Recursion

Martin Steffen

Autumn 2017

Recursion

- Latin: *recurrere*

> Linguistics/Mathematics:
> Relating to or involving
> the repeated application
> of a rule, definition, or
> procedure to successive
> results.
>
> Oxford Dictionary

Recursion

The reduction of a problem to a simpler problem

Martin Steffen

Recursion

- Latin: *recurrere*

> Linguistics/Mathematics:
> Relating to or involving
> the repeated application
> of a rule, definition, or
> procedure to successive
> results.
>
> Oxford Dictionary

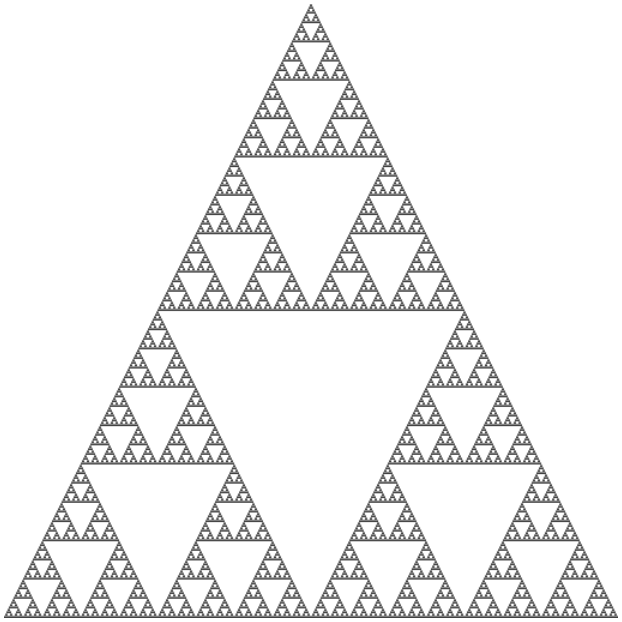
Recursion

The reduction of a problem to a (simpler) version of itself

- self-referentiality: recursive problems, algorithms, definitions, data structures, ...

Martin Steffen

Recursion



Martin Steffen

Multiplication

Martin Steffen

From elementary school: $a \times b$

"a times b" means: add b onto itself, and do that a-times.

$$a \times b = \underbrace{b + b + \ldots + b}_{a \geq 0}$$

Example (4×3)

3
plus 3 gives 6,
plus 3 gives 9,
plus 3 gives 12, and done.

Multiplication

Martin Steffen

From elementary school: $a \times b$

"a times b" means: add b onto itself, and do that a-times.

$$a \times b = 0 + \underbrace{b + b + \ldots + b}_{a \geq 0}$$

Example (4×3)

3
plus 3 gives 6,
plus 3 gives 9,
plus 3 gives 12, and done.

Multiplication

Martin Steffen

From elementary school: $a \times b$

"a times b" means: add b onto itself, and do that a-times.

$$a \times b = 0 + \underbrace{b + b + \ldots + b}_{a \geq 0} = \sum_{i=1}^{a} b$$

Example (4×3)

3

plus 3 gives 6,

plus 3 gives 9,

plus 3 gives 12, and done.

As static method

```
public class Timesiter {
    public static long times_iter(int a, int b) {
        int r = 0;
        for  (int i = 1; i<= a;i++) {
            r = r + b;
        };
        return r;
    }

  public static void main(String args[]) {
      System.out.println(times_iter (4, 3));
  }
}
```

As static method

```
public class Timesiter {
    public static long times_iter(int a, int b) {
        int r = 0;
        for  (int i = 1; i<= a;i++) {
            r = r + b;
        };
        return r;
    }

  public static void main(String args[]) {
      System.out.println(times_iter (4, 3));
  }
}
```

As static method (2)

```
public static long times_iter(int a, int b) {
    int r = 0;
    for  (int i = 1; i<= a;i++) {
        r = plus(r,b);
    };
    return r;
}
public static int plus(int x, int y) {
    return x + y;}
```

Just to focus

```
times_iter(a, b) {
  int r = 0;
  for  (i = 1; i<= a;i++) {
       r = plus(r,b);  };
  return r;
}

plus(x, y) { return x + y;}
```

$$a \times b = 0 + \underbrace{b + b + \ldots + b}_{a \geq 0} = \sum_{i=1}^{a} b$$

Multiplication: one more time

$$a \times b = 0 + \underbrace{b + b + \ldots + b}_{a \geq 0} = \sum_{i=1}^{a} b$$

Martin Steffen

Multiplication: one more time

Martin Steffen

$$a \times b = b + \underbrace{b + \ldots + b}_{a-1 \text{ “mal” b}} \quad = \quad b + ((a-1) \times b)$$

Self-referential definition (= recursive)

Multiplication calculated with the help of Multiplication (and addition)

Multiplication: one more time

$$a \times b = b + \underbrace{b + \ldots + b}_{a-1 \text{ “mal” b}} \quad = \quad b + ((a-1) \times b)$$

Self-referential definition (= recursive)

Multiplication of $a \times b$ calculated with the help of multiplication of $a - 1 \times b$ (and addition)

Martin Steffen

In Java

Martin Steffen


```
public static long times(int a, int b) {
    if (a == 0) {
        return 0;
    } else {
        return b + times (a-1, b);
    }
}
```

In Java

```
times(a, b) {
        if  (a == 0) {
            return 0;
        } else {
            return b + times (a-1, b);
        }
    }
```

$$a \times b = b + ((a-1) \times b)$$

Martin Steffen

In Java

```
times(a, b) {
        if  (a == 0) {
            return 0;
        } else {
            return b + times (a-1, b);
        }
    }
```

$$a \times b \Leftarrow b + ((a-1) \times b)$$

Martin Steffen

And what about negative numbers?

```
public static long times(int a, int b) {
    if (a >= 0) {
        if (a == 0) {
            return 0;
        } else {
            return b + times (a-1, b);
        }
    } else {
        return - (times (-a, b));
    }
}
```

Martin Steffen

Take home message

Martin Steffen

Recursion

A recursive method is defined by via itself (= calling itself).

Take home message

Martin Steffen

Recursion

A recursive method is defined by via itself (= calling itself).

"Proper recursion" (= termination)

To solve a problem:

1. define a method via calling itself of a "simpler" version of the problem.
2. and: there is a simplest problem which is directly solvable (= without further recursion), such that the recursion "finds the exit" and terminates

Take home message

Martin Steffen

Recursion

A recursive method is defined by via itself (= calling itself).

"Proper recursion" (= termination)

To solve a problem:

1. define a method via ~~an~~ applications of itself on ~~one~~ "simpler" versions of the problem
2. and: there ~~is one~~ are simplest problems, which ~~is~~ are directly (= without further recursion) solvable, such that the recursion "finds the exit" and terminates.

Factorial

Martin Steffen

Iteratively: "1 times 2 times 3... until n"

$$n! = 1 \times 2 \times \ldots \times (n-1) \times n = \prod_{i=1}^{n} i$$

Factorial

Martin Steffen

Iteratively: "1 times 2 times 3... until n"

$$n! = 1 \times 2 \times \ldots \times (n-1) \times n = \prod_{i=1}^{n} i$$

```
factorial_iter(n) {
  long result = 1;
  for  (i = 1; i<= n;i++) {
     result = result * i;
  };
  return result;
```

One more time, but recursive

$$n! = n \times (n-1) \times \ldots \times 2 \times 1$$

Martin Steffen

One more time, but recursive

$$n! = n \times \underbrace{(n-1) \times \ldots \times 2 \times 1}_{(n-1)!}$$

```
factorial(n) {
    if (n == 1) return 1;
    return n * factorial(n-1);
}
```

Martin Steffen

Side by side comparison

```
factorial_iter(n) {
  long result = 1;
  for (i = 1; i<= n;i++) {
     result = result * i;
  };
  return result;
}
```

- variable `result` updated step by step (= iteratively)

```
factorial(n) {
   if (n == 1) return 1;
   return n * factorial(n-1);
}
```

- variable `result` local local to method body
- exactly *one* value per call

Side by side comparison

```
factorial_iter(n) {
  long result = 1;
  for (i = 1; i<= n;i++) {
    result = result * i;
  };
  return result;
}
```

- variable `result` updated step by step (= iteratively)

```
factorial(n) {
  long result;
  if (n == 1) { result = 1; }
  else {
    result = n * factorial(n-1);
  };
  return result;
}
```

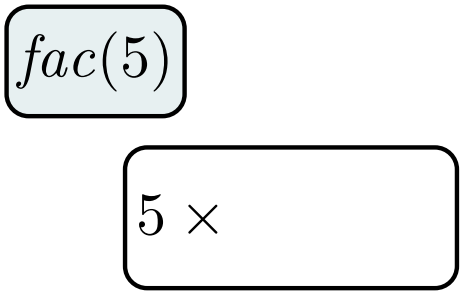
- variable `result` local
  local to method body
- exactly *one* value per call

