Recursion

Overview: Recursion

Recursion:

- Recursive methods
- Base case
- Recursive cases
- Depth of recursion
- Infinite recursion
- Method Activations
- Different types of recursion:
  - Direct recursion
  - Indirect recursion
  - Simple recursion
  - Tail recursion
- Iteration vs. recursion

Examples:

- Adviser
- Factorial function
- Binary search
- Tower of Hanoi

Recursion

- *Recursive definition*: a definition which is defined in terms of itself.
  - Definition of a statement i Java is a recursive definition: for example, a `for` loop can have a `for` loop i the loop body, and this inner loop can have a `for` loop in its loop body, and so on.

```
for (...; ...; ...) {
    for (...; ...; ...) {
        for (...; ...; ...) {
        ...
        }
    }
}
```

statement --> for (...) *statement*

  - Factorial function can be defined by a recursive definition:

    n! = n * (n-1)!

    0! = 1

- *Recursive method*: A method which is defined with a call to itself in the method body.

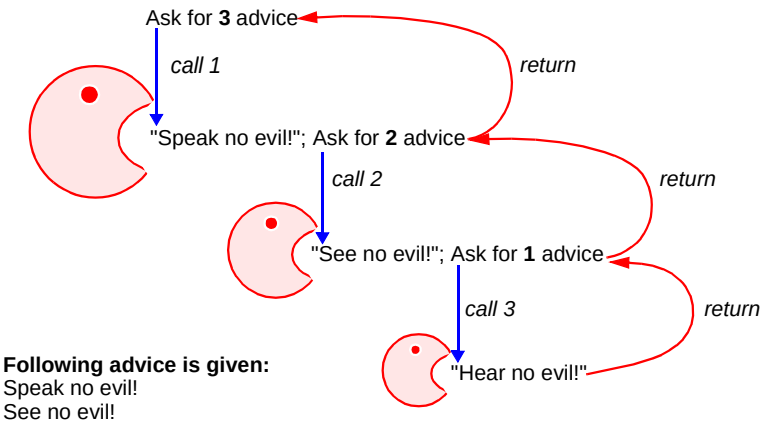
```
public void callMe(String msg) { ... callMe("Play it again, SaM") ... }
// direct recursion
```


Recursion as problem solving technique

- Model problems which can be split into smaller, simpler subtasks which all resemble the original problem.
  - each subtask can be solved by applying the same technique.
- The whole problem is solved by combining the solutions to the subtasks.
  - It is an example of *divide and conquer.*
- In Java we model this with the help of recursive methods.
  - requires a *base case* to terminate the recursion.

Example: Adviser

Sequence of recursive calls (and returns)



Following advice is given:
Speak no evil!
See no evil!
Hear no evil!

Tail recursion: Last statement executed is a recursive call.

Example: Adviser (continued)

Version 1:

```
class AdviserI {
    public void askAdvice(int noOfAdvice) {
        // give one advice:
        giveOneAdvice();
        // and ask for more advice:
        askAdvice(noOfAdvice - 1); // infinite recursion!
    }
    private void giveOneAdvice() {
        // code to dispense one advice
    }
}
public class ClientI {
    public static void main(String args[]) {
        AdviserI adviser = new AdviserI();
        adviser.askAdvice(3);
    }
}
```

- Logical error: A new advice is always given, followed by a new call to the method `askAdvice`.
  - We have *infinite recursion*.

Designing a recursive (`void`) method definition (*no return value*)

- One or more cases which do not use a recursive call.
  - These cases are called *base cases*.
  - Base cases stop the recursion.
- One or more cases which use a *recursive call*.
  - Recursive calls solve even smaller versions of the same problem.

