Lecture 12: Recursion

1 Recursion

Recursion is a powerful tool for solving certain kinds of problems. Recursion breaks a problem into smaller problems that are, in some sense, identical to the original, in such a way that solving the smaller problems provides a solution to the larger one. Every recursive solution to a problem can be rewritten as an iterative solution and every iterative solution can be written as a recursive algorithm.

Every recursive function consists of two parts:

- base case the case for which the solution can be stated non-recursively (this is the trivial case, but it is necessary for the recursion to terminate)
- recursive case the case for which the solution is expressed in terms of a smaller version of itself.

In direct recursion the recursive function makes calls to itself. In indirect recursion, there is a chain of two or more function calls that eventually returns to the function that originated the chain.

Recursion comes with a price tag. In most case when recursion is implemented a function/method is called over and over again (by itself in direct recursion, or by another function/method in indirect recursion). As a programmer you need to worry about the function call overhead that this brings to your programs. Recursion is a powerful tool, but there are many problems (computing factorials, computing Fibonacci numbers) for which the iterative solution is as simple as the recursive solution. In all such cases, you should opt for iterative implementation. On the other hand, there are problems for which recursive solution is extremely simple (towers of Hanoi), and iterative solution is prohibitively complicated - these are the good candidates for recursive implementations.

2 Computing Factorials

Computation of factorials is a very intuitive example of recursive function:

\[ n! = n \times (n-1)! \]

(The factorial of number \( n \) is a product of all the integers between 1 and \( n \).) The special case is that factorial of zero is equal to 1, i.e., \( 0! = 1 \). The recursive algorithm that implements factorials is just Java implementation of the mathematical formula.

```java
public static long factorial ( long number ) {
```

The iterative solution is equally simple and avoids the overhead of a function/method call:

```java
public static long factorial ( long number) {
    long tmpResult = 1;
    for ( ; number > 0; number--)
        tmpResult = tmpResult * number;
    return tmpResult;
}
```

Source code Factorial_Recursive.java, Factorial_Iterative.java and Factorial_Competition.java show examples of code that implements computation of the factorial method and compares their performance.

3 Fibonacci Numbers

Computation of Fibonacci numbers is another mathematical concept that is defined recursively and tends to be among first recursive problems that are introduced. The Fibonacci numbers are defined as follows:

\[
\begin{align*}
\text{fib}(0) &= 0, \\
\text{fib}(1) &= 1, \\
\text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2).
\end{align*}
\]

The interesting thing about this definition is that we have two base cases, not just one. This definition, once again, translates trivially to Java implementation that uses recursion

```java
public static long fibonacci ( long number ) {
    //base case
    if (number == 0 )
        return 0;
    else if (number == 1 )
        return 1;
    //recursive case
    else
        return fibonacci( number - 1 ) + fibonacci(number - 2);
}
```
The iterative solution requires a bit more thinking and making sure that two previous values are stored, but it is only a bit longer than the above recursive solution.

```java
public static long fibonacci ( long number ) {
    // base case
    if (number == 0 )
        return 0;
    else if (number == 1 )
        return 1;
    // recursive case
    else {
        long tmp1 = 0;
        long tmp2 = 1;
        long result = 0;
        int counter = 2;
        while ( counter <= number ) {
            result = tmp1 + tmp2;
            tmp1 = tmp2;
            tmp2 = result;
            counter++;
        }
        return result;
    }
}
```

The computation time gained when running the iterative solution is very large. Particularly because it lets us avoid repeated computations of the same values. See the source code example for the details.

**Source code**  Fibonacci_Recursive.java, Fibonacci03_Iterative.java and Fibonacci_Competition.java show examples of code that implements computation of the Fibonacci numbers and compares their performance.

## 4 Fractals

A fractal is a geometric figure that can be divided into parts, each of which is a smaller copy of the whole (if you could zoom in on a fractal, it still would look like the original shape).

Displaying a fractal is an ideal task for recursion, because fractals are inherently recursive.

### 4.1 Simple Circle Fractal

(You might have seen this example if you took the class with Craig Kapp.)
This figure may look complicated, but in fact it is very simple to produce if you know about recursion. There are very few steps required to produce this fractal. Assume that the point (0,0) in the image above is in the upper left corner.

1. for a given center (X, Y) and radius R, draw a circle
2. pick the center and a radius for the inner two circles as follows
   (a) circle 1: center = (X - R/2, Y), radius = R/2
   (b) circle 1: center = (X + R/2, Y), radius = R/2
   for each of (a) and (b) go back to step 1

In practice the program cannot recurse indefinitely, so we need to end somewhere. In the following code, we end when the radius becomes smaller than 10 (whatever units we operate in).

```java
public void drawCircle(int x, int y, int radius)
{
    // draw a circle of the desired size
    ellipse(x, y, 2*radius, 2*radius);

    // base case is to stop when radius is < 10
    if (radius >= 10)
    {
        int newRadius = radius / 2;
        int offset = radius / 2;
        drawCircle(x - offset, y, newRadius);
        drawCircle(x + offset, y, newRadius);
    }
}
```

Source Code: Circles_GUI.java shows the Processing code that produces the above circle fractal.