At run time

$fac(5)$

Martin Steffen

At run time



Martin Steffen

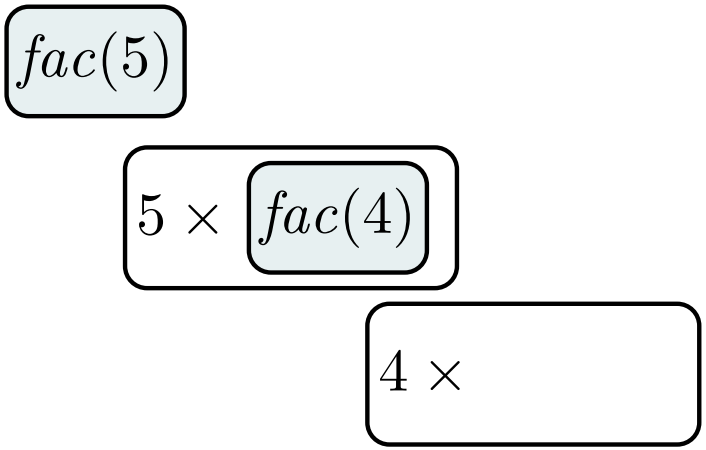
At run time

$$\boxed{fac(5)}$$

$$\boxed{5 \times \boxed{fac(4)}}$$

Martin Steffen

At run time



Martin Steffen

At run time

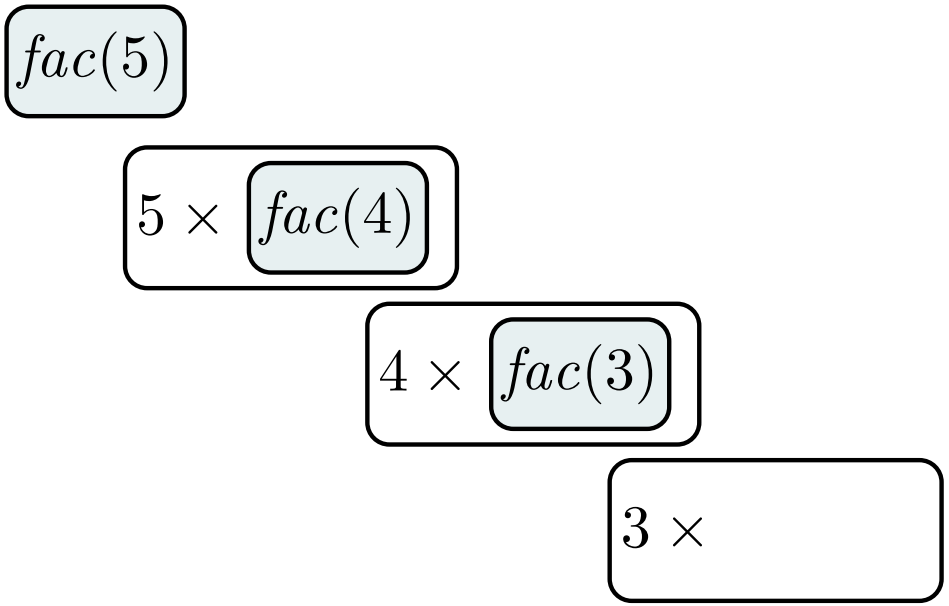
$$fac(5)$$

$$5 \times fac(4)$$

$$4 \times fac(3)$$

Martin Steffen

At run time



Martin Steffen

At run time

$fac(5)$

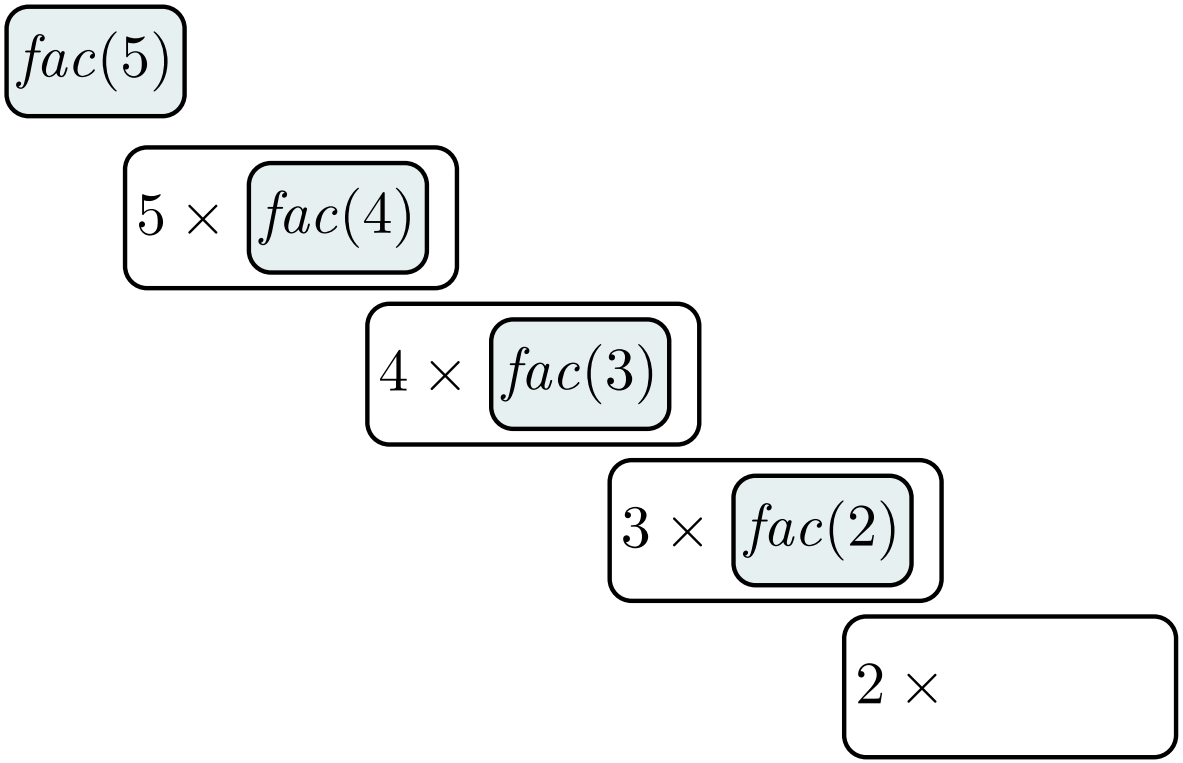
$5 \times fac(4)$

$4 \times fac(3)$

$3 \times fac(2)$

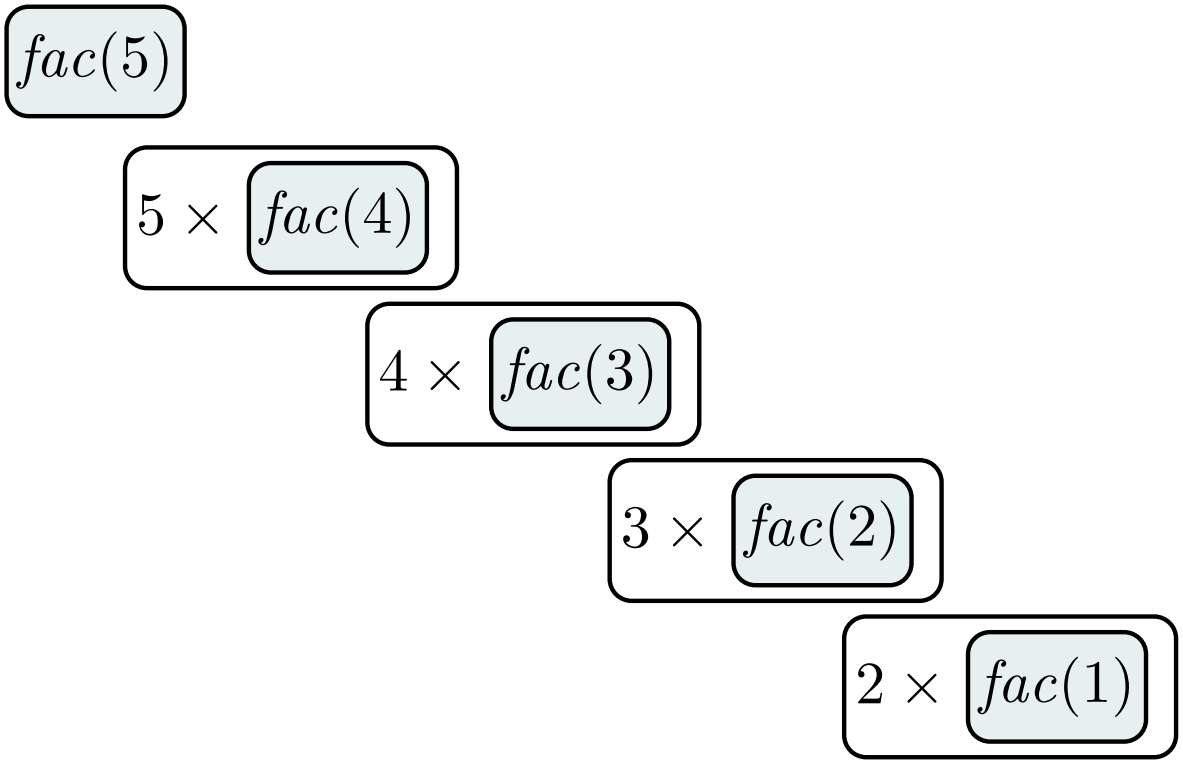
Martin Steffen

At run time



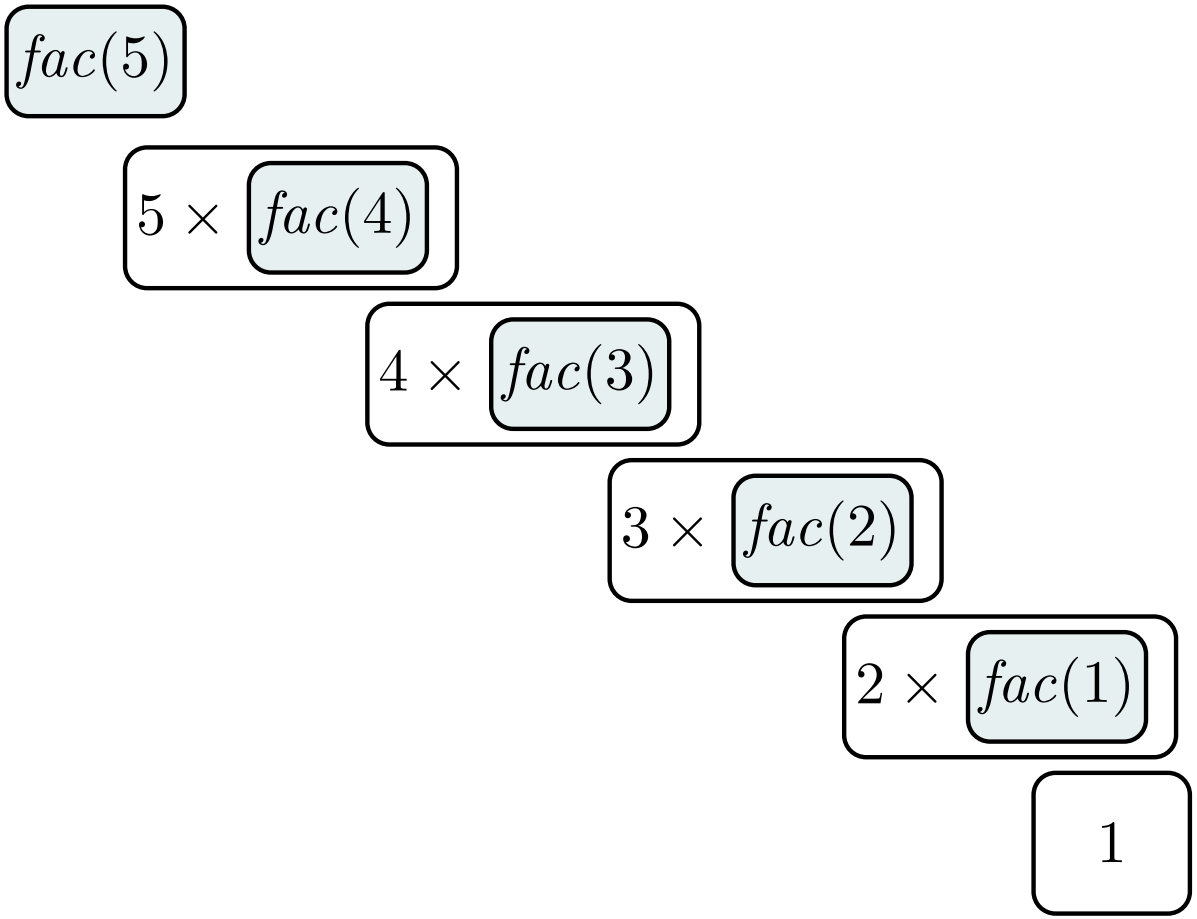
Martin Steffen

At run time



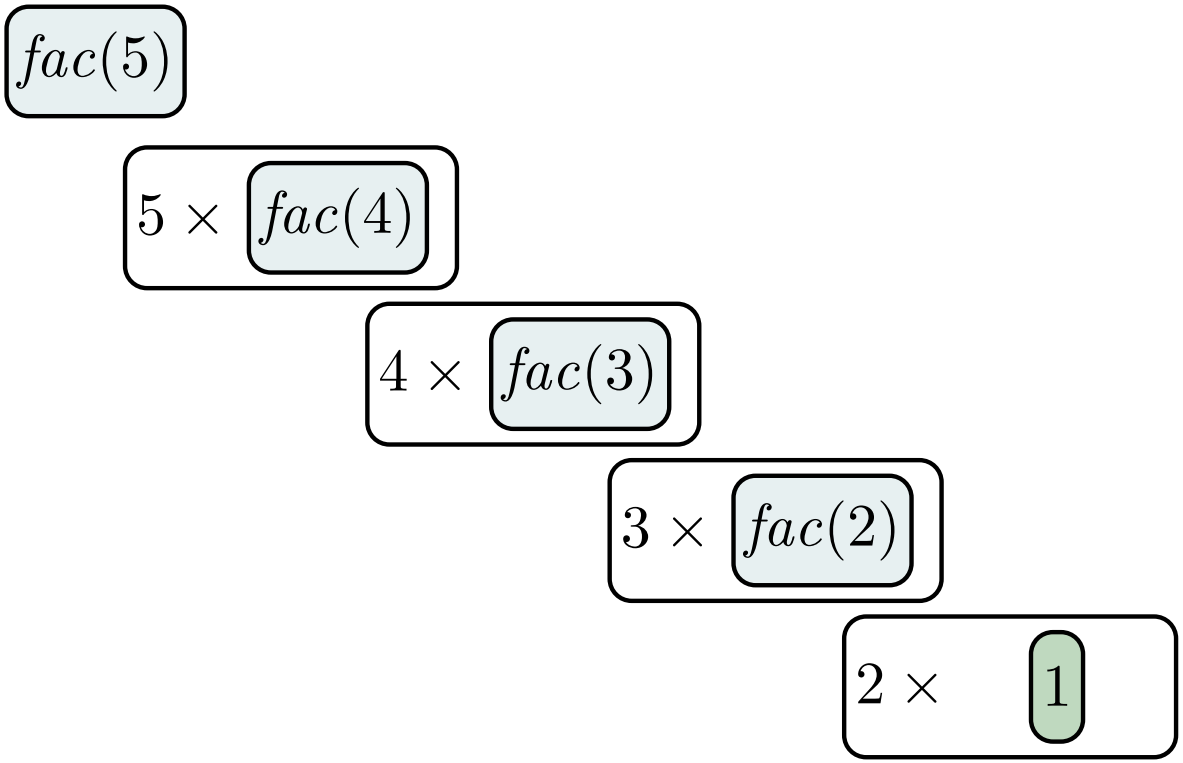
Martin Steffen

At run time



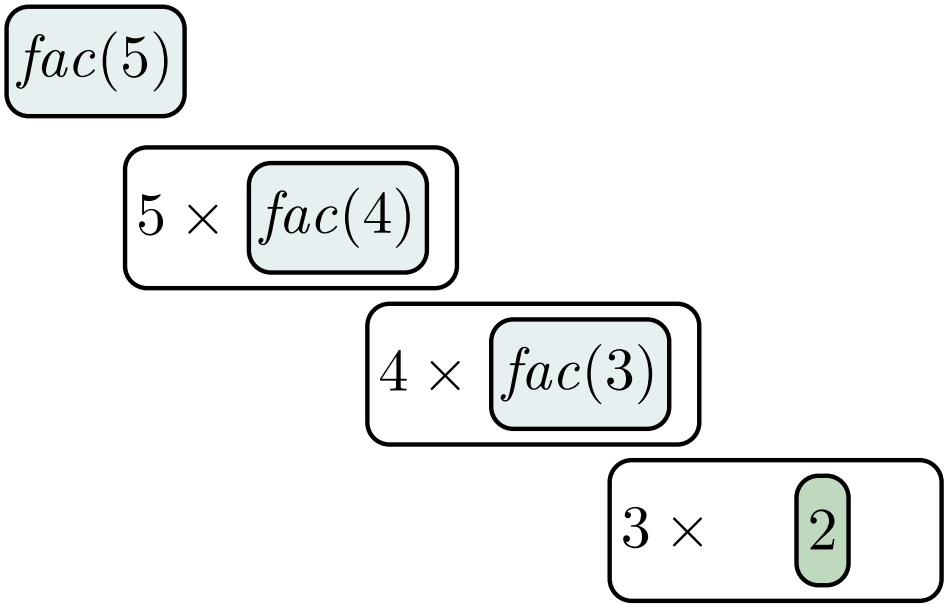
Martin Steffen

