Power of Recursion and Backtracking

Michael Inden

https://github.com/Michaeli71/PowerOfRecursionAndBacktracking

Agenda

Workshop Contents

- **PART 1: Schnelleinstieg Rekursion**
  - Rekursion im Überblick
  - Grafische Gebilde
  - Zustand abbilden
- **PART 2: Suchen und Sortieren**
  - Binärsuche
  - MergeSort
  - QuickSort

Workshop Contents

- **PART 3: 2-D**
  - Flood Fill
  - Komplexere Figuren
    - Schneeflocke
    - Hilbert
    - Sierpinski
    - …
- **PART 4: Fallstricke und Abhilfen**
  - Fallstricke
  - Abhilfe: Memoization

Workshop Contents

- **PART 5: Backtracking**
  - Weg aus Labyrinth
  - Sudoku Solver
  - n-Damenproblem
  - Knights Tour

PART 1:
Schnelleinstieg Rekursion

Rekursion im Überblick

Rekursion

- **Rekursion ist eine Vorgehensweise, bei der eine Methode sich selbst aufruft.**

```
static void printCountDownRec(int value)
{
    // rekursiver Abbruch
    if (value < 0)
    {
        System.out.println("FINISH");
        return;
    }

    System.out.println(value);
    // rekursiver Abstieg
    printCountDownRec(value - 1);
}

public static void main(String[] args)
{
    printCountDownRec(5);
}
```

```
5
4
3
2
1
0
FINISH
```

Rekursion

- Klingt merkwürdig, aber ist manchmal sehr elegant
- komplizierte Probleme in einfachere Teilprobleme zerlegen, die leichter zu lösen sind (manchmal ist das tatsächlich bereits nur ein um eins reduzierter Wert eines Parameters)

- Beispiele aus der Mathematik: Fakultät und Fibonacci-Zahlen

$$n! = \begin{cases} 1, & n = 0,\ n = 1 \\ n \cdot (n-1)!, & \forall n > 1 \end{cases}$$

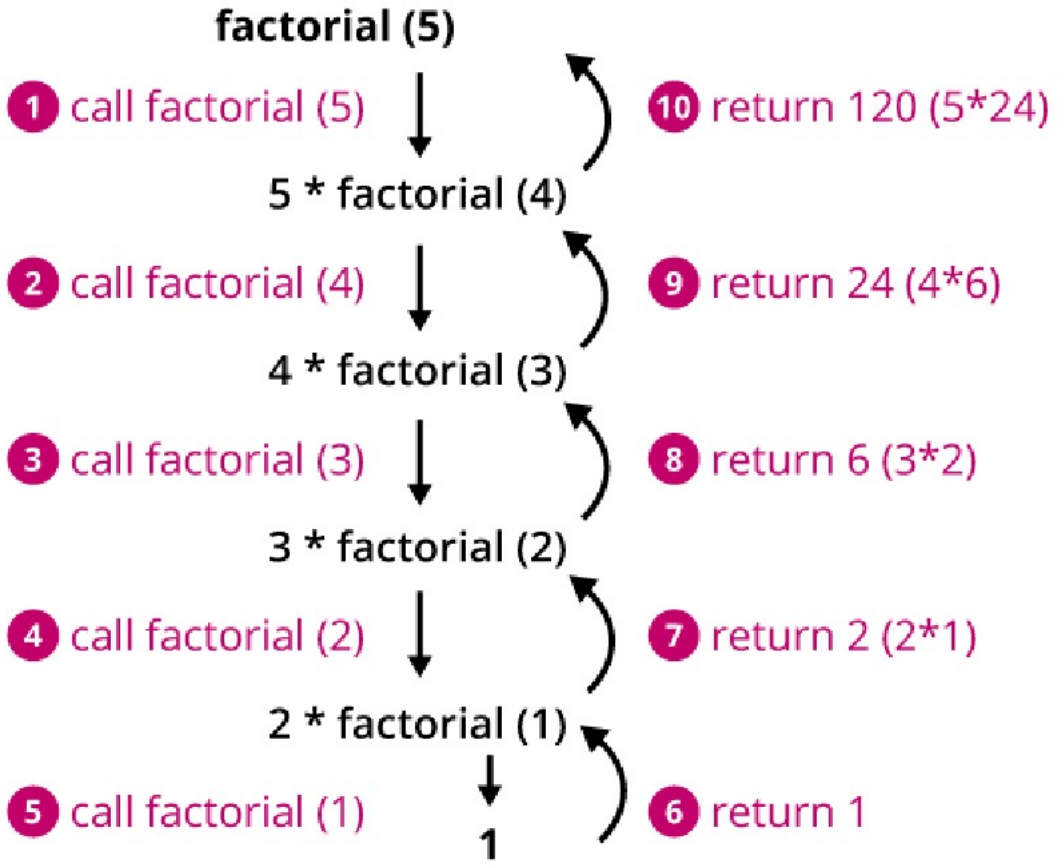
$$fib(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ fib(n-1) + fib(n-2), & \forall n > 2 \end{cases}$$

Rekursion

- **Beispiele aus der Mathematik**

$$n! = \begin{cases} 1, & n = 0,\ n = 1 \\ n \cdot (n-1)!, & \forall n > 1 \end{cases}$$

```
static int factorial(final int n)
{
    if (n == 1)
        return 1;

    return n * facatorial(n-1);
}
```



Rekursion

- **Beispiele aus der Mathematik**

$$fib(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ fib(n-1) + fib(n-2), & \forall n > 2 \end{cases}$$

```
static long fib(final int n)
{
    if (n <= 0)
        throw new IllegalArgumentException("n must be positive and >= 1");

    // rekursiver Abbruch
    if (n == 1 || n == 2)
        return 1;

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

- **Beispiele aus der Algorithmik: String umdrehen**

```
static String reverseString(final String input)
{
    if (input.length() <= 1)
        return input;

    final char firstChar = input.charAt(0);
    final String remaining = input.substring(1);

    return reverseString(remaining) + firstChar;
}


static String reverseStringShort(final String input)
{
    return input.length() <= 1 ?
           input : reverseStringShort(input.substring(1)) + input.charAt(0);
}
```

Rekursion

- **Don’t forget to test ☺**

