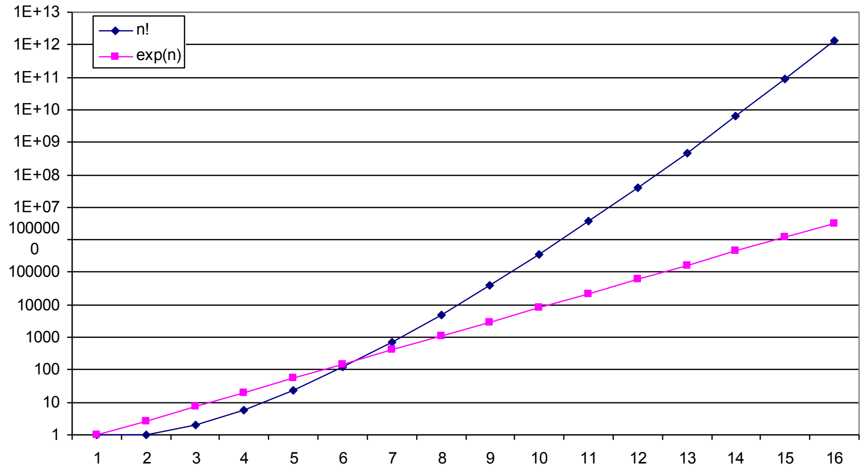
Software Structures and Models/Data Structures and Algorithms TX00CK91

Lecture 08 - 26.04.2016

Jarkko.Vuori@metropolia.fi

Recursion

- Widely used in mathematics and in computer science
  - Used in (mathematical) definitions
  - E.g. factorial (kertoma) *n*! is a product of numbers 1,2,3,…,*n*
    - Describes the number of permutations (different ways to order a given set of elements) of a finite set
      - e.g. there are six permutations of the set {1,2,3}, namely [1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], and [3,2,1]
    - Increases at an even higher rate than the exponential function
    - Definition $$n! = \begin{cases} 1, & \text{when n} = 0 \\ n \cdot (n-1)!, & \text{when n} > 0 \end{cases}$$
    - It seems at first that we go around a circle because in the definition we are using the definition itself (factorial of *n* is *n* times the factorial of *n*-1)
      - Kernel of the definition is the case where the factorial is explicitly defined, e.g. the case where *n*=0
        - » This is called as the base of the recursion
      - The other important point in the definition is, that in the recursive part, factorial is taken from the smaller number than the original number
        - » When the recursion is applied repeatedly, we finally reach the base of the recursion



Exponent function seems to be linear here because y-axis is logarithmic

Recursion

$$n! = \begin{cases} 1, & \text{when n} = 0 \\ n \cdot (n-1)!, & \text{when n} > 0 \end{cases}$$

- Using the definition of the factorial, we find out what is 5!

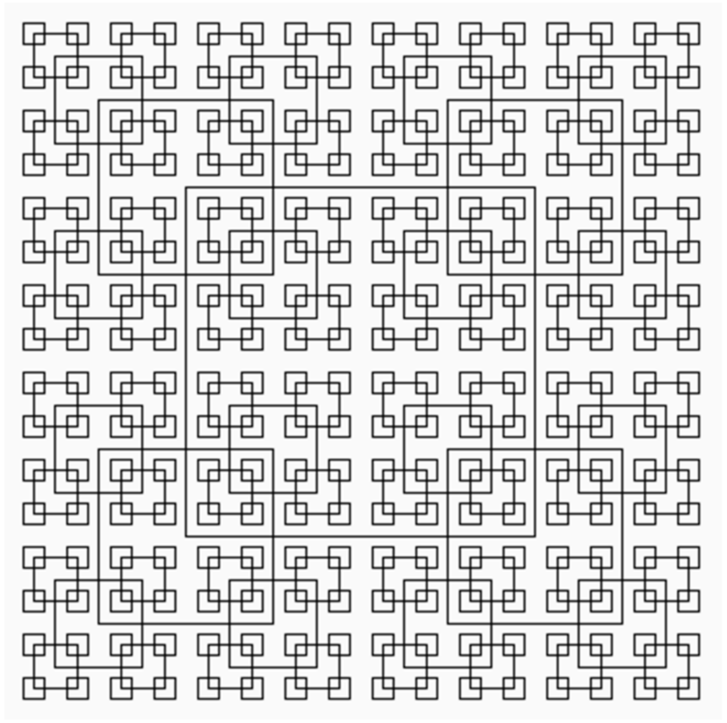
Step 1	if 5 = 0?	no $\rightarrow$ 5!=5·(5-1)!=5·4!
Step 2	if 4 = 0?	no $\rightarrow$ 4!=4·(4-1)!=4·3!
Step 3	if 3 = 0?	no $\rightarrow$ 3!=3·(3-1)!=3·2!
Step 4	if 2 = 0?	no $\rightarrow$ 2!=2·(2-1)!=2·1!
Step 5	if 1 = 0?	no $\rightarrow$ 1!=1·(1-1)!=1·0!
Step 6	if 0 = 0?	yes$\rightarrow$ 0!=1

