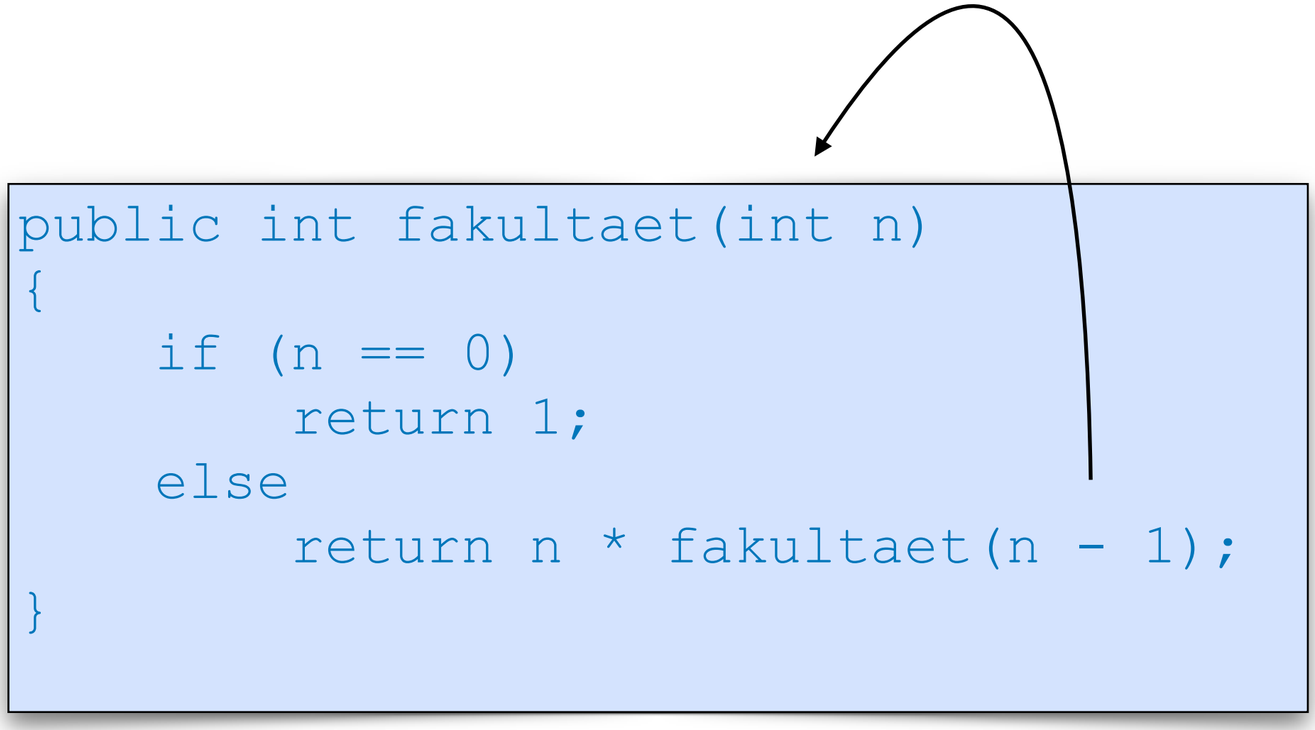
5. Sortieren und Suchen

5.8 Rekursionen

5.8 Rekursion

```
public int fakultaet(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakultaet(n - 1);
}
```

5.8 Rekursion



```
public int fakultaet(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakultaet(n - 1);
}
```

Die Methode ruft sich selbst auf = Rekursion

5.8 Rekursion

```
fakultaet(4)
n != 0 → fakultaet(3)
```

```
public int fakultaet(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakultaet(n - 1);
}
```

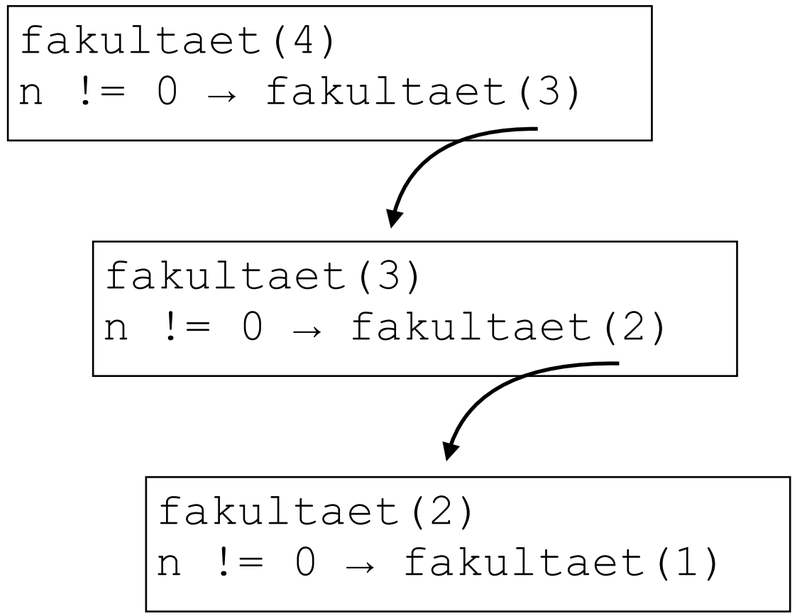
5.8 Rekursion

```
fakultaet(4)
n != 0 → fakultaet(3)
```

```
fakultaet(3)
n != 0 → fakultaet(2)
```

```
public int fakultaet(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakultaet(n - 1);
}
```

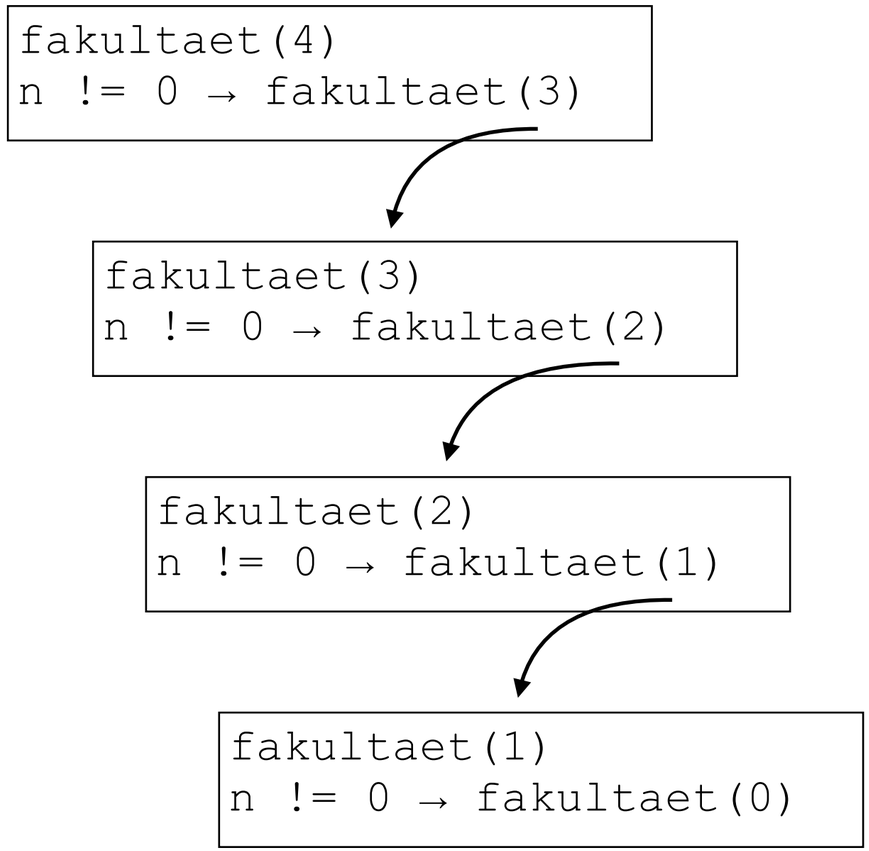
5.8 Rekursion




```
public int fakultaet(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakultaet(n - 1);
}
```

