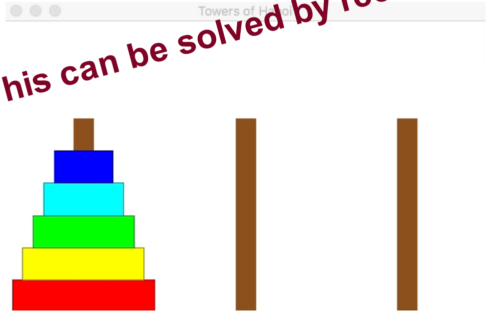
Recursion

Instructor: Jeeho Ryoo

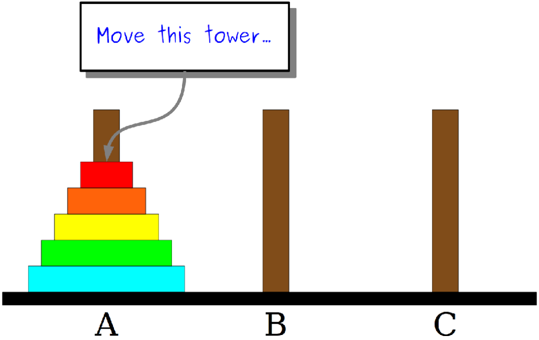
The Towers of Hanoi Puzzle



This can be solved by recursion!

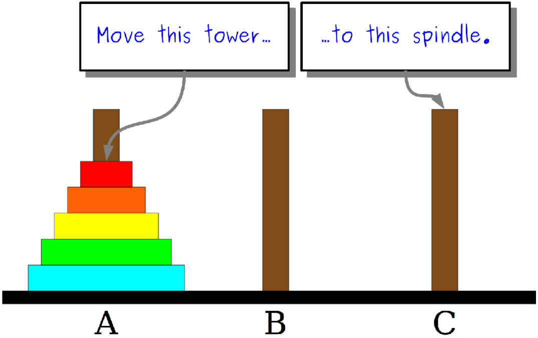
Towers of Hanoi

Here is the way the game is played:



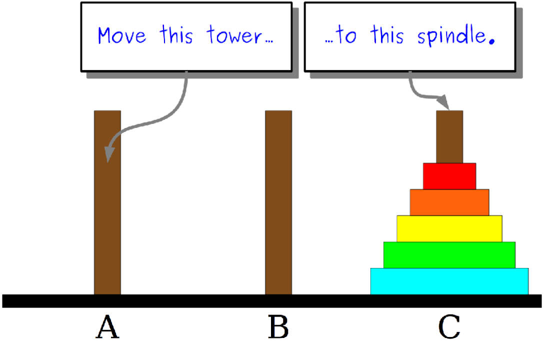
Towers of Hanoi

Here is the way the game is played:



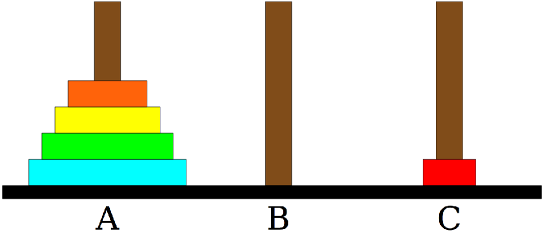
Towers of Hanoi

Here is the way the game is played:



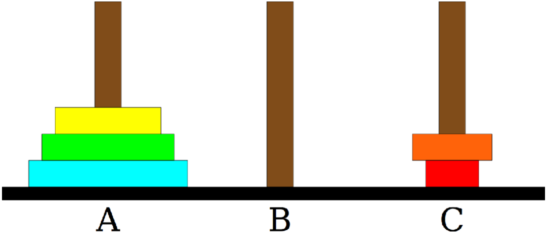
Towers of Hanoi

Here is the way the game is played:



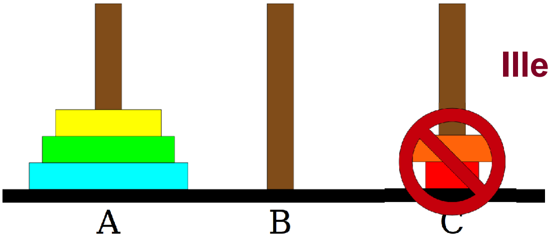
Towers of Hanoi

Here is the way the game is played:



Towers of Hanoi

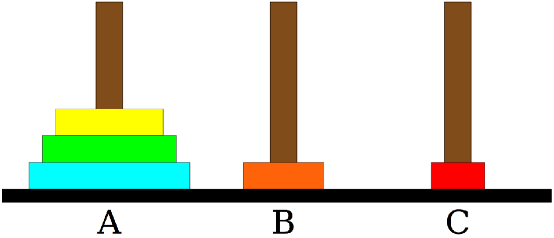
Here is the way the game is played:



Illegal move!