- Here we get the 5!=5·(4·(3·(2·(1·1))))
  - Calculation is done backwards, first 1·1, then 2·1, then 3·2, etc.

- In evaluation of the recursion we return back to the definition itself, but in every return, we have a smaller problem (as we had in the preceding factorial example)
  - When this returning has been done enough times, the problem is so small that it can be easily solved

Recursion

- Principle of the recursion
  1. First we find a simple case (base of the recursion)
  2. Then we reduce the problem back to itself, but in smaller size
  3. After that we repeat the case 2, until we'll get to the simple case (base of the recursion)
  4. Returning back to the original problem in reversed order
- ”In order to understand recursion, one must first understand recursion”

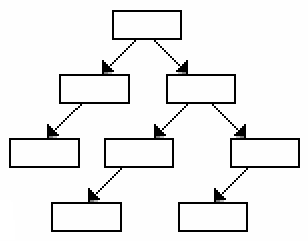


Recursion in definitions

- List
  - List (L) is
    1. Empty
    2. List and a new element aL
- Binary search tree
  - Binary tree is
    1. Empty
    2. Node with a type T which has binary (sub)trees (with no mutual connections) with type T
- Factorial *n*! is

0! = 1
1! = 1·0!
2! = 2·1!
.
.
.
n! = n·(n-1)!

3
3, 4
3, 4, 7
3, 4, 7, 11
.
.
.



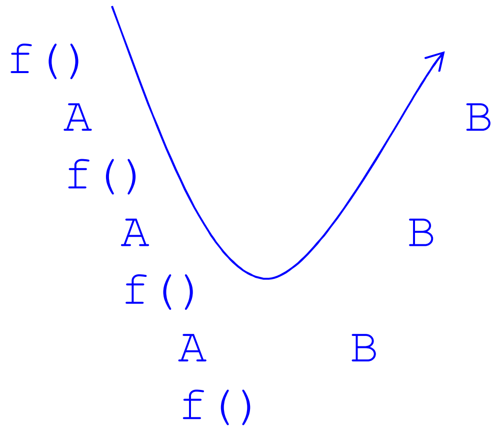
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
.
.
.

Recursion as programming method

- We can use recursion as a repetition structure
- Recursive function calls itself

```
float f(float x) {
    if (!<base_condition>) {
        // here we have the code A to be repeated
        // in forwards (when entering the recursion)
        f(…);
        // here is the code B to be repeated
        // in backwards (when leaving)
    } else
        // base of the recursion here
}
```

  - Base of the recursion does not call the recursion, this is the ending statement of the recursion



Recursive function

- For example, factorial calculation

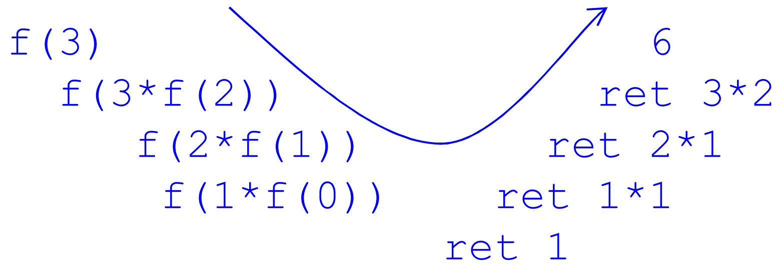
```
int factorial_r(int n) {
      if (n == 0)
            return (1); // base
      else
            return (n*factorial_r(n-1));
}
```

Terminating condition

  - Compare to iterative realization

```
int factorial_i(int n) {
       int i = 1;

       while (n > 0)
             i = i * n--;

       return (i);
}
```