rekursiver
Abstieg

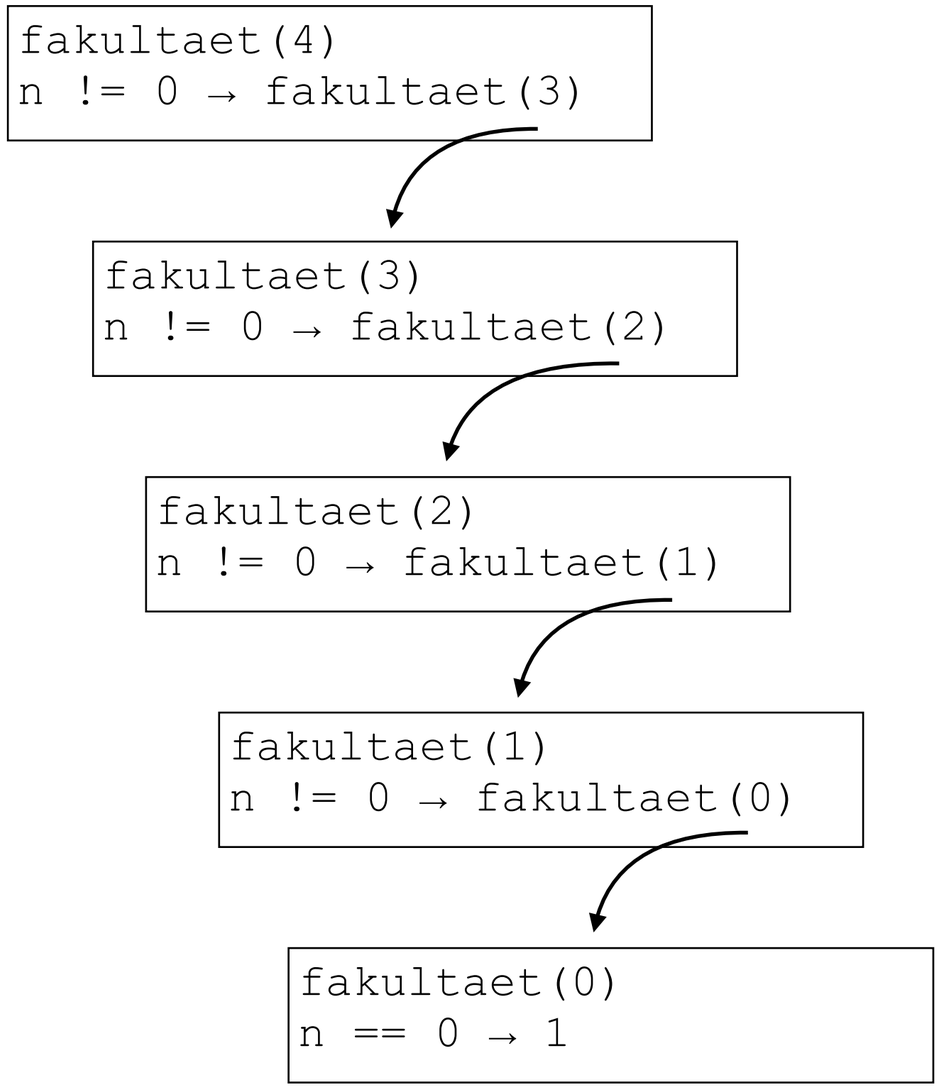
5.8 Rekursion




```
public int fakultaet(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakultaet(n - 1);
}
```

