Recursive Methods

When you are writing a function to solve a task, one basic design technique is to break
the task into subtask. Sometimes it turns out that one of the subtasks is a smaller example
of the same task. For example, binary searching an array: you divide the array into
halves, and then search each of them. Searching the halves is a smaller version of the
original task. If the function definition contains the call to itself, the function is said to be
recursive. As example, consider factorial function, which is defined as a product of
natural numbers:

\[ \text{factorial}(5) = 1 \times 2 \times 3 \times 4 \times 5 \]

The conventional Java implementation would be

```java
public static int factorial(int n)
{
    int res = 1;
    for(int i = 1; i <= n; i++)
        res *= i;
    return res;
}
```

The recursive implementation

```java
public static int factorial(int n)
{
    if(n<=1)
        return 1;
    else
        return n * factorial(n-1);
    // return (n<=1) ? 1 : n * factorial(n-1);
}
```

When the argument reaches 1, the exit condition is satisfied, and the recursion stops.
Otherwise, a recursive call is made to \text{factorial}(n-1), the returned value is multiplied
by \( n \), and the result returned.

One might wonder how the run time system handles recursive functions. There is a lot of
bookkeeping information that one has to keep track of: for each call one has to record
who made the call and what arguments are to be handed over. Most importantly, though,
one has to keep track of all the pending calls, which may be very deeply nested inside
each other. As it turns out, all that is needed is a single \textit{stack}. We will learn stacks later
on. Whenever a function call is made (recursive or not), all the necessary bookkeeping
information is pushed onto the stack. When the execution of the function terminates, the
return value is handed over to whoever made the call (pop from the stack). Consider the
following function
public static int fun (int x, int y)
{
    if (x == 0)
        return y;
    else
        return x + fun (x - 1, y / 2);
}

Suppose this function is called with arguments 3 and 32. Here is the bookkeeping information:

fun (3, 32) -> 3 + fun (2, 16)
fun (2, 16) -> 2 + fun (1, 8)
fun (1, 8) -> 1 + fun (0, 4)
fun (0, 4) -> 4

Example 1 Count the number of digits in the decimal representation of a positive integer.

public int count(int num)
{
    if(num == 0)
        return num;
    else
        return 1 + count(num/10);
}

Example 2 Convert a given decimal number to its binary form.

public void convert(int n)
{
    if (n > 0)
    {
        System.out.print(n % 2);
        convert(n / 2);
    }
}

Exercise 2 Assume the Node and LinkedList classes from the previous lecture. Implement the following methods recursively

append(Node tmp);
getLast();
deleteLast();
remove(int p);
insertInOrder(Comparable x);
Towers of Hanoi:

In the great temple of Brahma in Benares group of spiritually advanced monks have to move 64 golden disks from one diamond needle to another. And, there is only one other location in the temple (besides the original and destination locations) sacred enough that a pile of disks can be placed there. The 64 disks have different sizes, and the monks must obey two rules:
1) only one disk can be moved at a time
2) a bigger disk can never be placed on a top of a smaller disk.
The legend is that, before the monks make the final move to complete the new pile in the new location, the next Maha Pralaya will begin and the temple will turn to dust and the world will end. Is there any truth to this legend?

The Tower of Hanoi puzzle was invented by the French mathematician Edouard Lucas in 1883. The puzzle is well known to students of Computer Science since it appears in virtually any introductory text on data structures or algorithms.

Recursive solution:
Let T(N) be the minimum number of moves needed to solve the puzzle with N disks. One can easily convince oneself that T(1) = 1, and T(2) = 3. Now let us try to derive a general formula. We're moving disks from pole A to pole C by using pole B. We break down the problem into three steps:

1. move (N-1) disks from A to B
2. move one disk from A to C
3. move (N-1) disks from B to C.

We are ready to write a mathematical formula for number of moves:

\[ T(n) = T(n-1) + 1 + T(n-1) \]

Implement this in Java, and figure out the number of years needed to move 64 disks. Will the universe dissolve and when?

```java
public double hanoi(int n)
{
    return (n==1) ? 1 : 2 * hanoi(n-1) + 1;
}
```
Computational Complexity:

The most straightforward reasons for analyzing an algorithm are

- discover its characteristics like resources of time and space
- compare it with other algorithms.

The analysis generally is kept relatively independent of particular implementation. In practice, it is difficult to arrange. In real life implementation has dramatic effects on performance. Note, that analysis requires a far more complete understanding of an algorithm that its implementing.

The term analysis of algorithms has been used to describe two quite different general approaches to the study of the performance of computer programs:

- The first, by Aho and Ullman. A prime goal is to determine which algorithm is optimal. The idea is to find a "lower bound" - the worst-case performance of any algorithm for the same problem. This type of analysis is called computational complexity.

- The second, by Donald Knuth. A prime goal is to be able to predict the performance of a particular algorithm on a particular computer.

The goal of computational complexity is to classify algorithms according to their performance. We measure the run time of an algorithm by counting the number of steps. This model is useful and accurate in the same sense as the flat-earth model. The worst case complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size n. The best case complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size n. The average case complexity of the algorithm is the function defined by an average number of steps taken on any instance of size n. Each of these complexities defines a numerical function - time vs. size!

To represent the complexity we use the big-O notation, which takes into account ONLY the highest exponent of n.

**Big-Oh notation:**

**Definition.** We say that \( f(n) = O(g(n)) \) (read "f is big O of g") when there are constants c and n_0 such that \( f(n) \leq c \cdot g(n) \) for all \( n \geq n_0 \).

The meaning of the constant n_0 is that we do not care about initial values of n. The meaning of the constant c is that we do not care about constant factors, i.e., 3 \( n^2 \) and 4 \( n^2 \) are both O(n^2). Intuitively, \( f(n) = O(g(n)) \) means that g(n) is an upper bound for f(n), or, that f(n) grows no faster than g(n).
Note, \( f(n) = O(g(n)) \) is not an equation, you cannot switch sides.

Examples,

\[
\begin{align*}
3n + 4 &= O(n) \\
3n + 4 &= O(n^2) \\
5n^2 + 6 &= O(n^2) \\
\log(n) &= O(n) \\
n\log(n) &= O(n^2)
\end{align*}
\]

It has become standard to consider an algorithm as good (feasible, fast, practical) when its running time is **polynomial**, which means that \( T(n) = O(n^d) \) for some constant \( d \).

An algorithm is an **exponential** time algorithm when its running time is \( O(a^n) \) for some constant \( a \).