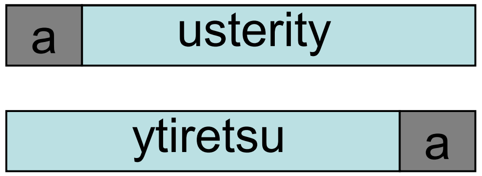
$$n! = \begin{cases} 1, & \text{when n} = 0 \\ n \cdot (n-1)!, & \text{when n} > 0 \end{cases}$$



```
f(3)                            6
  f(3*f(2))                ret 3*2
     f(2*f(1))          ret 2*1
      f(1*f(0))      ret 1*1
                  ret 1
```

Recursive function

- Printing in reversed order



```
private void print_inverse(int n) {
        char ch;

        if (n == 1) {
            ch = s.next().charAt(0); // s = new Scanner(System.in);
            System.out.print(ch);
        }
        else {
            ch = s.next().charAt(0); // s = new Scanner(System.in);
            print_inverse(n-1);
            System.out.print(ch);
        }
    }
```

base

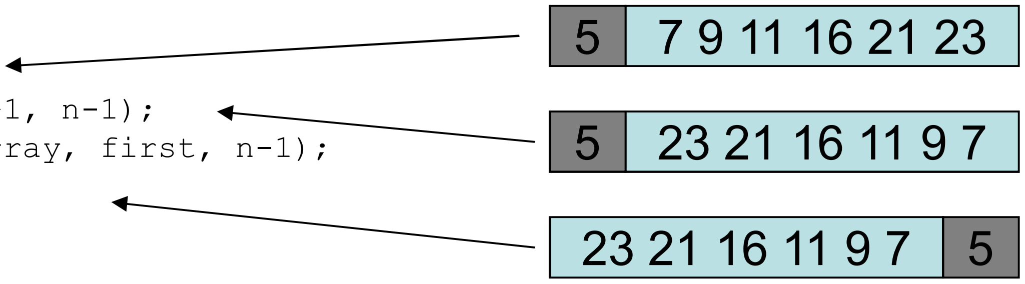
recursion

```
float f(float x) {
     if (!<base_condition>) {
         // here we have the code A to be repeated
         // in forwards (when entering the recursion)
         f(…);
         // here is the code B to be repeated
         // in backwards (when leaving)
     } else
         // base of the recursion here
  }
```

Recursive function

- Inverting an array

```
private void invert_array(int[] array, int first, int n) {
       int aux;

       if (n == 0)
           return;
       else {
           aux = array[first];
           invert_array(array, first+1, n-1);
           move_one_step_backwards(array, first, n-1);
           array[first+n-1] = aux;
       }
   }

   private void move_one_step_backwards(int[] array, int first, int n) {
       for (int i = 0; i < n; i++)
           array[first+i] = array[first+i+1];
   }
```



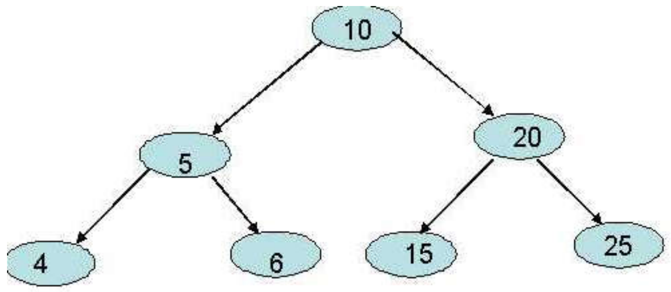
Recursive function

Traversing a binary tree iteratively

```
void in_order_traverse(Node tree) {
 stack<Node> pointerstack = new stack<Node>();
    Tree        p;

    p = tree; // tree is a Tree class variable
    while ((p != null) || !pointerstack.empty()) {
        while (p != null) {
            pointerstack.push(p);
            p = p.left;
        }
        p = pointerstack.top(); pointerstack.pop();
        System.out.print(p.item + " ");
        p = p.right;
    }
}
```

