Introduction to Computers & Programming

Ada: Recursion

Prof. Kristina Lundqvist
Dept. of Aero/Astro, MIT

Recursion

- Recursion means writing procedures and functions which call themselves.
- Recursion involves:
  - solving large problems
  - by breaking them into *smaller* problems
  - of *identical* forms
- Eventually a "trivial" problem is reached that can be solved immediately.
- Initially it may seem that the solution is incomplete.

General Algorithm

```
if stopping condition then
    solve simple problem
else
    use recursion to solve smaller problem(s)
    combine solutions from smaller problem(s)
```

Iterative vs Recursive

- Recursion is a way of implementing a loop
- Iteration and recursion can always be interchanged
- Iteration
  - Cognitively simple
- Recursion
  - Is not as intuitive
  - Demanding on machine time and memory
  - Sometimes simpler than iteration

Guess a number

- **Problem**: think of a number in the range 1 to N
- **Reworded**:
  - Given a set of N possible numbers to choose from
  - Guess a number from the set
  - If wrong, **guess again**
  - Continue until the number is guessed successfully
- **Recursion** comes into the "**guess again**" stage
  - A set of N-1 numbers remains from which to guess
  - This is a smaller version of the same problem

Print a number

- **Problem**: Print a number (an integer)

- **Recursive algorithm**:
  1. Print all of the number except the last digit
  2. Print the last digit

- **Pseudocode**
  -
```
procedure PrintNum (N : in integer) is
begin
  print the number N DIV 10;
  print the digit N MOD 10;
end PrintNum;
```

Print a number

- Stopping condition added

  - ```
    procedure PrintNum (N : in integer) is
    begin
      if N < 10 then
        print the digit N
      else print the number N DIV 10;
           print the digit N MOD 10
      end if;
    end;
    ```
    ```

Print a number

- Recursive procedure

-
```
procedure PrintNum (N : integer) is
begin
  if N < 10 then
    put(N,WIDTH=>1)
  else
    PrintNum (N DIV 10);
    put ((N MOD 10), WIDTH=>1)
  end if;
end;
```

Recursion

- Solution to problems uses either **iteration** or **recursion**
- What is meant by **iteration**?
  - Involves counting via **looping** within a module
- Not all recursive solutions are better than iteration.
- What is **recursion**?
  - It is a technique for performing a task **T** by performing another task **T'**.
  - Task **T'** has **exactly the same nature as** the original task **T**.
- Recursion can for example be used in **binary search**, such as looking for **word in a dictionary**.

Recursive

Psuedocode for recursive searching in a dictionary

```
-- Search the dictionary for a word
  if the dictionary contains only one page
    then scan the page for the word
  else
    begin
        Open the dictionary to a point near the middle to
             determine which half contains the word

        if the word is in the first half of the dictionary
          then search the first half of the dictionary
               for the word
        else search the second half of the dictionary for
             the word
    end
```

Factorial

- Functions are often useful for calculating mathematical equations
- Write a function that, given n, computes n!

  n! == 1 * 2 * ... * (n-1) * n
- Example:

  5! == 1 * 2 * 3 * 4 * 5 == 120

- Specification:

  Receive: *n*, an integer
  Precondition: *n* >= 0 (0! == 1 and 1! == 1)
  Return: *n!*

Preliminary Analysis

- At first glance, this is a counting problem, so we could solve it with **a for loop**:

```
function Factorial_Iterative (N : in Natural) return Positive is

   Result : Positive;

begin
   Result := 1;
   for Count in 2 .. N loop
      Result := Result * Count;
   end loop;

   return Result;

end Factorial_Iterative;
```

- But let’s instead use a different approach...

Analysis

- Consider: n! == 1 * 2 * ... * (n-1) * n
  - so: (n-1)! == 1 * 2 * ... * (n-1) → n! == (n-1)! * n

- **We have defined the ! function *in terms of itself***

Recursion

- A function that is defined in terms of itself is called *self-referential*, or ***recursive***.

- Recursive functions are designed in a 3-step process:
  1. Identify a **base case** -- an instance of the problem whose solution is ***trivial***
     - Example: The factorial function has **two base cases**:
       if n == 0 : n! == 1
       if n == 1 : n! == 1

Induction Step

2. Identify an **induction step** -- a means of solving the non-trivial instances of the problem using one or more "smaller" instances of the problem.

   Example: In the factorial problem, we solve the "big" problem using a "smaller" version of the problem:

   n! == (n-1)! * n

3. **Form an algorithm** from the base case and induction step.

Algorithm

```
-- Factorial(N)

0. Receive N

1. If N > 1
         Return Factorial(N-1) * N
   Else
         Return 1
```

Ada Code

```
function Factorial (N : in Natural) return Positive is
  begin - factorial
    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;
  end Factorial;
```

Behavior

Suppose the function is called with N == 4.

```
function Factorial (N : in Natural)
   return Positive is

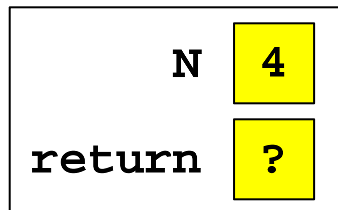
   begin - factorial

     if N > 1 then
       return N * Factorial(N-1);
     else
       return 1;
     end if;

  end Factorial;
```

Behavior

The function starts executing, with N == 4.