At run time



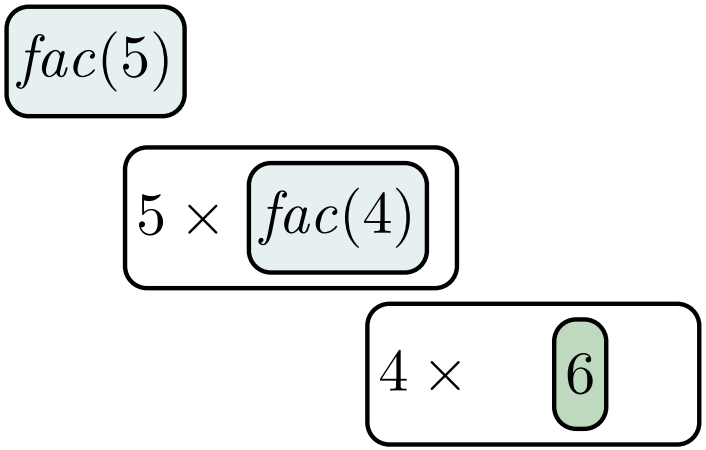
Martin Steffen

At run time



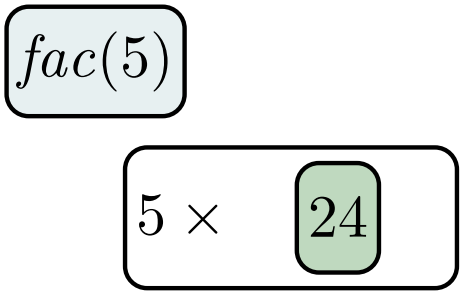
Martin Steffen

At run time



Martin Steffen

At run time

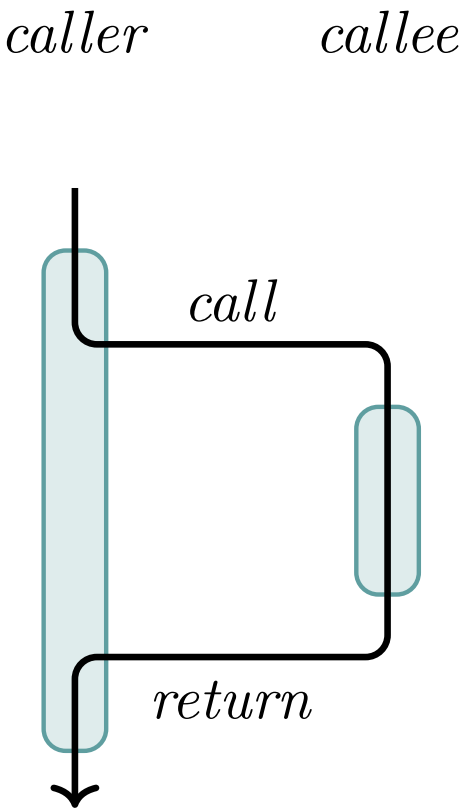


Martin Steffen

At run time

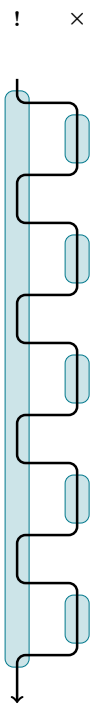
Martin Steffen

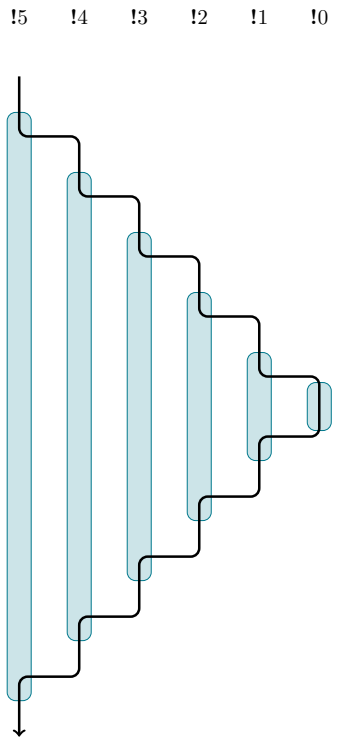
Caller and callee



Martin Steffen

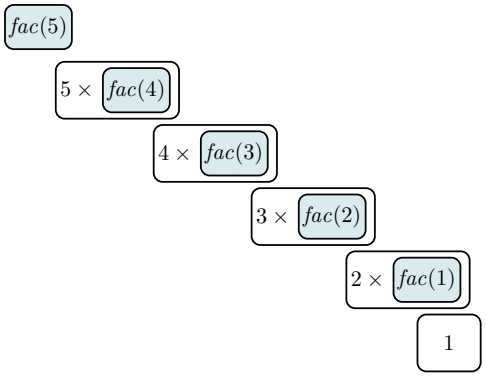
Life time of local variables





Martin Steffen

Run time stack




```
factorial(n) {
    long result;
    if (n == 1) { result = 1; }
    else {
      result = n * factorial(n-1);
    };
    return result;
}
```

- $n!$: n (here 5) incarnations of `result`
- allocation/deallocation: LIFO $\Rightarrow$ run-time stack
- *dynamic* memory management