Traverse until we ran out of subtrees

Traverse until a leaf reached



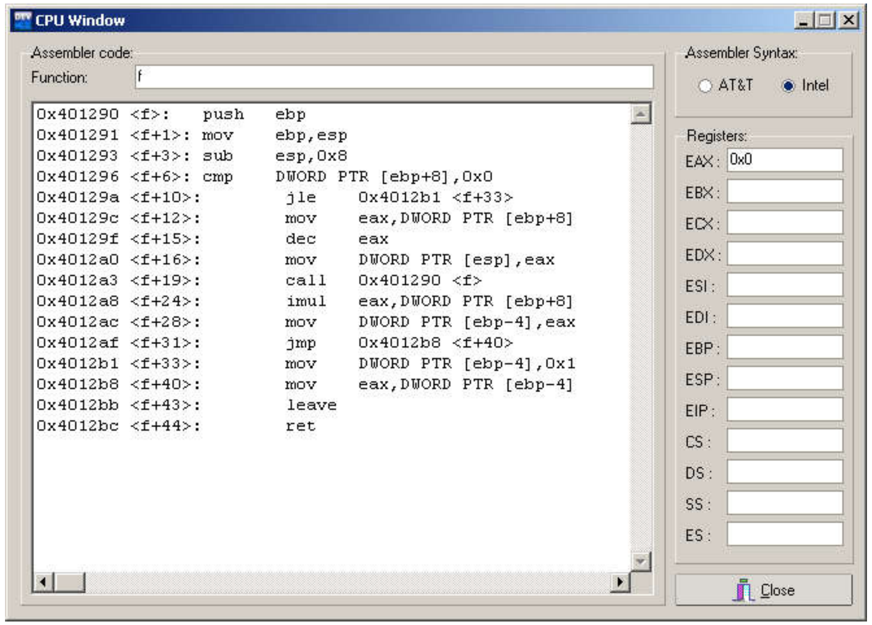
- Traversing binary tree recursively

```
void in_order_traverse(Node tree) {
    if (!is_tree_empty(tree)) {
        in_order_traverse(tree.left);
        System.out.print(tree.data + " ");
        in_order_traverse(tree.right);
    }
}
```

Recursive function

- When calling a function (subroutine), CPU push function arguments and the return address to the stack
- Inside the function, CPU reserves space for local variables (from stack)
- When returning from the function, all reservations from the stack are released
  - And old values of the local variables are restored

```
int f(int n) {
    if (n > 0)
        return (n*f(n-1));
    else
        return (1);
}
```



```
0x401290 <f>:   push   ebp
0x401291 <f+1>: mov    ebp,esp
0x401293 <f+3>: sub    esp,0x8
0x401296 <f+6>: cmp    DWORD PTR [ebp+8],0x0
0x40129a <f+10>:        jle    0x4012b1 <f+33>
0x40129c <f+12>:        mov    eax,DWORD PTR [ebp+8]
0x40129f <f+15>:        dec    eax
0x4012a0 <f+16>:        mov    DWORD PTR [esp],eax
0x4012a3 <f+19>:        call   0x401290 <f>
0x4012a8 <f+24>:        imul   eax,DWORD PTR [ebp+8]
0x4012ac <f+28>:        mov    DWORD PTR [ebp-4],eax
0x4012af <f+31>:        jmp    0x4012b8 <f+40>
0x4012b1 <f+33>:        mov    DWORD PTR [ebp-4],0x1
0x4012b8 <f+40>:        mov    eax,DWORD PTR [ebp-4]
0x4012bb <f+43>:        leave
0x4012bc <f+44>:        ret
```