rekursiver
Abstieg

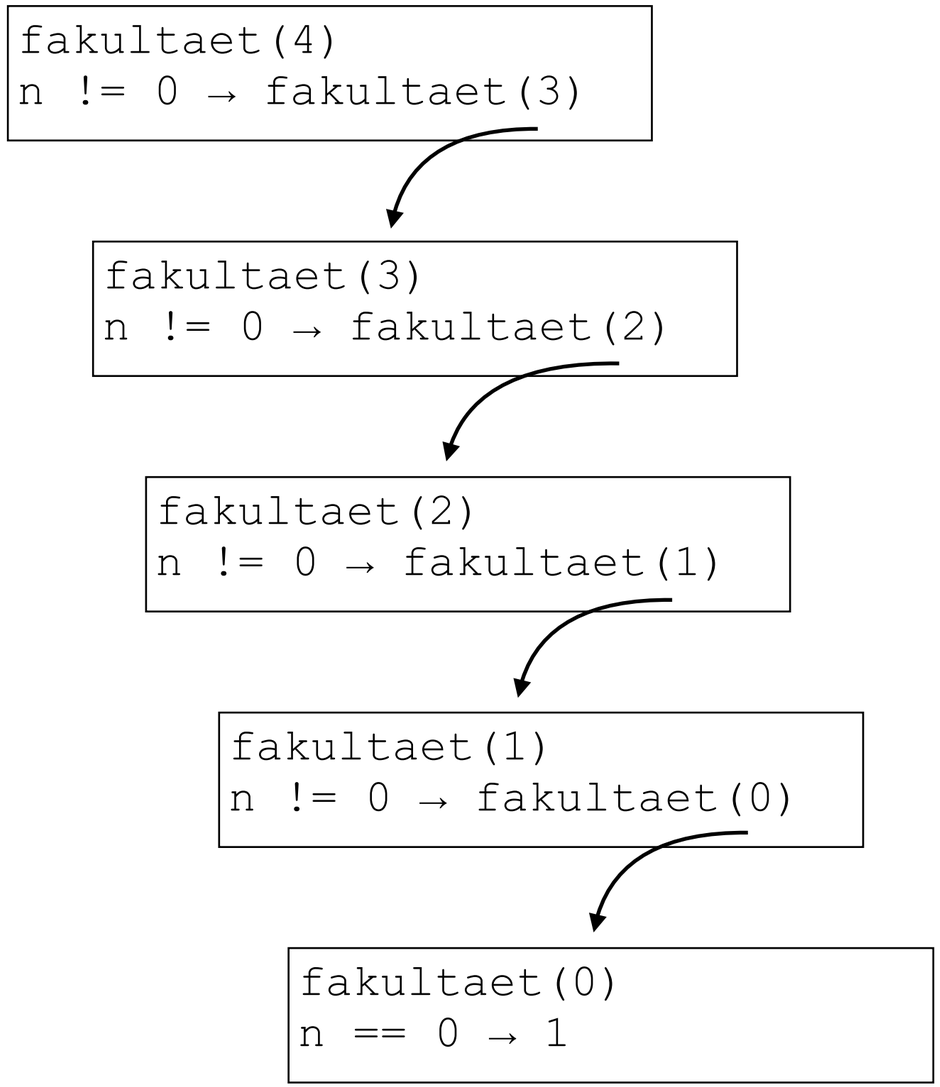
5.8 Rekursion



rekursiver
Abstieg

```
public int fakultaet(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakultaet(n - 1);
}
```

5.8 Rekursion



return 1

```
public int fakultaet(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakultaet(n - 1);
}
```

im "Keller" angekommen, da Abbruchbedingung n == 0 erfüllt ist.

5.8 Rekursion

```
fakultaet(4)
n != 0 → fakultaet(3)
```

```
fakultaet(3)
n != 0 → fakultaet(2)
```

```
fakultaet(2)
n != 0 → fakultaet(1)
```

```
fakultaet(1)
n != 0 → fakultaet(0)
```

```
fakultaet(0)
n == 0 → 1
```

return 1*1 = 1

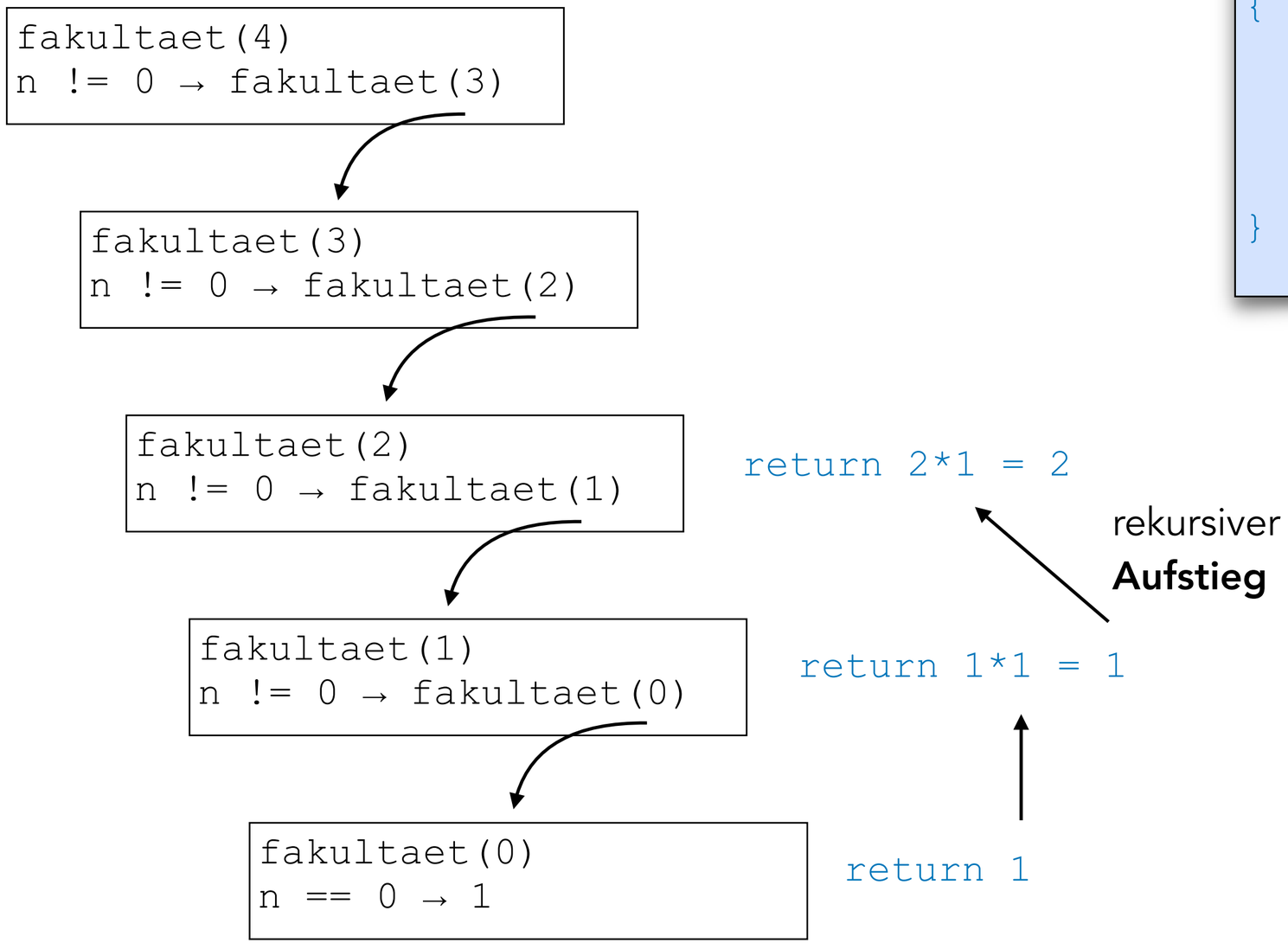
rekursiver **Aufstieg**

return 1

```
public int fakultaet(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakultaet(n - 1);
}
```

