Educational Objectives

- You understand how a solution to a recursive problem can be implemented in Java.
- You understand how methods are being executed in an execution stack.

12. Recursion

Mathematical Recursion, Termination, Call Stack, Examples, Recursion vs. Iteration, Lindenmayer Systems

Mathematical Recursion

Many mathematical functions can be naturally defined recursively.
The means, the function appears in its own definition

\[ n! = \begin{cases} 
1, & \text{if } n \leq 1 \\
 n \cdot (n - 1)!, & \text{otherwise} 
\end{cases} \]

Recursion in Java:

// POST: return value is n!
public static int fac (int n) {
    if (n <= 1)
        return 1;
    else
        return n * fac (n-1);
}
### Infinite Recursion

- is as bad as an infinite loop...
- ...but even worse: it burns time and memory

```java
private static void f() {
    f(); // f() -> f() -> ... stack overflow
}
```

### Recursive Functions: Termination

As with loops we need
- progress towards termination

```
fac(n):
terminates immediately for \( n \leq 1 \), otherwise the function is called
recursively with \(< n\).```

"n is getting smaller with each call."

### Recursive Functions: Evaluation

**Example:** \( \text{fac}(4) \)

// POST: return value is \( n! \)
```java
public static int fac(int n) {
    if (n <= 1) return 1;
    return n * fac(n-1); // n > 1
}
```

**Initialization of the formal argument:** \( n = 4 \)

*recursive call with argument \( n - 1 = 3 \)*

### The Call Stack

For each method call:
- push value of the actual parameter on the stack
- work with the upper most value
- at the end of the call the upper most value is removed from the stack

```
fac(4)
fac(3)
fac(2)
fac(1)
```

- \( n = 1 \)
  - \( 1! = 1 \)
- \( n = 2 \)
  - \( 2 \cdot 1! = 2 \)
- \( n = 3 \)
  - \( 3 \cdot 2! = 6 \)
- \( n = 4 \)
  - \( 4 \cdot 3! = 24 \)

Out.println(fac(4))
**Euclidean Algorithm**

- finds the greatest common divisor $\text{gcd}(a, b)$ of two natural numbers $a$ and $b$
- is based on the following mathematical recursion:

\[
\text{gcd}(a, b) = \begin{cases} 
  a, & \text{falls } b = 0 \\
  \text{gcd}(b, a \mod b), & \text{andernfalls}
\end{cases}
\]

**Euclidean Algorithm in Java**

```java
public static int gcd (int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd (b, a % b);
}
```

**Fibonacci Numbers**

\[
F_n := \begin{cases} 
  0, & \text{falls } n = 0 \\
  1, & \text{falls } n = 1 \\
  F_{n-1} + F_{n-2}, & \text{falls } n > 1
\end{cases}
\]

**Fibonacci Numbers in Java**

```java
public static int fib (int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib (n-1) + fib (n-2); // n > 1
}
```

**Laufzeit**

`fib(50)` takes “forever” because it computes

- $F_{48}$ two times,
- $F_{47}$ 3 times,
- $F_{46}$ 5 times,
- $F_{45}$ 8 times,
- $F_{44}$ 13 times,
- $F_{43}$ 21 times ...
- $F_1$ ca. $10^9$ times (!)

```java
public static int fib (int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib (n-1) + fib (n-2); // n > 1
}
```

Korrektheit und Terminierung sind klar.
Fast Fibonacci Numbers

Idea:
- Compute each Fibonacci number only once, in the order $F_0, F_1, F_2, \ldots, F_n$!
- Memorize the most recent two numbers (variables $a$ and $b$)!
- Compute the next number as a sum of $a$ and $b$!

Fast Fibonacci Numbers in Java

```java
public static int fib (int n){
if (n == 0) return 0;
if (n <= 2) return 1;
int a = 1; // F_1
int b = 1; // F_2
for (int i = 3; i <= n; ++i){
    int a_old = a; // F_i-2
    a = b; // F_i-1
    b += a_old; // F_i-1 += F_i-2 -> F_i
}
return b;
}
```

Recursion and Iteration

Recursion can always be simulated by
- Iteration (loops)
- explicit “call stack” (e.g. array)

Often recursive formulations are simpler, sometimes they are less efficient

The Power of Recursion

- Some problems appear to be hard to solve without recursion. With recursion they become significantly simpler.
- Examples: The towers of Hanoi, The $n$-Queens-Problem, Sudoku-Solver, Expression Parsers, Reversing In- or Output, Searching in Trees, Divide-And-Conquer (e.g. sorting) → Informatik II,
Experiment: The Towers of Hanoi

The Towers of Hanoi – Code