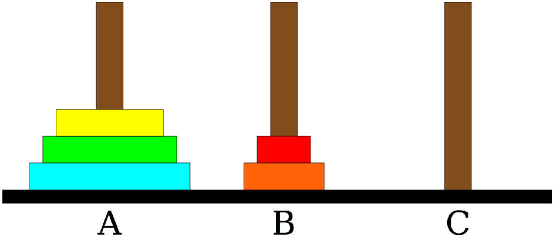
Towers of Hanoi

Here is the way the game is played:



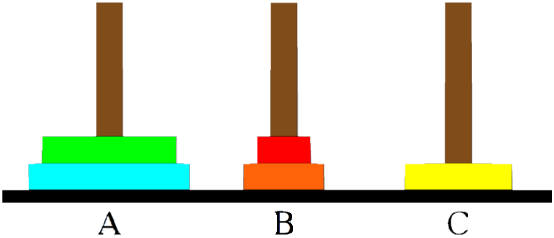
Towers of Hanoi

Here is the way the game is played:



Towers of Hanoi

Here is the way the game is played:



etc.

What is Recursion?

Recursion:

A problem solving technique in which problems are solved by reducing them to smaller problems *of the same form.*

Why Recursion?

1. Great style
2. Powerful tool
3. Master of control flow

Recursion In Programming

In programming, recursion simply means that a function will call itself:

```
int main() {
   main();
   return 0;
}
```

SEG FAULT!

(this is a terrible example, and will crash!)

main() isn't supposed to call itself, but if we do write this program, what happens?

We'll get back to programming in a minute...

Recursion In Real Life

Recursion

- How to solve a jigsaw puzzle recursively ("solve the puzzle")
  - Is the puzzle finished? If so, stop.
  - Find a correct puzzle piece and place it.
  - Solve the puzzle

ridiculously hard puzzle

Recursion In Real Life

Let's recurse on *you*.

How many students total are directly behind you in your "column" of the classroom?

Rules:

1. You can see only the people directly in front and behind you. So, you can't just look back and count.
2. You *are* allowed to ask questions of the people in front / behind you.

How can we solve this problem *recursively*?

Recursion In Real Life

Answer:

1. The first person looks behind them, and sees if there is a person there. If not, the person responds "0".
2. If there is a person, repeat step 1, and wait for a response.
3. Once a person receives a response, they add 1 for the person behind them, and they respond to the person that asked them.

In C

Every recursive algorithm involves at least **two** cases:

- base case: The simple case; an occurrence that can be answered directly; the case that recursive calls reduce to.

- recursive case: a more complex occurrence of the problem that cannot be directly answered, but can be described in terms of smaller occurrences of the same problem.

Three Musts of Recursion 🔑

1. Your code must have a case for all valid inputs

2. You must have a base case that makes no recursive calls

3. When you make a recursive call it should be to a simpler instance and make forward progress towards the base case.

More Examples!

The power() function:

Write a recursive function that takes in a number (*x*) and an exponent (*n*) and returns the result of x^n

Powers

$$x^0 = 1$$

$$x^n = x \cdot x^{n-1}$$

Powers

- Each previous call waits for the next call to finish (just like any function).

```
printf(power(5, 3));
```

```
// first call: power (5, 3)
// second call: power (5, 2)
// third call: power (5, 1)
// fourth call: power (5, 0)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

Powers

- Each previous call waits for the next call to finish (just like any function).

```
printf(power(5, 3));
```

```
// first call: power (5, 3)
```

```
// second call: power (5, 2)
```

```
// third call: power (5, 1)
```

```
// fourth call: power (5, 0)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

This call returns 1

Powers

- Each previous call waits for the next call to finish (just like any function).

```
printf(power(5, 3));
```

```
// first call: power (5, 3)
// second call: power (5, 2)
// third call: power (5, 1)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

equals 1 from call

this entire statement returns 5 * 1

Powers

- Each previous call waits for the next call to finish (just like any function).

