Recursion/ recursive programming

Definition

```
Recursion ~ see recursion
```

or

```
We need to got deeper (© Inception)
```

Method of solving problems by applying same solution algorithm to solve smaller instances of problem.

Wrt. programming - functions calling themselves from within own code.

Prominent examples of recursion include

- Divide-and-Conquer algorithms (quicksort)
- Tree traversal
- Fractals
- Definition of grammar for natural languages

Basically: A function or method is said to be **recursive** if it calls itself

Recursion types (examples)

- Single/direct recursion
  - `Recursion ~ see recursion`
- Multiple recursion
  - `fibonacci(n) = fibonacci(n-1) + fibonacci(n+2)`
- Indirect recursion
  - `f1(n) = f2(n-1); f2(m) = f1(m+1)`
  - example from literature, see https://en.wikipedia.org/wiki/Sepulka

Loops vs. Recursion

Recap `while` loop

```
int i = 10;
while (i > 0) {
    System.out.println(i);
    i--;
}
```

First example recursion

```
private void recursiveMethod(int v) {
    if (v <= 0) {
        return;
    }
    System.out.println(v);
    recursiveMethod(v - 1);
}

recursiveMethod(10);
```

Generally (see computability theory) it has been proven, that programming languages without support for imperative loops but with support for recursion are at least as powerful as imperative languages based on `for` and `while`.

See example `nl.yoink.courses.dev.java.recursion.demo1.WhileLoopTransformationTest` for a complete example of transformation of a `while` loop using recursive programming technique.

In certain cases (see further examples and exercises) recursive programming is more suitable and understandable due to a recursive nature of either data structure or algorithm itself.

Example factorial

`n! = 1 * 2 * … (n-1) * n; 0! = 1 (per convention)`

```
int factorial(int v) {
    if (v < 0) {
        throw new IllegalArgumentException();
    }

    if (v == 0 || v == 1) {
        return 1;
    }

    return v * factorial(v - 1);
}
```

Note: it’s important to properly define termination conditions on recursive functions, otherwise you might fall into the rabbit hole of recursive calls.

Exercise: try to omit

```
    if (v < 0) {
        throw new IllegalArgumentException();
    }
```

from the code above and compute `fibonacci(-1)`.

See `nl.yoink.courses.dev.java.recursion.demo1.FactorialTest`

Example fibonacci

`fibonacci(n) = fibonacci(n-1) + fibonacci(n-2); fibonacci(0) = 0; fibonacci(1) = 1`

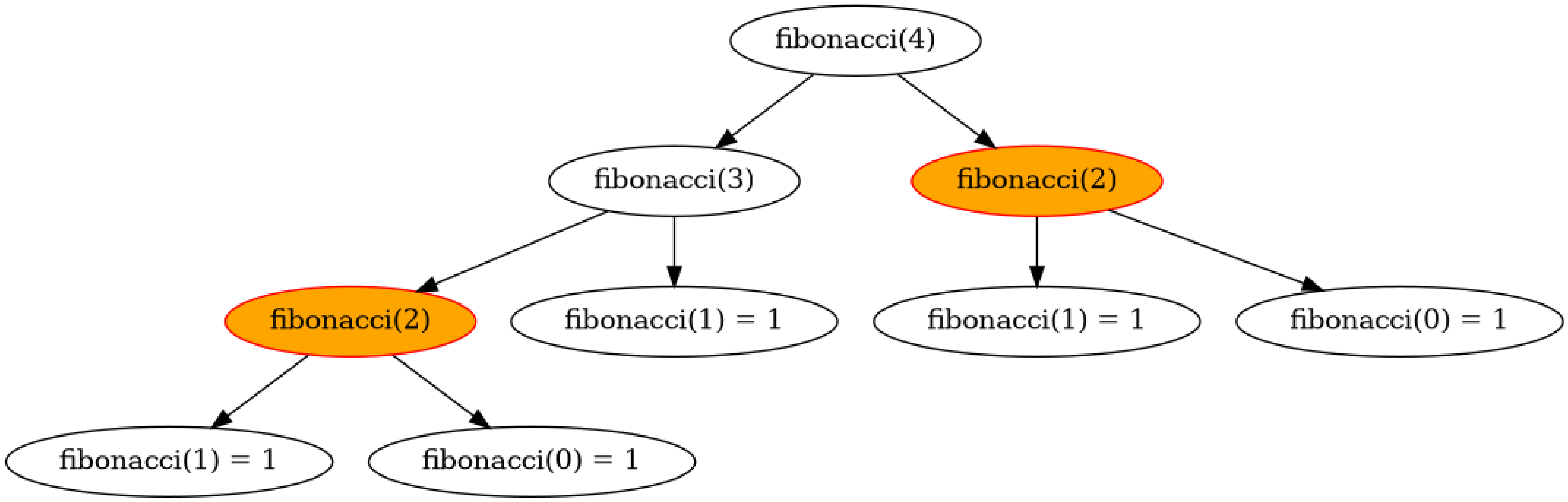
```
int fibonacci(int v) {
    if (v < 0) {
        throw new IllegalArgumentException();
    }

    if (v == 0 || v == 1) {
        return 1;
    }

    return fibonacci(v - 1) + fibonacci(v - 2);
}
```