```java
public static void move(int n, int source, int aux, int dest) {
    if (n == 1) {
        Out.println("move " + source + "−>" + dest);
    } else {
        move(n - 1, source, dest, aux);
        move(1, source, aux, dest);
        move(n - 1, aux, source, dest);
    }
}
```

The Towers of Hanoi – Code

1. Move 3 discs from 0 to 1 with auxiliary staple 2:
   `move(3, 0, 2, 1);`
2. Move 1 disc from 0 to 2
   `move(1, 0, 1, 2);`
3. Move 3 discs from 1 to 2 with auxiliary staple 0
   `move(3, 1, 0, 2);`

Move 4 discs vom 0 to 2 with auxiliary staple 1:
`move(4, 0, 1, 2);`
Lindenmayer-Systems (L-Systems)

Fractals from Strings and Turtles

L-Systems have been invented by the Ungarian Biologist Aristid Lindenmayer (1925 – 1989) to model growth of plants.

Definition and Example

- alphabet $\Sigma$
- $\Sigma^*$: finite words over $\Sigma$
- production $P: \Sigma \rightarrow \Sigma^*$
- initial word $s_0 \in \Sigma^*$

<table>
<thead>
<tr>
<th>$c$</th>
<th>$P(c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F$</td>
<td>$F + F +$</td>
</tr>
<tr>
<td>$+$</td>
<td>$+$</td>
</tr>
<tr>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>$F$</td>
<td></td>
</tr>
</tbody>
</table>

Definition

The triple $L = (\Sigma, P, s_0)$ is an L-System.

The Language Described

Wörter $w_0, w_1, w_2, \ldots \in \Sigma^*$:

$w_0 := s_0$

$w_1 := P(w_0)$

$w_2 := P(w_1)$

$\vdots$

$F(F) = F + F +$

Turtle Graphics

Turtle with position and direction

Turtle understands 3 commands:

- $F$: move one step forwards ✓
- $+$: rotate by 90 degrees ✓
- $-$: rotate by $-90$ degrees ✓

Definition

$P(c_1c_2 \ldots c_n) := P(c_1)P(c_2) \ldots P(c_n)$
word \( w_0 \in \Sigma^* \):

```java
public static void main(String[] args) {
    Out.print("Maximal Recursion Depth = ");
    int depth = In.readInt();
    Turtle t = new Turtle();
    produce(t, "F", depth);
    t.show();
}
```

\( w = w_0 = F \)

---

production

```java
// POST: recursively iterate over the production of the characters
// of a word.
// When recursion limit is reached, the word is "drawn"
static void produce (Turtle turtle, String word, int depth){
    if (depth > 0) {
        for (int k = 0; k < word.length(); ++k) {
            \( w_1 = F + F + \checkmark \)
            produce(turtle, replace(word.charAt(k)), depth-1);
        }
    } else {
        draw(turtle,word);
        draw \( w = w_n \).
    }
}
```

---

replace

```java
// POST: returns the production of c
static String replace (char c){
    switch (c) {
    case 'F':
        return "F+F+";
    default:
        return Character.toString(c); // trivial production c \( \rightarrow \) c
    }
}
```
lindenmayer: draw

// POST: draws the turtle graphic interpretation of word
static void draw (Turtle turtle, String word) {
    for (int k = 0; k < word.length(); ++k) {
        switch (word.charAt(k)) {
            case 'F':
                turtle.forward(1); // move one step forward
                break;
            case '+':
                turtle.left(90); // turn counterclockwise by 90 degrees
                break;
        }
    }
}

L-Systeme: Erweiterungen

- arbitrary symbols without graphical interpretation
- arbitrary angles (snowflake)
- saving and restoring the state of the turtle → plants (bush)

Challenge: we are looking forward to your contributions