Leonardo da Pisa

Martin Steffen

The rabbit problem

Martin Steffen

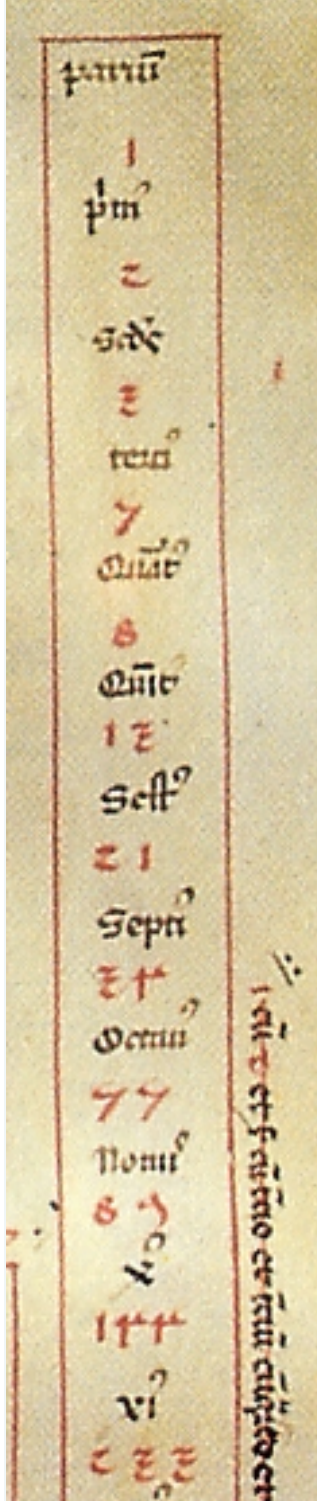
1. A rabbit grows up in *one* month
2. Each grown up pair of rabitt breeds a pair of rabitts each month

Question

Starting with one pair, how many pairs do we have after n months

Fibonacci's solution

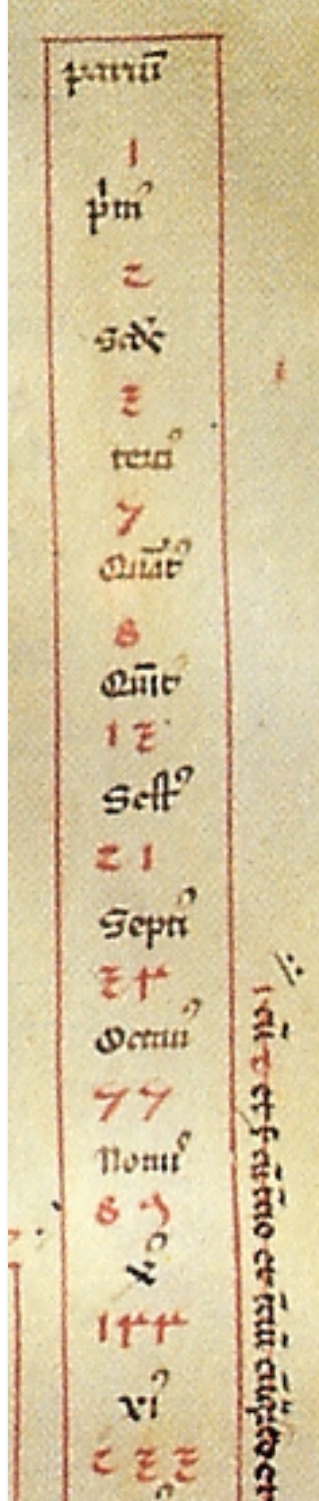
Month	rabbit pairs		
	newborn	grown up	total
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	8
	⋮		



Fibonacci's solution

Month	rabbit pairs		
	newborn	grown up	total
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	8
	⋮		

$$f_n = \begin{cases} 1 & \text{if } n = 0 \text{ or } n = 1 \\ f_{n-1} + f_{n-2} & \text{otherwise} \end{cases}$$

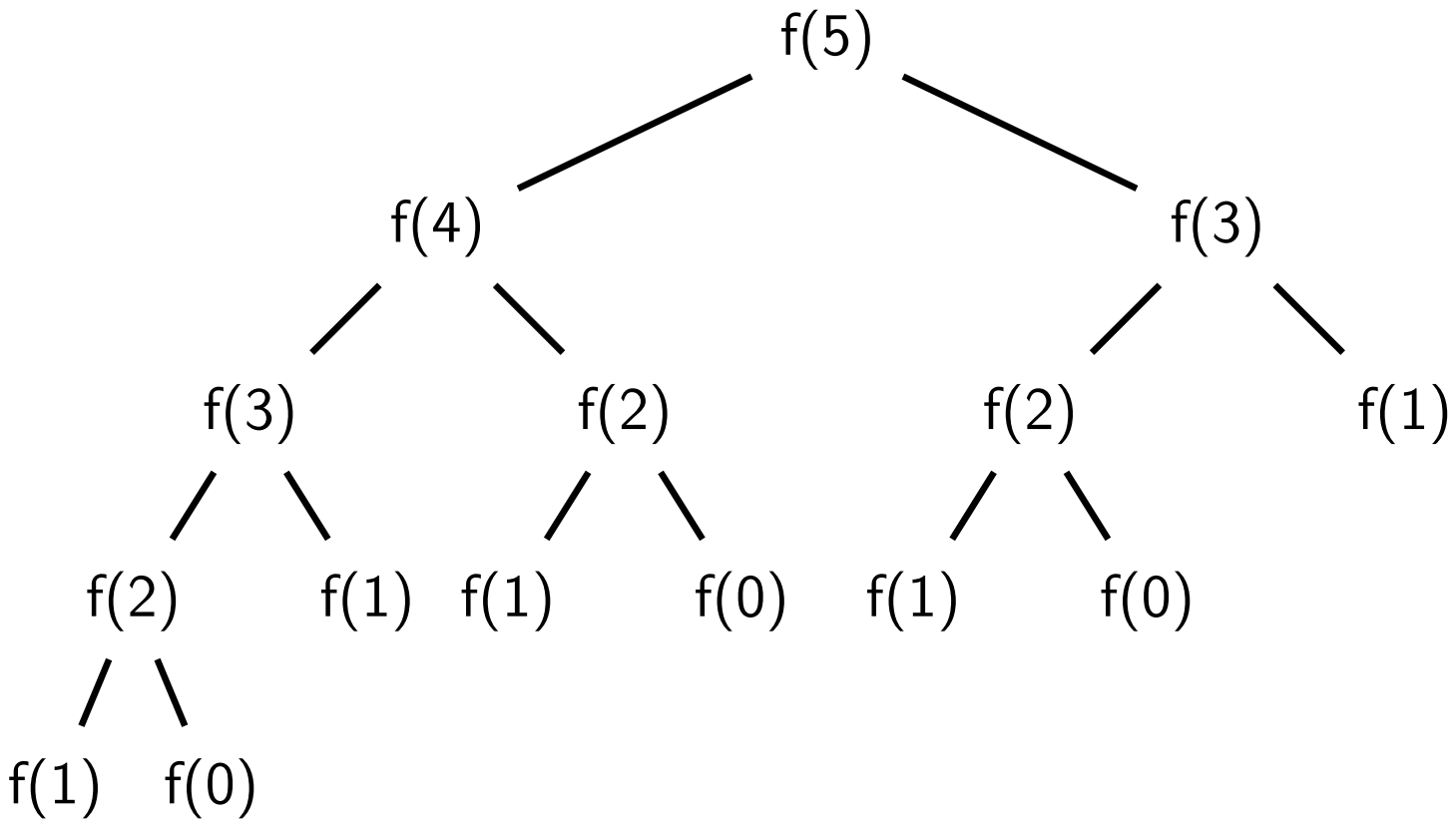


In Java

```
public static int fibonacci(int n) {
  if (n == 0) return 1;                    // base case
  if (n == 1) return 1;                    // base case
  return
    fibonacci (n-1) + fibonacci(n-2); // induction case
  }
```

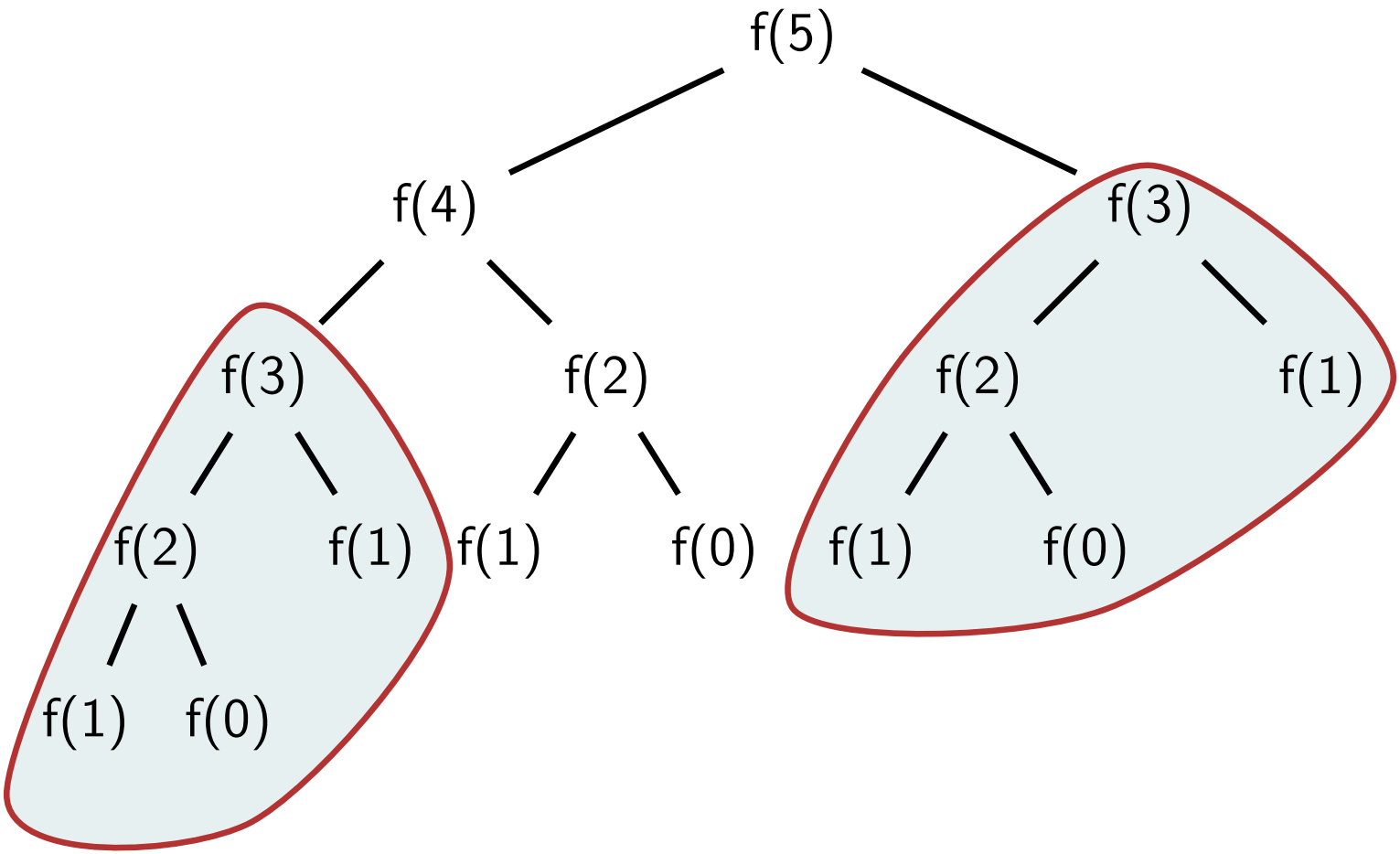
Martin Steffen

Calls



Martin Steffen

Calls



Martin Steffen

As an aside . . .

```
\node{f(5)}
child{node {f(4)}
  child{node {f(3)}
    child{node {f(2)}
      child{node {f(1)}}
      child{node {f(0)}}
    }
    child{node {f(1)}}
  }
  child{node {f(2)}
    child{node {f(1)}}
    child{node {f(0)}}
  }
}
child{node {f(3)}
  child{node {f(2)}
    child{node {f(1)}}
    child{node {f(0)}}
  }
  child{node {f(1)}}
}
;
U:---  fibonaccitree.tex
```

tree = rec. data structure

tree node =

- node without children (“leaf”), or
- with n (here 2) tree nodes as children

Martin Steffen

Trees


```
public class TreeO {
    private TreeO left, right = null;
    private Object data;
```

in practice: tree structure mostly more complex

- list of childred/sub-trees
- instead of `Object`: *"generics"*
- further methods, constructors, pointers
- avoid null-pointer
- interfaces
- ...

