Bisection method

- we can implement the bisection method using:
  - a loop to iterate until $f(c)$ is close to zero
  - a function handle to the function f

```
function [root] = bisect(f, a, b, tol)
%BISECT Root finding by bisection method
%   ROOT = BISECT(F, A, B, TOL) finds a root of
%   the function F known to lie in the range [A, B].
%   The root satisfies the inequality
%   ABS(F(ROOT)) <= TOL

if a == b
    error('range is zero');
elseif a > b
    tmp = a;
    a = b;
    b = tmp;
end
% continued on next slide
```

```
c = mean([a b]);
fc = f(c);
while abs(fc) > tol
    if sign(f(a)) ~= sign(fc) % root is to the left
        b = c;
    else                      % root is to the right?
        a = c;
    end
    c = mean([a b]);
    fc = f(c);
end
root = c;

end
```

Bisection method

- an alternate approach to implement the bisection method is to observe the following:
  - the bisection method repeatedly solves the same problem until it reaches the solution; i.e., finding a root via bisection looks something like:

1. bisect(*original range*)
2. bisect(*smaller range*)
3. bisect(*smaller range*)

   …

n) bisect(*smaller range*), done!

- can we have our bisection function call itself?
  - yes, we can make bisection be a *recursive* function

Recursive definitions

- in mathematics, a recursive definition is a definition that is defined in terms of itself
  - if you define something only in terms of itself, you end up with a circular definition; e.g.,
    - hill—a usually rounded natural elevation of land **lower than a mountain**
    - mountain—a landmass that projects conspicuously above its surroundings and is **higher than a hill**
- to prevent circular reasoning, a recursive definition requires one or more stopping points called *base cases*

Recursive definitions

- many mathematical entities can be defined recursively:
  - integer multiplication (positive m)

$0 \times n = 0$	base case
$m \times n = m + (m-1) \times n$	recursive definition

  - exponentiation (positive n)

$x^0 = 1$	base case
$x^n = x \times x^{n-1}$	recursive definition

  - factorial (positive n)

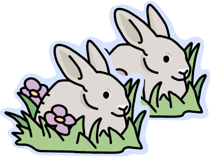
$0! = 1$	base case
$n! = n \times (n-1)!$	recursive definition

Factorial

- recursive definitions naturally lead to recursive implementations in functions:

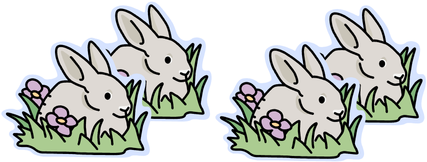
```
function f = fact(n)
%FACT Factorial of n
%   F = FACT(N) is the product of all of the integers
%   from 1 to N. N must be a positive integer.
if n == 0
    f = 1;
else
    f = n * fact(n - 1);
end
```

Rabbits



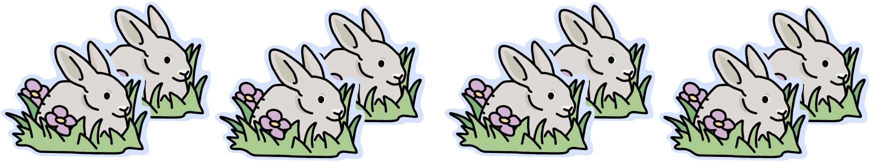
Month 0: 1 pair

0 additional pairs



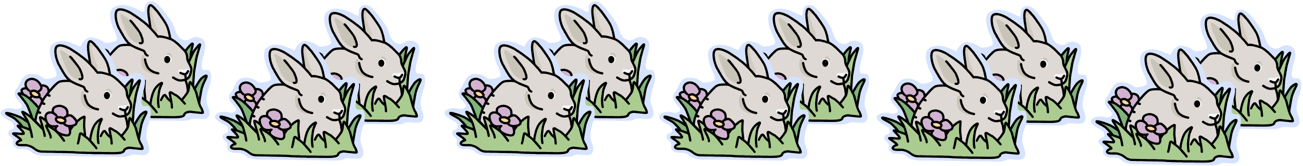
Month 1: first pair makes another pair

1 additional pair



Month 2: each pair makes another pair; oldest pair dies

1 additional pair



2 additional pairs

Month 3: each pair makes another pair; oldest pair dies

Fibonacci numbers

- the sequence of additional pairs
  - `0, 1, 1, 2, 3, 5, 8, 13, ...`

  are called Fibonacci numbers

- base cases
  - `F(0) = 0`
  - `F(1) = 1`
- recursive definition
  - `F(n) = F(n - 1) +  F(n - 2)`

Fibonacci numbers

- the recursive definition of the Fibonacci numbers leads naturally to a recursive implementation:

```
function fib = fibonacci(n)
% FIBONACCI nth Fibonacci number
%    FIB = FIBONACCI(N) computes the nth Fibonacci number
if n == 0
  fib = 0;
elseif n == 1
  fib = 1;
else
  fib = fibonacci(n - 1) + fibonacci(n - 2);
end
```