```
@ParameterizedTest(name = "reverseString({0}) => {1}")
@CsvSource({ "A, A", "ABC, CBA", "abcdefghi, ihgfedcba" })
public void reverseString(String input, String expected)
{
    String result = Ex04_ReverseString.reverseString(input);

    assertEquals(expected, result);
}

@ParameterizedTest(name = "reverseStringShort({0}) => {1}")
@CsvSource({ "A, A", "ABC, CBA", "abcdefghi, ihgfedcba" })
public void reverseStringShort(String input, String expected)
{
    String result = Ex04_ReverseString.reverseStringShort(input);

    assertEquals(expected, result);
}
```

Grafische Gebilde

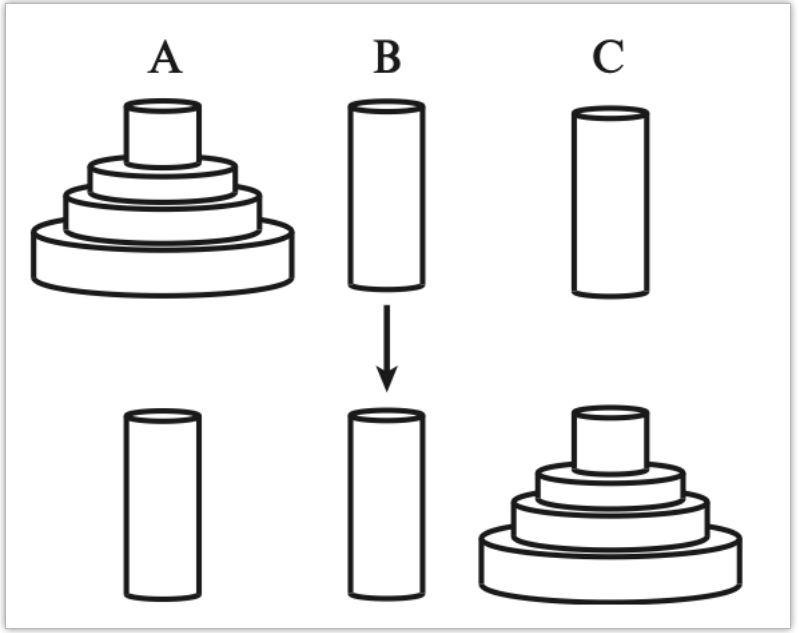
- **Beispiele**

```
static void fractalGenerator(final int n)
{
    if (n == 1)
    {
        System.out.println("-");
    }
    else
    {
        fractalGenerator(n - 1);
        System.out.println("=".repeat(n));
        fractalGenerator(n - 1);
    }
}
```

```
-
==
-
===
-
==
-
====
-
==
-
===
-
==
-
```

Rekursion

- Beim Türme-von-Hanoi-Problem gibt es drei Türme oder Stäbe A, B und C. Zu Beginn sind mehrere gelochte Scheiben der Größe nach auf Stab A platziert, die größte zuunterst.
- Ziel ist es nun, den gesamten Stapel, also alle Scheiben, von A nach C zu bewegen. Dabei darf immer nur eine Scheibe nach der anderen bewegt werden und niemals eine kleinere Scheibe unter einer größeren liegen.
- Deswegen benötigt man den Hilfsstab B.



Rekursion

- **Türme von Hanoi**

```
static void moveTower(int n, char source, char helper,
                      char destination)
{
    if (n == 1)
        System.out.println(source + " -> " + destination);
    else
    {
        // bewege um ein reduierten von Quelle auf Hilfstab,
        // (Ziel wird so zum neuen Hilfsstab)
        moveTower(n-1, source, destination, helper);

        // bewege die grösste Scheibe
        moveTower(1, source, helper, destination);

        // bewege um ein reduzierten Turm von Hilfsstab aud Ziel
        moveTower(n-1, helper, source, destination);
    }
}
```

```
Tower Of Hanoi 2
   A          B          C
   |          |          |
  #|#         |          |
 ##|##        |          |
-------    -------    -------
Moving slice 1: Tower [A] -> Tower [B]
   A          B          C
   |          |          |
   |          |          |
 ##|##       #|#         |
-------    -------    -------
Moving slice 2: Tower [A] -> Tower [C]
   A          B          C
   |          |          |
   |          |          |
   |         #|#       ##|##
-------    -------    -------
Moving slice 1: Tower [B] -> Tower [C]
   A          B          C
   |          |          |
   |          |         #|#
   |          |        ##|##
-------    -------    -------
```

DEMO

TowersOfHanoiGraphics.java

Zustand abbilden

- **Beispiel: Minimum eines Arrays rekursiv ermitteln**
- **Wie kann man Informationen mitliefern? => Parameter + Hilfsmethode**

```
static int min(final int[] values)
{
    return min(values, 0, Integer.MAX_VALUE);
}

static int min(final int[] values, final int pos, int currentMin)
{
    if (pos >= values.length)
        return currentMin;

    final int current = values[pos];
    if (current < currentMin)
        currentMin = current;

    return min(values, pos + 1, currentMin);
}
```

Rekursion

- **Beispiel: Maximum einer Liste rekursiv ermitteln**
- **Wie kann man Informationen mitliefern? => Parameter + Hilfsmethode**

```
static int max(final List<Integer> values)
{
    return max(values, Integer.MIN_VALUE);
}

static int max(final List<Integer> values, int currentMax)
{
    if (values.size() == 0)
        return currentMax;

    final int current = values.get(0);
    if (current > currentMax)
        currentMax = current;