Martin Steffen

We can do more efficiently

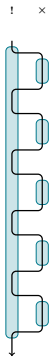
```
public static int fibonacci(int n, int a, int b) {
    if (n == 0) return b;     // let's start with 1
    return fibonacci (n-1,b,a+b);
}
```

- a = "newborn", b = "grown up"
- initial call with `fibonacci`(n, 0, 1)

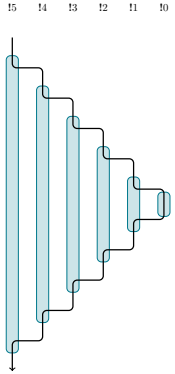
Martin Steffen

Life time of local variables

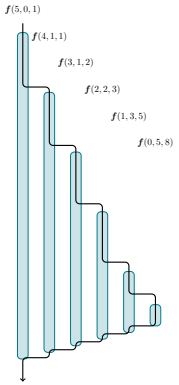
Fac. iteratively



Fac. recursively



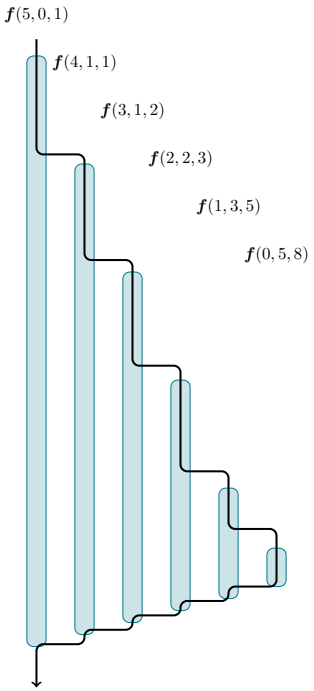
Fib. recursively




```
public static int fibonacci(int n, int a, int b) {
    if (n == 0) return b;     // let's start with 1
    return fibonacci (n-1,b,a+b);
}
```

Martin Steffen

Tail recursion

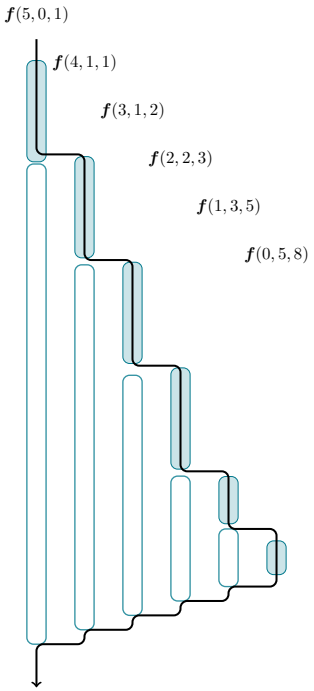


- recursive call: last in the method body

```
public static int fibonacci(int n, int a, int b) {
    if (n == 0) return b;     // let's start with 1
    return fibonacci (n-1,b,a+b);
}
```

Martin Steffen

Tail recursion



- recursive call: last in the method body
- consequently:
  - Stack unnecessary ...
  - iterative-rekursively
- often: compiler optimization (not in Java, but on the language's todo-list)

```
public static int fibonacci(int n, int a, int b) {
    if (n == 0) return b;     // let's start with 1
    return fibonacci (n-1,b,a+b);
}
```

Martin Steffen

“tail-recursive” calls?

```
public static int fibonacci(int n, int a, int b) {
    if (n != 0) return fibonacci (n-1,b,a+b);
    return b;
}
```

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

Martin Steffen

Rekursion vs. Iteration

LEARN ERLANG

absolute beginners

Erlang Tutorial

- Erlang - Home
- Erlang - Overview
- Erlang - Environment
- Erlang - Basic Syntax
- Erlang - Shell
- Erlang - Data Types
- Erlang - Variables
- Erlang - Operators
- Erlang - Loops
- Erlang - Decision Making
- Erlang - Functions
- Erlang - Modules
- Erlang - Recursion
- Erlang - Numbers
- Erlang - Strings
- Erlang - Lists
- Erlang - File I/O
- Erlang - Atoms
- Erlang - Maps

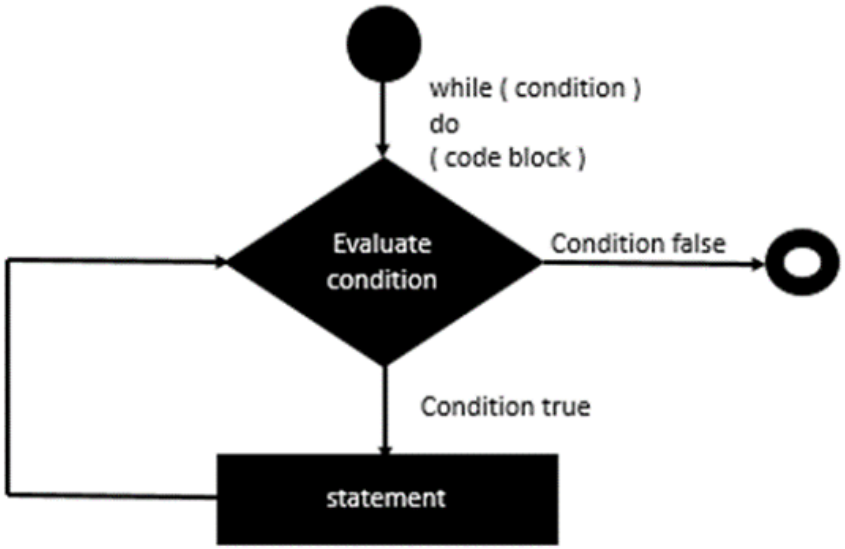
Previous Page Next Page

Erlang is a functional programming language and what needs to be remembered about all functional programming languages is that they don't offer any constructs for loops. Instead, functional programming depends on a concept called recursion.

while Statement Implementation

Since there is no direct while statement available in Erlang, one has to use the recursion techniques available in Erlang to carry out a while statement implementation.

We will try to follow the same implementation of the while loop as is followed in other programming languages. Following is the general flow which will be followed.



Martin Steffen

Rekursion vs. Iteration

Martin Steffen

- Recursion and iteration: in principle *equally expressive*

Recursion, only interesting for “number theoreticians”?

Git: Merge

```
[msteffen@rijkaard mmgo]$ git pull
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 9 (delta 7), reused 9 (delta 7), pack-reused 0
Unpacking objects: 100% (9/9), done.
From github.com:dfava/favasynthesis
   0d9fa6c..154dd44  master     -> origin/master
Merge made by the 'recursive' strategy.
 papers/mmgo/intro.tex | 52 ++++++++++++++++++++++++++++++----------------------
 1 file changed, 30 insertions(+), 22 deletions(-)
[msteffen@rijkaard mmgo]$ ▌
```

Recursion, only interesting for “number theoreticians”?

Martin Steffen

Internet DNS

Recursive DNS is essentially the opposite of Dyn Standard DNS which is an authoritative DNS service that allows others to find *your* domain while Recursive DNS allows *you* to resolve other people’s domains.

The Longer Answer

Recursive DNS provides recursive DNS. Yes, that’s recursive (something which repeats or refers back to itself) and confusing. In order to make a distinction between the service we provide and the general concept of recursive DNS, here’s an explanation.

To better illustrate how recursive DNS works, let’s imagine you are sitting at a computer in your study at home. You’re connected to the Internet by a cable connection and you are surfing the web looking for widgets. You have no idea where to find widgets, so you open your web browser and type in http://www.google.com.

Recursion, only interesting for “number theoreticians”?

Martin Steffen

The multi-million $ heist via recursion[1]

Deconstructing theDAO Attack: A Brief Code Tour

18 JUNE 2016 on thedao, security, ethereum, solidity

TheDAO was attacked today, and the attacker seems to have made off with 3.5mm ether (at time of writing in excess of $45mm). The vulnerability was the Race To Empty or Recursive Call attack.

[1] in virtual money (ether/“blockchain”) and temporary.