```
function Factorial (N : in Natural)
   return Positive is

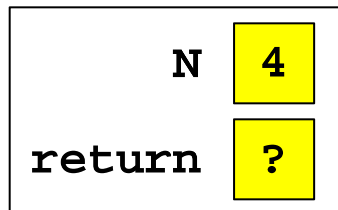
   begin - factorial

    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

Behavior

The if executes, and N (4) > 1, ...




```
function Factorial (N : in Natural)
   return Positive is

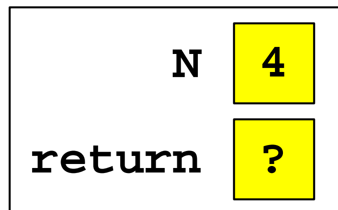
   begin - factorial

    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

Behavior

and computing the return-value calls Factorial(3).




```
function Factorial (N : in Natural)
   return Positive is

   begin - factorial

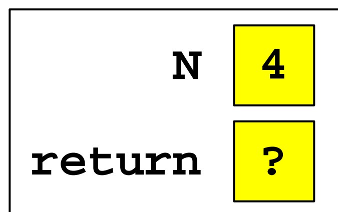
     if N > 1 then
       return N * Factorial(N-1);
     else
       return 1;
     end if;

   end Factorial;
```

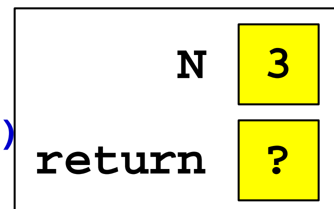
Behavior

This begins a *new execution*, in which N == 3.

Factorial(4)



Factorial(3)




```
function Factorial (N : in Natural)
   return Positive is

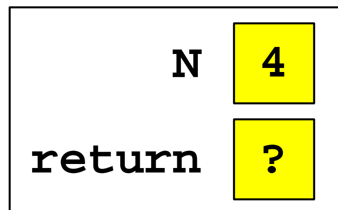
   begin - factorial

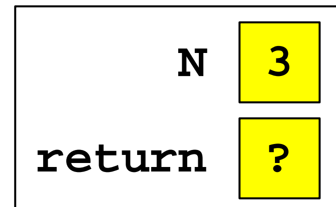
    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

Behavior

Its if executes, and n (3) > 1, ...






```
function Factorial (N : in Natural)
   return Positive is

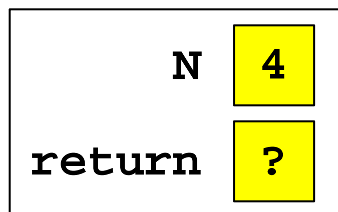
   begin - factorial

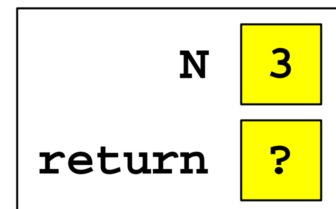
    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

Behavior

and computing its return-value calls Factorial(2).






```
function Factorial (N : in Natural)
   return Positive is

   begin - factorial

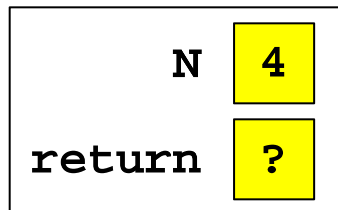
    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

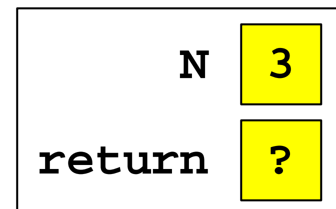
Behavior

This begins a new execution, in which N == 2.

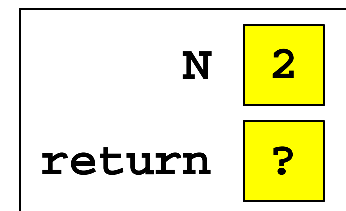
Factorial(4)



Factorial(3)



Factorial(2)




```
function Factorial (N : in Natural)
   return Positive is

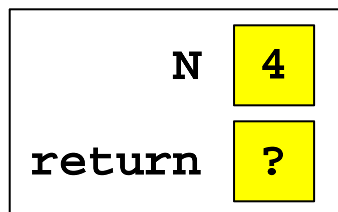
   begin - factorial

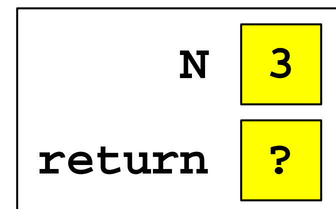
    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

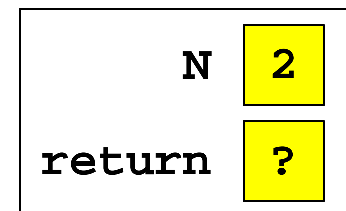
  end Factorial;
```

Behavior

Its if executes, and N (2) > 1, ...








```
function Factorial (N : in Natural)
   return Positive is

   begin - factorial

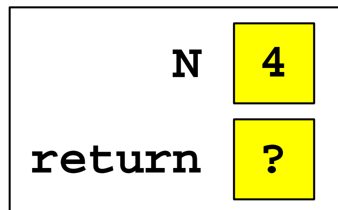
    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

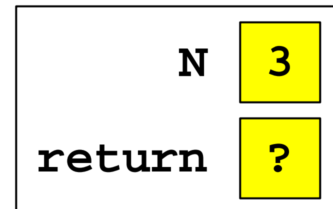
Behavior

and computing its return-value calls Factorial(1).

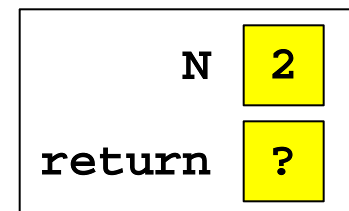
Factorial(4)



Factorial(3)



Factorial(2)