Bisection as a recursive function

```matlab
function [root] = bisect2(f, a, b, tol)
%BISECT2 Root finding by recursive bisection method
%   ROOT = BISECT(F, A, B, TOL) finds a root of
%   the function F known to lie in the range [A, B].
%   The root satisfies the inequality
%   ABS(F(ROOT)) <= TOL

if a == b
    error('range is zero');
elseif a > b
    tmp = a;
    a = b;
    b = tmp;
end                          % continued on next slide
```

Bisection as a recursive function

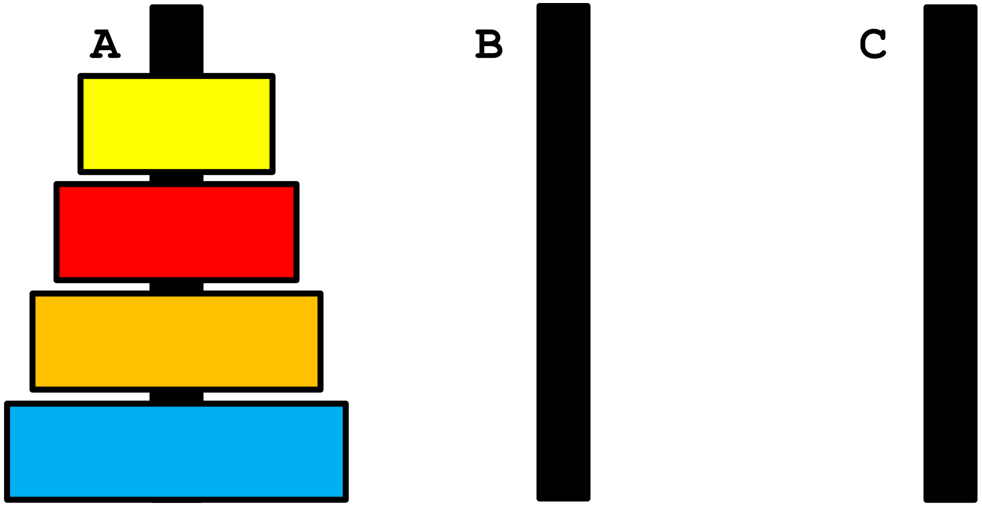
```
c = mean([a b]);
fc = f(c);
if abs(fc) <= tol
    root = c;
elseif sign(f(a)) ~= sign(fc)
    root = bisect2(f, a, c, tol); % root is to the left
else
    root = bisect2(f, c, b, tol); % root is to the right?
end

end
```

Recursion

- any problem that can be solved using recursion can also be solved using iteration
  - however, the recursive solution is often easier to implement

Towers of Hanoi



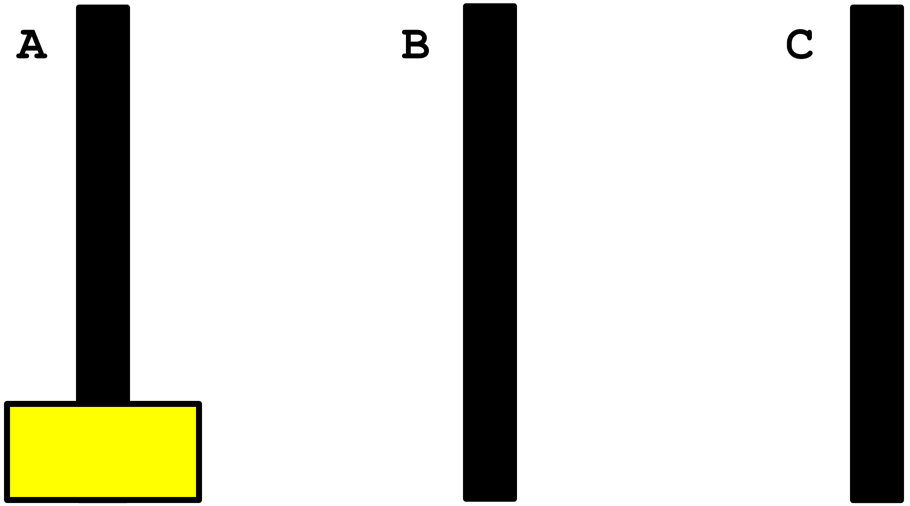
- move the stack of *n* disks from A to C
  - can move one disk at a time from the top of one stack onto another stack
  - cannot move a larger disk onto a smaller disk

Towers of Hanoi

- legend says that the world will end when a 64 disk version of the puzzle is solved
- several appearances in pop culture
  - Doctor Who
  - Rise of the Planet of the Apes
  - Survior: South Pacific