Search

Martin Steffen

Goal

- Input: integer-array + number
- Output: if number in the array: index *where*

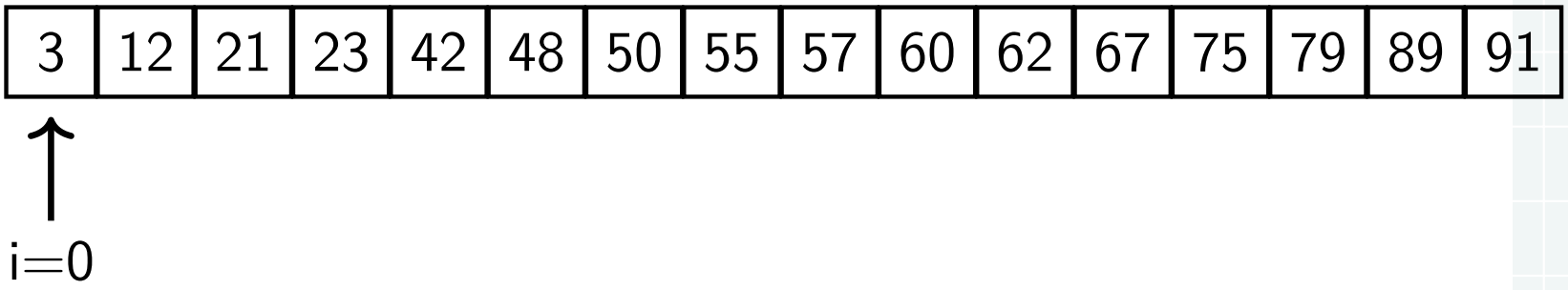
3	12	21	23	42	48	50	55	57	60	62	67	75	79	89	91

Search

Martin Steffen

Goal

- Input: integer-array + number
- Output: if number in the array: index *where*

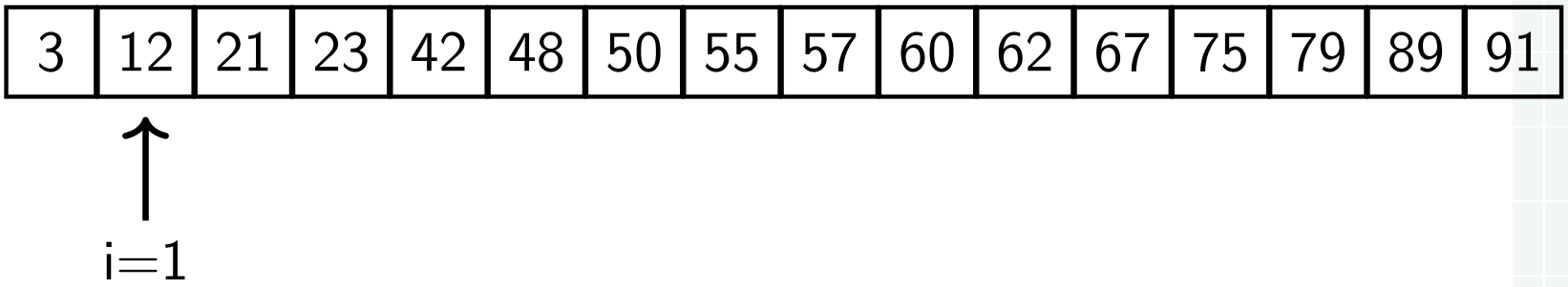


Search

Martin Steffen

Goal

- Input: integer-array + number
- Output: if number in the array: index *where*

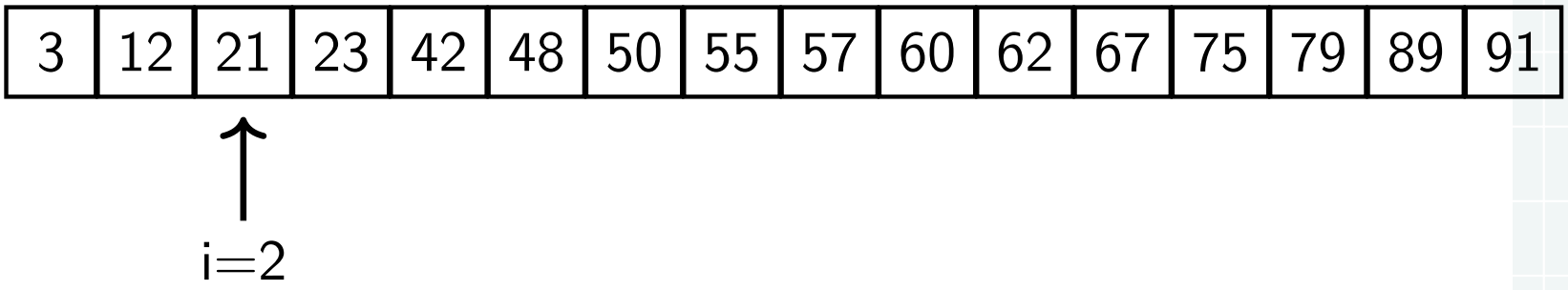


Search

Martin Steffen

Goal

- Input: integer-array + number
- Output: if number in the array: index *where*

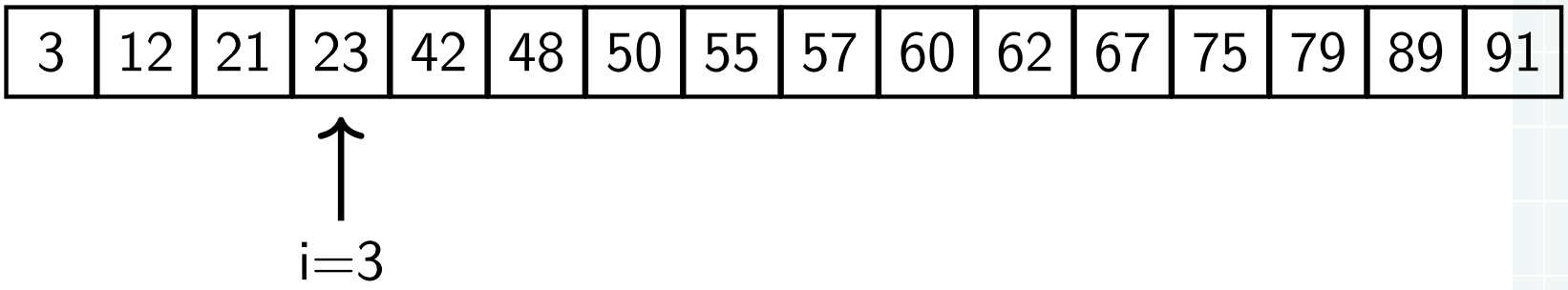


Search

Martin Steffen

Goal

- Input: integer-array + number
- Output: if number in the array: index *where*

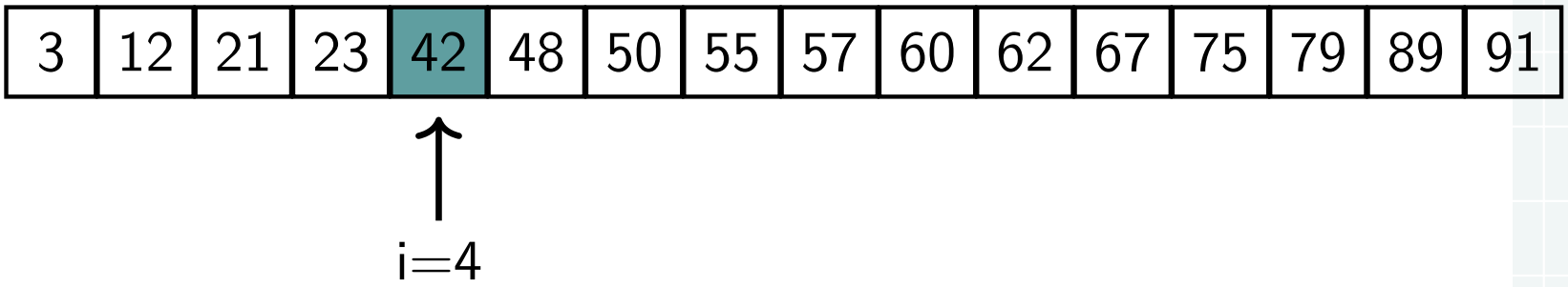


Search

Martin Steffen

Goal

- Input: integer-array + number
- Output: if number in the array: index *where*



We can do better: binary search

Martin Steffen

Goal

- Input: sorted Integer-array + number
- Output: if number in the array: index *where*

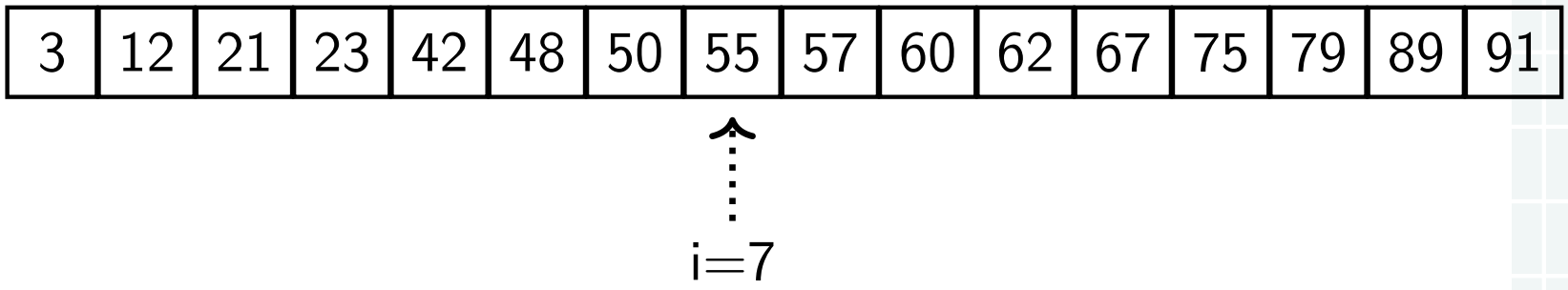
3	12	21	23	42	48	50	55	57	60	62	67	75	79	89	91

We can do better: binary search

Martin Steffen

Goal

- Input: sorted Integer-array + number
- Output: if number in the array: index *where*

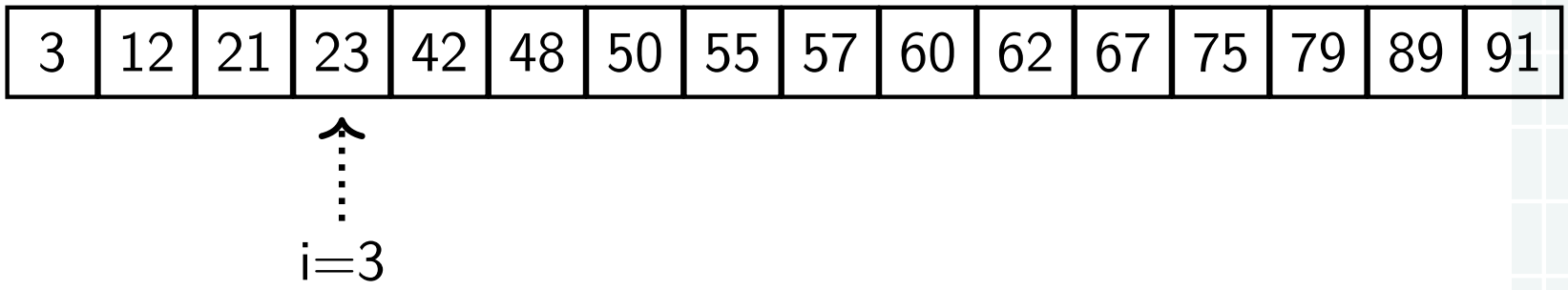


We can do better: binary search

Martin Steffen

Goal

- Input: sorted Integer-array + number
- Output: if number in the array: index *where*

