Introductory Programming
Recursion

IMM, DTU

a. Parts of this material are inspired by/originate from a course at ITU developed by Niels Hallenberg and Peter Sestoft on the basis of a course at KVL developed by Morten Larsen and Peter Sestoft. Translated into English and edited by Anne Haxthausen.

Recursion: methods that call themselves

A method can invoke another method.

A method can also invoke itself. Such a method is said to be *recursive*.

Example: factorial function

The function $n!$, pronounced n factorial, gives the number of permutations of n items.
E.g. 3 persons can sit on a bench in $3! = 3 \cdot 2 \cdot 1 = 6$ different permutations.

The factorial function $n!$ can be defined as follows

$$n! = n \cdot (n-1) \cdot \ldots \cdot 1$$

However, the factorial function can also be defined recursively:

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

The first definition is most easily implemented as a loop (i.e. as an iterative method).

The second definition is most easily implemented as a method that calls itself (i.e. as a recursive method).

Iterative and recursive calculations of $n!$

```
public class Factorial {
  public static void main(String args[]) {
    int n = Integer.parseInt(args[0]);
    System.out.println(n + "! is " + ifac(n));
    System.out.println(n + "! is " + rfac(n));
  }

  static int ifac(int n) {       // Iterative factorial function
    int result = 1;
    for (int i=n; i >= 1; i=i-1)
      result = result * i;
    return result;
  }

  static int rfac(int n) {       // Recursive factorial function
    if (n == 0) return 1;
    else
      return n * rfac(n - 1);
  }
}
```

How `rfac(3)` is evaluated in the machine

```
    rfac(3)
→   3 * rfac(2)
→   3 * (2 * rfac(1))
→   3 * (2 * (1 * rfac(0)))
→   3 * (2 * (1 * 1))
→   3 * (2 * 1)
→   3 * 2
→   6
```

During the evaluation, the recursive method invocations require allocation of memory for not completed expressions.

Recursive method invocations are a little slower than repetitions in a loop.

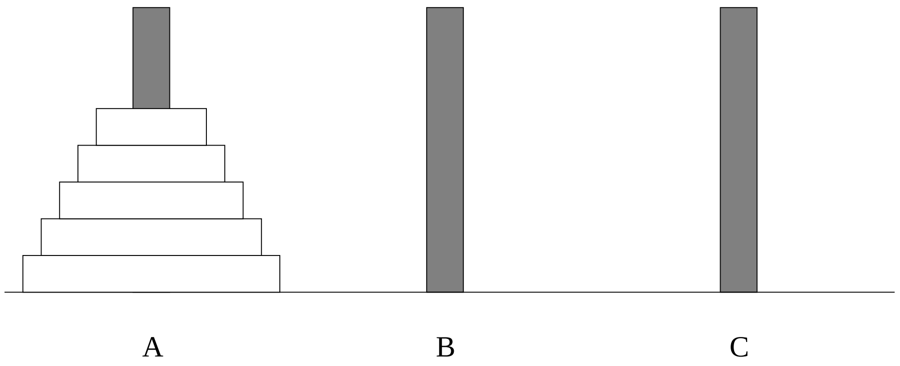
Some problems are more elegantly solved using recursive methods

Example: the towers of Hanoi

All disks should be moved from tower A to tower B.

Only move one disk at a time.

Never place a larger disk on the top of a smaller disk.



Example: the towers of Hanoi, algorithm

Move n disks from tower A to tower B via an extra tower C:
1. Move the $n-1$ topmost disks from tower A to tower C.
2. Move a disk from tower A to tower B.
3. Move the $n-1$ topmost disks from tower C to tower B.

To make step 1 use the recipe replacing 'n' with '$n-1$', 'B' with 'C' and 'C' with 'B':

Move $n-1$ disks from tower A to tower C via an extra tower B:
1. Move the $n-2$ topmost disks from tower A to tower B.
2. Move a disk from tower A to tower C.
3. Move the $n-2$ topmost disks from tower B to tower C.

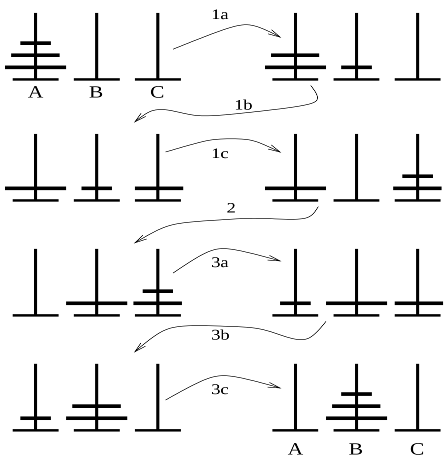
Similarly, to make step 3 replace 'n' with '$n-1$', 'A' with 'C' and 'C' with 'A':

Move $n-1$ disks from tower C to tower B via an extra tower A:
1. Move the $n-2$ topmost disks from tower C to tower A.
2. Move a disk from tower C to tower B.
3. Move the $n-2$ topmost disks from tower A to tower B.

In order to move zero disks you do not need to do anything.

Hanoi – example with 3 disks

1. Move disk 1, 2 from A to C via B
   (a) Move disk 1 from A to B
   (b) Move disk 2 from A to C
   (c) Move disk 1 from B to C
2. Move disk 3 from A to B
3. Move disk 1, 2 from C to B via A
   (a) Move disk 1 from C to A
   (b) Move disk 2 from C to B
   (c) Move disk 1 from A to B



Example: ‘the towers of Hanoi’ in Java (`Hanoi.java`)

```
public class Hanoi {

  public static void move(int n, String from, String to, String via) {
    if (n > 0) {
      move(n-1, from, via, to);
      System.out.println("Move disk " + n +
                         " from " + from + " to " + to);
      move(n-1, via, to, from);
    }
  }

  public static void main(String args[]) {
    int n = Integer.parseInt(args[0]);

    System.out.println("Moving " + n + " disks from A to B via C:");
    System.out.println();
    move(n, "A", "B", "C");
  }
}
```


Main idea in recursion

Some problems can be solved by solving one or more subproblems of the same kind.

E.g. the towers of Hanoi: move a stack of n disks by

- moving a stack of $n-1$ disks,
- moving 1 disk, and
- moving a stack of $n-1$ disks

Such a strategy for problem solving is called *‘divide-and-conquer’* (Danish: ‘Del og hersk’).

It is well-suited for implementation with recursive methods.

The strategy only works if:

- each subproblem is simpler than the original one, and
- it terminates with a trivial subproblem (e.g. move a stack of 0 disks)