Towers of Hanoi

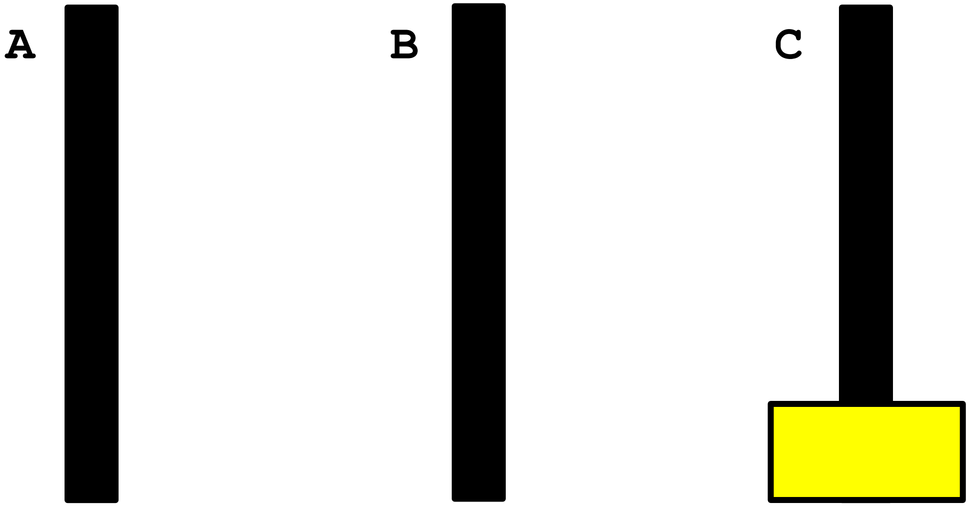
- n = 1



- move disk from A to C

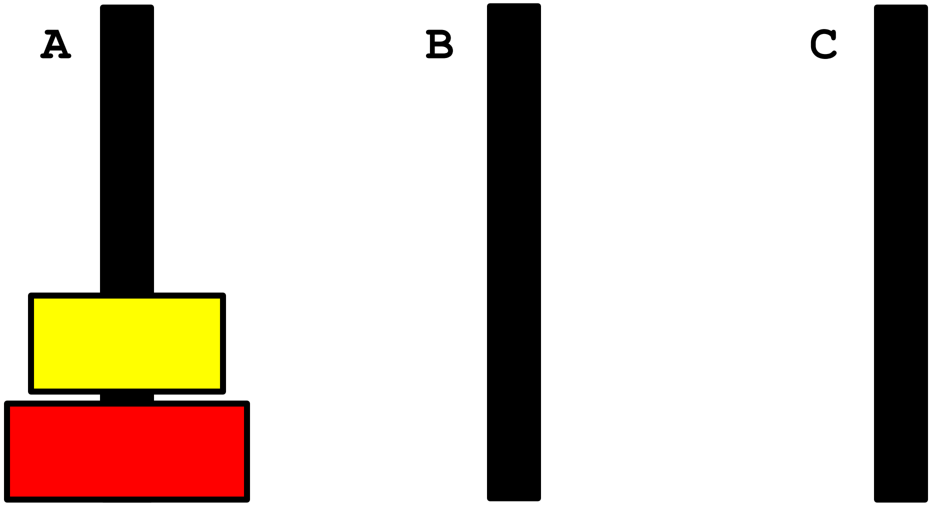
Towers of Hanoi

- n = 1



Towers of Hanoi

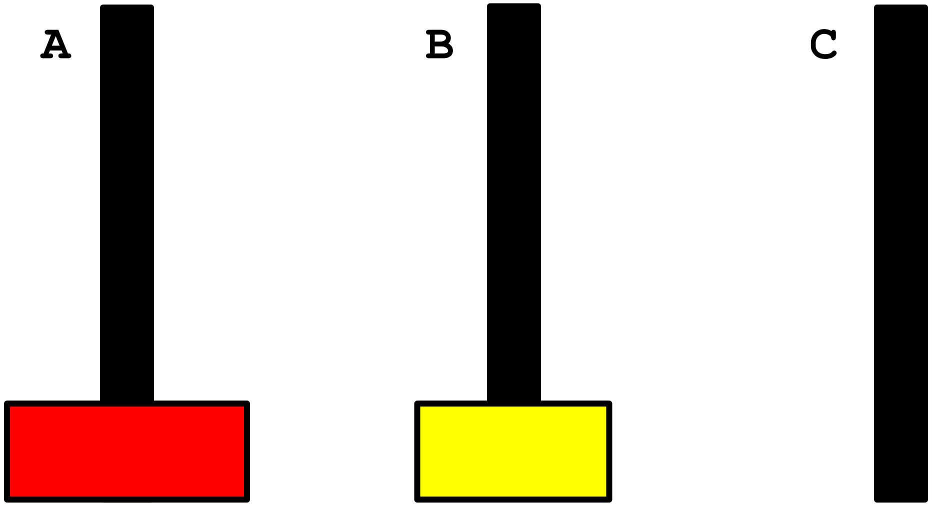
- n = 2



- move disk from A to B

Towers of Hanoi

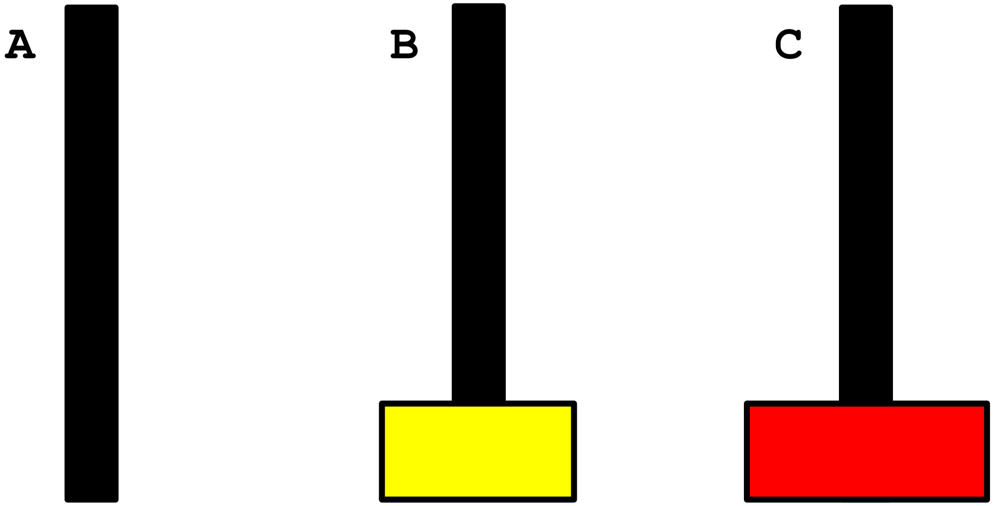
- n = 2



- move disk from A to C

Towers of Hanoi

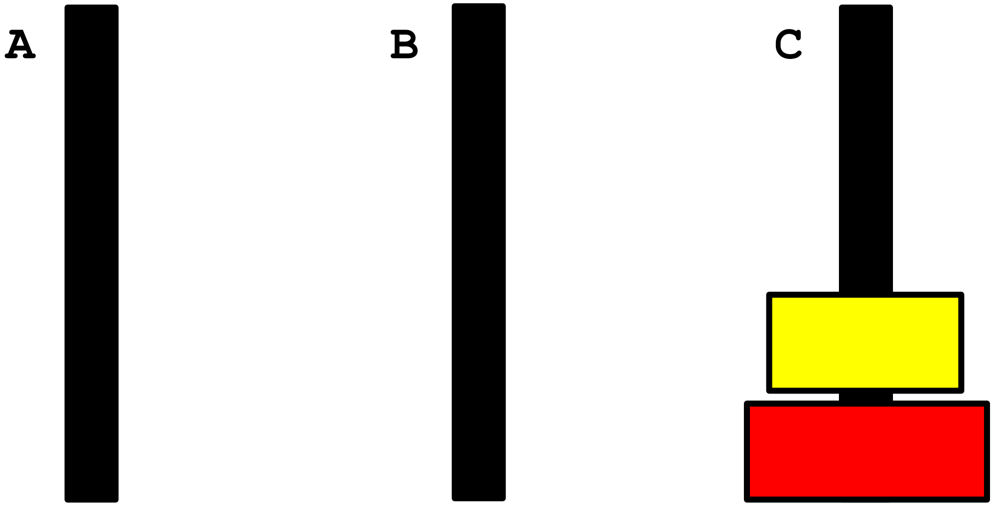
- n = 2



- move disk from B to C