    return max(values.subList(1, values.size()), currentMax);
}
```

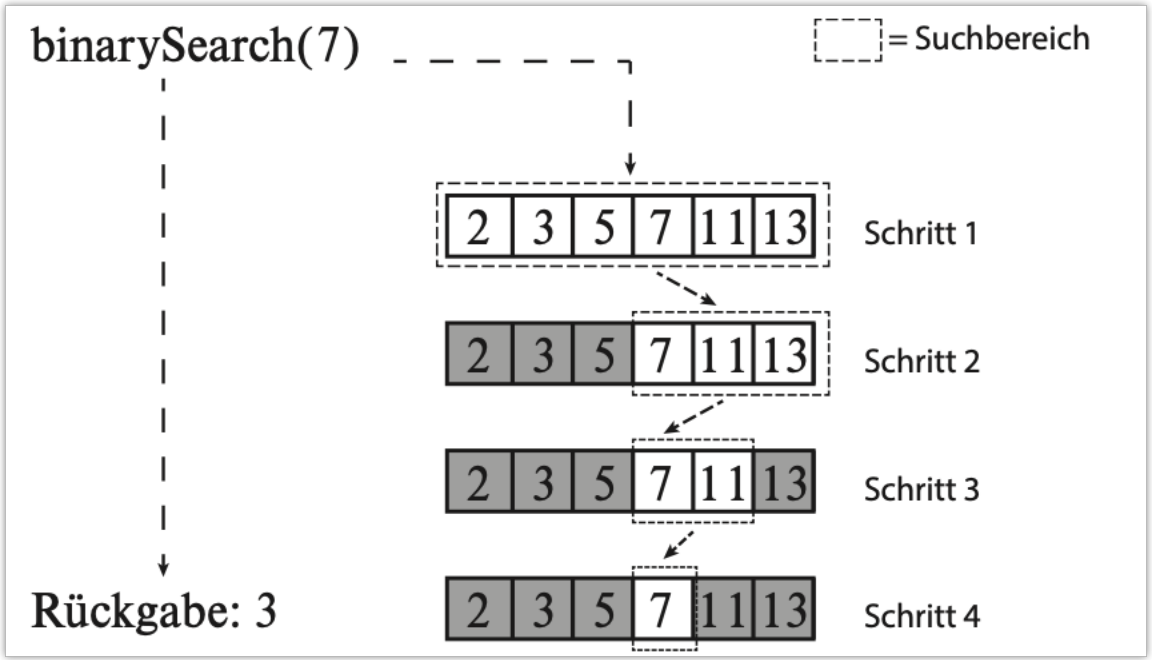
Exercises Part 1

https://github.com/Michaeli71/PowerOfRecursionAndBacktracking

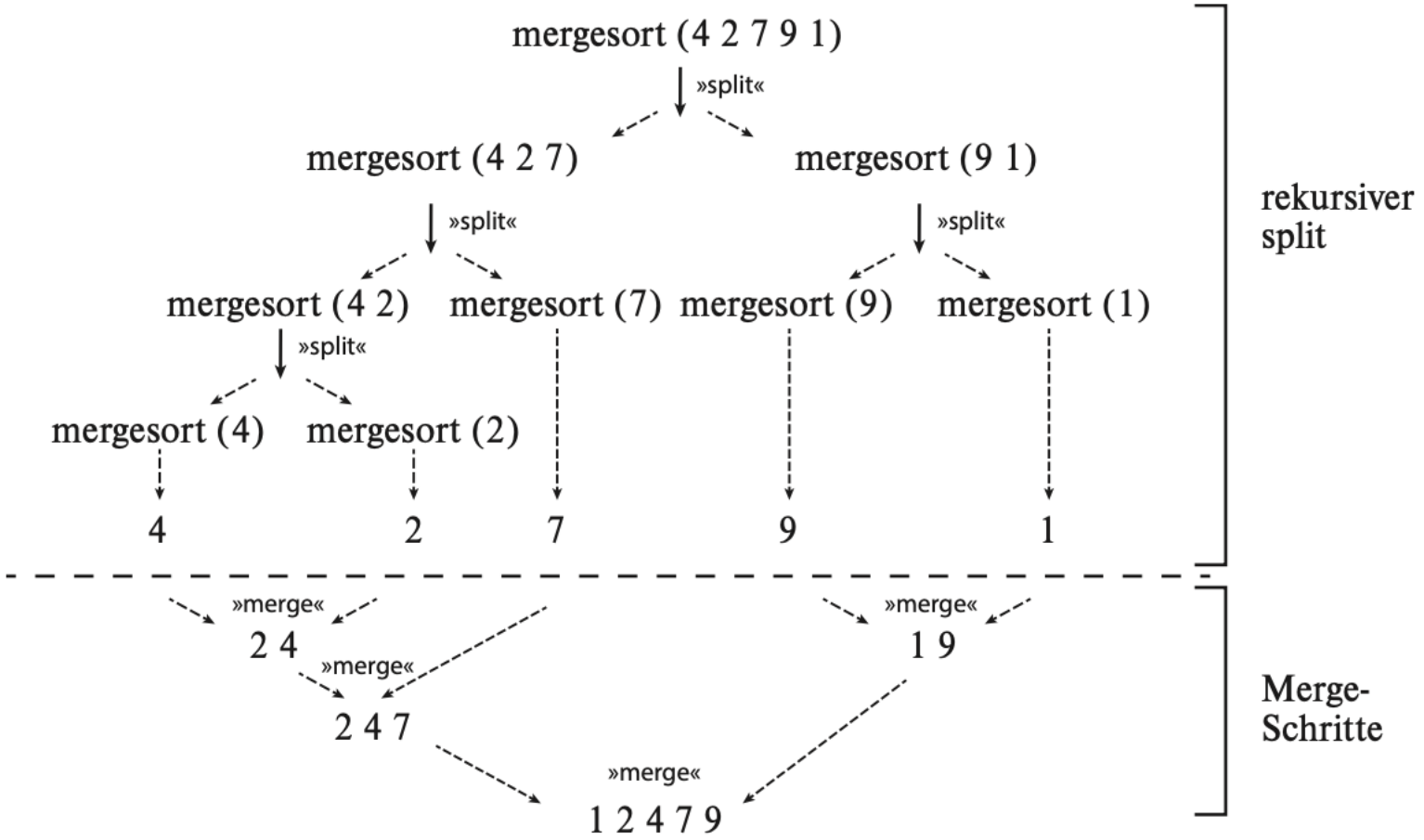
PART 2:
Suchen und Sortieren

Rekursion

- Binärsuche arbeitet auf sortierten Datenbeständen
- Effiziente Suche in logarithmischer Zeit
- Algorithmus: die jeweils zu verarbeitenden Bereiche werden halbiert und danach wird im passenden Teilstück weitergesucht.



- **Merge Sort**


mergesort (4 2 7 9 1)
»split«
mergesort (4 2 7)
mergesort (9 1)
»split«
»split«
mergesort (4 2)
mergesort (7)
mergesort (9)
mergesort (1)
»split«
mergesort (4)
mergesort (2)
4
2
7
9
1
»merge«
2 4
»merge«
1 9
»merge«
2 4 7
»merge«
1 2 4 7 9
rekursiver split
Merge-Schritte

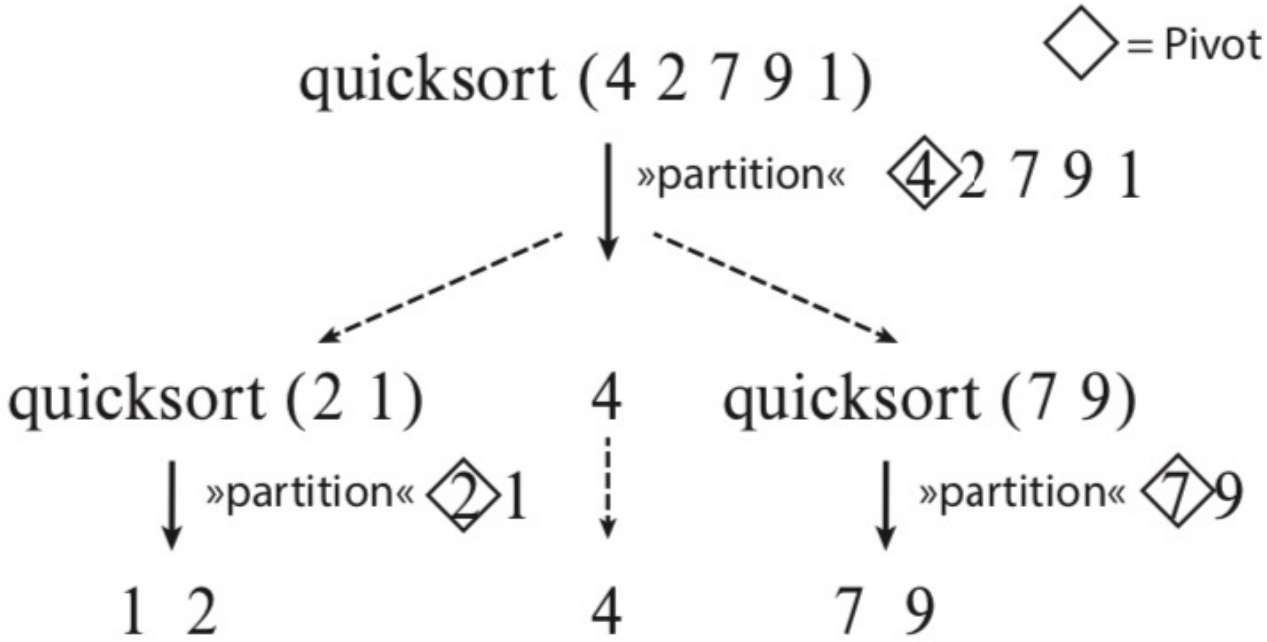
Quick Sort

Implementieren Sie eine Sortierung von Zahlen mithilfe des Quick Sort Verfahrens.

Eingabe	Resultat
[5, 2, 7, 1, 4, 3, 6, 8]	[1, 2, 3, 4, 5, 6, 7, 8]
[5, 2, 7, 9, 6, 3, 1, 4, 8]	[1, 2, 3, 4, 5, 6, 7, 8, 9]
[5, 2, 7, 9, 6, 3, 1, 4, 2, 3, 8]	[1, 2, 2, 3, 3, 4, 5, 6, 7, 8, 9]

Lösungsstrategie / Algorithmus

- **Quick Sort** basiert auf einem **Divide-and-Conquer**-Ansatz
- **Zerteilt** den zu sortierende Datenbestand **in** immer **kleinere Teile**.
- Spezielles Element (***Pivot***) **legt** die **Unterteilung fest**.
- Alle **Elemente** des Teilbereichs mit Wert **kleiner oder gleich** werden **links** bzw. die **größeren** werden **rechts** vom Pivot **anordnet**.
- **Rekursive Wiederholung** bis die Teilbereiche nur noch einelementig sind.



