Recursion: Towers of Hanoi

Last time we saw recursive solutions for a couple of simple problems, and for the Sierpinski triangle. Today we'll look at another problem that is seemingly hard; but recursion allows for a nice solution. This is the problem of the *Towers of Hanoi*.

1. *TOWERS OF HANOI*

In the Towers of Hanoi problem there are three pegs (posts) and n disks of different sizes. Each disk has a hole in the middle so that it can fit on any peg. At the beginning of the game, all n disks are all on the first peg, arranged such that the largest is on the bottom, and the smallest is on the top (so the first peg looks like a tower).

The goal of the game is to end up with all disks on the third peg, in the same order, that is, smallest on top, and increasing order towards the bottom.

But, there are some restrictions to how the disks are moved (which make the problem non-trivial):

(1) The only allowed type of move is to grab *one* disk from the top of one peg and drop it on another peg. That is, you cannot grab several disks at one time.

(2) A larger disk can never lie above a smaller disk, on any post.

The legend says that the world will end when a group of monks, somewhere in a temple, will finish this task with 64 golden disks on three diamond pegs.

2. SOLVING THE PROBLEM

The goal is to write a program to list the set of moves for this problem.

We'll first answer an easier question: How would *you* solve this problem? Try to solve it by hand for $n = 1, 2, 3, ...$ It quickly becomes hard to keep track of the moves, no? But it gives you the intuition.

3. THE RECURSIVE FORMULATION

While you find a solution for small values of n, think of the moves you are doing, and try to identify smaller sub-problems. The goal is to express the process of moving n disks from one post to another, in terms if moving $n - 1$ disks between two posts.

Try to come up with a recursive formulation of your solution. The trick when thinking recursively is to assume that *you know how to solve the problem on a smaller input*. All you have to do is:

(1) figure out what is/are the subproblem that come up in solving your problem;

(2) figure out how to compose the solution to your oroginal problem from the solution to the subproblem(s); and

(3) provide a base-case.

4. JAVA SKELETON

Attached you'll find a skeleton for the Towers of Hanoi that is supposed to ask you for the number of disks, and print the moves. It does no graphics, in order to keep things simple (and focus on recursion). The goal is to fill in the details for

```
public void move (sourcePeg, storagePeg, DestinationPeg)
```

5. CORRECTNESS: MATHEMATICAL INDUCTION

Why does this work? Can you argue that the solution you came up with (the recursive formulation) is correct?

Recursive programming is related to *mathematical induction*, a technique of proving that some statement is true for n, known from the ancient times (the greeks). The steps in a proof by mathematical induction are th efollowing:

(1) The base case: prove that the statement is true for some value of n, usually $n = 1$;
(2) The induction step: assume that teh statement is true for all integers $\leq n - 1$. Then prove that this implies that it is true for n.

Try proving by induction that $1 + 2 + 3 + ... + n = n(n + 1)/2$.

A recursive solution is similar to an inductive proof; just that instead of "inducting" from values smaller than n to n, we "reduce" from n to values smalles than n (think n = input size). The importance of the base-case becomes more apparent, because without it the recursion goes onto an infinite loop.

Recursive algorithm have very easy-to-see correctness proofs by mathematical induction.

Proof sketch for correctness of Towers of Hanoi: It works correctly for moving 1 disk (base-case). Assume it works correctly for moving $n - 1$ disks. Then argue that it follows that it works correctly for moving n disks.

6. ANALYSIS: HOW CLOSE IS THE END OF THE WORLD?

How long does your move() method take to compute the sequence of moves to move n disks from one peg to the other?

The running time of recursive algorithms is estimated using *recurrence relations* as follows:

Let $T(n)$ be the time it takes to to compute the sequence of moves to move n disks from one peg to the other. Then it must be that

$$T(n) = 2T(n - 1) + 1, \forall n > 1$$

and $T(1) = 1$ (the base case).

It can be shown by induction that $T(n) = 2^n - 1$ (*math200*). This means that the runing time is exponential in n.

With a 1GHz machine and 64 disks this would take $2^{64} \cdot 10^{-9} = ...$ a long time; hundreds of years.

7. PITFALLS OF RECURSION

7.1 Missing base case.

Gives infinite recursion....and stack overflow.

7.2 No convergence.

Another common problem is to include a call to a problem that is not smaller than the original problem.

7.3 Excessive recomputation.

Sometimes recursive solutions may take exponential time. Sometimes this can be fixed...

8. PERSPECTIVE

Recursion leads to compact solutions.

Recursion leads to easy-to-understand and easy-to-prove-correct solutions.

Most of the programming time is spent in debugging, so easy-to-understand solutions are crucial.

Recursion allows to think about the problem at a higher level of abstraction.

Recursion has an overhead (system keeps track of the list of active functions); modern compilers can sometimes eliminate recursion; efficiency on moren systems is less of an issue. Unless you write super-optimized code for large problem instances, recursion is good.

Mastering recursion is essential to understanding computation.