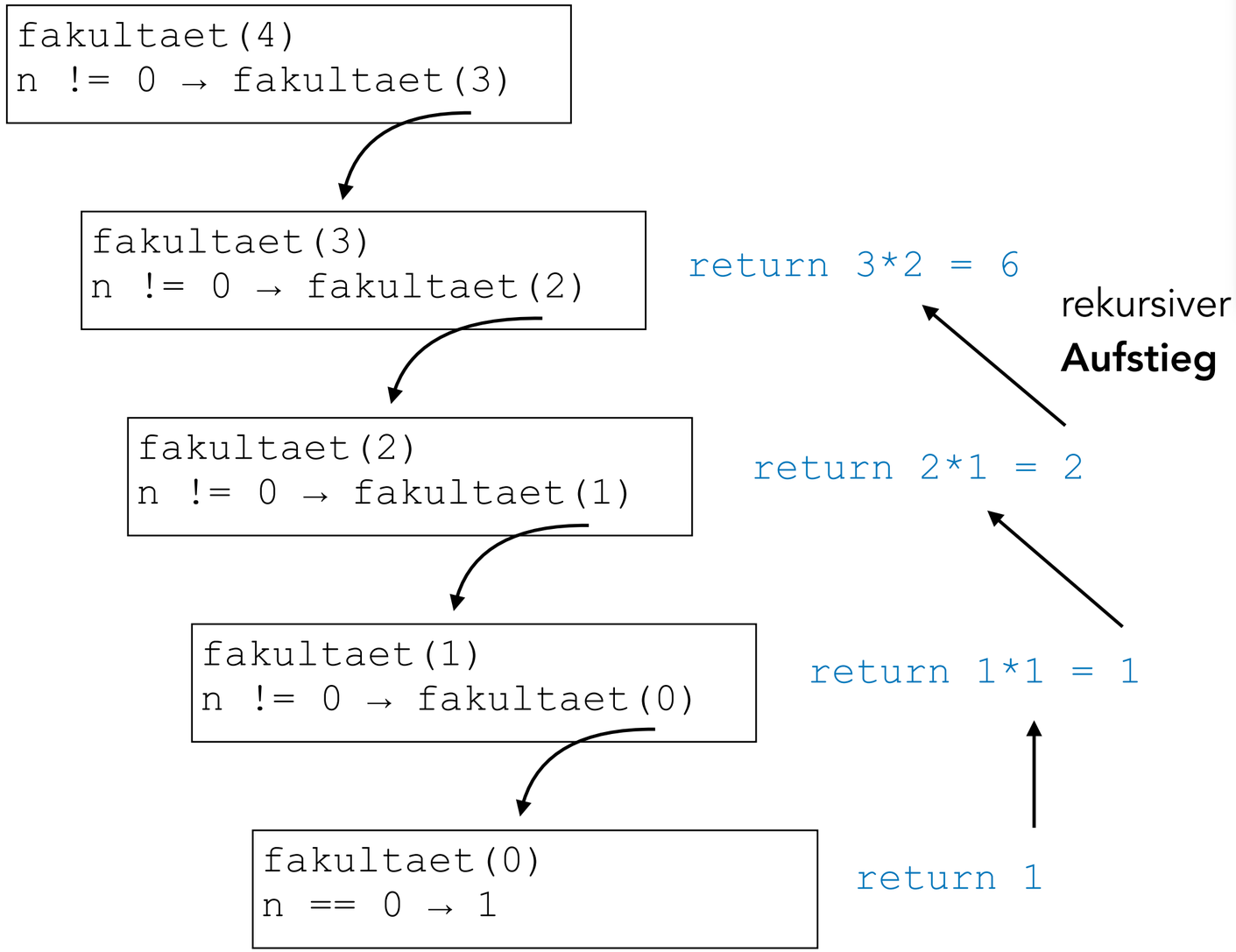
5.8 Rekursion

```
public int fakultaet(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakultaet(n - 1);
}
```

