Unit 10

Recursion

Summary

- Inductively defined domains
- Recursion and recursive methods
- Run-time memory management
- Multiple recursion

10.1 Inductively defined domains

Very often, data that are manipulated by a program belong to an *inductively defined domain*. A domain of this type has the property that the set of its elements can be characterized as follows:

- one or more elements (a finite number) belong to the domain;
- one or more rules allow us to obtain from one or more elements of the domain a new element of the domain.

The fact that the set of elements of the domain is *characterized* as specified above means that no other element besides those mentioned explicitly and those specified by the rules belongs to the domain.

Example: Natural numbers:

- 0 is a natural number;
- if n is a natural number, then the successor of n is a natural number;
- nothing else is a natural number.

Example: Strings:

- the empty string `""` is a string;
- if s is a string, then, by adding to s a character in the first position, we obtain a string;
- nothing else is a string.

Example: Text files:

- the empty file is a text file;
- if f is a text file, then, by placing at the beginning of f a new line, we obtain a text file;
- nothing else if a text file.

Other inductively defined domains commonly used in computer science are *lists* and *trees*.

10.2 Inductively defined domains and recursion

The elements of an inductively defined domain can be easily manipulated through **recursion**.

A method is said to be *recursive* if it contains an activation of itself (either directly, or indirectly through the activation of other methods).

Let us see some examples of mathematical functions on natural numbers that are defined inductively, exploiting the fact that these functions operate on elements of an inductively defined domain.

Example: Factorial:

$$fact(n) = \begin{cases} 1, & \text{if } n = 0 \qquad \text{(base case)} \\ n \cdot fact(n-1), & \text{if } n > 0 \qquad \text{(recursive case)} \end{cases}$$

The recursive definition of a function reflects the structure of the inductive definition of the domain on which the function operates; hence we have:

- one (or more) *base cases*, for which the result of the function can be determined directly;

- one (or more) *recursive cases*, for which the computation of the result is reduced to the computation of the same function on a smaller/simpler value of the domain.

The fact that the domain on which the function operates is defined inductively guarantees us that, by repeatedly applying the recursive cases, we will reach in a finite number of steps one of the base cases.

Starting from the recursive definition of a function, we can usually provide rather easily an implementation by means of a recursive method.

Example: Implementation of the factorial by means of a recursive method:

```
public static long factorial(long n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

10.3 Example: recursive implementation of the sum between two integers

We exploit the following recursive definition of the sum between two non-negative integers:

$$sum(x, y) = \begin{cases} x, & \text{if } y = 0 \\ 1 + sum(x, y-1), & \text{if } y > 0 \end{cases}$$

Implementation:

```
public static int sum(int x, int y) {
  if (y == 0)
    return x;
  else
    return 1 + sum(x, y-1);
}
```

10.4 Example: recursive implementation of the product between two integers

We exploit the following recursive definition of the product between two non-negative integers:

$$product(x, y) = \begin{cases} 0, & \text{if } y = 0 \\ sum(x, product(x, y-1)), & \text{if } y > 0 \end{cases}$$

Implementation (we assume that the method `sum` is defined in the same class):

```
public static int product(int x, int y) {
  if (y == 0)
    return 0;
  else
    return sum(x, product(x, y-1));
}
```

10.5 Example: recursive implementation of the power between two integers

We exploit the following recursive definition of the power between two non-negative integers:

$$power(b, e) = \begin{cases} 1, & \text{if } e = 0 \\ product(b, power(b, e-1)), & \text{if } e > 0 \end{cases}$$

Implementation (we assume that the method `product` is defined in the same class):

```
public static int power(int b, int e) {
  if (e == 0)
    return 1;
  else
    return (product(b, power(b, e-1)));
}
```

10.6 Comparison between recursion and iteration

Some methods implemented using recursion can also be directly implemented using iteration.

Example: Iterative implementation of the factorial, exploiting the following iterative definition:

$$fact(n) = n \cdot (n-1) \cdot (n-2) \cdot \ \cdots \ \cdot 2 \cdot 1$$

```
public static long factorialIterative(long n) {
  int res = 1;
  while (n > 0) {
    res = res * n;
    n--;
  }
  return res;
}
```

Characteristics of the iterative implementation:

- initialization:
  Ex. `res = 1;`
- loop operation, executed a number of times equal to the number of repetitions of the loop:
  Ex. `res = res * n;`
- termination:
  Ex. `n--;` allows the condition `(n > 0)` of the `while` loop to become false

Recursive implementation of the factorial, exploiting the recursive definition shown previously:

```
public static long factorial(long n) {
  if (n == 0)
    return 1;
  else
    return n * factorial(n-1);
}
```

Characteristics of the recursive implementation:

- base step:
  Ex. `return 1;`
- recursive step:
  Ex. `return n * factorial(n-1) ;`
- termination is guaranteed by the fact that the recursive call `factorial(n-1)` decreases by one the value passed as parameter; hence, if initially we have `n>0`, sooner or later we reach an activation in which the condition `n==0` is true and hence only the code for the base step is executed.

Actually, it is not always possible to implement a recursive method in a simple way without using recursion. However, all recursive methods can be implemented iteratively by simulating recursion through the use of a specific data structure (a stack).

10.7 Example: number of occurrences of a character in a string

Recursive characterization of the operation of counting the occurrences of the character c in the string s:

- if s is the empty string `""`, then return 0;
- otherwise, if the first character of s is equal to c, then return 1 plus the number of occurrences of c in the string equal to s without the first character;
- otherwise (i.e., if the first character of s is different from c), return the number of occurrences of c in the string equal to s without the first character.

Implementation:

```
public static int countChars(String s, char c) {
  if (s.length() == 0)
    return 0;
  else if (s.charAt(0) == c)
    return 1 + countChars(s.substring(1), c);
```

```
    else
      return countChars(s.substring(1), c);
  }
```

10.8 Example: maximum of positive integers read from a file

Recursive characterization of the operation of finding the maximum among a set of values read from a file that contains positive integers:

- if the file is finished, return 0;
- otherwise,
  1. read an integer i from the file;
  2. find the maximum m among the remaining values in the file;
  3. return the bigger one between i and m.

Implementation: we access the file through a `BufferedReader`, and we assume that each integer is written on a separate line.

```
public static int maximum(BufferedReader br) throws IOException {
  String s = br.readLine();
  if (s == null)
    return 0;
  else {
    int i = Integer.parseInt(s);
    int m = maximum(br);
    return (m > i)? m : i;
  }
}
```

10.9 Comparison between loop for reading and recursive reading

Structure of the loop for reading:

```
read the first element
while (element is valid) {
  process the element;
  read the following element;
}
```

Recursive reading:

```
read an element
if (element is valid) {
  process the element;
  call the reading method recursively;
}
```

Example: Copy of a file (accessed through a `BufferedReader`) to an output stream.

Iterative implementation:

```
public static void copyIterative(BufferedReader br, PrintStream p) throws IOException {
  String s = br.readLine();
  while (s != null) {
    p.println(s);
    s = br.readLine();
  }
}
```

Recursive implementation:

```
public static void copy(BufferedReader br, PrintStream p) throws IOException {
  String s = br.readLine();
  if (s != null) {
    p.println(s);
```

```
    copy(br, p);
  }
  // else don't do anything
}
```

10.10 Example: the last ones will be the first ones

We want to read the lines of a file (which we access through a `BufferedReader`) and copy them to an output stream, inverting the order of the lines of the file.

By using recursion, this operation is straightforward.

```
public static void copyInverse(BufferedReader br, PrintStream p) throws IOException {
  String s = br.readLine();
  if (s != null) {
    copyInverse(br, p);
    p.println(s);
  }
}
```

The method `copyInverse` cannot be realized easily in an iterative way. The reason is that, in order to print the lines in inverse order, we have to read and store all lines read in a suitable data structure, before starting to print. We will look at this example later on, and we will show how the memory locations associated to the local variables of the recursive activations act as temporary memory locations for the lines read from the file.

Note the difference between this type of recursion and the simpler cases seen before, in which we could easily obtain an iterative implementation (e.g., for the `copy` method). The simple cases are those in which the last statement executed before the method terminates is the recursive call (*tail recursion*). Some compilers are able to detect the cases of tail recursion and replace the recursion with iteration, thus obtaining a more efficient machine code.

Instead, in general, a recursive implementation is less efficient than the corresponding iterative one, due to the necessity to handle the recursive calls (see later).

10.11 Counting elements using recursion

We want to count elements read from a file. A recursive method to do so has the following structure:

```
read an element;
if (element is not valid)
  return 0;
else
  return 1 + result-of-the-recursive-call;
```

Example: Counting the number of lines of a file

```
public static int countLines(BufferedReader br) throws IOException {
  String s = br.readLine();
  if (s == null)
    return 0;
  else
    return 1 + countLines(br);
}
```

10.12 Conditional counting of elements using recursion

We want to count only those elements of a file that satisfy a given condition. A recursive method to do so has the following structure:

```
read an element;
if (element is not valid)
  return 0;
else if (condition)
  return 1 + result-of-the-recursive-call;
else
```

```
  return result-of-the-recursive-call ;
```

Example: Counting the number of lines of a file that start with ':'

```
public static int countLinesColon(BufferedReader br) throws IOException {
  String s = br.readLine();
  if (s == null)
    return 0;
  else if (s.charAt(0) == ':')
    return 1 + countLinesColon(br);
  else
    return countLinesColon(br);
}
```

10.13 Computing values using recursion

We want to perform an operation (e.g., a sum) between all elements of an inductively defined structure (e.g., a file of integers). A recursive method to do so has the following structure:

```
read an element ;
if (element is not valid)
  return neutral-element-for-op ;
else
  return value-of-element op result-of-the-recursive-call ;
```

where *neutral-element-for-op* is the neutral element for the operation we want to perform (e.g., 0 for the sum, 1 for the product, "" for string concatenation, etc.).

Example: Sum of integers read from a file, one per line.

```
public static int sum(BufferedReader br) throws IOException {
  String s = br.readLine();
  if (s == null)
    return 0;
  else
    return Integer.parseInt(s) + sum(br);
}
```

Example: Verifying the presence of a certain value in a file of integers, stored one per line.