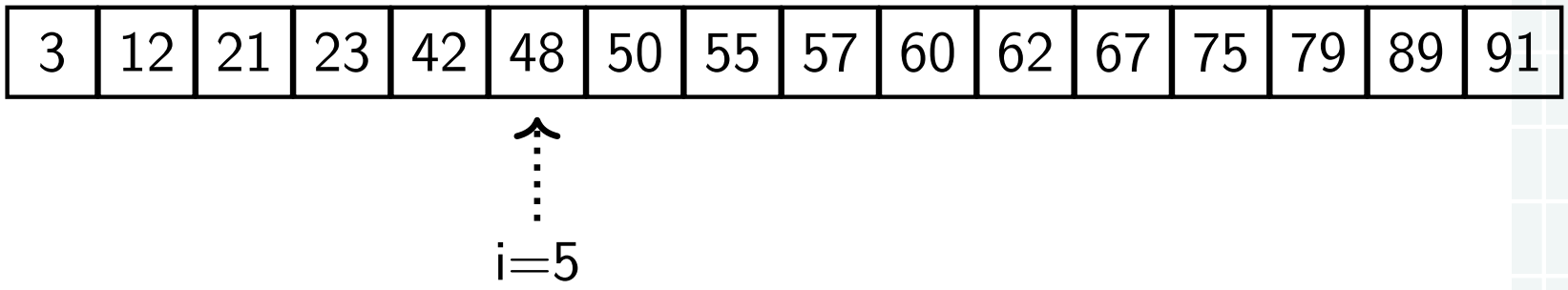


We can do better: binary search

Martin Steffen

Goal

- Input: sorted Integer-array + number
- Output: if number in the array: index *where*

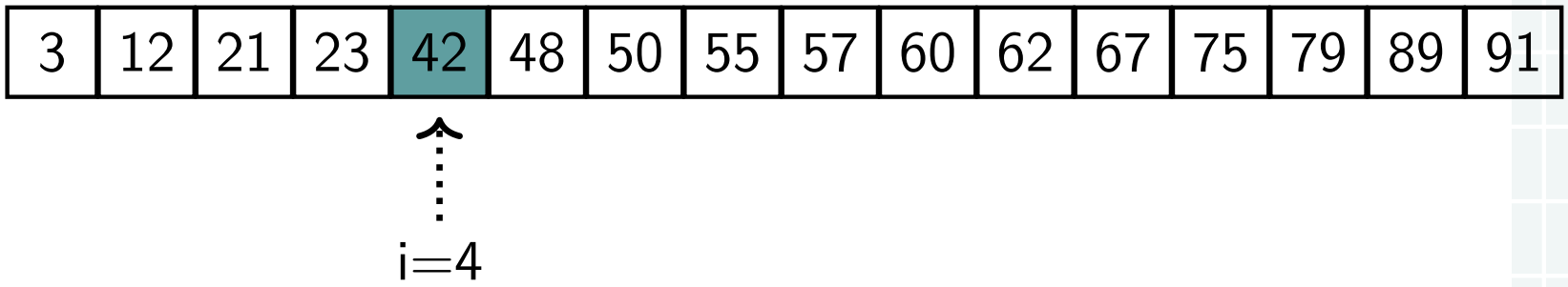


We can do better: binary search

Martin Steffen

Goal

- Input: sorted Integer-array + number
- Output: if number in the array: index *where*



Recursive approach

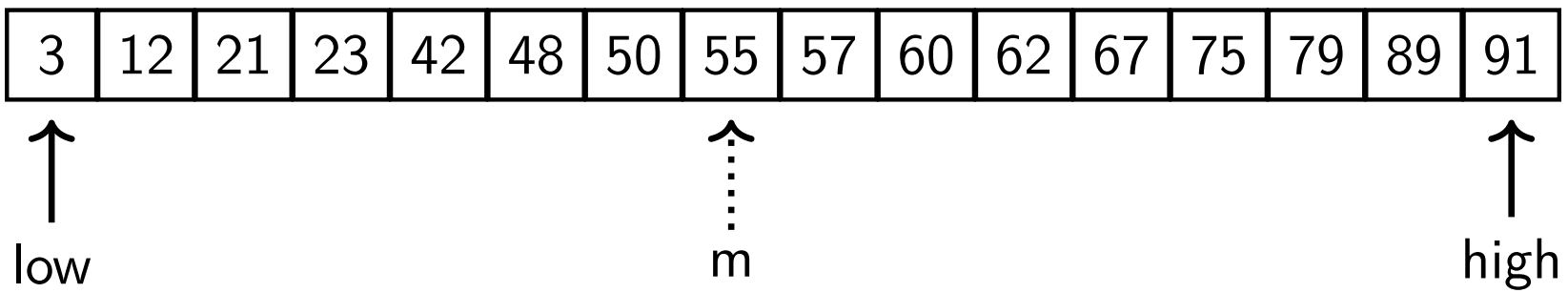
Martin Steffen

Divide & conquer: search for `elem`

- look up in the *middle* of the array
- if equal to `elem` $\Rightarrow$ done
- if smaller than `elem` $\Rightarrow$ search rekursively in the right half
- if larger than `elem` $\Rightarrow$ search rekursively in the left half

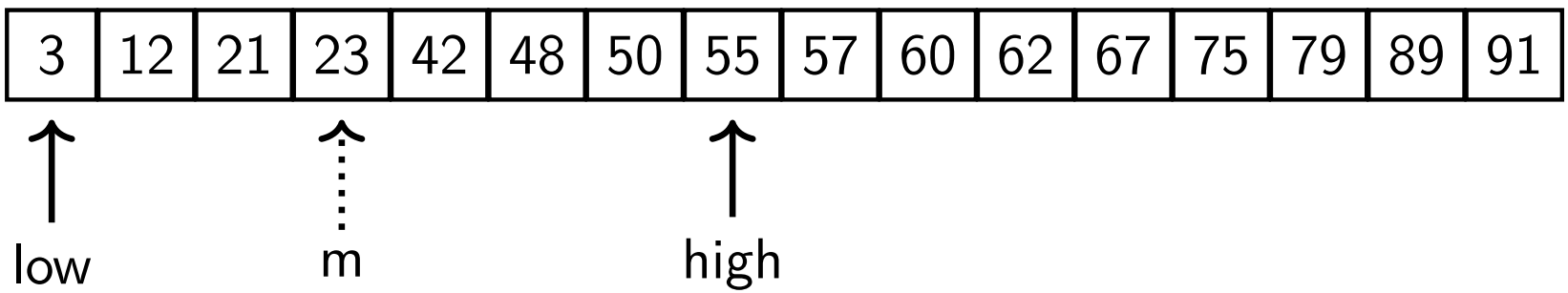
In Java

```
static int search (int elem, int[] a, int low, int high) {
    System.out.println(low);
    System.out.println(high);
    if (low == high) {
        if (elem == a[low]) {
            return low;
        } else {
            return -1;
        }
    } else {  // low ≠ high
        int m = (low + high) / 2;
        if (elem < a[m]) {
            return search(elem, a, low, m-1);
        } else {
            return search(elem, a, m+1, high);
```



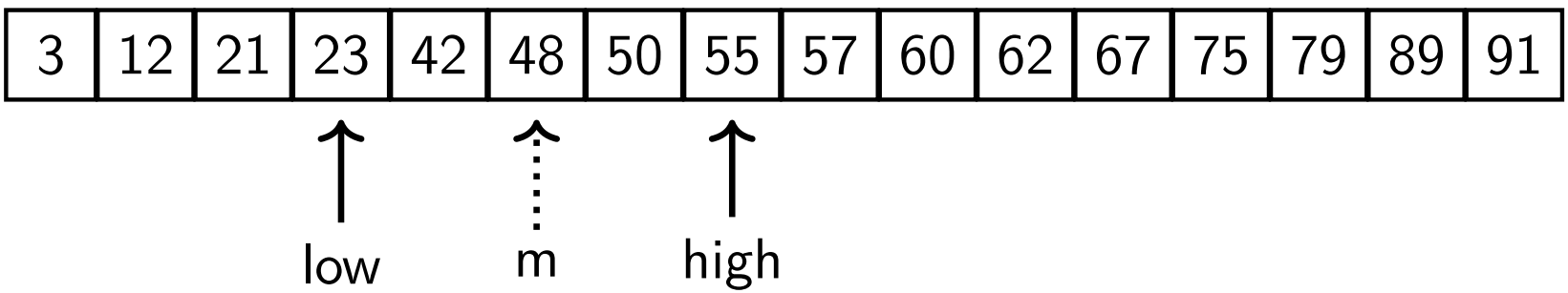
In Java

```
static int search (int elem, int[] a, int low, int high) {
    System.out.println(low);
    System.out.println(high);
    if (low == high) {
        if (elem == a[low]) {
            return low;
        } else {
            return -1;
        }
    } else {  // low ≠ high
        int m = (low + high) / 2;
        if (elem < a[m]) {
            return search(elem, a, low, m-1);
        } else {
            return search(elem, a, m+1, high);
```



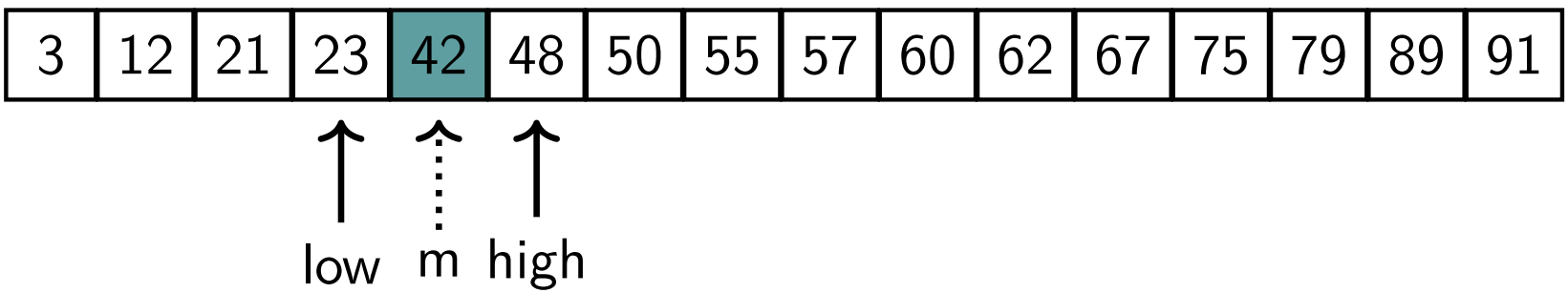
In Java

```
static int search (int elem, int[] a, int low, int high) {
    System.out.println(low);
    System.out.println(high);
    if (low == high) {
        if (elem == a[low]) {
            return low;
        } else {
            return -1;
        }
    } else {  // low ≠ high
        int m = (low + high) / 2;
        if (elem < a[m]) {
            return search(elem, a, low, m-1);
        } else {
            return search(elem, a, m+1, high);
```



In Java

```
static int search (int elem, int[] a, int low, int high) {
    System.out.println(low);
    System.out.println(high);
    if (low == high) {
        if (elem == a[low]) {
            return low;
        } else {
            return -1;
        }
    } else {  // low ≠ high
        int m = (low + high) / 2;
        if (elem < a[m]) {
            return search(elem, a, low, m-1);
        } else {
            return search(elem, a, m+1, high);
```



“Correctness”

Argument

1. The “divide-and-conquer” idea seems plausibly sound
2. Termination
   - each recursive call renders the problem smaller (induktion case)
   - there is a smallest problem (base case)

Martin Steffen

“Correctness”

Argument

1. The “divide-and-conquer” idea seems plausibly sound
2. Termination
   - each recursive call renders the problem smaller (induktion case)
   - there is a smallest problem (base case)
   - Alas: it only looks like that . . .