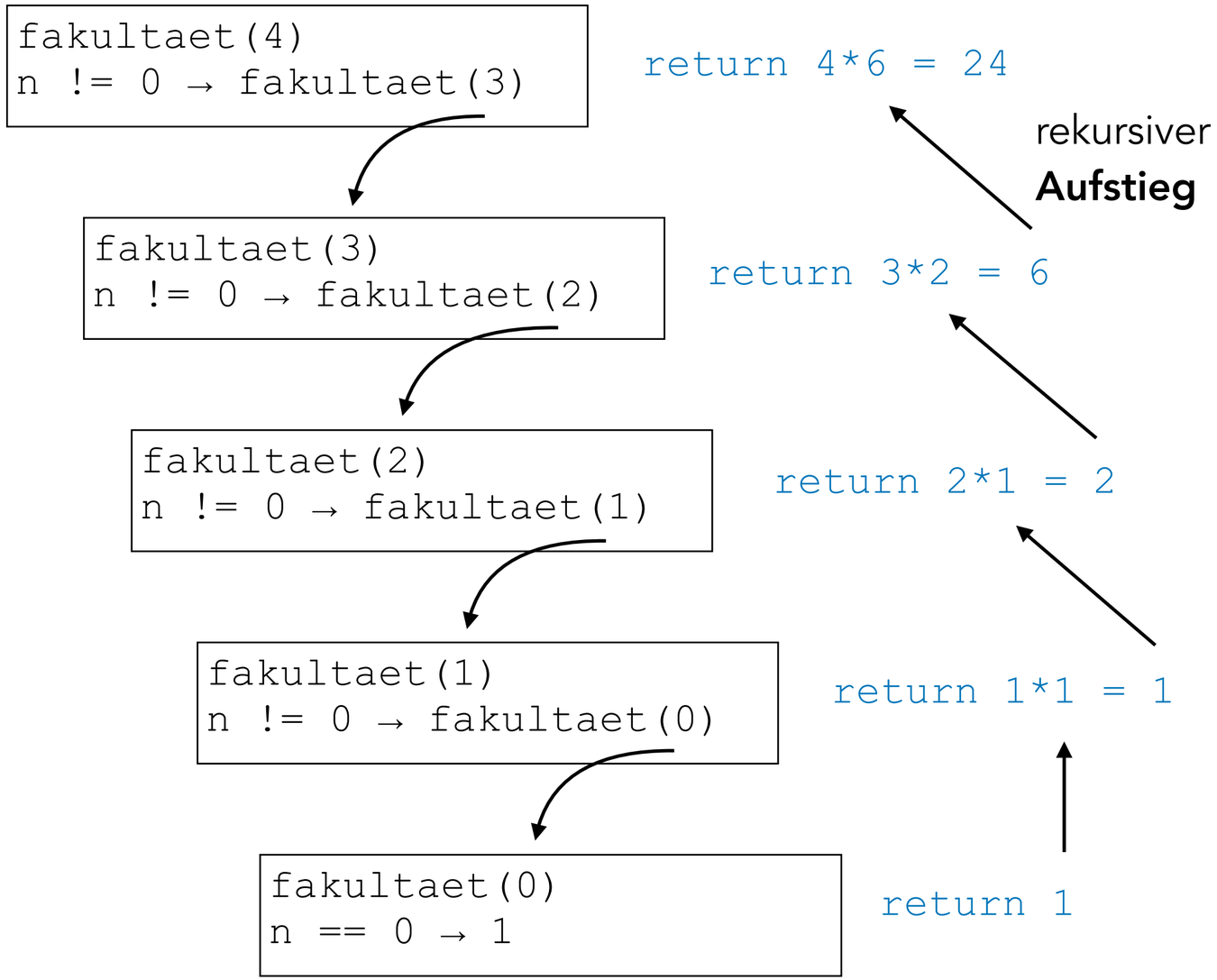


5.8 Rekursion

```
public int fakultaet(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakultaet(n - 1);
}
```



5.8 Rekursion



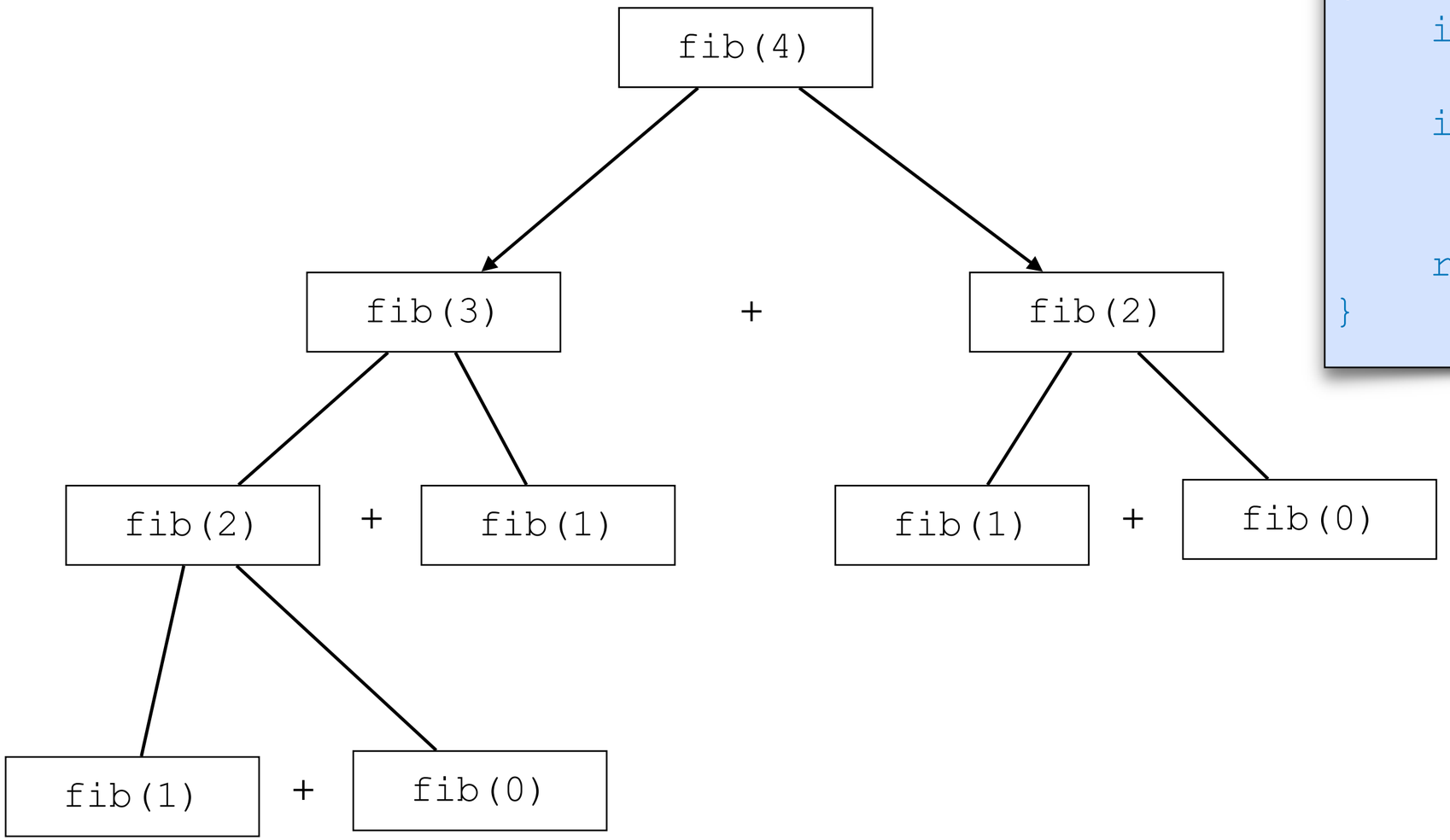

```
public int fakultaet(int n)
{
    if (n == 0)
        return 1;
    else
        return n * fakultaet(n - 1);
}
```

5.8 Rekursion: Fibonacci-Zahlen

```
public static int fib(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;

    return fib(n-1) + fib(n-2);
}
```

5.8 Rekursion: Fibonacci-Zahlen



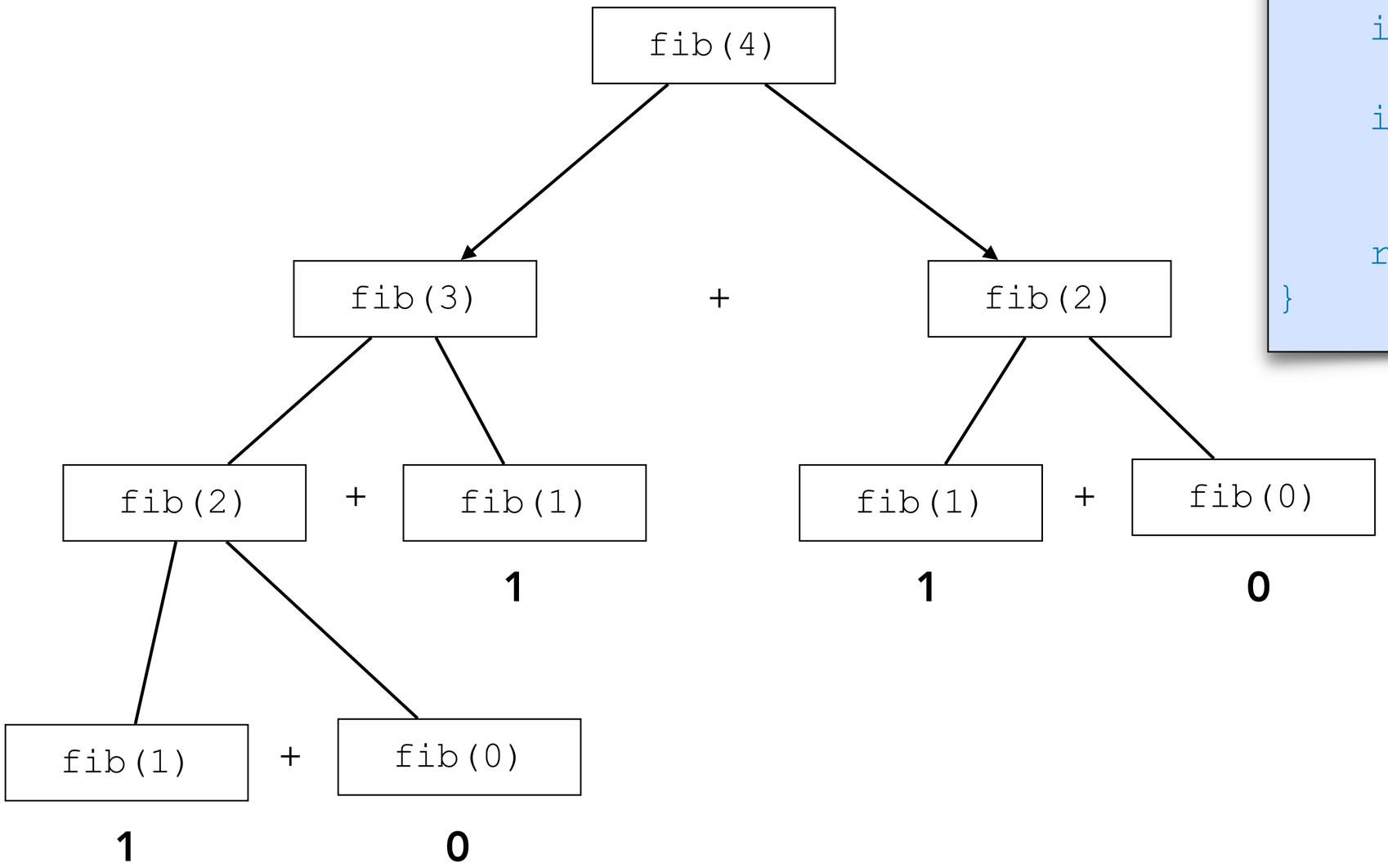

```
public static int fib(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;

    return fib(n-1) + fib(n-2);
}
```