```java
static List<Integer> quickSort(final List<Integer> values)
{
    if (values.size() <= 1)
        return values;

    Integer pivot = values.get(0);
    var belowOrEquals = values.stream().skip(1).filter(cur -> cur <= pivot).toList();
    var aboves = values.stream().skip(1).filter(cur -> cur > pivot).toList();

    var sortedLowersPart = quickSort(belowOrEquals);
    var sortedUppersPart = quickSort(aboves);

    final List<Integer> result = new ArrayList<>();
    result.addAll(sortedLowersPart);
    result.add(pivot);
    result.addAll(sortedUppersPart);

    return result;
}
```

```
static List<Integer> quickSort2(final List<Integer> values)
{
    if (values.size() <= 1)
        return values;

    // collect all below / above pivot
    Integer pivot = values.get(0);
    var belowOrEquals = collectAll(values, cur -> cur <= pivot);
    var aboves = collectAll(values, cur -> cur > pivot);

    // rekursiver Abstieg
    var sortedLowersPart = quickSort(belowOrEquals);
    var sortedUppersPart = quickSort(aboves);

    final List<Integer> result = new ArrayList<>();
    result.addAll(sortedLowersPart);
    result.add(pivot);
    result.addAll(sortedUp

    return result;
}
```

```
static List<Integer> collectAll(final List<Integer> values,
                                 final Predicate<Integer> condition)
{
    return values.stream().skip(1).
                  filter(condition).collect(Collectors.toList());
}
```

```
static List<Integer> quickSort2(final List<Integer> values)
{
    if (values.size() <= 1)
        return values;

    // collect all below / above pivot
    Integer pivot = values.get(0);
    var belowOrEquals = collectAll(values, cur -> cur <= pivot);
    var aboves = collectAll(values, cur -> cur > pivot);

    // rekursiver Abstieg
    var sortedLowersPart = quickSort(belowOrEquals);
    var sortedUppersPart = quickSort(aboves);

    final List<Integer> result = new ArrayList<>();
    result.addAll(sortedLowersPart);
    result.add(pivot);
    result.addAll(sortedUp

    return result;
}
```

```
static List<Integer> collectAll(final List<Integer> values,
                                 final Predicate<Integer> condition)
{
    return values.stream().skip(1).
                  filter(condition).toList();
}
```

```
def quick_sort(values):
    if len(values) <= 1:
        return values

    pivot = values[0]
    below_or_equals = [value for value in values[1:] if value <= pivot]
    aboves = [value for value in values[1:] if value > pivot]

    sorted_lowers_part = quick_sort(below_or_equals)
    sorted_uppers_part = quick_sort(aboves)

    return sorted_lowers_part + [pivot] + sorted_uppers_part
```

Geht es noch kompakter?