```
public static boolean present(int val, BufferedReader br) throws IOException {
  String s = br.readLine();
  if (s == null)
    return false;
  else
    return (Integer.parseInt(s) == val) || present(val, br);
}
```

10.14 Run-time memory management

At execution time, the Java virtual machine (JVM) must handle different parts of memory:

- the part that contains the **Java bytecode** (i.e., the code that is executed by the JVM)
  - determined at execution time when the class is loaded into the JVM
  - the dimensione is fixed for each method at compilation time
- **heap**: part of memory that contains the objects
  - dynamically grows and shrinks during execution
  - each object is allocated and deallocated independently from the other objects
- **stack of activation records**: part of memory for the data local to methods (variables and parameters)
  - dynamically grows and shrinks during execution
  - is managed in the form of a stack (Last-In-First-Out strategy)

10.15 Heap management and garbage collection

- An object is created by invoking a constructor by means of the `new` operator. The moment an object is created, the memory locations for the object are allocated on the heap.
- When an object is not used anymore by the program, the memory locations allocated on the heap for the object can be freed and made available for other objects. In Java, differently from other languages, the programmer cannot choose to do such an operation explicitly. It is done automatically by the garbage collector when the object is not accessible anymore.

The *garbage collector* is a component of the JVM that is able to detect when an object has no more references that can be used to access the object, and hence is not usable anymore and can be deallocated. Typically, the garbage collector is invoked automatically by the JVM, without any control by the programmer, when it is necessary to make memory available. However, the programmer can also choose to invoke the garbage collector explicitly, by calling the static method `gc` of the class `System`.

10.16 Stack of activation records

A *stack* is a linear data structure with LIFO access strategy: LIFO stands for Last-In-First-Out, which means that the last element that entered the stack is the first element that leaves the stack among those present (confront with a stack of plates).

At run-time, the JVM manages the **stack of activation records** (AR):

- for *each method activation* a new AR is created on top of the stack;
- at the end of the method activation, the AR is removed from the stack.

Each AR contains:

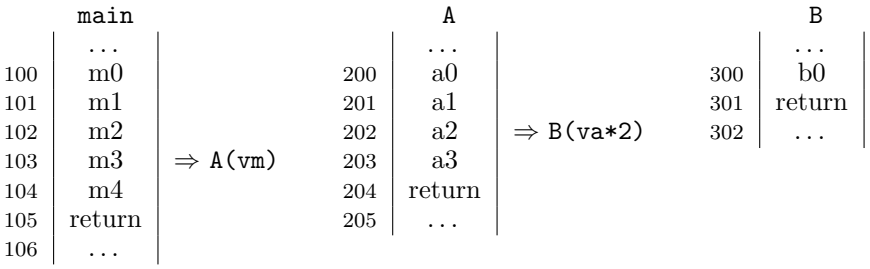
- the memory locations for the formal parameters, including the reference to the invocation object (if the method is not a static one);
- the memory locations for the local variables (if present);
- the return value for the method invocation (if the method has a return type different from `void`);
- a memory location for the return address, i.e., the address of the next statement to execute in the calling method.

10.17 Example of evolution of the stack of activation records

Let us consider the following methods `main`, `A`, and `B` and see what happens during the execution of the `main` method.

```
public static int B(int pb) {
  /* b0 */ System.out.println("In B. Parameter pb = " + pb);
  /* b1 */ return pb+1;
}

public static int A(int pa) {
  /* a0 */ System.out.println("In A. Parameter pa = " + pa);
  /* a1 */ System.out.println("Call of B(" + (pa * 2) + ").");
  /* a2 */ int va = B(pa * 2);
  /* a3 */ System.out.println("Again in A. va = " + va);
  /* a4 */ return va + pa;
}

public static void main(String[] args) {
  /* m0 */ System.out.println("In main.");
  /* m1 */ int vm = 22;
  /* m2 */ System.out.println("Call of A(" + vm + ").");
  /* m3 */ vm = A(vm);
  /* m4 */ System.out.println("Again in main. vm = " + vm);
  /* m5 */ return;
}
```

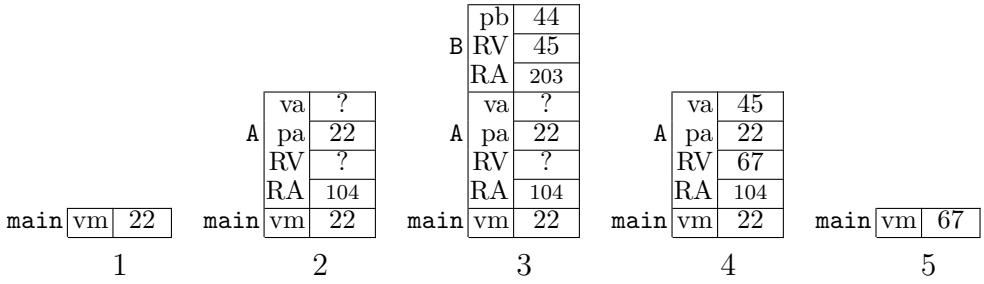
For simplicity, we ignore the invocation of the `println` method, and consider it as if it were a simple statement; moreover, we assume that each statement of the Java source code corresponds to a single statement in the Java bytecode. We also assume that the bytecode is loaded by the JVM in the following memory locations:



Output generated by the program:

```
In main.
Call of A(22).
In A. Parameter pa = 22
Call of B(44).
In B. Parameter pb = 44
Again in A. va = 45
Again in main. vm = 67
```

Evolution of the stack of ARs:



To understand what happens during the execution of the code, it is necessary to refer, in addition to the stack of ARs, to the **program counter** (PC), whose value is the address of the next Java bytecode statement to execute.

We analyze in detail what happens when `A(vm)` is activated from the `main` method. Before the activation, the stack of ARs is as shown in 1 in the above figure:

1. *The actual parameters are evaluated*: in our case, the actual parameter is the expression `vm`, whose value is the integer 22.
2. *The method to execute is determined* based on the number and types of the actual parameters, by looking for the definition of a method whose signature corresponds to the invocation (the name of the method must be the same, and the actual parameters must correspond in number and types to the formal parameters): in our case, the method to execute must have the signature `A(int)`.
3. *The execution of the calling method is suspended*: in our case, it is the `main` method.
4. *The AR is created* for the current activation of the called method: in our case, the AR for the current activation of `A` is created; the AR contains:
   - the memory locations for the formal parameters: in our case, the parameter `pa` of type `int`;
   - the memory locations for the local variables: in our case, the parameter `va` of type `int`;
   - a memory location for the return value: in our case indicated with RV;
   - a memory location for the return address: in our case indicated with RA.
5. *The values of the actual parameters are assigned to the corresponding formal parameters*: in our case, the formal parameter `pa` is initialized to the value 22.
6. *The return address in the AR is set to the address of the next statement in the calling method that must be executed at the end of the current invocation*: in our case, the return address in the AR for the activation of `A` is set to the value 104, which is the address of the statement `m4` of `main`, to be executed when the activation of `A` will be finished; the AR at this point is as shown in 2 in the above figure.
7. *The address of the first statement of the invoked method is assigned to the program counter*: in our case, the address 200, which is the address of the first statement of `A`, is assigned to the program counter.
8. *The next statement indicated by the program counter is executed (this is the first statement of the invoked method)*: in our case, the statement at address 200, i.e., the first statement of `A`.

After these steps, the statements of the called method, in our case of `A`, are executed in sequence. Specifically, if the method contains itself method calls, such methods will be activated and executed, and will terminate. In our case, the method `B` will be activate and, executed, and will terminate, with a mechanisms analogous to that for method `A`; the stack of ARs evolves as shown above in 3 and 4.

Let us now analyze in detail what happens when the activation of `A` **terminates**, i.e., when the statement `return va+pa;` is executed. Before the execution of this statement, the stack of ARs is as shown in 4 in the figure above (to be precise, the memory location reserved for the return value, indicated with RV in the figure, is initialized the moment the `return` statement is executed, and not before).

1. *The value stored in the memory location reserved for the return address in the current AR is assigned to the program counter*: in our case, such a value is equal to 104, which is precisely the address, stored in AR, of the next statement in `main` that should be executed.
2. *If the called method needs to return a value, such a value is stored in a specific memory location in the current AR*: in our case, the value 67, which is the result of the evaluation of the expression `va+pa`, is stored in the memory location indicated with RV, which is suited to contain the return value.
3. *The AR for the current activation is removed from the stack of ARs, and the current AR becomes the one immediately below it in the stack; together with the elimination of the AR from the stack of ARs, the return value, if present, is copied into a memory location of the AR of the calling method*: in our case, the AR for the activation of `A` is removed from the stack, and the current AR becomes the one for the activation of `main`; moreover, the value 67, stored in the memory location RV, is assigned to the variable `vm` in the AR of `main`; the stack of ARs is as shown in 5 in the figure above.
4. *The next statement indicated by the program counter is executed (i.e., the statement specified in step 1)*: in our case, the statement with address 104 is executed, and this corresponds to continuing the execution of `main`.

10.18 Evolution of the stack of ARs in the case of recursive methods

In the case of recursive methods, the way in which the stack of ARs and the program counter evolve is exactly the same as for non-recursive methods. However, it is important to remember that an AR is associated to *an activation of a method*, and not to a method.

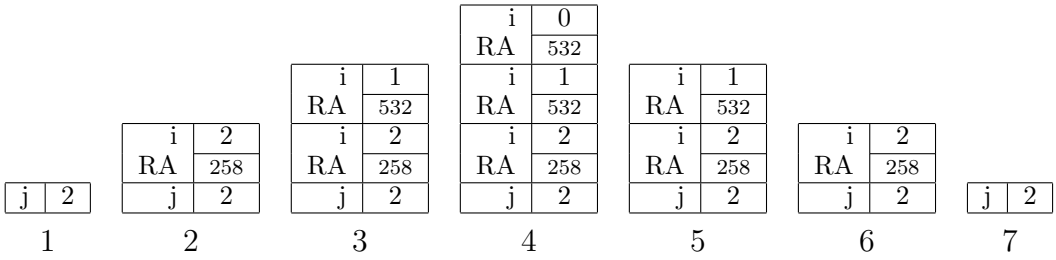
Example: Let us consider the following method `recursive` and its activation from the `main` method:

```
public static void recursive(int i) {
  System.out.print("In recursive(" + i + ")");
  if (i == 0)
    System.out.println(" - Finished");
  else {
    System.out.println(" - Activation of recursive("
                       + (i-1) + ")");
    recursive(i-1);
    System.out.print("Again in recursive(" + i + ")");
    System.out.println(" - Finished");
  }
  return;
}

public static void main(String[] args) {
  int j;

  System.out.print("In main");
  j = Integer.parseInt(JOptionPane.showInputDialog(
                          "Insert a non-negative integer"));
  System.out.println(" - Activation of recursive(" + j + ")");
  recursive(j);
  System.out.print("Again in main");
  System.out.println(" - Finished");

  System.exit(0);
}
```

Output generated by the program when the user inputs the value 2:

```
In main - Activation of recursive(2)
In recursive(2) - Activation of recursive(1)
In recursive(1) - Activation of recursive(0)
In recursive(0) - Finished
Again in recursive(1) - Finished
Again in recursive(2) - Finished
```

The evolution of the stack of ARs is shown below. we have assumed that 258 is the address of the statement that follows the activation of `recursive(j)` in `main`, and that 532 is the address of the statement that follows the activation of `recursive(i-1)` in `recursive`. Since the invoked methods do not return any value (the return type is `void`), the ARs do not contain a memory location for such a value. Moreover, we have not indicated to which method each AR refers, since the AR at the bottom of the stack is for the `main` method, and all the others are for the successive activations of `recursive`.



Note that, for the various recursive activations, different ARs are created on the stack, with successively decreasing values of the parameter `i`, up to the last recursive activation, for which the parameter `i` has value 0. In the latter case, no further recursive call is made, the string `" - Finished"` is printed, and the activation terminates. This causes the previous activations to print the message `"Again in recursive(`*i*`) - Finished"`, and to terminate.

Also note that the Java bytecode associated to the various recursive activations is always the same, i.e., the one of the method `recursive`. Hence, the return address stored in the AR for the various recursive activations is always the same (namely 532), except for the first activation, for which the return address is that of a statement in the `main` method (namely 258).

10.19 Example: the last ones will be the first ones (cont'd)

Let us consider again the example in which we read the lines of a file (which we access through a `BufferedReader`) and copy them to an output stream, inverting the order of the lines in the file.

For simplicity, we repeat here the recursive implementation we have seen before:

```
public static void copyInverse(BufferedReader br, PrintStream p) throws IOException {
  String s = br.readLine();
  if (s != null) {
    copyInverse(br, p);
    p.println(s);
  }
}
```

At this point it is clear that the successive lines of the file are stored in the strings that we can access through successive occurrences of the variable `s` in the ARs of the successive recursive calls of `copyInverse`. Hence, the stack of ARs is used as a temporary "data structure" in which to store the lines of the file before printing them.

To implement this method in an iterative way, we would have to read all lines of the file and store them, before we could start printing. Since we could not use the stack of ARs, we would need an additional data structure, e.g., an array of strings.

10.20 Example: palindrome string

A string is said to be a *palindrome* if the string read from left to right is equal to the string read from right to left. For example, ignoring the difference between uppercase and lowercase letters, the string `"iTopiNonAvevanoNipoti"` is a palindrome, while the string `"iGattiNonAvevanoCugini"` is not so.

The following is an inductive characterization of a palindrome string:

- the empty string is a palindrome;

- a string constituted only by a single character is a palindrome;
- a string $c\,s\,d$ is a palindrome, if s is a palindrome and c is a character equal to d;
- nothing else is a palindrome.

Using such a characterization, we can implement a recursive method that verifies whether a string passed as a parameter is a palindrome.

```
public static boolean palindrome(String s) {
  if (s.length() <= 1)
    return true;
  else
    return (s.charAt(0) == s.charAt(s.length()-1)) &&
           palindrome(s.substring(1,s.length()-1));
}
```

10.21 Example: symmetric sequence of integers

A sequence of integers that are all positive except for a single 0 in the central position is said to be *symmetric* if the sequence coincides with the same sequence when we invert it. For example, the sequence `5 8 3 0 3 8 5` is symmetric, while the sequence `5 8 3 0 8 5 3` is not so.

Suppose we have a text file (which we access through a `BufferedReader`) containing a sequence of integers, one per line, all positive, except for a 0 in the central position, and we want to check whether it is symmetric. One possibility is to store the whole sequence in an array, and do the check by directly accessing the elements of the array. This can be done either by using a loop, or through recursion, similarly as to what we have done for palindrome strings. However, using recursion, we can do the check also without using (explicitly) an additional data structure.

Let us first provide an inductive characterization of a sequence of integers with a 0 in the middle that is symmetric, similar to the one given for palindrome strings:

- the sequence constituted by a single 0 is symmetric;
- a sequence $n\ s\ m$ is symmetric, if s is symmetric and n and m are two equal positive integers;
- nothing else is a symmetric sequence.

Using such a characterization, we can provide a recursive implementation of the check whether the file contains a sequence of positive integers with a 0 in the middle that is symmetric (if the file contains additional lines at the end these are ignored).