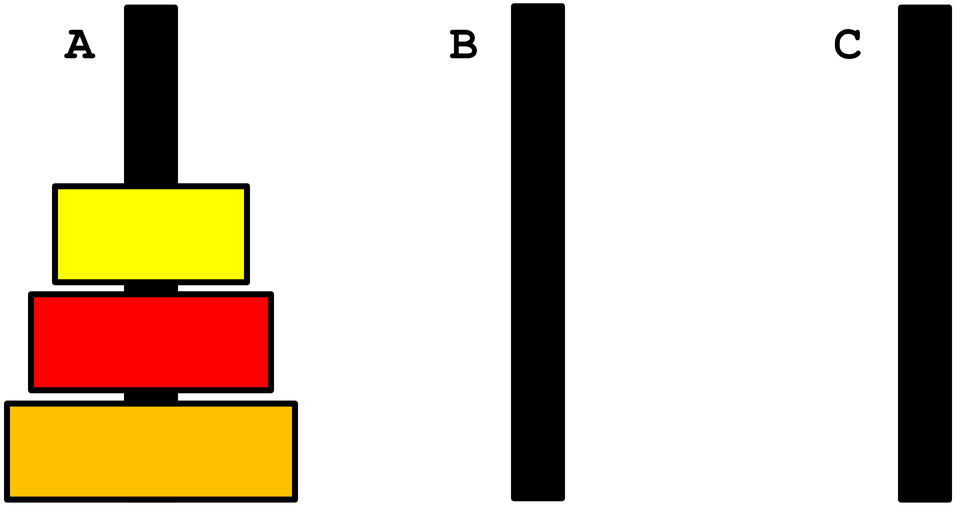
Towers of Hanoi

- n = 2



Towers of Hanoi

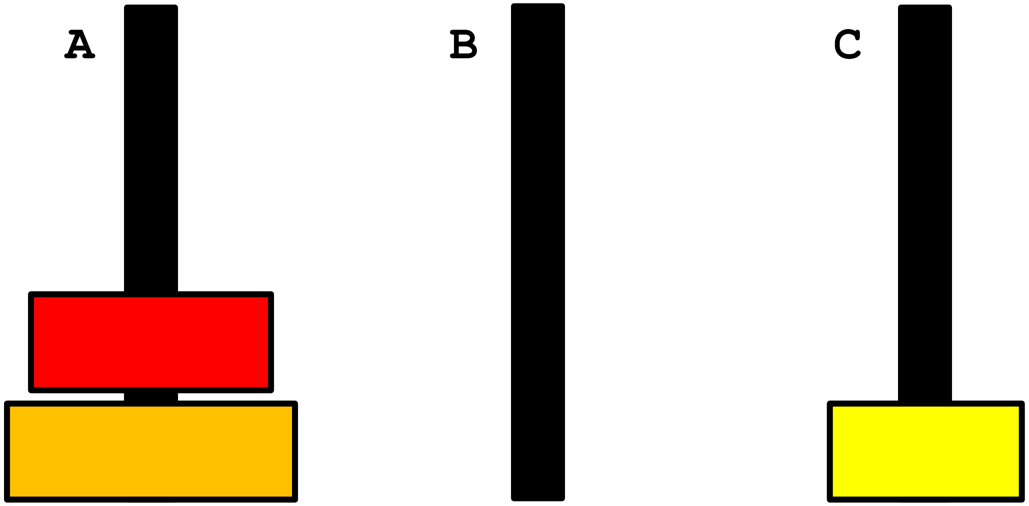
- n = 3



- move disk from A to C

Towers of Hanoi

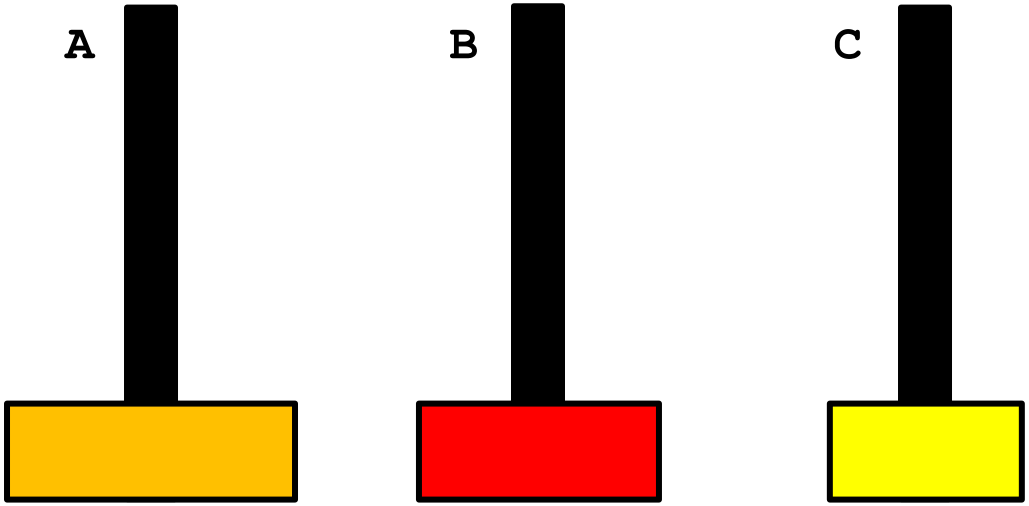
- n = 3



- move disk from A to B

Towers of Hanoi

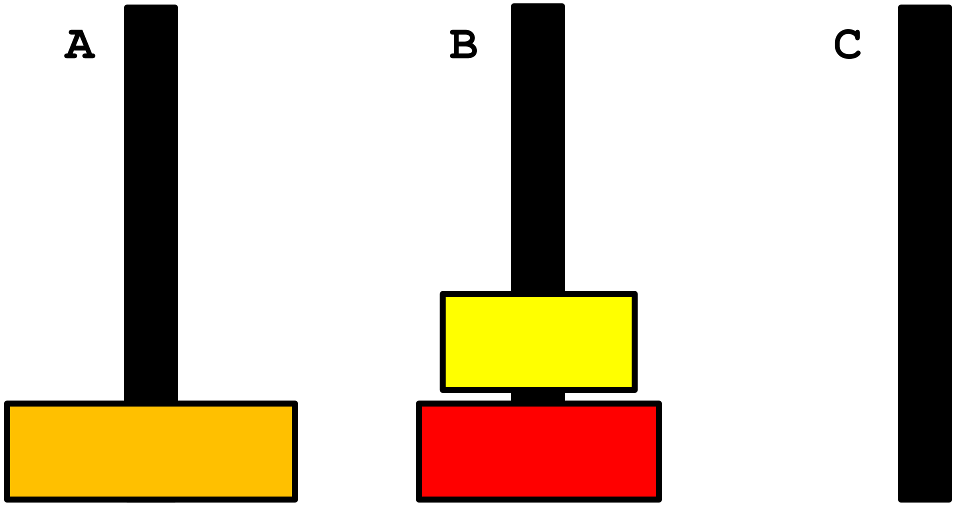
- n = 3



- move disk from C to B

Towers of Hanoi

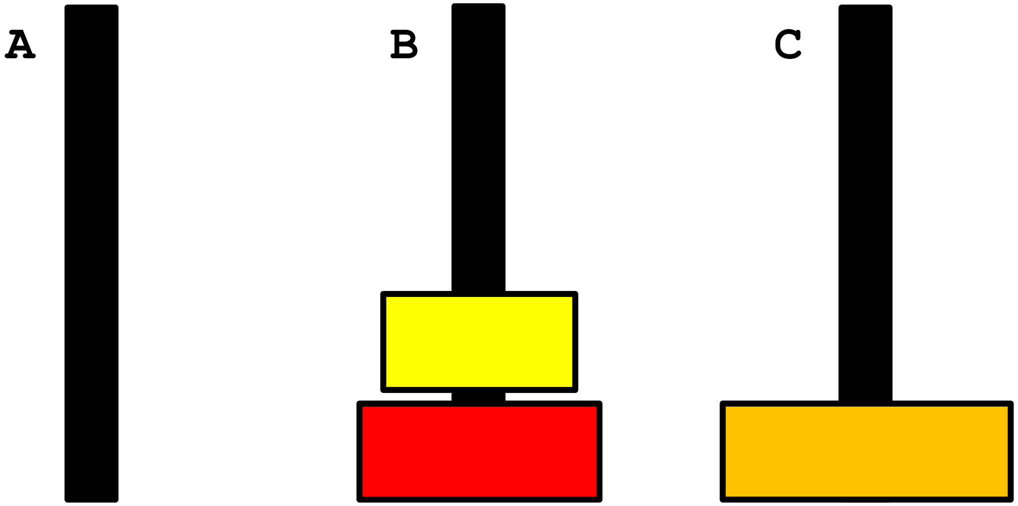
- n = 3



- move disk from A to C