```
def quick_sort_short(values):
    if len(values) <= 1:
        return values

    return quick_sort_short([val for val in values[1:] if val <= values[0]]) + \
           [values[0]] + \
           quick_sort_short([val for val in values[1:] if val > values[0]])
```

```java
@ParameterizedTest(name = "{0} should be sorted to {1}")
@MethodSource("createInputAndExpected")
void testQuickSort(int[] values, int[] expected)
{
    var sortedValues = Ex06_Quicksort.quickSort(values);

    assertArrayEquals(expected, sortedValues);
}


private static Stream<Arguments> createInputAndExpected()
{
    return Stream.of(Arguments.of(new int[] { 5, 2, 7, 1, 4, 3, 6, 8 },
                                  new int[] { 1, 2, 3, 4, 5, 6, 7, 8 }),
                     Arguments.of(new int[] { 5, 2, 7, 9, 6, 3, 1, 4, 8 },
                                  new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 }),
                     Arguments.of(new int[] { 5, 2, 7, 9, 6, 3, 1, 4, 2, 3, 8 },
                                  new int[] { 1, 2, 2, 3, 3, 4, 5, 6, 7, 8, 9 }));
}
```

```
@pytest.mark.parametrize("values, expected",
                         [([5, 2, 7, 1, 4, 3, 6, 8],
                           [1, 2, 3, 4, 5, 6, 7, 8]),
                          ([5, 2, 7, 9, 6, 3, 1, 4, 8],
                           [1, 2, 3, 4, 5, 6, 7, 8, 9]),
                          [[5, 2, 7, 9, 6, 3, 1, 4, 2, 3, 8],
                           [1, 2, 2, 3, 3, 4, 5, 6, 7, 8, 9]]])
def test_quick_sort_inplace(values, expected):
    sorted_values = quick_sort(values)

    assert sorted_values == expected
```

DEMO

Exercises Part 2

https://github.com/Michaeli71/PowerOfRecursionAndBacktracking

PART 3:
2-D

Flood Fill
(sogar mit Muster)

Aufgabenstellung

Schreiben Sie eine Methode / Funktion `floodFill()`, die in einem Array alle freien Felder mit einem bestimmten Wert befüllt.

```
##              ##
#  ########     #
   #          #
   #o         #
#  ########     #
##              ##
```

```
## o            ##
#  ########     #
   #          #
   #          #
#  ########     #
##              ##
```

```
##              ##
#  ########     #
   #********#
   #********#
#  ########     #
##              ##
```

```
##############
#**##########***#
**#          #***
**#          #***
#**##########***#
##############
```

```java
void floodFill(char[][] values, int x, int y)
{
    if (x < 0 || y < 0 ||
        y >= values.length ||
        x >= values[y].length)
        return;

    if (values[y][x] == ' ')
    {
        values[y][x] = '*';

        floodFill(values, x, y - 1);
        floodFill(values, x + 1, y);
        floodFill(values, x, y + 1);
        floodFill(values, x - 1, y);
    }
}
```

```python
def flood_fill(values2dim, x, y):
    max_y, max_x = get_dimension(values2dim)

    if x < 0 or y < 0 or \
       x >= max_x or y >= max_y:
        return

    if values2dim[y][x] == ' ':
        values2dim[y][x] = '*'

        flood_fill(values2dim, x, y - 1)
        flood_fill(values2dim, x + 1, y)
        flood_fill(values2dim, x, y + 1)
        flood_fill(values2dim, x - 1, y)


def get_dimension(values2dim):
    if isinstance(values2dim, list):
        return (len(values2dim), len(values2dim[0]))

    if isinstance(values2dim, np.ndarray):
        return values2dim.shape

    raise ValueError("unsupported type", type(values2dim))
```

Was ist denn nun mit dem Muster?

Aufgabenstellung

Erweitern wir also die Implementierung so, dass nun eine Fläche auch mit Muster gefüllt werden kann!

```
## o                          #
#  ###################      ##
  #          ##             ###
  #          ##              ##
#  ################          #
##                           ##
###                         ###
##                           ##
#                            #
```

```
##..|..|..|..|..|..|..|..#
#*-###################--*-##
.|#          ##|..|..|..|###
-*#          ##*--*--*--*-##
#|.#################|..|..#
##--*--*--*--*--*--*--*-##
###.|..|..|..|..|..|..|###
##--*--*--*--*--*--*--*-##
#|..|..|..|..|..|..|..|..#
```

```java
void floodFill(char[][] values, int x,
               int y, char[][] pattern)
{
    if (x < 0 || y < 0 ||
        y >= values.length ||
        x >= values[y].length)
        return;

    if (values[y][x] == ' ')
    {
        values[y][x] = findFillChar(x, y, pattern);


        floodFill(values, x, y - 1, pattern);
        floodFill(values, x + 1, y, pattern);
        floodFill(values, x, y + 1, pattern);
        floodFill(values, x - 1, y, pattern);
    }
}
```

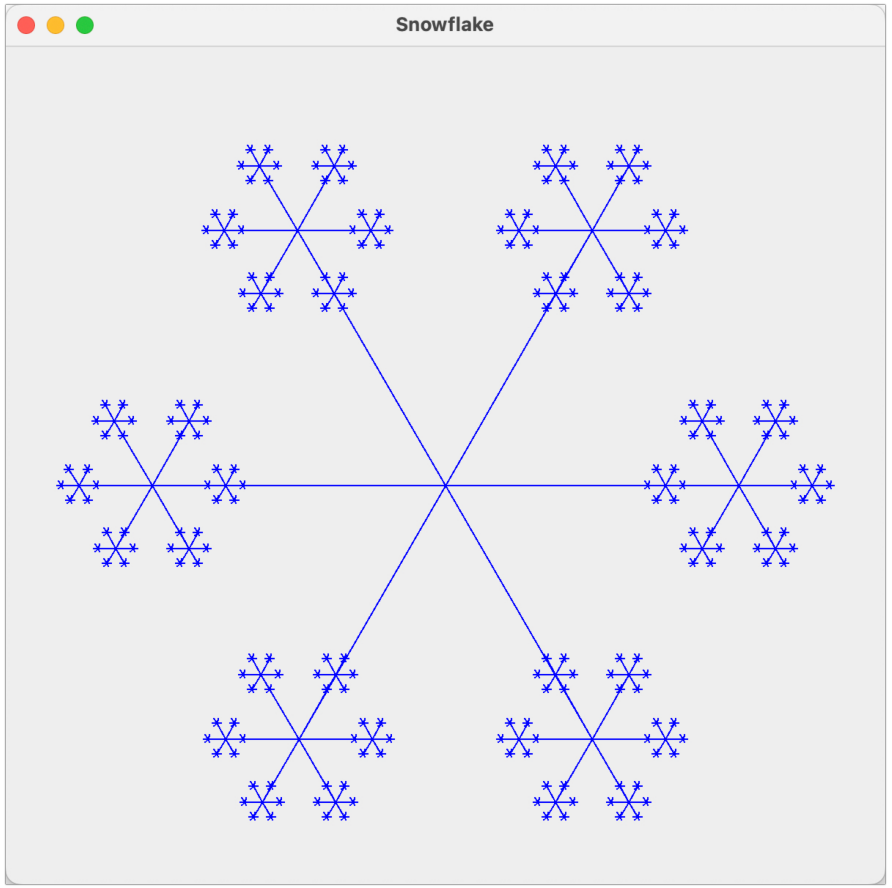
```python
def flood_fill(values2dim, x, y, pattern):
    max_y, max_x = get_dimension(values2dim)