Example: Adviser (continued)

Version 2:

```
class AdviserII {
    public void askAdvice(int noOfAdvice) {
        if (noOfAdvice == 1) { // base case: stops recursion
            // give one advice:
            giveOneAdvice();
       } else {// recursive case: divide into smaller, similar subtasks.
            // give one advice:
            giveOneAdvice();
            // and ask for more advice:
            askAdvice(noOfAdvice - 1);
        }
    } // ...
}
public class ClientII {
    public static void main(String args[]) {
        AdviserII adviser = new AdviserII();
        adviser.askAdvice(3);
    }
}
```

Design by recursion

- The logic in the method must result in a *base case* to be executed: the sequence of smaller and smaller subtasks must *converge* to a base case.
  - This is normally achieved by a conditional statement in a condition stops the recursion.
  - This condition is usually dependent on a parameter of the method.

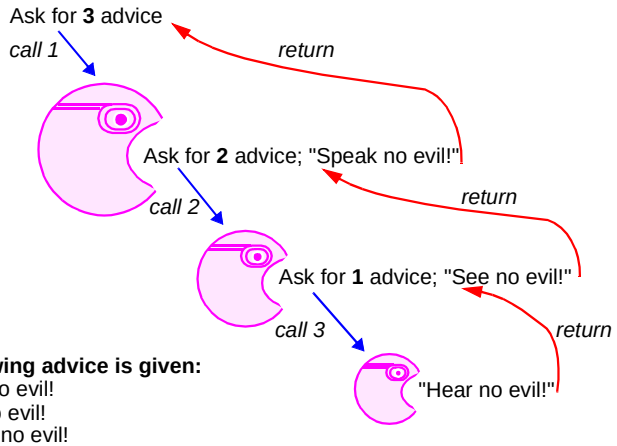
Example: Adviser (continued)

```
class AdviserIII {
    public void askAdvice(int noOfAdvice) {
        if (noOfAdvice == 1) { // base case: stops recursion
            giveOneAdvice(1);
        } else { // recursive case: divide into smaller, similar subtasks.
            giveOneAdvice(noOfAdvice);
            askAdvice(noOfAdvice - 1);
        }
    }
    private void giveOneAdvice(int adviceNo) {
        switch (adviceNo) {
            case 1:  System.out.println("Hear no evil!");   break;
            case 2:  System.out.println("See no evil!");    break;
            case 3:  System.out.println("Speak no evil!");  break;
            default: System.out.println("Sorry, no advice.");
        } // switch
    }
}
public class ClientIII {
    public static void main(String args[]) {
        AdviserIII adviser = new AdviserIII();
        adviser.askAdvice(3);
    }
}
```

Ask for more that 3 advice. What happens?

Example: New Adviser

Sequence of recursive calls (and returns)



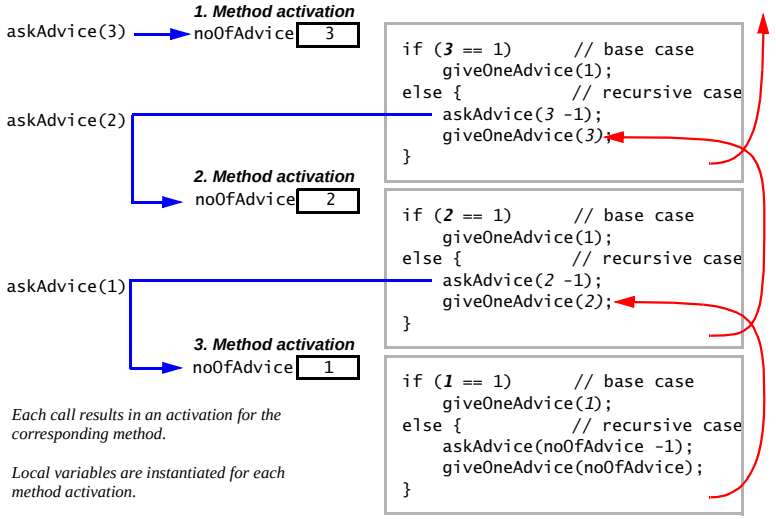
Following advice is given:
Hear no evil!
See no evil!
Speak no evil!