```
public static boolean symmetric(BufferedReader br) throws IOException {
  int n = Integer.parseInt(br.readLine());    // read the first integer
  if (n == 0)
    return true;         // we are in the middle of the sequence
  else {
    // read the sequence in the middle and check whether it is symmetric
    boolean sim = symmetric(br);
    int m = Integer.parseInt(br.readLine());  // read the last integer
    return (n == m) && sim;
  }
}
```

10.22 Multiple recursion

We are in the presence of **multiple recursion** when the activation of a method can cause *more than one recursive activations* of the same method.

Example: Recursive method for computing the n-th Fibonacci number.

Fibonacci was a mathematician from Pisa that lived around 1200, who was interested to population growth. He developed a mathematical model to estimate the number of individuals at each generation:

$F(n)$ ... number of individuals at the n-th generation

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-2) + F(n-1), & \text{if } n > 1 \end{cases}$$

$F(0)$, $F(1)$, $F(2)$, ... is called the sequence of Fibonacci numbers, and it starts with:
0, 1, 1, 2, 3, 5, 8, 13, 21, ...

We implement a recursive method that takes a positive integer n as parameter and returns the n-th Fibonacci number.

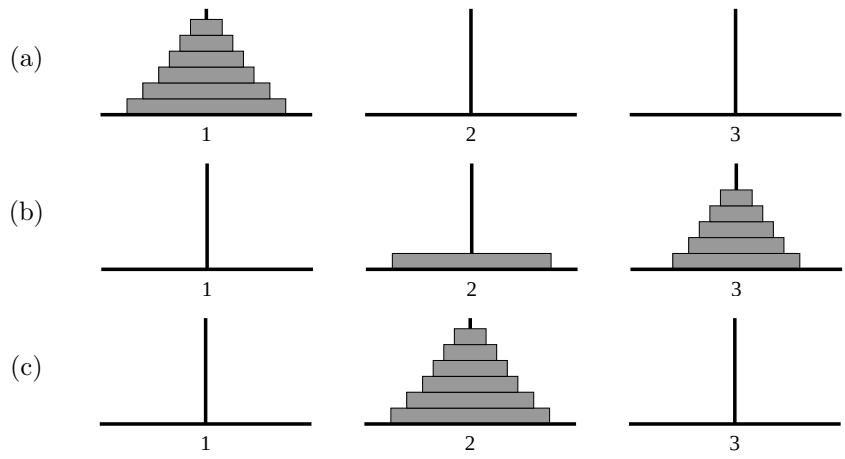
```
public static long fibonacci(long n) {
  if (n < 0) return -1;  // F(n) is not defined when n is negative
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fibonacci(n-1) + fibonacci(n-2);
}
```

10.23 Example: Towers of Hanoi

The problem of the Towers of Hanoi originates from an ancient legend from Vietnam, according to which a group of monks is moving around a tower of 64 disks of different sizes according to certain rules. The legend says that, when the monks will have finished moving around the disks, the end of the world will come. The rules according to which the disks have to be moved are the following:

- initially, the disks are placed in decreasing size on support 1
- the objective is to move them to support 2, making also use of an auxiliary support 3;
- the conditions for moving the disks are:
  - all disks (except the one to be moved) have to be on one of the three supports;
  - it is possible to move only one disk at a time, taking it from the top of the tower on one of the supports and placing it on the top of the tower on another support;
  - a disk can never be placed on a smaller disk.

The initial state (a), an intermediate state (b), and the final state (c) for a set of 6 disks are shown in the following figures:



We want to realize a program that prints the sequence of moves to be done. For each move we want to print a statement as follows (where `x` and `y` are either 1, 2, or 3):

```
move a disk from support x to support y
```

Idea: to move $n > 1$ disks from support 1 to support 2, using 3 as auxiliary support:

1. move $n - 1$ disks from 1 to 3 (without moving the n-th disk)
2. move the l'n-th disk from 1 to 2
3. move $n - 1$ disk from 3 to 2 (without moving the n-th disk)

Implementation (this is another example of multiple recursion):

```
import javax.swing.JOptionPane;

public class Hanoi {

  private static void moveADisk(int source, int dest) {
    System.out.println("move a disk from " + source + " to " + dest);
  }

  private static void move(int n, int source, int dest, int aux) {
    if (n == 1)
      moveADisk(source, dest);
    else {
      move(n-1, source, aux, dest);
      moveADisk(source, dest);
      move(n-1, aux, dest, source);
    }
  }

  public static void main (String[] args) {
    int n = Integer.parseInt(
      JOptionPane.showInputDialog("How many disks do you want to move?"));
    System.out.println("To move " + n +
                       " disks from 1 to 2 using 3 as auxiliary disk:");
    move(n, 1, 2, 3);
    System.exit(0);
  }
}
```

10.24 Number of activations in the case of multiple recursion

When we make use of multiple recursion, we have to consider that the number of recursive activations could become exponential in the depth of the recursive calls (i.e., in the maximum height of the stack of ARs).

Example: Towers of Hanoi

$act(n)$ = number of activations of `moveADisk` for n disks
= number of moves of a disk

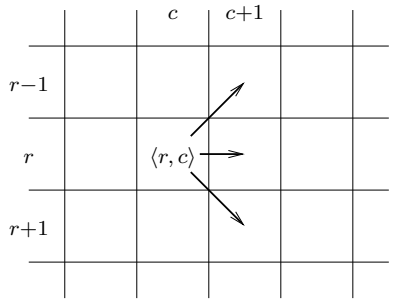
$$act(n) = \begin{cases} 1, & \text{if } n = 1 \\ 1 + 2 \cdot act(n-1), & \text{if } n > 1 \end{cases}$$

Even if we ignore "$1+$" in the case for $n > 1$, we have that $act(n) = 2^{n-1}$. Hence we have that $act(n) > 2^{n-1}$.

Note that, in the case of the problem of the Towers of Hanoi, the exponential number of activations (i.e., moves) is an intrinsic characteristic of the problem, i.e., there is no better solution.

10.25 Example: traversal of a moor

Consider a moor constituted by $R \times C$ square zones (for given R and C), each of which is either a land zone (which can be crossed) or a water zone (which cannot be crossed). Each zone of the moor is identified by a pair of coordinates $\langle r, c \rangle$, with $0 \leq r < R$ and $0 \leq c < C$. We say that r represents the *row* and c the *column* of the zone $\langle r, c \rangle$. A *traversal* is a path across the moor, i.e., a sequence of adjacent land zones that cross the moor from left (column 0) to right (column $C-1$). We are interested in traversals in which at each step we move to the right, i.e., from a zone in column c we move to a zone in column $c+1$. In other words, the zone in position $\langle r, c \rangle$ is considered adjacent to the zones in position $\langle r-1, c+1 \rangle$, $\langle r, c+1 \rangle$ and $\langle r+1, c+1 \rangle$, as shown in the following figure.



In the following figure, the character `'*'` represents a land zone, while the character `'o'` represents a water zone. Moor 1 has no traversal, while moor 2 has a traversal (shown in the figure).

	0	1	2	3	4	5
0	*	o	o	*	o	o
1	o	*	o	o	o	o
2	o	o	*	*	o	o
3	*	*	o	o	o	o
4	*	*	*	o	*	*

Moor 1

	0	1	2	3	4	5
0	*	o	o	*	o	o
1	*	o	o	o	o	o
2	o	*	o	o	o	*
3	o	o	*	*	*	o
4	o	*	o	o	o	o

Moor 2

We want to check the existence of at least one traversal, and print it out, if it exists (if there is more than one traversal, it is sufficient to print out the first one that we find).

10.26 Moor: representation of a moor

To represent a moor, we realize a class `Moor` that exports the following functionalities:

- construction of a random moor, given the number of rows, the number of columns, and a real value in the range [0..1] representing the probability that a generic zone is a zone of land;
- return of the number of rows;
- return of the number of columns;
- check if the zone with coordinates $\langle r, c \rangle$ is of land.

Moreover, in the class we override the `toString` method of `Object` in such a way that it prints a moor by using the `'*'` character for a land zone, and the `'o'` character for a water zone.

In realizing the class, we choose to represent the moor by means of a matrix of `boolean`, in which the value `true` represents a land zone, and the value `false` represents a water zone.