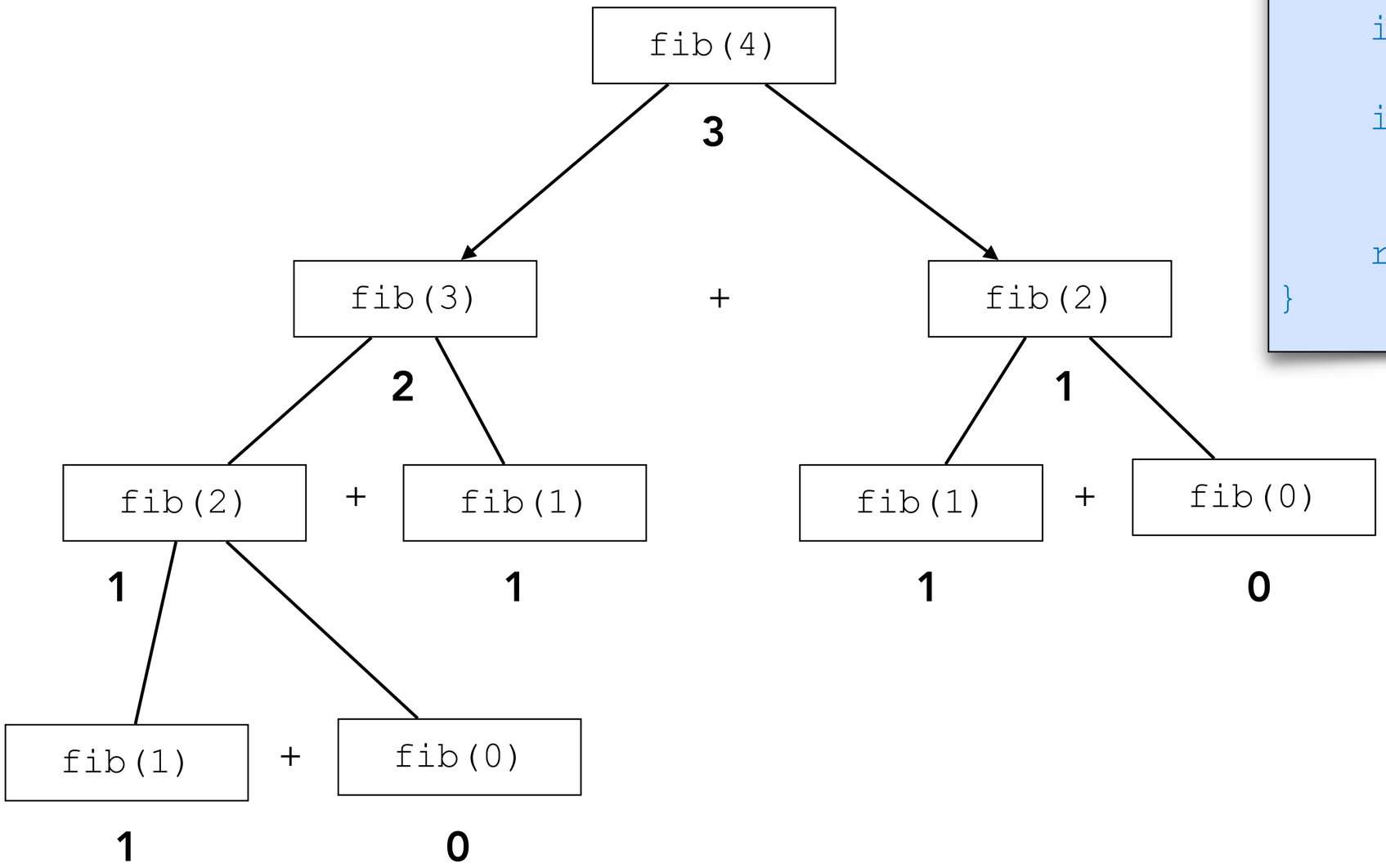
5.8 Rekursion: Fibonacci-Zahlen

```
public static int fib(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;

    return fib(n-1) + fib(n-2);
}
```



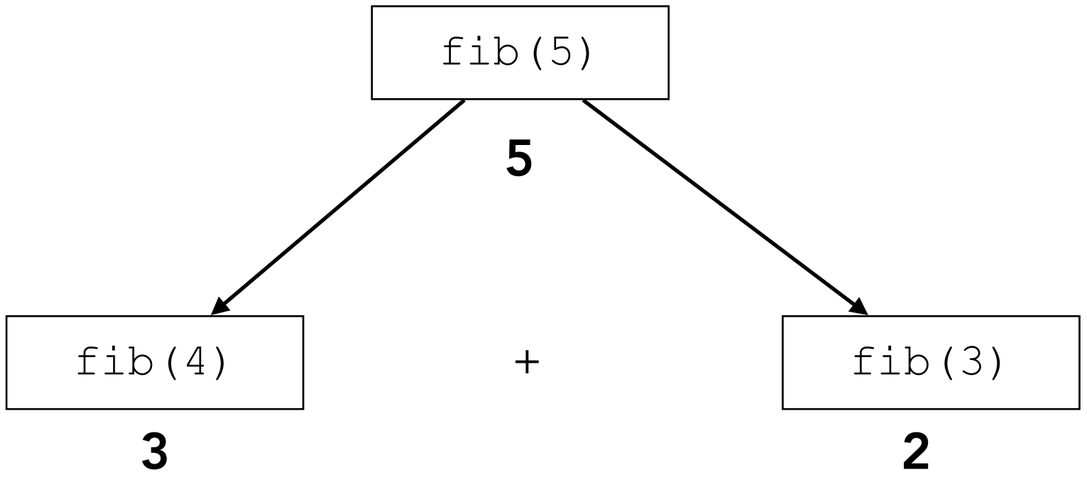
5.8 Rekursion: Fibonacci-Zahlen




```
public int fib(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;

    return fib(n-1) + fib(n-2);
}
```