Example: New Adviser (continued)

```
class NewAdvisorIII {
    public void askAdvice(int noOfAdvice) {
        if (noOfAdvice == 1) { // base case: stops recursion
            giveOneAdvice(1);
        } else { // recursive case: divide into smaller, similar subtasks.
            askAdvice(noOfAdvice - 1);
            giveOneAdvice(noOfAdvice);
        }
    }
    private void giveOneAdvice(int adviceNo) {
        switch (adviceNo) {
            case 1:  System.out.println("Hear no evil!");   break;
            case 2:  System.out.println("See no evil!");    break;
            case 3:  System.out.println("Speak no evil!");  break;
            default: System.out.println("Sorry, no advice.");
        } // switch
    }
}
public class NewClientIII {
    public static void main(String args[]) {
        NewAdvisorIII adviser = new NewAdvisorIII();
        adviser.askAdvice(3);
    }
}
```

Ask for more that 3 advice. What happens?

Example: New Advisor - Execution



Method Activations

- *Recursive Methods*:
  - An object can invoke (different) methods on itself, but now a method call must lead to this method being called again, before the previous call of this method has completed.
  - The method must also handle smaller versions of the original problem.
- *Method Activations*:
  - A *method activation frame* is created which stores the local variables (inclusive parameters) each time the method is called.
  - The method activation exits as long as the method is executed.
- Consequences of using recursion:
  - Many activations for a method can exist at the same time.
  - Each activation has its own instances of local variables (inclusive parameters).
- *Recursion depth*: number of times a method is called recursively before a base case is executed.

Check-list for recursion

- Check that *infinite recursion* does not occur.
- Check that each *base case* executes the correct actions for the case.
- Check that *for each case which uses recursion, if* all recursive calls execute their correct actions, *then* the whole case is executed correctly.

Example: Factorial function

Problem: Given `n` parking places and `n` cars, find the number of *permutations* in which to park the cars.

Iterative definition:

```
n! = n * (n-1) * ... * 2 * 1
```

Recursive definition:

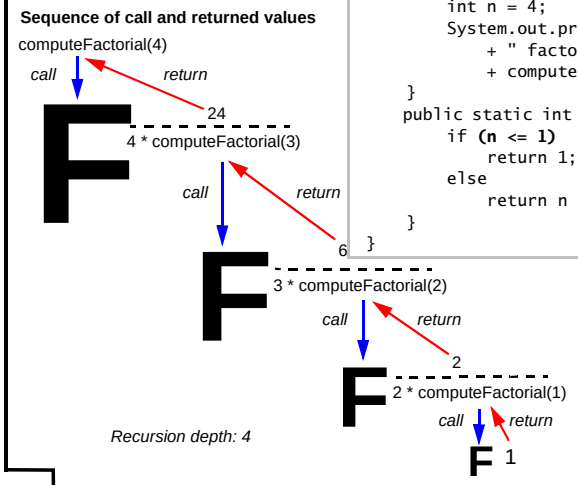
```
n! = n * (n-1)!
0! = 1
```

Factorial function: iterative solution

```
public class FI { //Factorial Iterative
    public static void main(String args[]) {
        int n = 4;
        System.out.println(n + " factorial is " +
                           computeFactorial(n));
    }
    public static int computeFactorial(int n) {
        // Watch out for "overflow".
        int result = 1;
        for (int i = n; i > 1; i--) {
            result *= i;
        }
        return result;
    }
}
```


Factorial function: recursive solution

```
public class FR { // Factorial Recursive
    public static void main(String args[]) {
        int n = 4;
        System.out.println(n
            + " factorial is "
            + computeFactorial(n));
    }
    public static int computeFactorial(int n) {
        if (n <= 1)
            return 1;
        else
            return n * computeFactorial(n-1);
    }
}
```



Designing a recursive method definition (*with return value*)

- One or more cases where the return value is computed without the use of a recursive call.
  - These cases are called *base cases*.
- One or more cases where the return values is computed with a *recursive call*.
  - Recursive calls solve even smaller versions of the same problem.