Towers of Hanoi

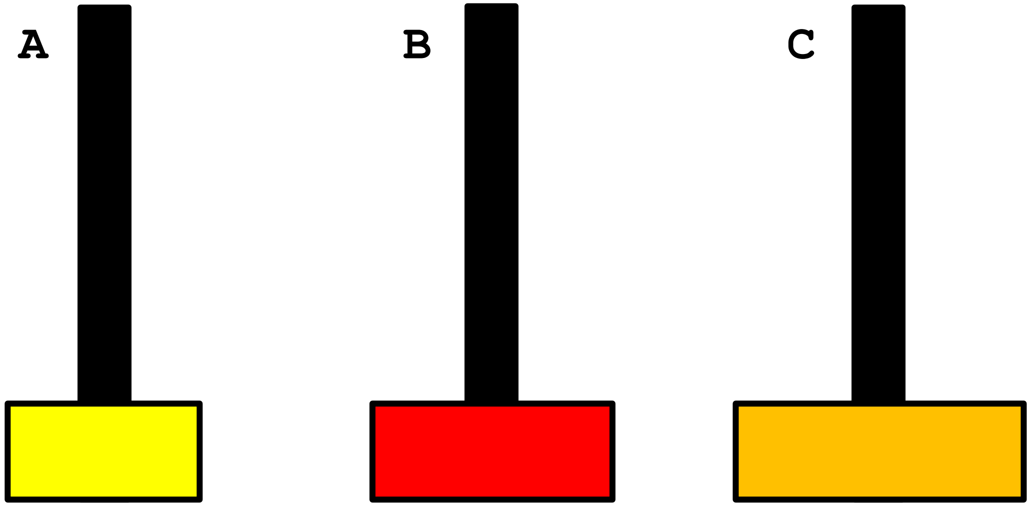
- n = 3



- move disk from B to A

Towers of Hanoi

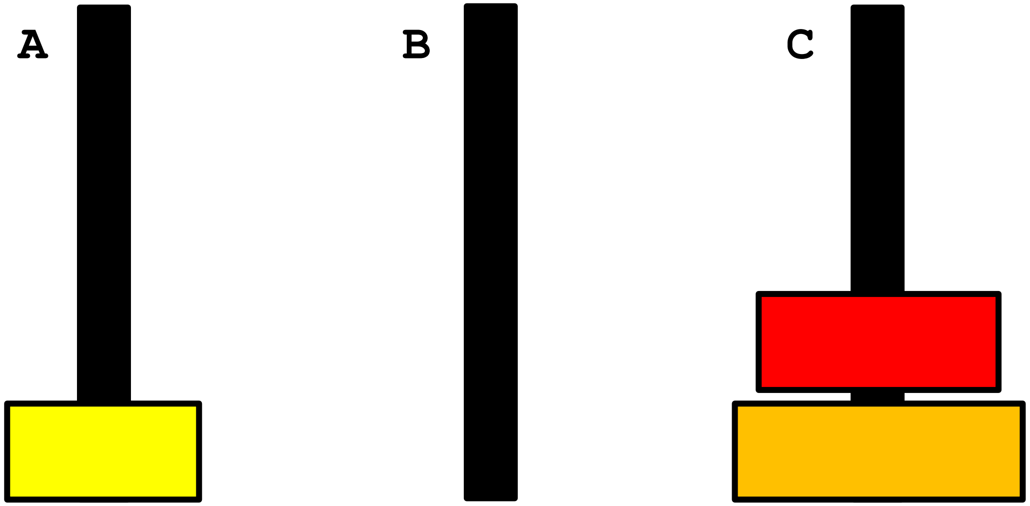
- n = 3



- move disk from B to C

Towers of Hanoi

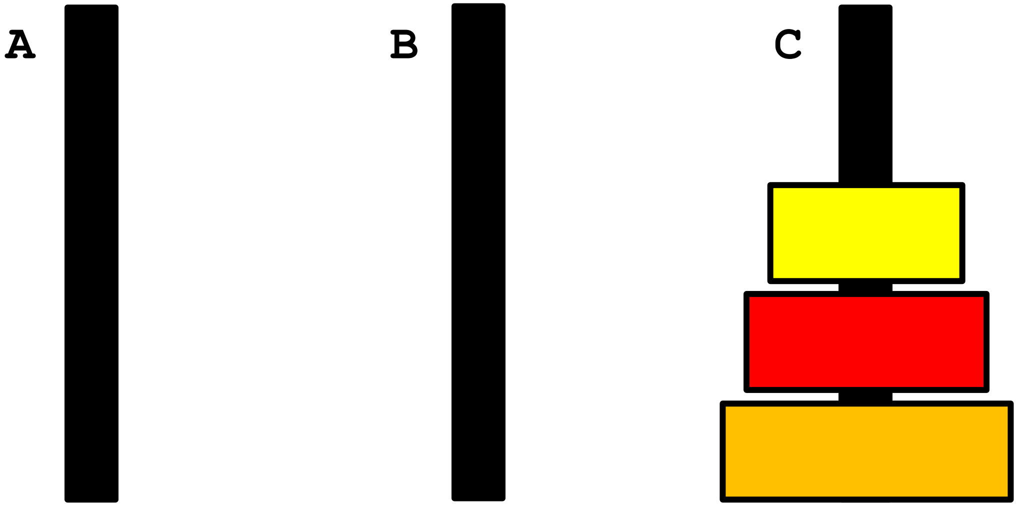
- n = 3



- move disk from A to C

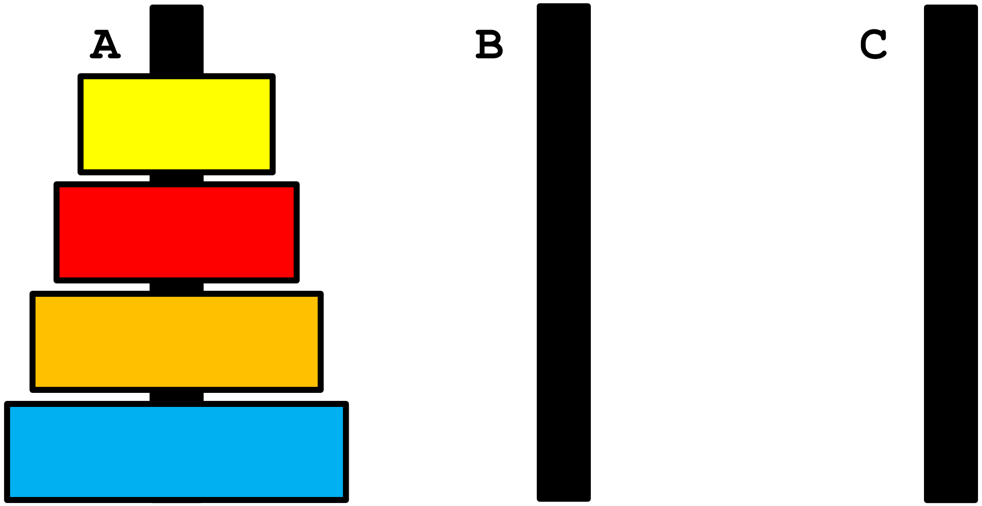
Towers of Hanoi

- n = 3



Towers of Hanoi

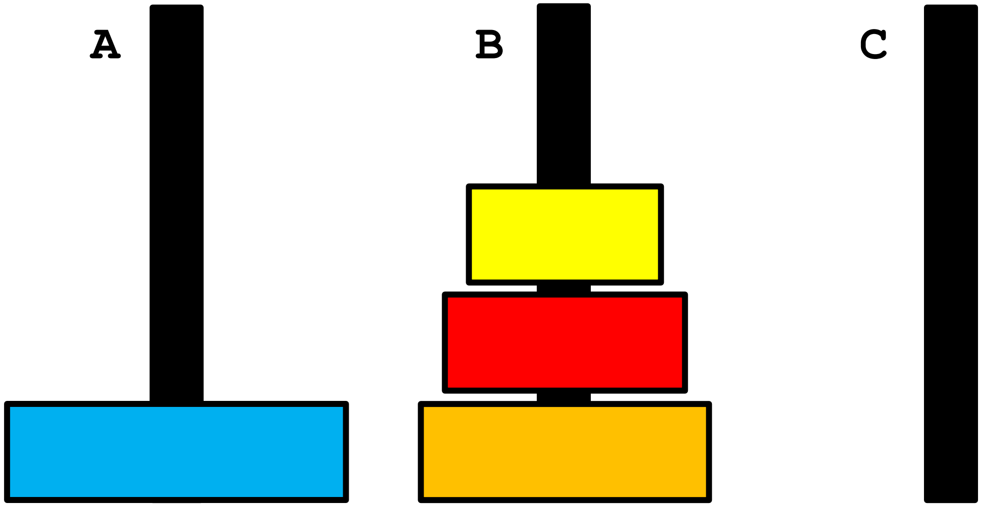
- n = 4