```
function Factorial (N : in Natural)
   return Positive is

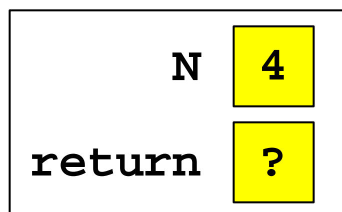
   begin - factorial

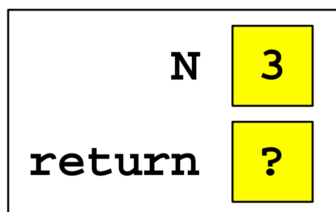
    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

   end Factorial;
```

Behavior

This begins a new execution, in which N == 1.



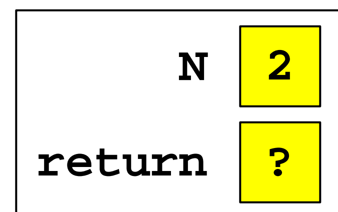


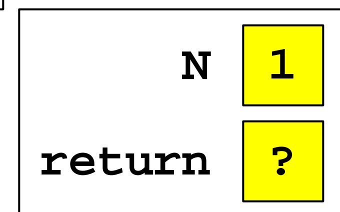

```
function Factorial (N : in Natural)
   return Positive is

   begin - factorial

    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

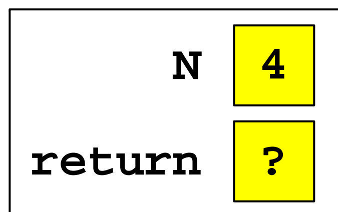
  end Factorial;
```

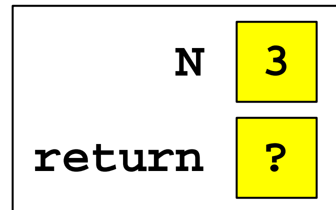




Behavior

The if executes, and the condition N > 1 is *false*, ...



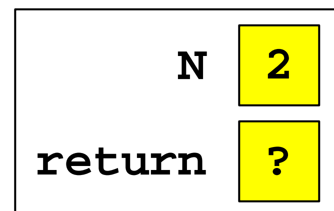


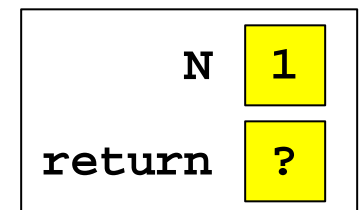

```
function Factorial (N : in Natural)
   return Positive is

   begin - factorial

    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

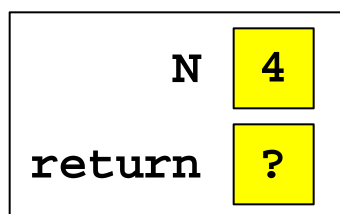
  end Factorial;
```

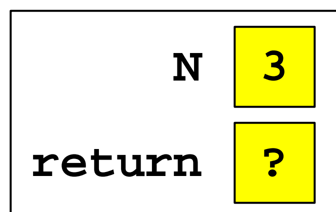




Behavior

so its return-value is computed as 1 (the base case)



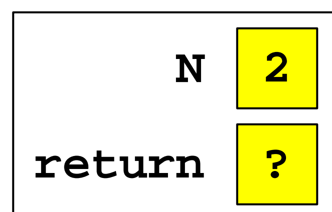


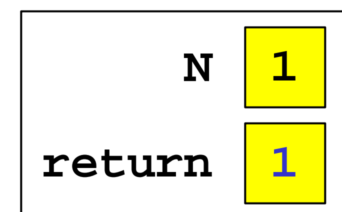

```
function Factorial (N : in Natural)
   return Positive is

   begin - factorial

    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

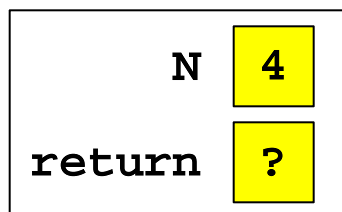




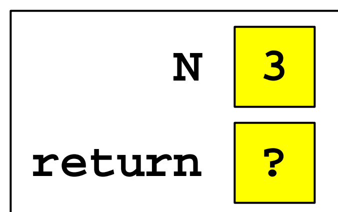
Behavior

Factorial(1) terminates, returning 1 to Factorial(2).