Some terminology

- Direct recursion:
  - definition of a method A contains (one or more) calls to method A.
  - Example: all examples in these notes.
- Indirect recursion:
  - One or more methods cooperate recursively instead of just one.
  - Methods can be implemented in the same or in different objects.
  - For example, definition of a method A contains (one or more) calls to method B (in the same or different objects) which in turn has a call to method A.
- *Simple recursion*:
  - Call to a method A leads to a recursive call of method A, in other words no "splitting" of recursion – each subtasks leads to only one other smaller subtask.
  - Example of simple recursion: Adviser, Factorial function, Binary search.
  - Example of splitting recursion: Tower of Hanoi.
- *Tail recursion*:
  - definition of a method is such that all recursive calls occur as the last statement which is executed before return from the method.
  - Example: Adviser (but not New Adviser), Factorial function, Binary search.

Comparison of recursion and iteration

- Recursion is usually more expensive because it can involve many method calls.
- Recursion is a concise and elegant solution for problems that naturally lend themselves to this technique.
- *Simple recursion* and *iteration* are computationally equivalent (see Factorial example).

	Iteration	**Recursion**
Based on a control structure:	• uses a loop structure (`for`, `while`, `do-while`)	• uses a conditional structure (`if`, `if-else`, `switch`)
Uses repetition:	• *Explicit* use of a loop structure for repetition	• Repeated method calls to achieve repetition.
Stop criteria:	• stop when the loop condition is false • can lead to *infinite loop*	• stop when a base case is executed • can lead to *infinite recursion*
Convergence towards termination:	• counter-based repetition converges towards final value which terminates the loop	• simpler versions of subtasks converge towards a base case which terminates recursion

Binary search - iterative solution

```
public class BinarySearchII {
    public static void main(String args[]) {
       int a[] = {11,19,24,30,32,48,50,55,72,99}; // sorted in increasing order.
        int key = 48;
        int index = binSearch(a, key);
        if (index >= 0)
            System.out.println("Value " + key +
                               " found in element with index " + index);
        else
            System.out.println("Value " + key + " not found!");
    }

    public static int binSearch(int a[], int key) {
        int l = 0, h = a.length - 1, m = -1;
        for (;;) {
            m = (l + h) / 2;                  // middle of array
            if (l > h) return -1;             // key not found
            else if (a[m] == key) return m;// key found in a[m]
           else if (a[m] < key) l = m + 1;// if present, key is in the upper half
            else h = m - 1;                // if present, key is in the lower half
        }
    }
}
```

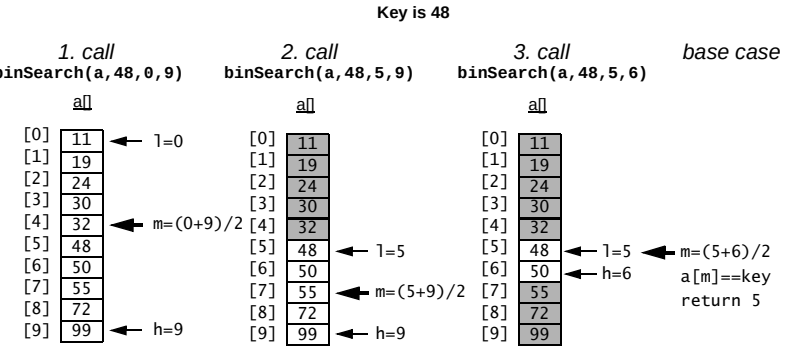

Binary search - recursive solution

```
public class BinarySearchIII {
    public static void main(String args[]) {
       int a[] = {11,19,24,30,32,48,50,55,72,99}; // sorted in increasing order.
        int key = 48;
        int index = binSearch(a, key, 0, a.length - 1);
        if (index >= 0)
            System.out.println("Value " + key +
                               " found in element with index "+ index);
        else
            System.out.println("Value " + key + " not found!");
    }

    public static int binSearch(int a[], int key, int l, int h) {
        int m = (l + h) / 2;                  // middle of array
        if (l > h) return -1;             // base case: key not found
        else if (a[m] == key) return m;  // base case: key is in a[m]
        else if (a[m] < key)             // if present, key is in the upper half
            return binSearch(a, key, m + 1, h);
        else                             // if present, key is in the lower half
            return binSearch(a, key, l, m - 1);
    }
}
```

See check-list for recursion!