    if x < 0 or y < 0 or \
       x >= max_x or y >= max_y:
        return

    if values2dim[y][x] == ' ':
        values2dim[y][x] = find_fill_char(y, x,
                                          pattern)

        flood_fill(values2dim, x, y - 1, pattern)
        flood_fill(values2dim, x + 1, y, pattern)
        flood_fill(values2dim, x, y + 1, pattern)
        flood_fill(values2dim, x - 1, y, pattern)
```

```java
char findFillChar(int x, int y,
                  char[][] pattern)
{
    final int maxY = pattern.length;
    final int maxX = pattern[0].length;

    return pattern[y % maxY][x % maxX];
}
```

```python
def find_fill_char(y, x, pattern):
    max_y, max_x = get_dimension(pattern)

    return pattern[y % max_y][x % max_x]
```

- **Grafische Figuren**

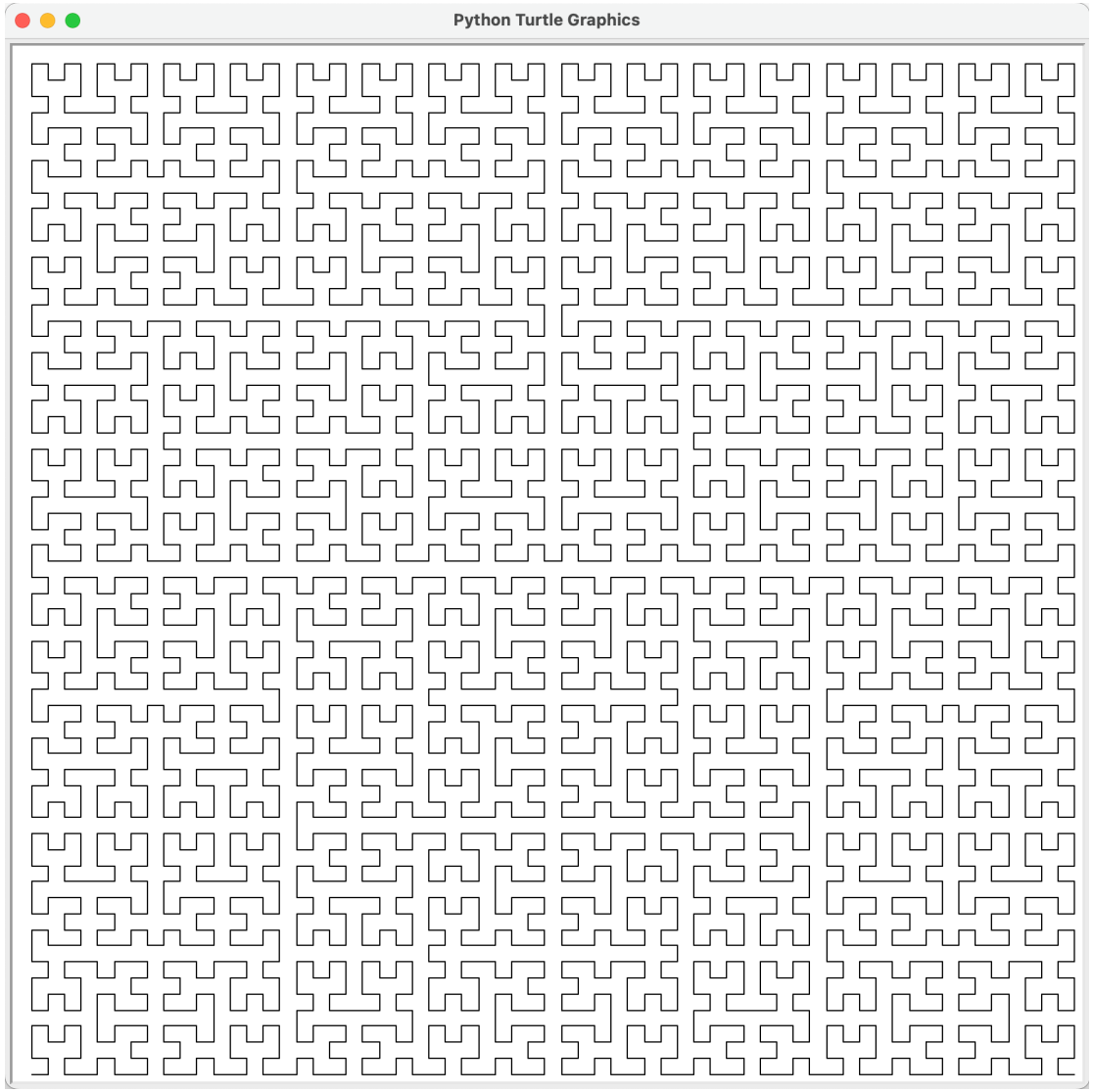
```
static void drawSnowflake(final Graphics graphis,
                          final int startX, final int startY,
                          final int length, final int depth)
{
    for (int degree = 0; degree < 360; degree += 60)
    {
        double rad = degree * Math.PI / 180;
        int endX = (int) (startX + Math.cos(rad) * length);
        int endY = (int) (startY + Math.sin(rad) * length);

        graphis.drawLine(startX, startY, endX, endY);