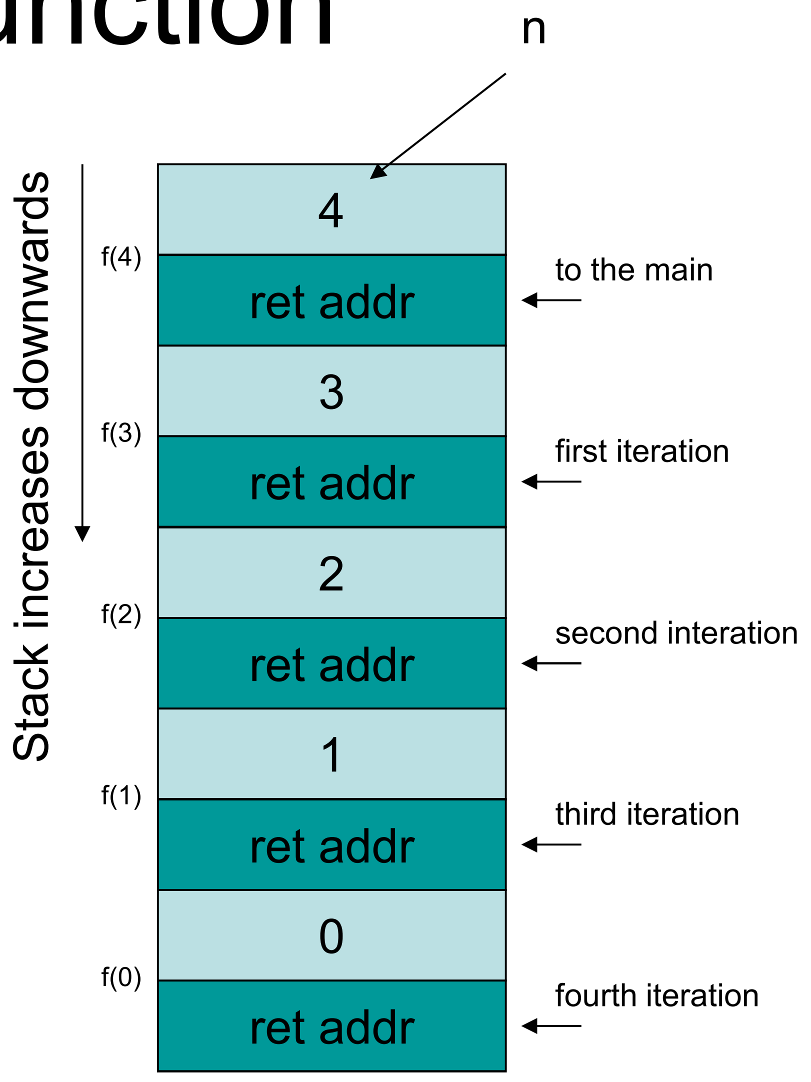
Stack frame

Stack

```
000063A4H: 67 FF 00 C7   n (param 1)
000063A0H: C4 2F FB F9   return address
0000639CH: A7 FE 89 B7   EBP
00006398H: FC 45 45 F0   local variable 1
00006394H: 3F FE 39 47   local variable 2 (esp)
00006390H: 21 7F DF CB
0000638CH: 3F F4 4D BE
```

Recursive function

```
int f(int n) {
    if (n > 0)
        return (n*f(n-1));
    else
        return (1);
}

int main() {
    System.out.println("4! is " + f(4));
}
```



Recursive functions for linked list

- Inserting element to the end of the list
  - We know only the starting node of the list, list end must be searched for
    - Traditional iterative method: traversing while we'll encounter node that has NULL in the next-field
    - Recursive model is derived from the definition of series
      - Empty set is a serie (base)
      - If L is a serie, and a is an element, then aL is a serie
    - In dynamic linked list representation, the node which contains an element, has also a pointer to the next node
      - First node's next field represents the list L, where all the remaining nodes are
        - » Similarly the first node's address represents the whole list
    - With this model, inserting an element to the end of the list can be represented recursively
      - Insert an element to the first node of the list, if the list is empty
      - Otherwise insert an element to the end of the list which follows the first element
  - We find a base where the solution is simple (inserting an element to an empty list), and the recursive part solves smaller problem than the original problem (inserting an element to one element shorter list)

```
void insert_to_list_end(Node list, T data) {
    if (list.next == NULL)
        list.next = new Node(data);
    else
        insert_to_list_end(list.next, data);
}
```

  - Solution is simple, and describes recursive modeling clearly
  - When considering the efficiency, this function is not the best possible
    - All nodes addresses are stored to the stack until we encounter the base
    - We need as many elements from the stack than we have nodes in the list
      - And we don't need them in this task
  - Recursive solution can also be less efficient than the iterative solution (waste of stack)

Recursive functions for linked list

- Printing linked list's elements in reversed order

```
void print_list_in_reverse(Node list) {
    if (list == NULL)
        System.out.println("List in reverse order :");
    else {
        print_list_in_reverse(list->next);
        System.out.print(list.data + " ");
    }
}
```

  – In this case recursion fits, because pointers must be stored somewhere in every case (because we must print the content of the node later)