Factorial(4)



Factorial(3)



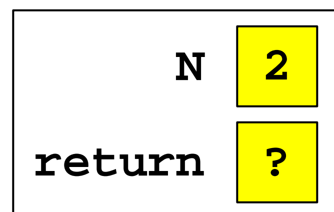

```
function Factorial (N : in Natural)
   return Positive is

   begin - factorial

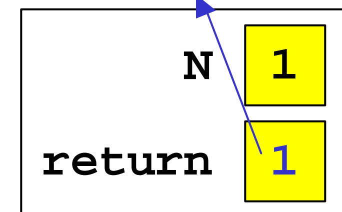
    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

Factorial(2)



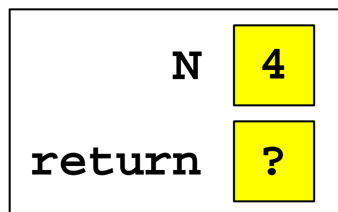
= 2 * 1



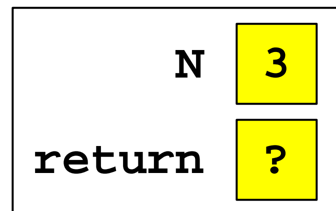
Behavior

Factorial(2) resumes, computing its return-value:

Factorial(4)



Factorial(3)



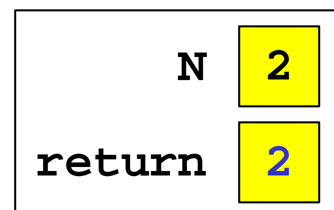

```
function Factorial (N : in Natural)
   return Positive is

   begin - factorial

    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

Factorial(2)

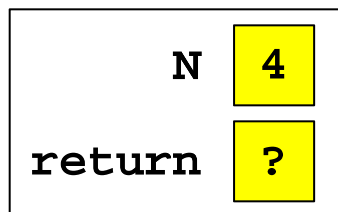


= 2 * 1

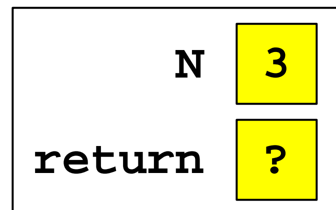
Behavior

Factorial(2) terminates, returning 2 to Factorial(3):

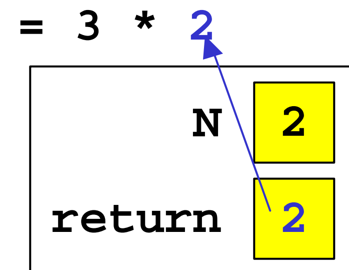
Factorial(4)



Factorial(3)



= 3 * 2




```
function Factorial (N : in Natural)
   return Positive is

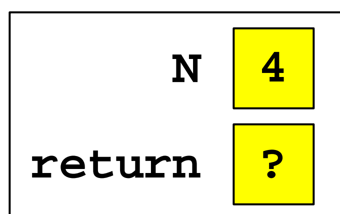
   begin - factorial

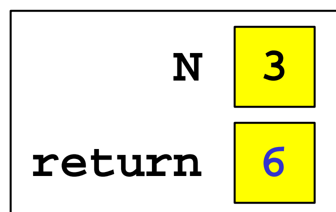
    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

Behavior

Factorial(3) resumes, and computes its return-value:






```
function Factorial (N : in Natural)
   return Positive is

   begin - factorial

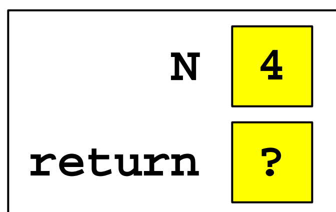
    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

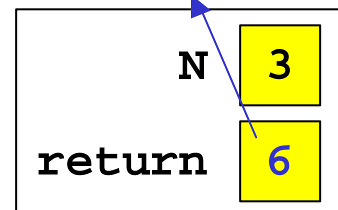
Behavior

Factorial(3) terminates, returning 6 to Factorial(4):

Factorial(4)



= 4 * 6




```
function Factorial (N : in Natural)
   return Positive is

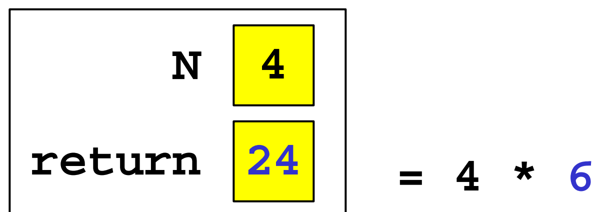
   begin - factorial

    if N > 1 then
      return N * Factorial(N-1);
    else
      return 1;
    end if;

  end Factorial;
```

Behavior

Factorial(4) resumes, and computes its return-value:



= 4 * 6

```
function Factorial (N : in Natural)
   return Positive is

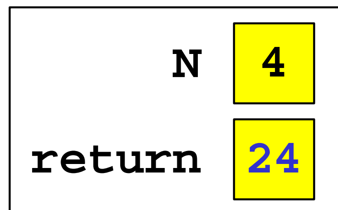
   begin - factorial

     if N > 1 then
       return N * Factorial(N-1);
     else
       return 1;
     end if;

  end Factorial;
```

Behavior

Factorial(4) terminates, returning 24 to its caller.




```
function Factorial (N : in Natural)
   return Positive is

   begin - factorial

     if N > 1 then
       return N * Factorial(N-1);
     else
       return 1;
     end if;

   end Factorial;
```

- If we time the for-loop version and the recursive version, the for-loop version will usually win, because the overhead of a function call is far more time-consuming than the time to execute a loop.
- However, there are problems where the recursive solution is more efficient than a corresponding loop-based solution.

For example the exponentiation problem:

Given two values x and n, compute x^n.