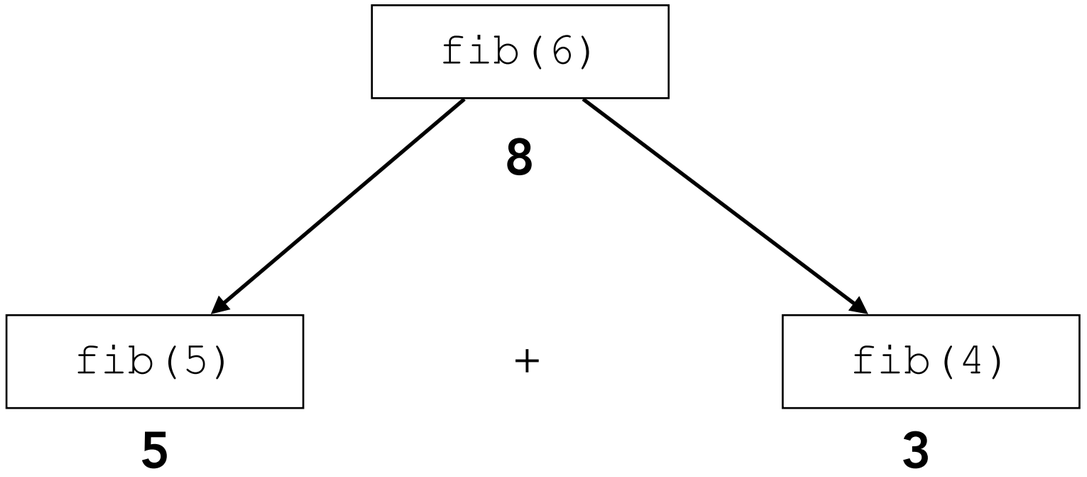
5.8 Rekursion: Fibonacci-Zahlen




```
public int fib(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;

    return fib(n-1) + fib(n-2);
}
```

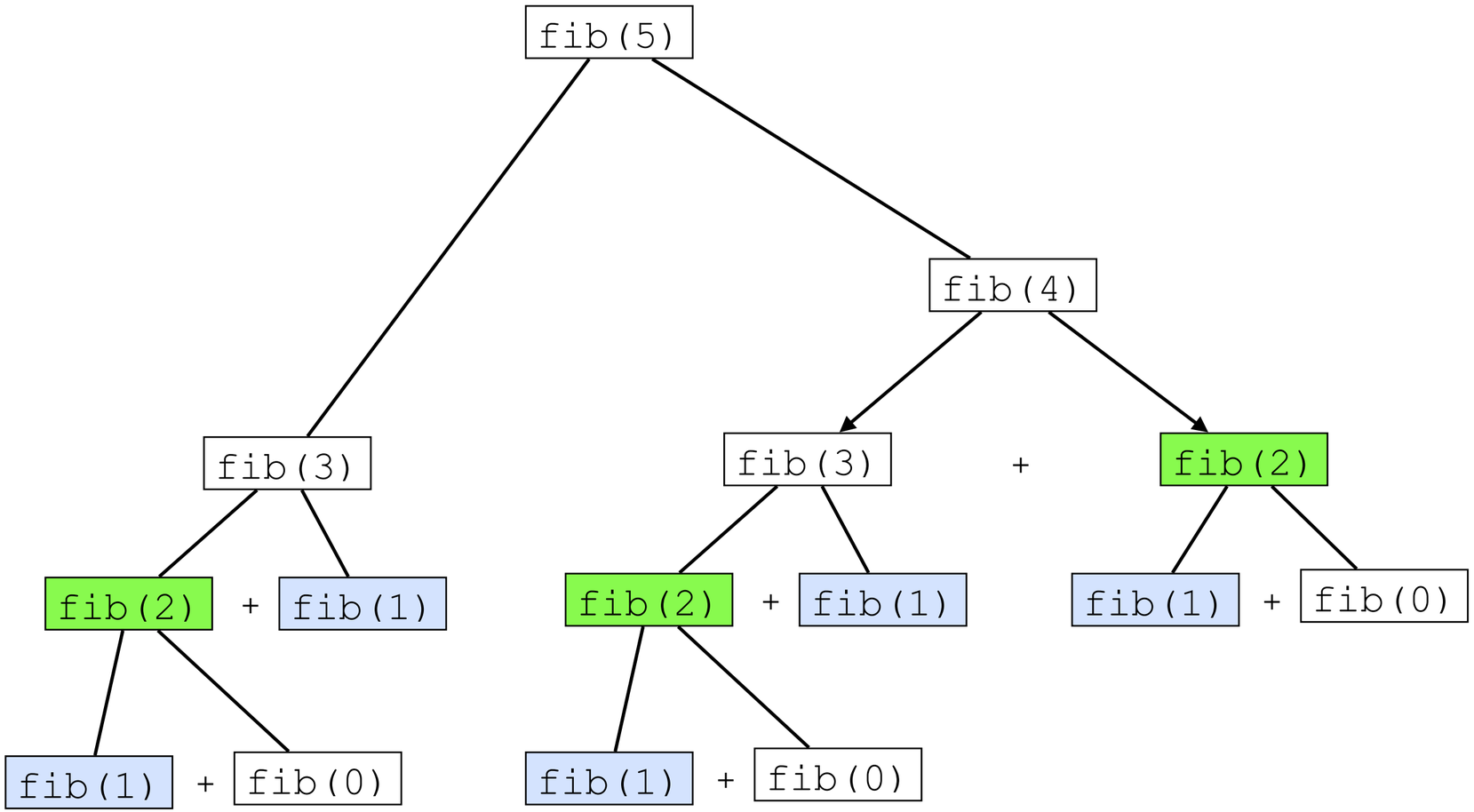
5.8 Rekursion: Fibonacci-Zahlen




```
public int fib(int n)
{
    if (n < 2) return n;
    return fib(n-1) + fib(n-2);
}
```

1 - 1 - 2 - 3 - 5 - 8 - 13 - 21 - 24 - 55 - 89 - 144 - 233 - 377 - ...