Recursive functions for binary search tree

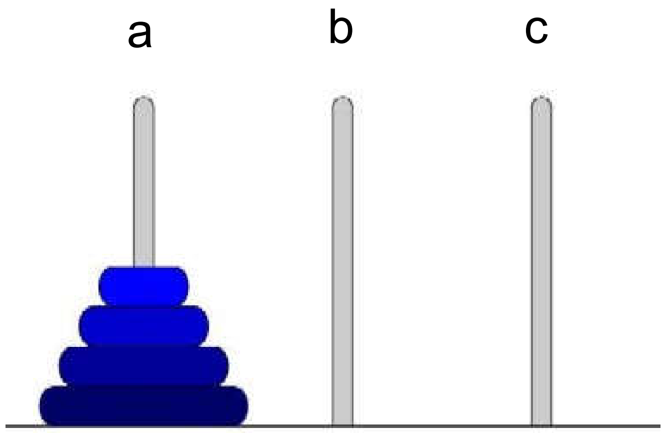
- Traversing binary search tree in order (in_order_traverse)

```
void in_order_traverse(Node tree) {
    if (!is_tree_empty(tree)) {
        in_order_traverse(tree.left);
        System.out.print(tree.data + " ");
        in_order_traverse(tree.right);
    }
}
```

- This is also an efficient solution, because on the stack there is information only the amount of depth (of the tree)

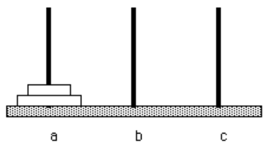
Recursion in problem solution

- Towers of Hanoi
  - Three vertical pegs (a,b,c) where we have *n* disks of different sizes (with a hole in the middle)
    - Disks are on the pegs in such a way that the largest disk is on the bottom and the smallest disk is on the top
    - All the times a disk should be smaller than the disk under it
  - Our task is to move all disks from the peg a to peg c
    - Peg b can be used also (for the help)
  - The following rules must be obeyed
    - Only one disk can be removed from the pegs at time
    - No peg can contain disks in such a way that larger disk is on the top of smaller disk
  - There is a legend about an Indian temple which contains a large room with three time-worn posts in it surrounded by 64 golden discs
    - The priests of Hanoi, acting out the command of an ancient prophecy, have been moving these discs, in accordance with the rules of the puzzle
    - According to the legend, when the last move of the puzzle is completed, the world will end.

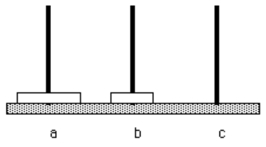


Recursion in problem solution

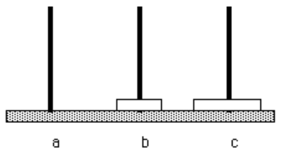
- Solving this problem is difficult at first
  - Number of moves required increases exponentially when the number is disks increases
  - With a recursion we can manage the problem
    - Solution is simple (but of course the number of disk moves is still large)
- With recursion
  - First we find a small problem which is simple (base of the recursion)
    - In this problem, the case with only one disk
      - All the requirements are met if we simply move the disk from peg a to the peg c
  - In the recursive part we state how to move *n* disks
    - In the beginning we assume that the solution can be done for a little smaller amount of disks (*n*-1 disks). If there were *n* disks, they are moved in this way
      - First we move *n*-1 disks obeying the rules from peg a to peg b using peg c
      - Then we move the largest disk in peg a to the peg c
      - Finally move *n*-1 disks according the rules from peg b to peg c using peg a
    - The question how to move *n*-1 disks obeying the rules is still an open question
      - With recursion the problem of *n*-1 disks, can be further divide to two parts
        - » First we move *n*-2 disks in the above mentioned way
        - » Then the remaining one disk can be moved
      - The problem gets smaller in every round, and finally it will reduce to the simplest case of moving only one disk
        - » After this case, all the necessary moves can be found in reverse action