Example: $3^3 == 27$

Dual procedure recursion

- Sometimes have algorithms with multiple states
  - **Example**:
    p(x) = p(x-1) + q(x/2) {x>1}
    2 {x<=1}
    q(x) = q(x-3) * p(x-5) {x>3}
    x/3 {x<=3}
- This involves two recursive functions
  - **directly** recursive:
    - p calls itself
    - q calls itself
  - also **indirectly** recursive
    - p calls q which call p again
    - q calls p which call q again
- Ada requires that things must be **declared before they can be used**
  - You cannot both define p before q and define q before p.
  - must predefine one or both of the functions

Dual procedure recursion

```
-- first declare q, so p can refer to it
function q (x: in float) return float;

-- now define p. Full definition is possible
function p (x: in float) return float is
begin
  if (x<=1) then
     return 2;
  end if;
  return p(x-1) + q(x/2);
end p;

-- now provide full definition of q
function q(x: in float) return float is
begin
  if (x<=3) then
    return x/3;
  end if;
  return q(x-3) * p(x-5);
end q;
```

Merge sort

- **Merge sort** is a more efficient sorting algorithm than either *selection sort* or *bubblesort*
- Where bubblesort and select sort are **$O(n^2)$**, merge sort is **O(n log n)**
- The basic ides is that if you know you have 2 sorted lists, you can combine them into a single large sorted list by just looking at the first item in each list. Whichever is smaller is moved to the single list being assembled. There is then a new first item in the list from which the item was moved, and the process repeats.
- **The process overall is thus**:
  - Split the original list into two halves
  - Sort each half (using merge sort)
  - Merge the two sorted halves together into a single sorted list

Algorithm

```
procedure mergesort(first,last,array)
  mid= (first+last)/2
  mergesort(first,mid,array)
  mergesort(mid+1,last,array)
  rejoin_two_halves(mid,array)
end mergesort
```

Code 1(3)

```
procedure main is

type int_array is array(1..100) of integer;
tosort:int_array;

  procedure merge (a:in out int_array; low,mid,high:in integer) is
    temp: int_array;
    choose1: boolean;
    loop_low,loop_high:integer;

  begin
    loop_low:=low;
    loop_high:=high;

    for i in low..high loop
      if (loop_low>mid) then choose1:=false;
      elsif (loop_high>high) then choose1:=true;
      else choose1:= a(loop_low)<a(loop_high);
      end if; -- choose which side
```

Code 2(3)

```
    if choose1 then -- choose from low side
      temp(i):=a(loop_low);
      loop_low:=loop_low+1;
    else
      temp(i):=a(loop_high); -- choose from high side
      loop_high:=loop_high+1;
    end if;
  end loop;
  a:=temp;

end merge;
```

Code 3(3)

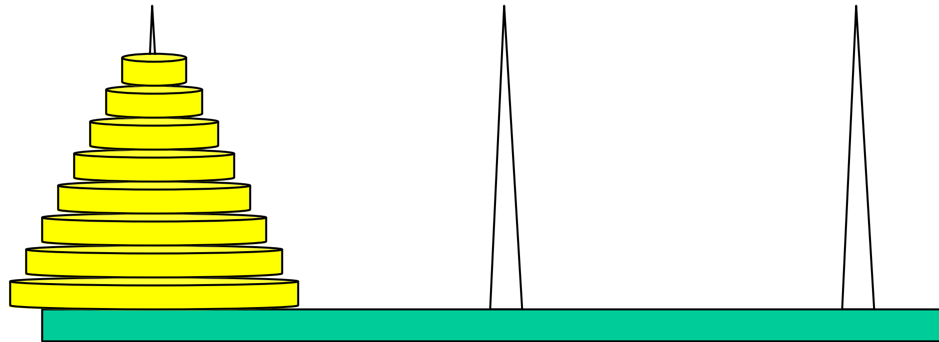
```
procedure mergesort(a: in out int_array;low,high:integer) is
   mid:integer;

begin
   if low<high then
     mid:= (high+low)/2;
     mergesort(a,low,mid);
     mergesort(a,mid+1,high);
     merge(a,low,mid,high);
   end if;
end mergesort;

begin
  -- something happens here to get the initial values of tosort
  -- then use mergesort to sort the array mergesort(tosort,1,100);
end main;
```

A Legend

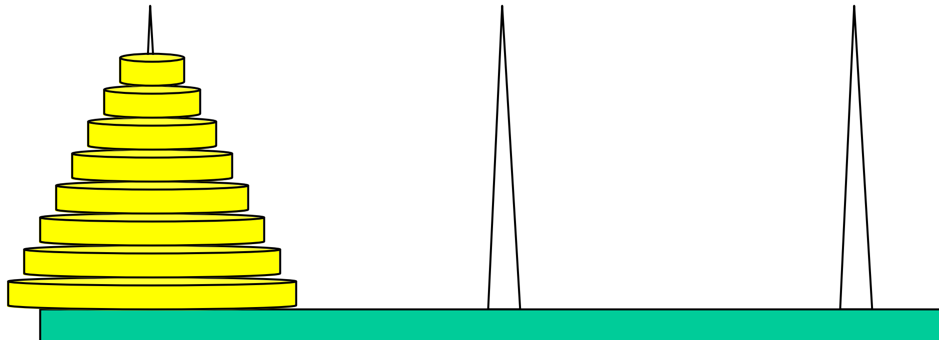
Legend has it that there were three diamond needles set into the floor of the temple of Brahma in Hanoi.



Stacked upon the leftmost needle were 64 golden disks, each a different size, stacked in concentric order:

A Legend (*Ct'd*)

The priests were to transfer the disks from the first needle to the second needle, using the third as necessary.

