**Workshop 7 - Recursion**

Start the workshop with the familiar, first recursion example from lecture 8-1:

public static int factorial(int n) {

int ans;

if (n == 0) {

ans = 1;

} else {

ans = n \* factorial(n-1);

}

return ans;

}

Remark: the function’s code was changed slightly. The version above provides a better way to track the recursive calls (the lecture version used *tail recursion*, which ends up with an optimized code that eliminates some of the recursive calls – **don’t discuss it with the students).**

To better help understand the working of the function, present the following variant which helps visualization by adding print statements.

public static int printedFactorial(int n) {

System.out.println("Start of factorial function with input: " + n);

int ans;

if (n == 0) {

System.out.println("Base condition reached");

ans = 1;

} else {

ans = n \* printedFactorial(n-1);

}

System.out.println("end of factorial function with input: " + n);

return ans;

}

Run this function with several inputs and explain how the print statements help track the flow of execution.

Proceed to another familiar example from lecture 8-1:

public static int fibo(int n) {

int ans;

if (n == 0 || n == 1) {

ans = n;

} else {

ans = fibo(n-1) + fibo(n-2);

}

return ans;

}

As before, we’ll add some print statements to help visualize the flow.

public static int printedFibo(int n) {

System.out.println("Start of Fibbonaci's function with input: " + n);

int ans;

if (n == 0 || n == 1) {

System.out.println("Base condition reached");

ans = n;

} else {

ans = printedFibo(n-1) + printedFibo(n-2);

}

System.out.println("end of Fibbonaci's function with input: " + n);

return ans;

}

Again, run the code with several (small) inputs and discuss the difference between this program and the previous one (recursion tree is of one\w children per node). Execute with a relatively large number, say 30, so that the students can appreciate the inefficiency of what happens here.

Points for discussion:

1. Remind the students that we implemented the Fibo function iteratively, with a loop, and discuss the differences (actually recursion like this is inefficient)
2. discuss the importance of the base / stop condition and what could happen if we don’t put it.

Printing Arrays

Challenge the students to implement the task of printing the contents of an array, **without using loops.** In particular, they have to implement the function with following signature:

public static void print(int[] arr)

Before they get started, discuss the main ideas, and explain how this function should wrap the actual recursive function. This might be a good place to remind them about overloading.

Here is the solution:

public static void print(int[] arr) {

print(arr, 0);

}

public static void print(int[] arr, int i) {

if (i < arr.length) {

System.out.print(arr[i] + " ");

print(arr, i + 1);

}

}

Next, challenge the students to implement the function:

public static void printReverse(int[] arr)

Challenge the students to implement it by making *as few changes as possible* to the code of the print array function implemented before (basically, it’s enough to switch two lines).

public static void printReverse(int[] arr) {

printReverse(arr, 0);

}

public static void printReverse(int[] arr, int i) {

if (i < arr.length) {

printReverse(arr, i + 1);

System.out.print(arr[i] + " ");

}

}

Points for discussion:

1. Emphasize again the importance of overloading when using recursion implementation.
2. Explain that when going through an array using recursion, we need to plan how to initialize and use the running index properly.

Sum digits:

The next function that we will write takes a positive integer and calculates the sum of all digits. The solution should use neither loops, nor strings.

Points to discuss before they begin:

* What should be the base / stop condition?
* How to extract a single digit from a number?
* What will the input look like at each stage?

Here is the solution:

public static int sumDigits(int n) {

if (n == 0) return 0;

return n % 10 + sumDigits(n/10);

}

Catalan Numbers (time permitting):

“Catalan numbers” can be defined recursively, as follows:

for .

Challenge the students to write a program that calculates the *n*’th Catalan number.

They can (and should) use loops this time. Obviously they should do it with recursion, but ask them to think how one might do it without recursion. The recursive solution is actually more straightforward.

Here is the solution:

public static int catalan (int n){

if (n == 0) return 1;

int ans = 0;

for (int i = 0; i < n; i++) {

ans += catalan(i) \* catalan(n - 1 - i);

}

return ans;

}

Notice that this number “grows” very quickly, so run it with small *n* values. You can discuss briefly the idea of memorization, which the students will learn in the recitations.

You can also mention that this particular problem has a direct closed form solution:

.