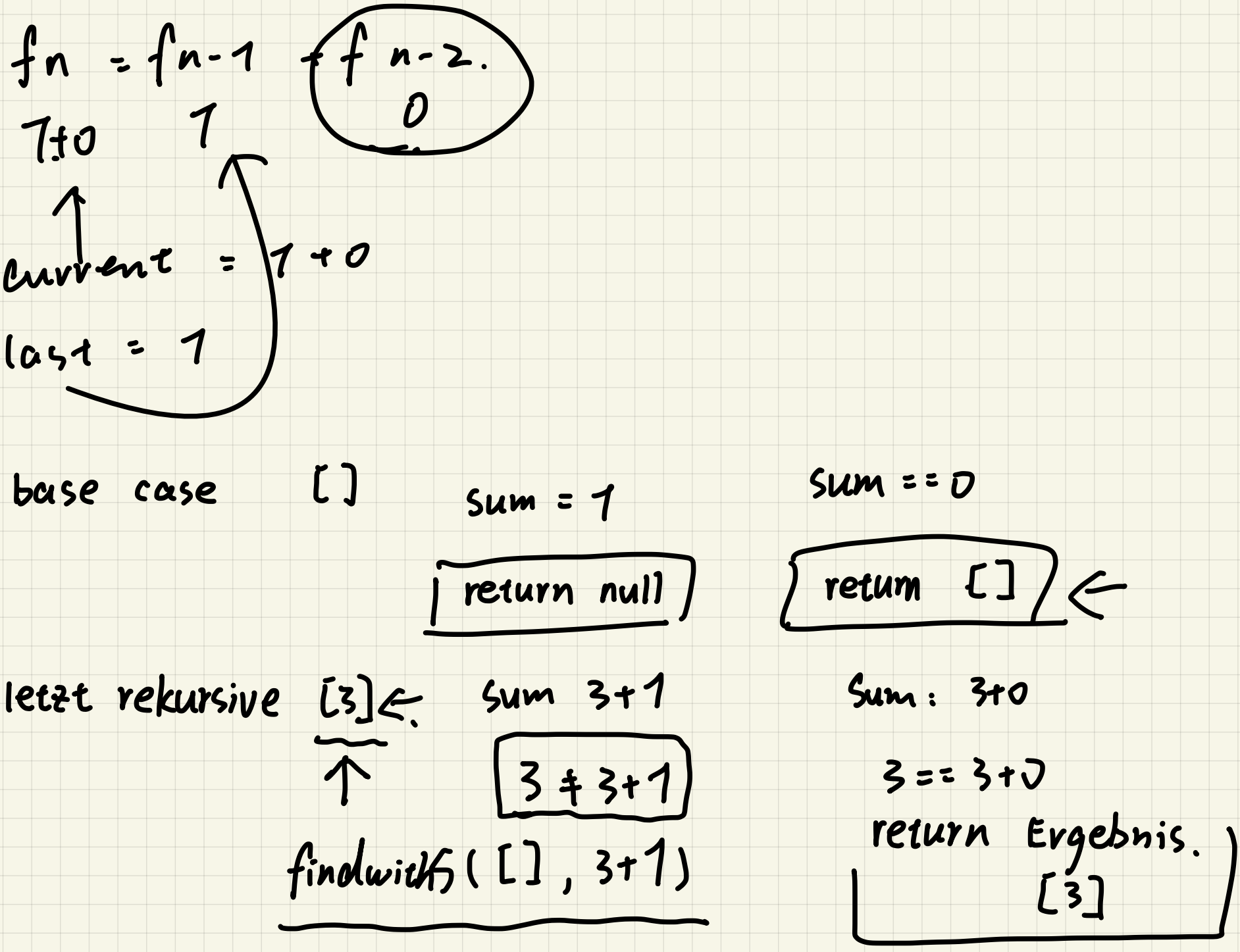
7	10	5	21	3	2	8	0

Post-Order:

5	21	10	8	2	0	3	**7**

W08P04 - Rekursive Datenstrukturen

Bearbeite nun die Aufgabe W08P04 auf Artemis. Die Aufgabe enthält viele Methoden und muss nicht innerhalb des Tutoriums fertig implementiert werden. Der Rest ist für Übung da.


fn = fn-1 + fn-2.
1+0 1 0
current = 1+0
last = 1
base case []
sum = 1
return null
sum == 0
return []
letzt rekursive [3]
sum 3+1
3 + 3+1
findwithSum([], 3+1)
Sum: 3+0
3 == 3+0
return Ergebnis.
[3]