```
public class Moor {

  private boolean[][] moor;

  public Moor(int rows, int columns, double probLand) {
    moor = new boolean[rows][columns];
    for (int r = 0; r < rows; r++)
      for (int c = 0; c < columns; c++)
        moor[r][c] = (Math.random() < probLand);
  }

  public int getNumRows() {
    return moor.length;
  }

  public int getNumColumns() {
    return moor[0].length;
  }

  public boolean land(int r, int c) {
    return (r >= 0) && (r < moor.length) &&
      (c >= 0) && (c < moor[0].length) &&
```

```
        moor[r][c];
    }

    public String toString() {
      String res = "";
      for (int r = 0; r < moor.length; r++) {
        for (int c = 0; c < moor[0].length; c++)
          res = res + (moor[r][c]? "*" : "o");
        res = res + "\n";
      }
      return res;
    }
}
```

10.27 Moor: solution of the traversal problem

The solution requires to find a sequence of zones of the moor such that the first zone is in column 1 and the last zone is in column C. Each position in the sequence must be adjacent to the following one. For example, if the first position is $\langle 3, 1\rangle$, the second one could be $\langle 4, 2\rangle$, but not $\langle 3, 3\rangle$. Since at each step we have to move to the right, the sequence will have exactly C steps.

To explore the moor, we use a recursive method. This is the most intuitive choice, since the search process is inherently recursive. The algorithm can be summarized as follows: in the first step we look for a land zone in the first column. If there is one, we start from that zone. In the generic recursive step we are in a zone $\langle r, c\rangle$. If the zone is a land zone, we can go on and we continue the search recursively from the adjacent positions, namely $\langle r-1, c+1\rangle$, $\langle r, c+1\rangle$, and $\langle r+1, c+1\rangle$. Instead, if the zone is a water zone, we cannot go on and the search from that zone terminates. The overall search terminates with success when we arrive on a zone on the last column (i.e., c is equal to $C-1$) and such a zone is of land.

The generic search step can be implemented through the following recursive method `searchPath`, that takes as parameters a moor and the coordinates $\langle r, c\rangle$ of the zone from where to start searching the path through the moor.

```
private static boolean searchPath(Moor m, int r, int c) {
  if (the coordinates <r,c> of m are not valid ||
      in m <r,c> is a water zone)
    return false;
  else if (<r,c> is on the right border of m)
    return true;
  else
    return searchPath(p, r-1, c+1) ||
           searchPath(p, r  , c+1) ||
           searchPath(p, r+1, c+1);
}
```

The method `searchPath` checks only if there is a path from a generic position $\langle r, c\rangle$ to the last column. Since a path could start from an arbitrary position on the first column, we have to repeatedly call this method on the positions $\langle r, 0\rangle$ of the first column, until we have found a traversal or we have searched without success by starting in the first column of all rows up to the last one. This is done by a method `traverseMoor`.

10.28 Moor: construction of the traversal

Besides checking the existence of a traversal, we also want to return such a traversal. Therefore, we realize a class `Traversal` that exports the following functionalities:

- construction of a traversal, given a moor; if the moor has at least one traversal, the constructed traversal should be one of these;
- return of the moor associated to a traversal;
- checking the existence of a traversal;
- return of the length of a traversal, if it exists;
- return of the i-th step of the traversal, where i is an integer between 0 and the length of the traversal minus one, if the traversal exists;
- return of a string representing a traversal, realized by overriding the `toString` method of `Object`.

To represent a traversal, we exploit the fact that its length must be equal to C, i.e., equal to the number of columns of the moor, and that the zones that belong to the traversal are on successive columns, starting from 0 up to $C-1$. Hence, we can use an array of C integer elements in which the value of the generic element of index c is equal to the index r of the row of zone $\langle r, c\rangle$ belonging to the traversal. For example, the traversal shown in moor 2 above is represented by the array `{1,2,3,3,3,2}`.

In the following implementation, we have chosen to add to the methods `searchPath` and `traverseMoor` an additional parameter, of type array of integers, that represents a path, and have the methods do side-effect by updating the path in an appropriate way. The implementation of the method `toString` is left as an exercise.