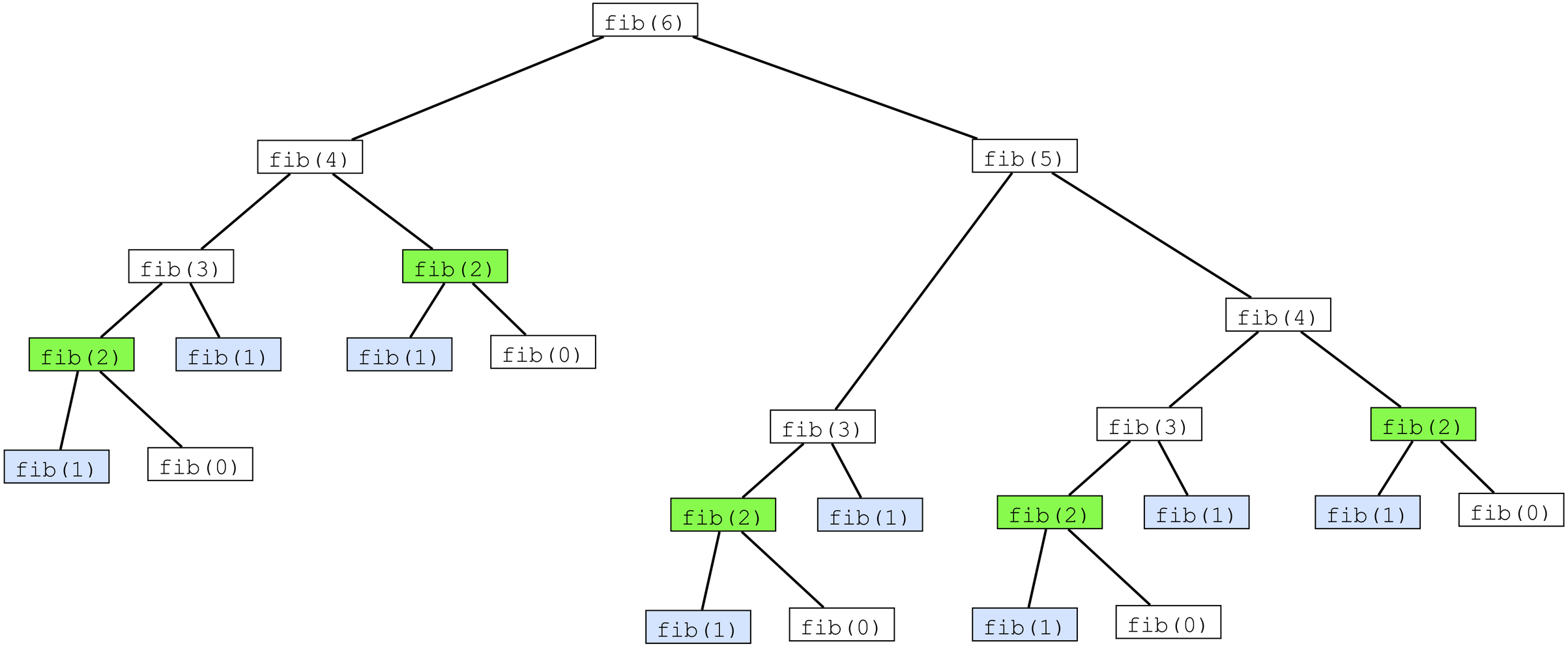
5.8 Rekursion: Fibonacci-Zahlen



Die rekursive Implementierung der Methode hat durchaus Nachteile!

Welche?

5.8 Rekursion: Fibonacci-Zahlen



5.8 Rekursion: mergeSort()

Wo ist hier die Abbruchbedingung?

```java
public void mergeSort(int[] zahlen)
{
    if (zahlen.length > 1)
    {
        int mitte = zahlen.length / 2;

        // linkes und rechtes Teilarray erzeugen
        int[] links  = new int[mitte];
        int[] rechts = new int[zahlen.length - mitte];

        // Elemente aufteilen
        for (int i = 0; i < mitte; i++)
            links[i] = zahlen[i];
        for (int i = mitte; i < zahlen.length; i++)
            rechts[i - mitte] = zahlen[i];

        // rekursiv sortieren
        mergeSort(links);
        mergeSort(rechts);

        // zusammenführen
        merge(zahlen,links, rechts);
    }
}
```

5.8 Rekursion: mergeSort()

Abbruchbedingung!

```java
public void mergeSort(int[] zahlen)
{
    if (zahlen.length > 1)
    {
        int mitte = zahlen.length / 2;

        // linkes und rechtes Teilarray erzeugen
        int[] links  = new int[mitte];
        int[] rechts = new int[zahlen.length - mitte];

        // Elemente aufteilen
        for (int i = 0; i < mitte; i++)
            links[i] = zahlen[i];
        for (int i = mitte; i < zahlen.length; i++)
            rechts[i - mitte] = zahlen[i];

        // rekursiv sortieren
        mergeSort(links);
        mergeSort(rechts);

        // zusammenführen
        merge(zahlen,links, rechts);
    }
}
```

5.8 Rekursion: mergeSort()

```java
public void mergeSort(int[] zahlen)
{
    if (zahlen.length > 1)
    {
        int mitte = zahlen.length / 2;

        // linkes und rechtes Teilarray erzeugen
        int[] links  = new int[mitte];
        int[] rechts = new int[zahlen.length - mitte];

        // Elemente aufteilen
        for (int i = 0; i < mitte; i++)
            links[i] = zahlen[i];
        for (int i = mitte; i < zahlen.length; i++)
            rechts[i - mitte] = zahlen[i];

        // rekursiv sortieren
        mergeSort(links);
        mergeSort(rechts);

        // zusammenführen
        merge(zahlen,links, rechts);
    }
}
```

```
mergeSort([38, 27, 43, 3])
│
├── Teilschritt: Aufteilung in zwei Hälften
│       links:  [38, 27]
│       rechts: [43, 3]
│
├── mergeSort([38, 27])
│   │
│   ├── Aufteilung
│   │       links:  [38]
│   │       rechts: [27]
│   │
│   ├── mergeSort([38])   → Länge 1 → Abbruch
│   ├── mergeSort([27])   → Länge 1 → Abbruch
│   │
│   └── merge([38, 27])   → Ergebnis: [27, 38]
│
└── mergeSort([43, 3])
    │
    ├── Aufteilung
    │       links:  [43]
    │       rechts: [3]
    │
    ├── mergeSort([43])   → Länge 1 → Abbruch
    ├── mergeSort([3])    → Länge 1 → Abbruch
    │
    └── merge([43, 3])    → Ergebnis: [3, 43]
```

Beispiel erzeugt von ChatGPT