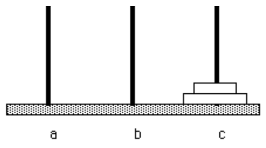
1



2



3



4

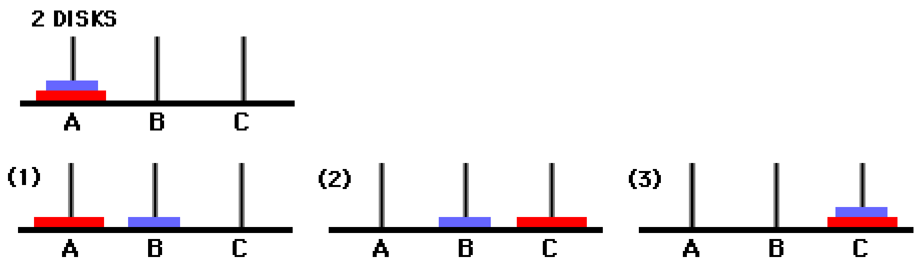
Recursion in problem solution

```
// move disks from the peg1 to peg3 using peg2 as a help
void move(int n, char peg1, char peg2, char peg3) {
    if (n == 1)
        System.out.println("Move from peg " + peg1 + " to peg " + peg3);
    else {
        // move first n-1 disks from the top to the helper peg, then only nth disk is remaining
        move(n-1,peg1,peg3,peg2);
        // move the remaining disk to its final place
        System.out.println("Move from peg " + peg1 + " to peg " + peg3);
        // move n-1 disks from the helper peg to the top of the nth disk
        move(n-1,peg2,peg1,peg3);
    }
}

void main() {
    int n;

    System.out.println("How many disks :"); n = Integer.parseInt(System.console().readLine());
    System.out.println("Use the following movements ");
    move(n, ’a’, ’b’, ’c’);
}
```

The number of moves needed is $H_N = 2H_{N-1} + 1$; $H_1 = 1$, e.g. $H_N=2^N-1$.
If it takes about 1 minute to move one disk, and we have 64 disks
the world will end after $1{,}84 \cdot 10^{19}$ minutes, e.g. $3{,}51 \cdot 10^{13}$ (35000 billion years).
Lifetime of our sun is estimated to be about $6{,}5 \cdot 10^{12}$ years.
(http://image.gsfc.nasa.gov/poetry/ask/a10395.html)

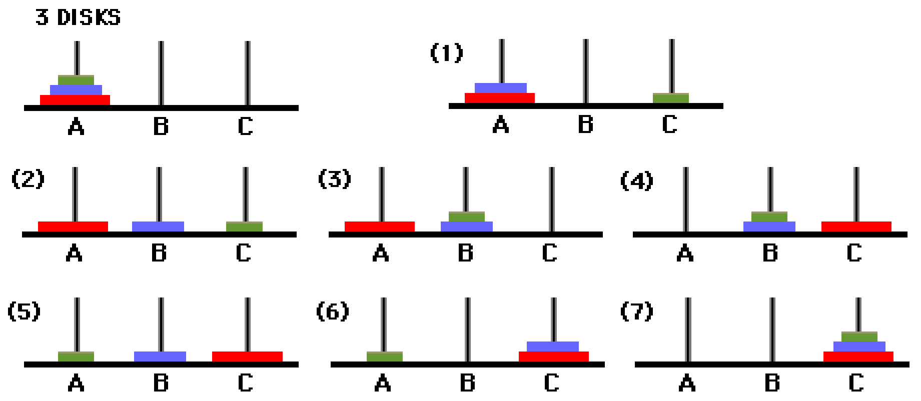
Recursion in problem solution

- 2 disks: 2^2-1 = 3 moves
  - move 1: move the disk 2 to the peg B
  - move 2: move the disk 1 to the peg C
  - move 3: move the disk 2 to the peg C



```
// move disks from the peg1 to peg3 using peg2 as a help
void move(int n, char peg1, char peg2, char peg3) {
    if (n == 1)
        System.out.println("Move from peg " + peg1 + " to peg " + peg3);
    else {
        // move first n-1 disks from the top to the helper peg, then only nth disk is remaining
        move(n-1,peg1,peg3,peg2);
        // move the remaining disk to its final place
        System.out.println("Move from peg " + peg1 + " to peg " + peg3);
        // move n-1 disks from the helper peg to the top of the nth disk
        move(n-1,peg2,peg1,peg3);
    }
}
```

- 3 disks: 2^3-1 = 7 moves
  - move 1: move the disk 3 to the peg C
  - move 2: move the disk 2 to the peg B
  - move 3: move the disk 3 to the peg B
  - move 4: move the disk 1 to the peg C
  - move 5: move the disk 3 to the peg A
  - move 6: move the disk 2 to the peg C
  - move 7: move the disk 3 to the peg C


3 DISKS
A
B
C
(1)
(2)
(3)
(4)
(5)
(6)
(7)