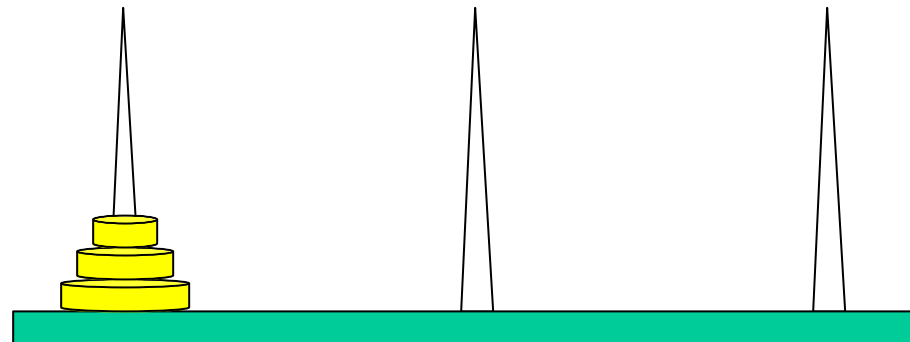


But they could *only move one disk at a time*, and could *never put a larger disk on top of a smaller one*.

When they completed this task, **the world would end**!

To Illustrate

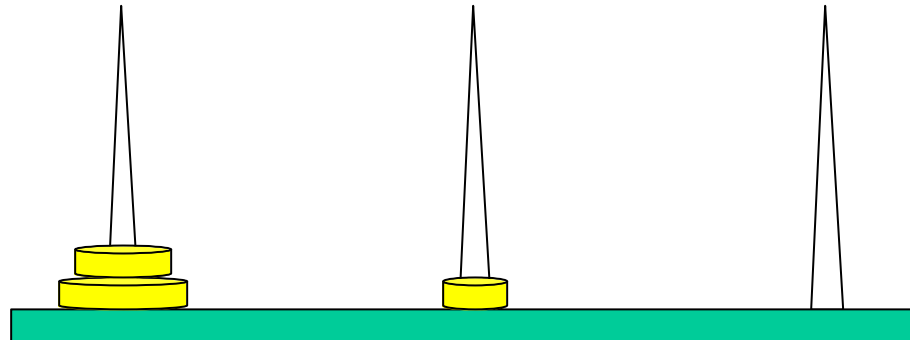
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



Since we can only move one disk at a time, we move the top disk from A to B.

Example

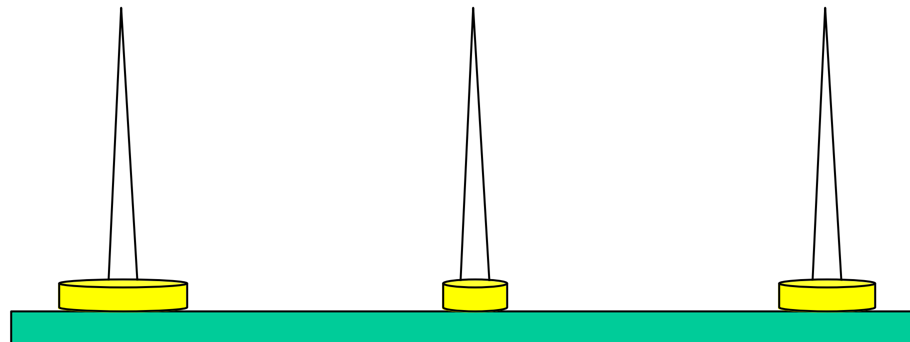
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



We then move the top disk from A to C.

Example (*Ct'd*)

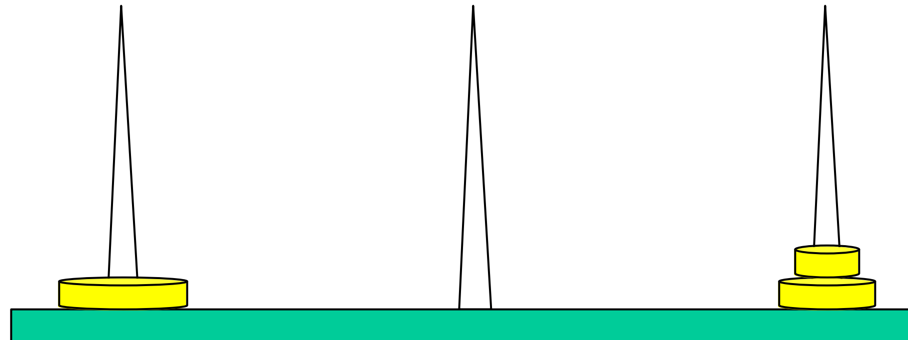
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



We then move the top disk from B to C.

Example (*Ct'd*)

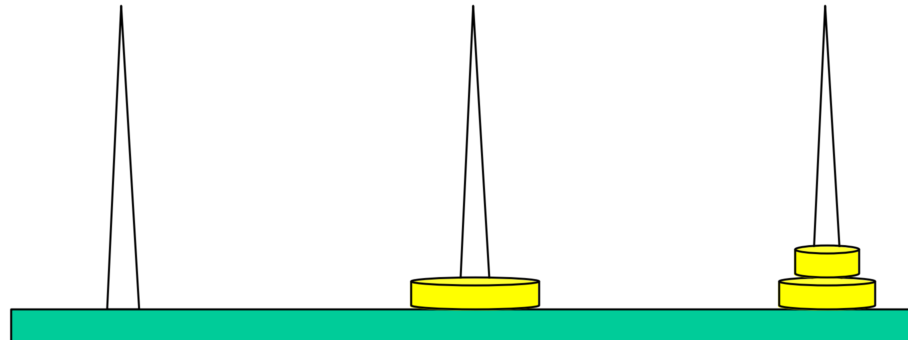
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



We then move the top disk from A to B.