        // rekursiver Abstieg
        if (depth > 0)
        {
            drawSnowflake(graphis, endX, endY,
                          length / 4, depth - 1);
        }
    }
}
```



Rekursion

- **Grafische Figuren**

```
def hilbert_curve(n, turtle, angle=90):
    if n <= 0:
        return

    turtle.left(angle)
    hilbert_curve(n - 1, turtle, -angle)
    turtle.forward(1)
    turtle.right(angle)
    hilbert_curve(n - 1, turtle, angle)
    turtle.forward(1)
    hilbert_curve(n - 1, turtle, angle)
    turtle.right(angle)
    turtle.forward(1)
    hilbert_curve(n - 1, turtle, -angle)
    turtle.left(angle)
```



DEMO

Exercises Part 3

https://github.com/Michaeli71/PowerOfRecursionAndBacktracking

PART 4:
Fallstricke und Abhilfen

Fallstricke

Rekursion

- **Endlosaufrufe**

```
// Achtung: Zur Demonstration bewusst falsch
static void infiniteRecursion(final String value)
{
    infiniteRecursion(value);
}

static int factorialNoAbortion(final int number)
{
    return number * factorialNoAbortion(number - 1);
}
```

Rekursion

- **Endlosaufrufe**

```
static int calcLengthParameterValues(final String value, int count)
{
    if (value.length() == 0)
        return count;

    System.out.println("Count: " + count);
    final String remaining = value.substring(1);

    return calcLengthParameterValues(remaining, count++);
}
```

Rekursion

- **Berechnungsdauer**

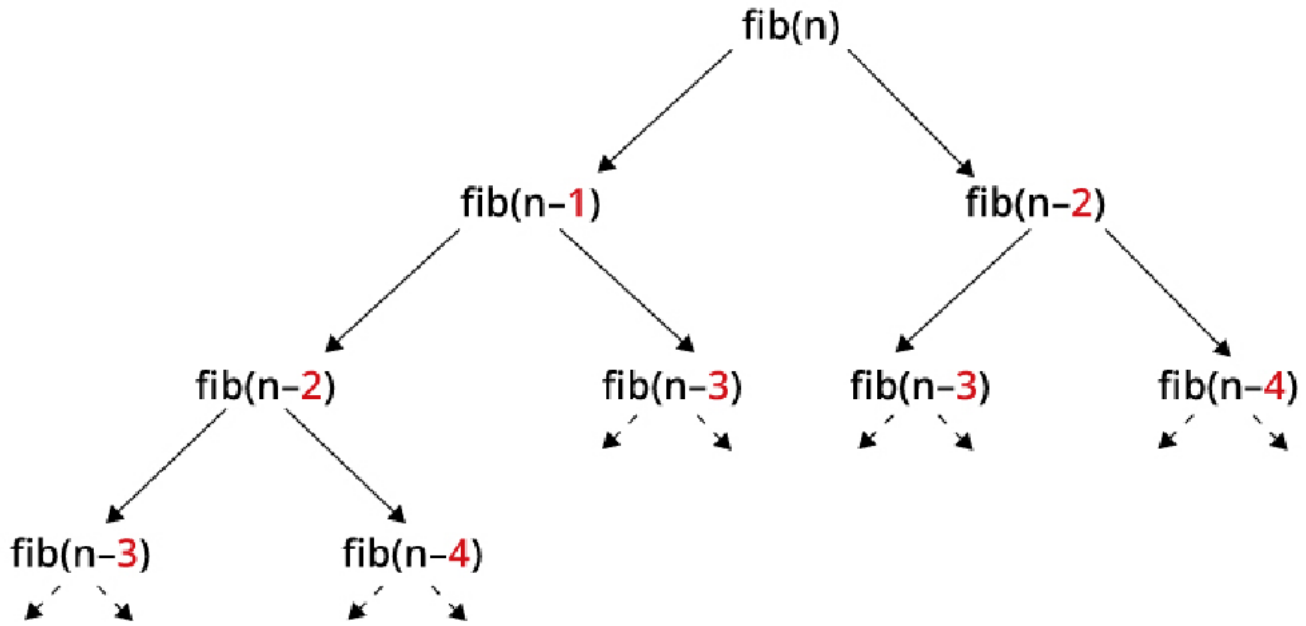
```
static long fibRec(final int n)
{
    if (n <= 0)
        throw new IllegalArgumentException("must be positive and >= 1");

    // rekursiver Abbruch
    if (n == 1 || n == 2)
        return 1;