Binary search (continued)



Binary Search Analysis

Given `n` items.

Call no.	No. of items left to search
1	n /2
2	n /4
3	n /16
...	...
m	$n / 2^m$

In the worst case (i.e. the key is not found) the recursive calls stop when there are no more elements left to search.

If `m` is the maximum no. of recursive calls for the worst case, then `m` is the smallest integer s.t.

$n / 2^m < 1 \Rightarrow n < 2^m \Rightarrow \log n < m$

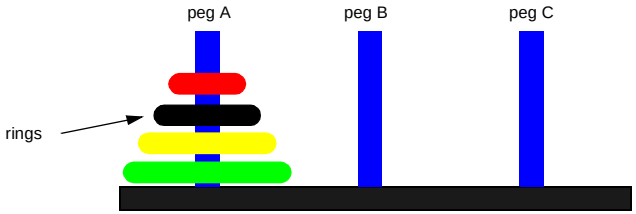
$m > \log n$

$m = \lfloor \log n \rfloor + 1$

In the worst case, binary search runs in $O(\log n)$ times.

Tower of Hanoi

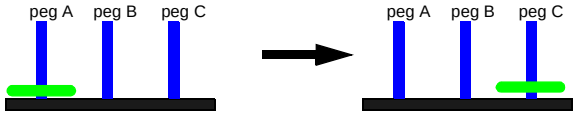
- Game devised by the French mathematician Edouard Lucas in 1883.



- Objective: move the tower with *n* rings of different sizes from peg A to peg C (using peg B).
- Rule 1: No ring can be placed on a smaller ring.
- Rule 2: Only one ring can be moved at a time.

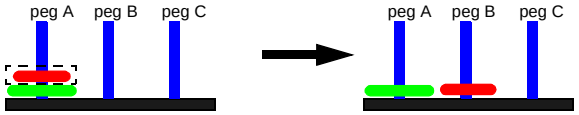
Tower of Hanoi: recursive algoritme

- One ring:
  - move the ring from peg A to peg C

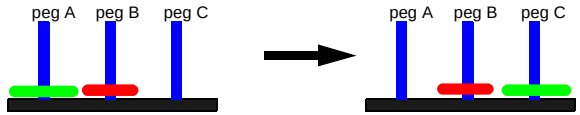


Tower of Hanoi: recursive algoritme (continued)

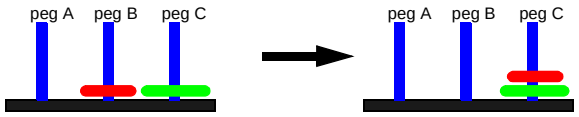
- Two rings:
  - use one ring solution to move the top ring from peg A to peg B:



  - use one ring solution to move the ring on the bottom from peg A to peg C:

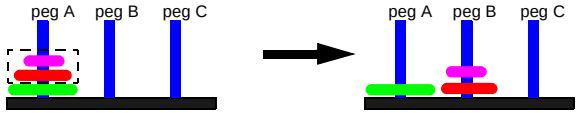


  - use one ring solution to move the ring from peg B to peg C

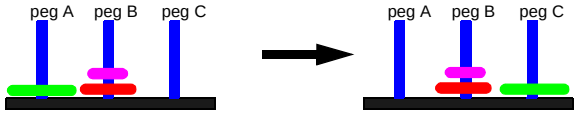


Tower of Hanoi: recursive algoritme (continued)

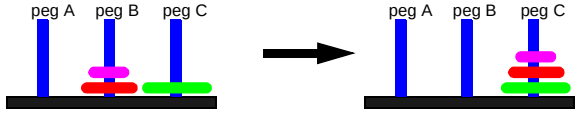
- Three rings:
  - use two ring solution to move 2 rings from peg A to peg B:



  - use one ring solution to move the ring on the bottom from peg A to peg C:

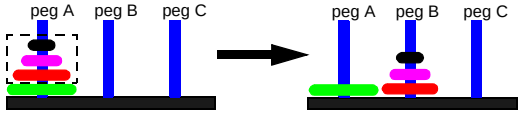


  - use two ring solution to move the 2 rings from peg B to peg C



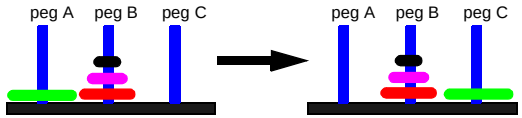
Tower of Hanoi: recursive algoritme (continued)

- *n* rings: **`tower(n, A, B, C); // from, using, to`**
  - use (n - 1) ring solution to move (n - 1) rings from peg A to peg B:



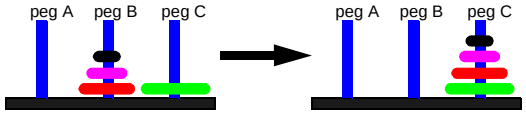
`tower(n-1, A, C, B);`

  - use one ring solution to move the ring at the bottom from peg A to peg C:



`A --> C`

  - use (n - 1) ring solution to move (n - 1) rings from peg B to peg C:



`tower(n-1, B, A, C);`

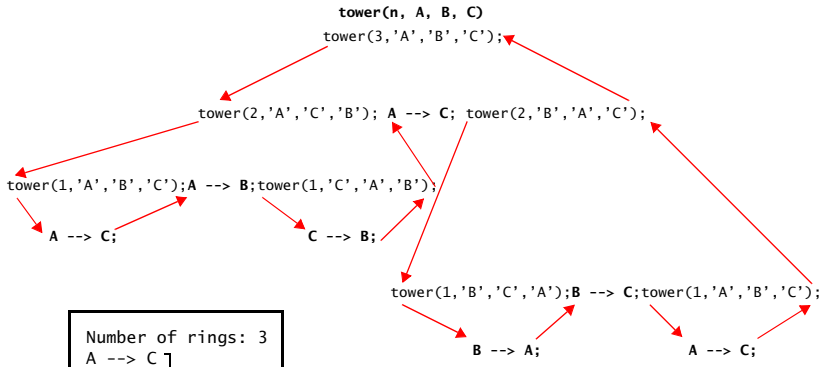
Tower of Hanoi: recursive solution (continued)

```
public class Hanoi {
    public static void main(String args[]) {
        int n = 3;
        System.out.println("Number of rings: " + n);
        tower(n, 'A', 'B', 'C');  // (n rings, from, using, to)
    }

    public static void tower(int n, char A, char B, char C) {
        if (n == 1) {       // base case
            System.out.println(A + " --> " + C);
            return;
        }
        tower(n-1, A, C, B);
        System.out.println(A + " --> " + C);
        tower(n-1, B, A, C);
    }
}
```

```
% javac Hanoi.java
% java Hanoi
Number of rings: 3
A --> C
A --> B
C --> B
A --> C
B --> A
B --> C
A --> C
```


Tower of Hanoi: recursive solution (continued)




```
Number of rings: 3
A --> C
A --> B
C --> B
A --> C
B --> A
B --> C
A --> C
```

Number of recursive calls: $2^n - 1$, there n is the number of rings. This is also the number of moves.

Tower of Hanoi: recursive solution (continued)

Each recursive calls gives rise to 2 new recursive calls, for number of rings greater than 1.

No. of rings	No. of recursive calls
1	1
2	$1 + 2 = 3 = 2^2-1$
3	$1 + 2 + 4 = 7 = 2^3-1$
4	$1 + 2 + 4 + 8 = 15 = 2^4-1$
...	...
n	$1 + 2 + 4 + 8 + ... + 2^{n-1} = 2^n-1$ (*geometric series*)

`n = 5`, 2^n-1 is 31.

`n = 10`, 2^n-1 is 1023

For `n` rings, the number of moves is also 2^n-1.