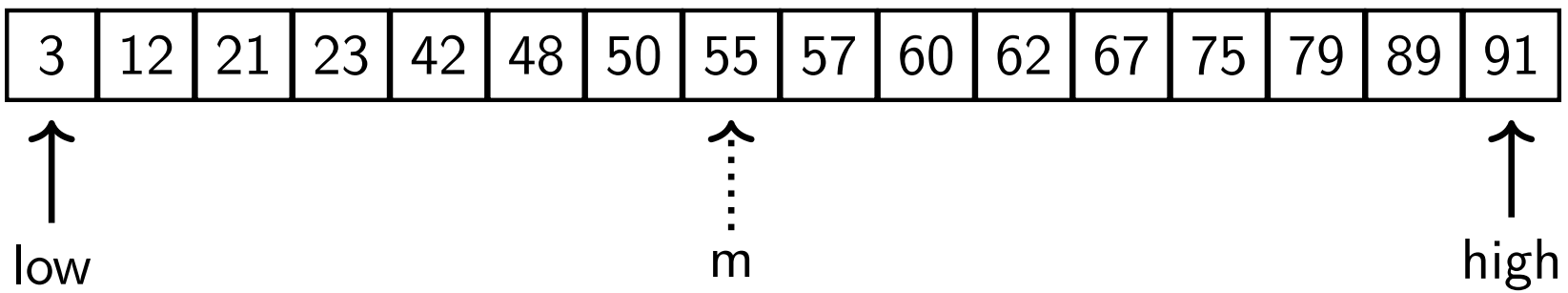
```
-*- mode: compilation; default-directory: "~/javaexamples/" -*-
Compilation started at Wed Oct 11 15:58:48

java Binsearch
Exception in thread "main" java.lang.StackOverflowError
        at Binsearch.search(Binsearch.java:12)
        at Binsearch.search(Binsearch.java:12)
        at Binsearch.search(Binsearch.java:12)
        at Binsearch.search(Binsearch.java:12)
        at Binsearch.search(Binsearch.java:12)
-:%*-  *compilation*   Top L1     [(Compilation:exit [1] Abbrev)]
```

Martin Steffen

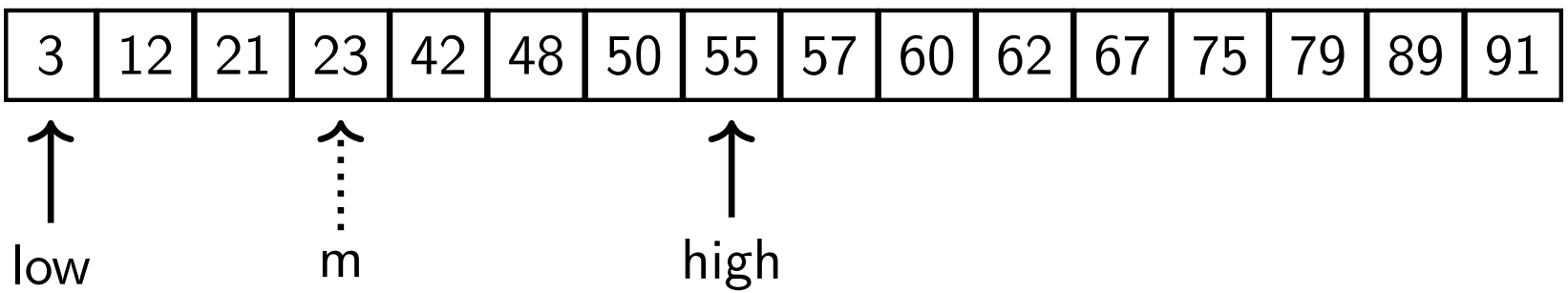
On second thought . . .

```
static int search (int elem, int[] a, int low, int high) {
    if (low > high) return -1; // empty
    int m = (low + high) / 2;
    if (elem == a[m]) return m;
    if (elem < a[m]) {
        return search(elem, a, low, m-1);
    } else {
        return search(elem, a, m+1, high);
    }
}
```



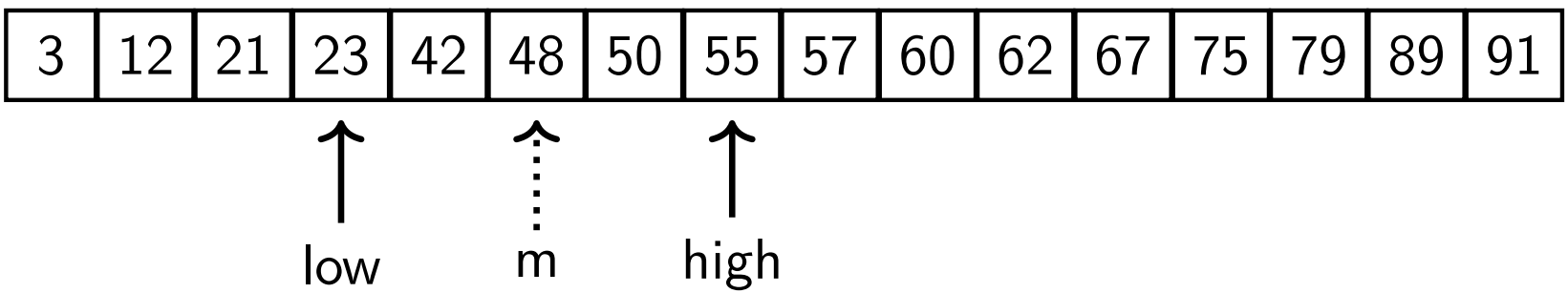
On second thought . . .

```
static int search (int elem, int[] a, int low, int high) {
    if (low > high) return -1; // empty
    int m = (low + high) / 2;
    if (elem == a[m]) return m;
    if (elem < a[m]) {
        return search(elem, a, low, m-1);
    } else {
        return search(elem, a, m+1, high);
    }
}
```



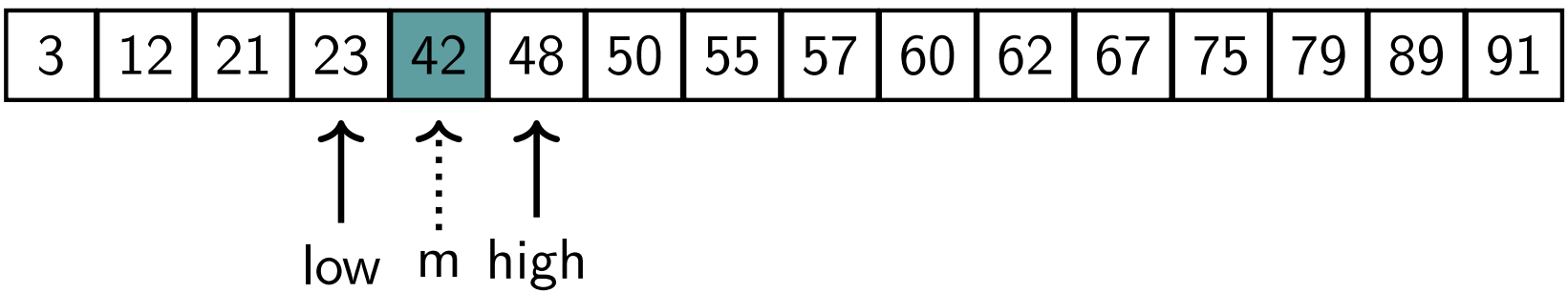
On second thought . . .

```
static int search (int elem, int[] a, int low, int high) {
    if (low > high) return -1; // empty
    int m = (low + high) / 2;
    if (elem == a[m]) return m;
    if (elem < a[m]) {
        return search(elem, a, low, m-1);
    } else {
        return search(elem, a, m+1, high);
    }
}
```



On second thought . . .

```
static int search (int elem, int[] a, int low, int high) {
    if (low > high) return -1; // empty
    int m = (low + high) / 2;
    if (elem == a[m]) return m;
    if (elem < a[m]) {
        return search(elem, a, low, m-1);
    } else {
        return search(elem, a, m+1, high);
    }
}
```



The bug that “fixed itself”

```
while (days > 365) {
  if (IsLeapYear(year)) {
    if (days > 366) {
      days -= 366;
      year += 1;
    }
  } else {
    days -= 365;
    year += 1;
  }
}
```

Martin Steffen

Perhaps beginner's glitches only . . . ?

Proving that Android's, Java's and Python's sorting algorithm is broken (and showing how to fix it)

February 24, 2015 Envisage Written by Stijn de Gouw. $s

Tim Peters developed the Timsort hybrid sorting algorithm in 2002. It is a clever combination of ideas from merge sort and insertion sort, and designed to perform well on real world data. TimSort was first developed for Python, but later ported to Java (where it appears as java.util.Collections.sort and java.util.Arrays.sort) by Joshua Bloch (the designer of Java Collections who also pointed out that most binary search algorithms were broken). TimSort is today used as the default sorting algorithm for Android SDK, Sun's JDK and OpenJDK. Given the popularity of these platforms this means that the number of computers, cloud services and mobile phones that use TimSort for sorting is well into the billions.

Martin Steffen

And as consequence?

- hands off recursion?
- base cases (and special cases) are particular *error prone* ("one-off" errors)
- working in most cases $\neq$ correct

> "Program testing can be used to show the presence of bugs, but never to show their absence"
>
> Dijkstra 1970, p. 7

- ultimately: careful reasoning needed ("correctness proof") angesagt.

Martin Steffen

Further issues

Martin Steffen

- in-direct recursion ("call-backs")
- induction & recursion
- inductive/rekursive data structures and corresponding algorithms (e.g. trees)
- complexity
- ...

Kudos

Martin Steffen

The slides where done with

- gnu emacs *org-mode* ("gnu's not Unix")
- LaTeX, und
- TikZ ("TikZ ist kein Zeichenprogram")

The "design" owes inspiration the elaborate style-files of the Uni Lübeck (M. Leucker, V. Stolz).

Sources

Martin Steffen

Most of the material is "common knowledge" and I did not base the lecture on any specific book or source. Similar examples can be found in basically all introductions to Java, or other programming languages, for that matter. Pictures, if not self-made graphics, are likewise from "creative commons". Particular internet finds might be clickable via embedded links.