- move (n – 1) disks from A to B using C

Towers of Hanoi

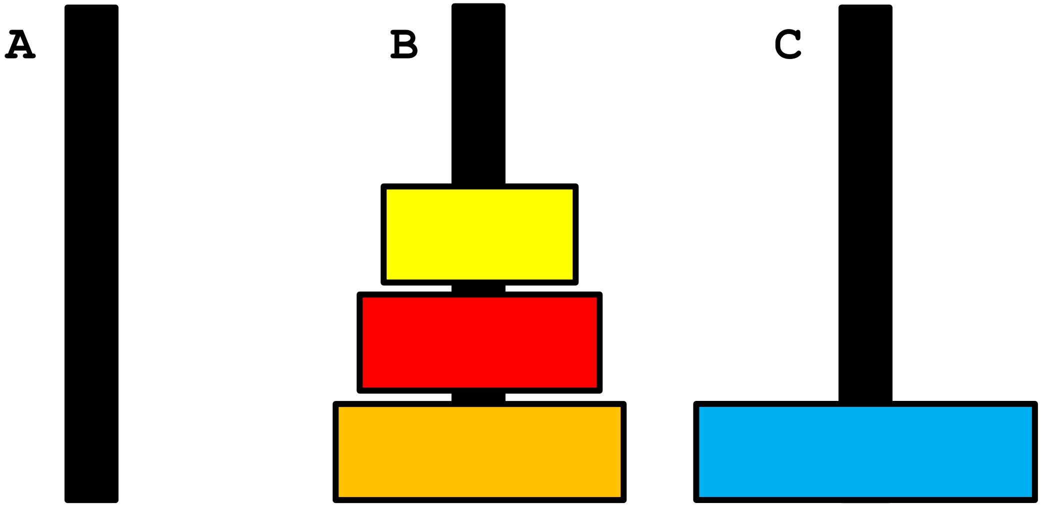
- n = 4



- move disk from A to C

Towers of Hanoi

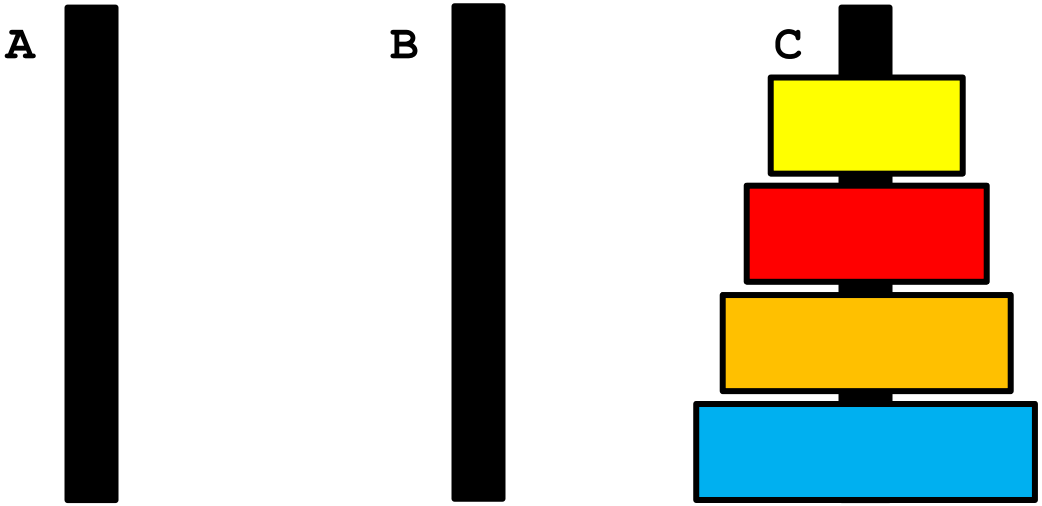
- n = 4



- move (n – 1) disks from B to C using A

Towers of Hanoi

- n = 4



- base case $n = 1$
  1. move disk from A to C
- recursive case
  1. move $(n - 1)$ disks from A to B
  2. move 1 disk from A to C
  3. move $(n - 1)$ disks from B to C

```
function [] = hanoi(n)
%HANOI Towers of Hanoi with n discs
%   HANOI(N) prints a solution for the Towers
%   of Hanoi problem for N discs.
move(n, 'A', 'C', 'B');
end

function [] = move(n, from, to, using)
%MOVE Recursive solution for Towers of Hanoi
if n == 1
    s = sprintf('move disc from %s to %s', from, to);
    disp(s);
else
    move(n - 1, from, using, to);
    move(1, from, to, using);
    move(n - 1, using, to, from);
end
end
```

Root finding in MATLAB

- MATLAB provides a function named **`fzero`** for root finding
  - “The **`fzero`** command is a function file. The algorithm, which was originated by T. Dekker, uses a combination of bisection, secant, and inverse quadratic interpolation methods.”
- **`fzero`** requires an initial estimate for the root and then finds a suitable interval to search for the root

```
fzero(@myf, 0.1)
ans =
    0.1421
```