```
public class Traversal {

  private int[] traversal;
  private Moor moor;
  private boolean found;

  public Traversal(Moor m) {
    moor = m;
    traversal = new int[moor.getNumColumns()];
    found = traverseMoor(moor, traversal);
  }

  public Moor moor() {
    return moor;
  }

  public boolean existsTraversal() {
    return found;
  }

  public int length() {
    if (found)
      return traversal.length;
    else
      throw new RuntimeException("Traversal: traversal does not exist");
  }

  public int step(int i) {
    if (found)
      return traversal[i];
    else
      throw new RuntimeException("Traversal: traversal does not exist");
  }

  public String toString() { ... }

  // auxiliary methods

  private static boolean traverseMoor(Moor m, int[] path) {
    for (int row = 0; row < m.getNumRows(); row++)
      if (searchTraversal(m, row, 0, path)) return true;
    return false;
  }

  private static boolean searchTraversal(Moor m, int r, int c, int[] path) {
    if (!m.land(r,c))
      return false;
    else {
      path[c] = r;
      if (c == m.getNumColumns()-1)
        return true;
      else
```

```
            return searchTraversal(m, r-1, c+1, path) ||
                   searchTraversal(m, r,   c+1, path) ||
                   searchTraversal(m, r+1, c+1, path);
        }
      }
    }
```

Exercises

Exercise 10.1. Provide an iterative implementation of a method that computes the n-th Fibonacci number.

Exercise 10.2. Modify the recursive implementation of the `fibonacci` method in such a way that, when it is called on the integer n, it computes besides the n-th Fibonacci number, also the total number of recursive activations of `fibonacci` used for the computation.

Exercise 10.3. Provide an implementation of the method that calculates the Ackermann function $A(m, n)$, which is defined as follows:

$$A(m,n) = \begin{cases} n+1, & \text{if } m = 0 \quad \text{(base case)} \\ A(m-1,1), & \text{if } n = 0 \quad \text{(recursive case)} \\ A(m-1, A(m,n-1)), & \text{otherwise} \quad \text{(recursive case)} \end{cases}$$

Note that the Ackermann function grows *very* fast (it is a non-elementary function): $A(x,x)$ grows faster than any tower of exponentials $2^{2^{\cdots^{2^x}}}$.

Exercise 10.4. Implement recursive methods that are based on the following inductive definitions:

- greatest common divisor:

$$gcd(x,y) = \begin{cases} x, & \text{if } y = 0 \\ gcd(y,r), & \text{if} y > 0 \text{ and } x = q \times y + r, \text{ with } 0 \leq r < y \end{cases}$$

- check whether two positive integers are relative prime:

$$prime(x,y) = \begin{cases} true, & \text{if } x = 1 \text{ or } y = 1 \quad \text{(base case)} \\ false, & \text{if } x \neq 1,\ y \neq 1 \text{ and } x = y \quad \text{(base case)} \\ prime(x, y-x), & \text{if } x \neq 1,\ y \neq 1 \text{ and } x < y \quad \text{(recursive case)} \\ prime(x-y, y), & \text{if } x \neq 1,\ y \neq 1 \text{ and } x > y \quad \text{(recursive case)} \end{cases}$$

- rest of the division between an integer and a positive integer:

$$rest(x,y) = \begin{cases} rest(x+y, y), & \text{if } x < 0 \quad \text{(recursive case)} \\ x & \text{if } 0 \leq x < y \quad \text{(base case)} \\ rest(x-y, y) & \text{if } x > y \quad \text{(recursive case)} \end{cases}$$

Exercise 10.5. Provide the implementation of a recursive method that counts how many occurrences of 1 appear in a sequence of integers read from a file (accessed through a `BufferedReader`).

Exercise 10.6. Provide the implementation of a recursive method that takes as parameters a string s and a character c and returns the length of the longest sequence of consecutive occurrences of character c in s.

Exercise 10.7. Provide the implementation of the method `toString` of the class `Traversal` in such a way that the returned string represents the whole moor, using:

- the character `'o'` for the water zones,
- the character `'#'` for the land zones that belong to the traversal, and
- the character `'*'` for the remaining land zones.

Exercise 10.8. In the implementation of the search of a traversal through a moor shown above, certain land zones could be visited several times. This makes the search for a traversal inefficient, in the worst case even exponential in the number of columns of the moor. Provide examples of a moor with a traversal and of a moor without a traversal for which the method `searchTraversal`, as implemented above, is activated a number of times that is exponential in the number of columns of the moor. Modify the class `Traversal` and the implementation of the method `searchTraversal` in such a way that zones of the moor that have already been visited are marked, and thus are not visited several times. Verify that, with the new implementation, the number of activations of `searchTraversal` is proportional to (rather than exponential in) the number of zones of the moor.

Exercise 10.9. Modify the class `Traversal` in such a way that it returns a traversal through the moor in the case where we can move in all directions (as long as we stay on land zones). Note that, in this case, it is necessary to modify the representation of a traversal, since it could be longer than the number of columns of the moor, and hence could traverse different zones on the same column. Moreover, in this case, it becomes unavoidable to mark the land zones that have already been visited during the search for a traversal, in order to avoid infinite looping on the same zones.