Example (*Ct'd*)

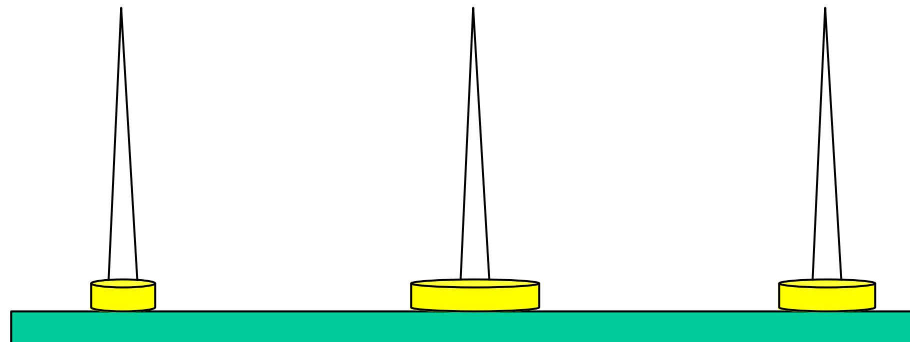
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



We then move the top disk from C to A.

Example (*Ct'd*)

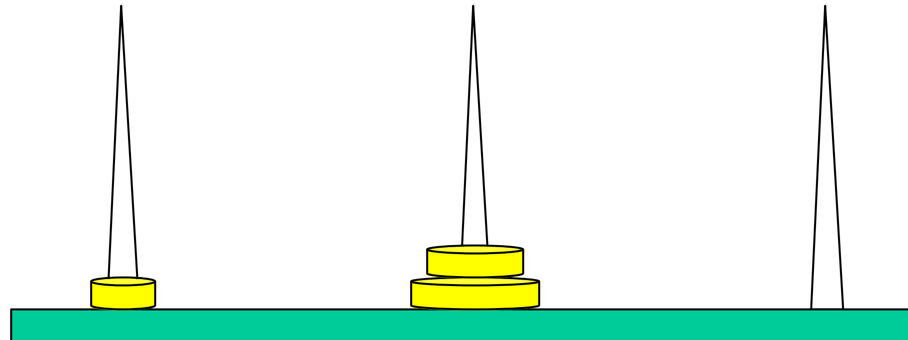
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



We then move the top disk from C to B.

Example (*Ct'd*)

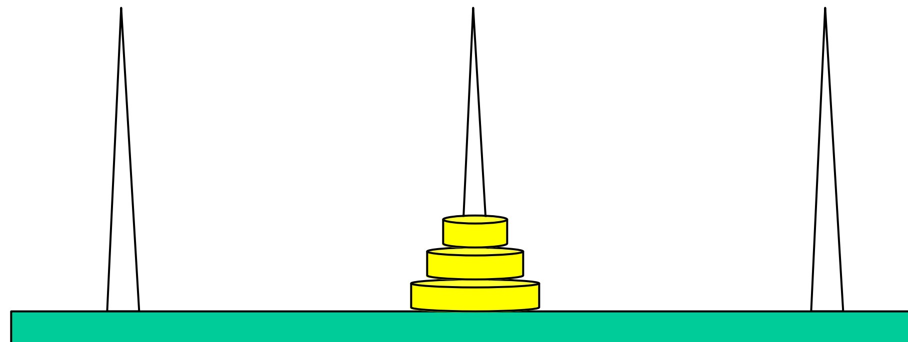
For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...



We then move the top disk from A to B.

Example (*Ct'd*)

For simplicity, suppose there were just 3 disks, and we'll refer to the three needles as A, B, and C...

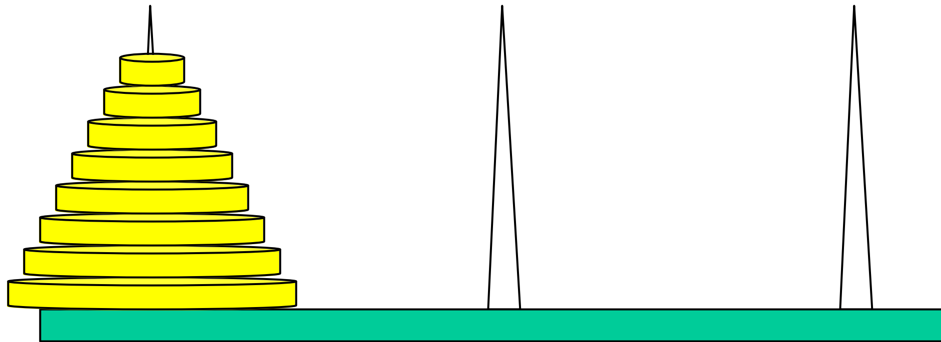


and we're done!

The problem gets more difficult as the number of disks increases...

Our Problem

Today's problem is to write a program that generates the instructions for the priests to follow in moving the disks.