See `nl.yoink.courses.dev.java.recursion.demo1.FibonacciTest`

Note: it’s important to understand, that the above implementation is not the most efficient one.

First, there is a so-called closed-form expression to computing `fibonacci(n)` (Binet’s formula, see https://mathworld.wolfram.com/BinetsFibonacciNumberFormula.html).

Second, some of the `fibonacci(x)` (e.g. `fibonacci(2)`, marked below) are computed multiple times.



Exercise: re-write method to cache [1] intermediate results.

Recursion pitfalls

Termination

As mentioned previously, one of the common pitfalls of recursive programming is providing/defining proper termination condition.

That’s a common reason to start the implementation of the recursive method with the explicit termination condition(s), e.g.

```
if (v < 0) {
    throw new IllegalArgumentException();
}

if (v == 0 || v == 1) {
    return 1;
}
```

In some other cases, the termination of the recursive algorithm is ensured by only execution the recursive call under certain conditions, see `nl.yoink.courses.dev.java.recursion.demo1.TreeAggregationTest`

Recursion depth

Another common problem is the recursion depth. Since recursive method calls are allocated on stack (HotSpot JVM, TODO: clarify whether stack is already known), depending on the stack size only a certain amount of stack frames (each frame - recursive call - consuming memory on stack) is available.

In Java the stack size is controlled by `-Xss` JVM argument, for more hands-on please see `nl.yoink.courses.dev.java.recursion.demo1.StackOverflowErrorTest`

Note: each thread in Java has its own stack (HotSpot JVM, TODO: link proof)

Tail recursion

Special case of direct recursion, when recursive call is the last operation of the function.

See below (`nl.yoink.courses.dev.java.recursion.demo1.ListAggregationTest`) and compare

```
int sum(List<Integer> integerList) {
    if (integerList == null || integerList.isEmpty()) {
        return 0;
    }

    return integerList.get(0) + sum(integerList.subList(1, integerList.size())); // last op is `+`
}
```

vs.

```
int sumTailRec(List<Integer> integerList) {
    return sumWithAggregate(0, integerList);
}

int sumWithAggregate(int intermediate, List<Integer> integerList) {
    if (integerList == null || integerList.isEmpty()) {
        return intermediate;
    }

    return sumWithAggregate(intermediate + integerList.get(0), integerList.subList(1, integerList.size()));
// last op is `sumWithAggregate`
}
```

Why should one consider this special case? Tail recursion is subject to optimization, since it doesn’t need all the frames on the stack but can instead be optimized to replace frames on the stack, thus avoiding one of previously mentioned pitfalls.

Note: AFAIK, Java does not (yet?) fully support tail call recursion optimization.

Exercise: set conditional breakpoint (`intermediate > 50`) in `sumWithAggregate` and check the frames.

Example: tree traversal BFS (breadth-first-search)

See `nl.yoink.courses.dev.java.recursion.demo1.TreeTraversalBFSTest`

Exercise: tree traversal DFS (depth-first-search)

See `nl.yoink.courses.dev.java.recursion.demo1.TreeTraversalDFSTest`

Example: tree value aggregation

When representing data in tree-like hierarchical structures (think of e.g. a car dealer chain, where a dealer can have sub-dealers or sell cars directly to customer) recursive algorithms come in handy when trying to aggregate data across this structures.

In the example below you will compute the aggregate sum following simple instruction

```
value at the node is equals to sum of the values of its children nodes
```

which could represent the joint revenue computation across all car dealers in the example above.

See `nl.yoink.courses.dev.java.recursion.demo1.TreeAggregationTest`

Exercise: implement the solution using imperative loop and compare the outcome wrt. readability.

Advanced: Visitor pattern

See https://www.gofpatterns.com/behavioral/patterns/visitor-pattern.php

Advanced: Recursive descent parser

See https://en.wikipedia.org/wiki/Recursive_descent_parser

Summary

Recursion/ recursive programming is a helpful tool to proceed in a divide-and-conquer manner.

Questions?

[1] https://en.wikipedia.org/wiki/Cache_(computing)