    // rekursiver Abstieg
    return fibRec(n - 1) + fibRec(n - 2);
}


public static void main(String[] args)
{
    System.out.println("fibRec(42) = " + fibRec(42)); // sofort
    System.out.println("fibRec(45) = " + fibRec(45)); // ca. 3 s
    System.out.println("fibRec(50) = " + fibRec(50)); // ca. 23 s
    System.out.println("fibRec(70) = " + fibRec(70)); // take a break ...
}
```

DEMO

`Fibonacci.java`

Woran könnte das liegen?

- **Ursache für die hohe Berechnungsdauer**



- **Selbst bei diesem exemplarischen Aufruf erkennt man, dass diverse Aufrufe mehrmals erfolgen, etwa für *fib(n − 4)* und *fib(n − 2)*, aber insbesondere dreimal für *fib(n − 3)*. Das führt sehr schnell zu aufwendigen und langwierigen Berechnungen. Später Optimierung durch Memoization.**

Rekursion

- **Beispiel: Berechne die möglichen Kombinationen, eine beliebige Summe aus 2 und 1 CHF / Euro zusammenzusetzen**

- **Beispiel 7 Euro u.a.:**

Rekursion

- **Beispiel: Berechne die möglichen Kombinationen, eine beliebige Summe aus 5, 2 und 1 CHF zusammenzusetzen**

- **Beispiel 7 CHF u.a.:**

- **Rekursive Berechnung**

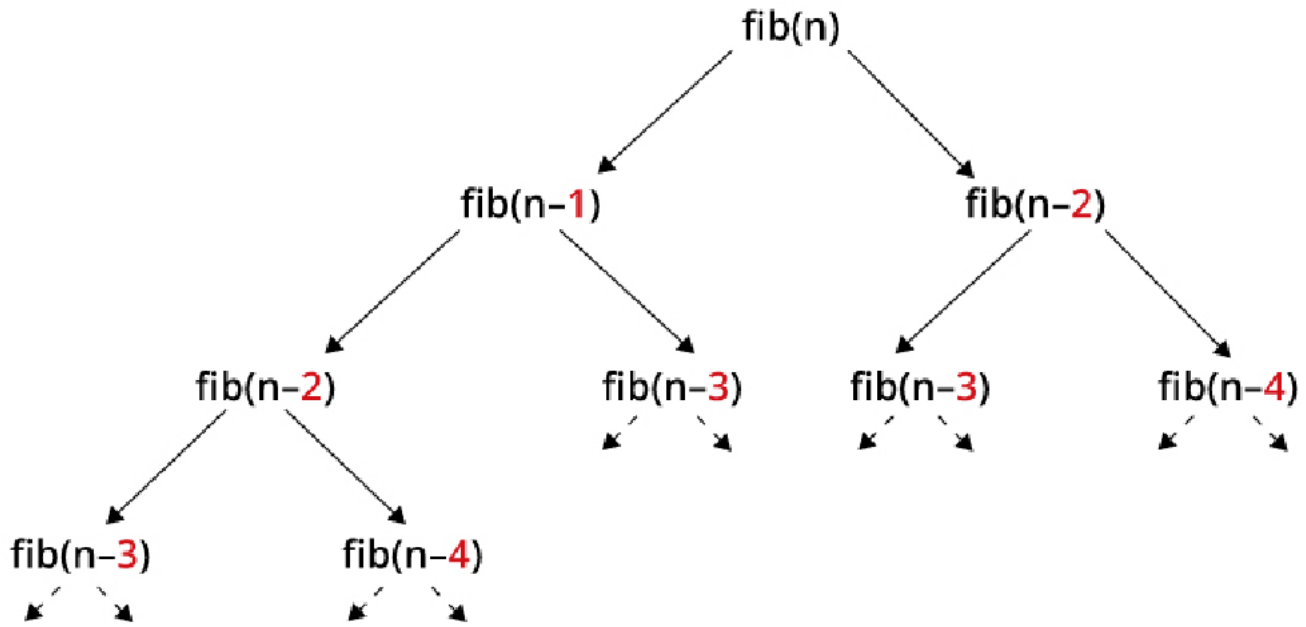
```
def coins_5_1(n):
    if n < 0:
        return 0

    if n == 0 or n == 4:
        return 1

    return coins_5_1(n - 5) + coins_5_1(n - 1)


# 5, 2, 1
def coins_5_2_1(n):
    if n < 0:
        return 0

    if n == 0 or n == 1 or n == 4:
        return 1

    return coins_5_2_1(n - 5) + coins_5_2_1(n - 2) + coins_5_2_1(n - 1)
```

DEMO

`coins.py`

Abhilfe: Memoization

- **Recap Ursache für die hohe Berechnungsdauer**



- **Selbst bei diesem exemplarischen Aufruf erkennt man, dass diverse Aufrufe mehrmals erfolgen, etwa für *fib(n − 4)* und *fib(n − 2)*, aber insbesondere dreimal für *fib(n − 3)*. Das führt sehr schnell zu aufwendigen und langwierigen Berechnungen. Später Optimierung durch Memoization.**

Habt ihr eine Idee, wie man vorgehen könnte?

Rekursion

- **Man speichert Daten zwischen und folgt den Ideen das Caching!**

- **Dazu muss man Zustand übergeben => dazu hatten wir schon Parameter kennengelernt und den Trick der Hilfsmethode**

```
static long fibonacciOptimized(final int n)
{
    return fibonacciMemo(n, new HashMap<>());
}
```

Rekursion

- **Dazu muss man Zustand übergeben => dazu hatten wir schon Parameter kennengelernt und den Trick der Hilfsmethode**

```
static long fibonacciMemo(final int n, final Map<Integer, Long> lookupMap)
{
    if (n <= 0)
        throw new IllegalArgumentException("must be > 0");

    // MEMOIZATION: prüfe, ob vorberechnetes Ergebnis
    if (lookupMap.containsKey(n))
        return lookupMap.get(n);

    // normaler Algorithmus mit Hilfsvariable für Resultat
    long result = 0;
    if (n == 1 || n == 2)
        result = 1;
    else
        result = fibonacciMemo(n - 1, lookupMap) + fibonacciMemo(n - 2, lookupMap);

    // MEMOIZATION: speichere berechnetes Ergebnis
    lookupMap.put(n, result);

    return result;
}
```

Rekursion

- **Dazu muss man Zustand übergeben => dazu hatten wir schon Parameter kennengelernt und den Trick der Hilfsmethode**

```
static long fibonacciOptimizedNicer(final int n)
{
    return fibonacciMemoOpt(n, new HashMap<>(Map.of(1, 1L, 2, 1L)));
}

static long fibonacciMemoOpt(final int n, final Map<Integer, Long> lookupMap)
{
    if (n <= 0)
        throw new IllegalArgumentException("must be > 0");

    // MEMOIZATION: prüfe, ob vorberechnetes Ergebnis
    if (lookupMap.containsKey(n))
        return lookupMap.get(n);

    // normaler Algorithmus mit Hilfsvariable für Resultat
    long result = fibonacciMemoOpt(n - 1, lookupMap) + fibonacciMemoOpt(n - 2, lookupMap);

    // MEMOIZATION: speichere berechnetes Ergebnis
    lookupMap.put(n, result);

    return result;
}
```

DEMO

MemoizationExamples.java

Rekursion

```
def coins_5_2_1(n):
    if n < 0:
        return 0

    if n == 0 or n == 1 or n == 4:
        return 1

    return coins_5_2_1(n - 5) + coins_5_2_1(n - 2) + coins_5_2_1(n - 1)
```

- **Mit Memoization**

```
def coins_5_2_1(n):
    return coins_5_2_1_memo(n, {0: 1, 1: 1, 4: 1})


def coins_5_2_1_memo(n, cache):
    if n < 0:
        return 0

    if n in cache:
        return cache[n]

    cache[n] = coins_5_2_1_memo(n - 5, cache) + coins_5_2_1_memo(n - 2, cache) + coins_5_2_1_memo(n - 1, cache)
    return cache[n]
```

DEMO

`coins_memo.py`

Exercises Part 4

https://github.com/Michaeli71/PowerOfRecursionAndBacktracking

PART 5:
Backtracking

Weg aus Labyrinth

n-Damenproblem

Sudoku Solver

Exercises Part 5

https://github.com/Michaeli71/PowerOfRecursionAndBacktracking