While quite difficult to solve iteratively, this problem has a simple and elegant *recursive* solution.

Example

- Consider six disks instead of 64
- Suppose the problem is to move the stack of six disks from needle 1 to needle 2.
  - Part of the solution will be to move the bottom disk from needle 1 to needle 2, as a single move.
  - Before we can do that, we need to move the five disks on top of it out of the way.
  - After we have moved the large disk, we then need to move the five disks back on top of it to complete the solution.

Example

- We have the following process:
  - Move the top five disks to stack 3
  - Move the disk on stack 1 to stack 2
  - Move the disks on stack 3 to stack 2
- Notice that part of solving the six disk problem, is to solve the five disk problem (with a different destination needle). Here is where recursion comes in.

Algorithm

- ```
  hanoi(from,to,other,number)
      -- move the top number disks
      -- from stack from to stack to
  if number=1 then
      move the top disk from stack from to stack to
  else
      hanoi(from,other,to, number-1)
      hanoi(from,to,other, 1)
      hanoi(other,to, from, number-1)
  end
  ```
  ```

Ada Code

```
TOT_DISK: constant INTEGER:=64;

type int_array is array(1..TOT_DISK) of integer;

type towers is record
    num:integer:=0;
    disk:int_array;
end record;

type all_towers is array(1..3) of towers;

main_tower: all_towers;
```

Ada Code

```
procedure hanoi(from,to,other,number: in integer;
                tower                : in out all_towers);
    disk_move: integer;
    disk_loc : integer;

begin
    if (number=1) then
      disk_loc:= tower(from).num;
      disk_mov:=tower(from).disk(disk_loc);
      tower(from).num:=tower(from).num-1;
      put("Moving disk ");  put(disk_mov);
      put(" from tower ");  put(from);
      put(" to tower ");    put(to);
      new_line;
      tower(to).num:=tower(to).num+1;
      disk_loc:=tower(to).num;
      tower(to).disk(disk_loc):=disk_mov;
    else
      hanoi(from,other,to,number-1,tower);
      hanoi(from,to,other,1,tower);
      hanoi(other,to,from,number-1,tower);
    end if;
end hanoi;
```

Ada Code

```
procedure main is
begin
  for i in 1..TOT_DISK loop
    main_tower(1).disks(i):=i;
  end loop;
  main_tower(1).num:=TOT_DISK;
  hanoi(1,2,3,TOT_DISK,main_tower);
end main;
```

Execution

- `set TOT_DISK =3`
  `hanoi(1,2,3,3,main_tower);`

  ```
  hanoi(1,3,2,2) hanoi(1,2,3,1) move
  tower 1 to tower 2 hanoi(1,3,2,1) move
  tower 1 to tower 3 hanoi(2,3,1,1) move
  tower 2 to tower 3 hanoi(1,2,3,1) move
  tower 1 to tower 2 hanoi(3,2,1,2)
  hanoi(3,1,2,1) move tower 3 to tower 1
  hanoi(3,2,1,1) move tower 3 to tower 2
  hanoi(1,2,3,1) move tower 1 to tower 2
  ```

Analysis

Let's see how many moves it takes to solve this problem, as a function of *n*, the number of disks to be moved.

n	Number of disk-moves required
1	1
2	3
3	7
4	15
5	31
...	
i	2^i-1
64	2^{64}-1 (a big number)

Analysis (*Ct'd*)

How big?

Suppose that our computer and "super-printer" can generate and print 1,048,576 (2^{20}) instructions/second.

How long will it take to **print** the priest's instructions?

- There are 2^{64} instructions to print.
  - Then it will take $2^{64}/2^{20} = 2^{44}$ *seconds* to print them.
- 1 minute == 60 seconds.
  - Let's take $64 = 2^6$ as an approximation of 60.
  - Then it will take $\cong 2^{44} / 2^6 = 2^{38}$ *minutes* to print them.

Analysis (*Ct'd*)

Hmm. 2^{38} minutes is hard to grasp. Let's keep going...

- 1 hour == 60 minutes.
  - Let's take $64 = 2^6$ as an approximation of 60.
  - Then it will take $\cong 2^{38} / 2^6 = 2^{32}$ *hours* to print them.
- 1 day == 24 hours.
  - Let's take $32 = 2^5$ as an approximation of 24.
  - Then it will take $\cong 2^{32} / 2^5 = 2^{27}$ *days* to print them.

Analysis (*Ct'd*)

Hmm. 2^{27} days is hard to grasp. Let's keep going...

- 1 year == 365 days.
  - Let's take $512 = 2^9$ as an approximation of 365.
  - Then it will take $\cong 2^{27} / 2^9 = 2^{18}$ *years* to print them.
- 1 century == 100 years.
  - Let's take $128 = 2^7$ as an approximation of 100.
  - Then it will take $\cong 2^{18} / 2^7 = 2^{11}$ *centuries* to print them.

Analysis (*Ct'd*)

Hmm. 2^{11} centuries is hard to grasp. Let's keep going...

- 1 millenium == 10 centuries.
  - Let's take $16 = 2^4$ as an approximation of 10.
  - Then it will take $\cong 2^{11} / 2^4 = 2^7 =$ *128 millenia* just to *print* the priest's instructions (assuming our computer doesn't crash, in which case we have to start all over again).

How fast can the priests actually *move* the disks?

I'll leave it to you to calculate the data of the apocalypse...