```
printf(power(5, 3));
```

```
// first call: power (5, 3)
```

```
// second call: power (5, 2)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

equals 5 from call

this entire statement returns 5 * 5

Powers

- Each previous call waits for the next call to finish (just like any function).

```
printf(power(5, 3));
```

```
// first call: power (5, 3)
int power(int x, int exp) {
    if (exp == 0) {
        return 1;
    } else {
        return x * power(x, exp - 1);
    }
}
```

equals 25 from call

this entire statement returns 5 * 25

the original function call was to this one, so it returns 125, which is 5^3

Faster Method!

```
int power(int x, int exp) {
    if(exp == 0) {
        // base case
        return 1;
    } else {
        if (exp % 2 == 1) {
        // if exp is odd
            return x * power(x, exp - 1);
        } else {
            // else, if exp is even
            int y = power(x, exp / 2);
            return y * y;
        }
    }
}
```

Factorial

```
n! = n * (n-1) * (n-2) * (n-3) * … * 1
```

- Write a function that computes and returns the factorial of a provided number, recursively (no loops).
  - e.g. **`factorial(4)`** should return **`24`**
  - You should be able to compute the value of any non-negative number. (**`0! = 1`**).

Factorial

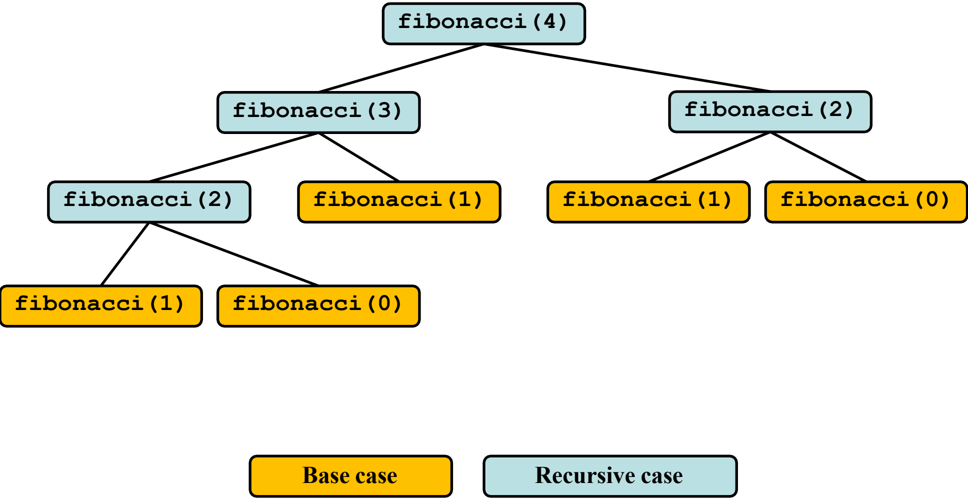
```
// Returns  n!, or 1  * 2 * 3 * 4 * ...  *n.
// Assumes n >= 0

int factorial(int n) {
    if (n < 0) {
        throw "illegal negative n";
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Fibonacci

`0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

- The Fibonacci sequence starts with 0 and 1, and each subsequent number is the sum of the two previous numbers.
- Write a function that computes and returns the nth Fibonacci number, recursively (no loops).
  - e.g. **`fibonacci(6)`** should return **`8`**

Recursive Tree



Fibonacci

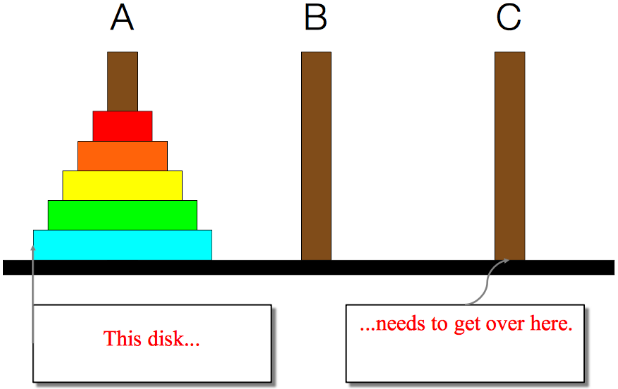
```
// Returns the i’th Fibonacci number in the sequence
// (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …)
// Assumes i >= 0.
int fibonacci(int i) {







}
```

Fibonacci

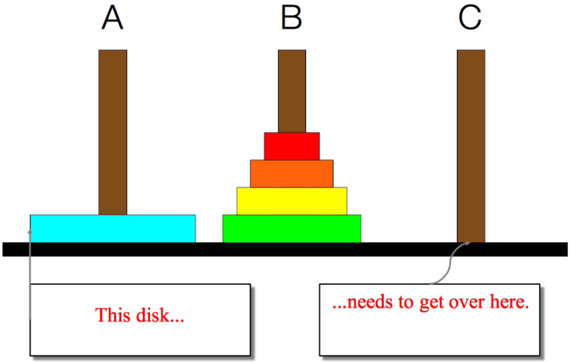
```
// Returns the i'th Fibonacci number in the sequence
// (0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …)
// Assumes i >= 0.
int fibonacci(int i) {
    if (i < 0) {
        throw "illegal negative index";
    else if (i == 0) {
        return 0;
    } else if (i == 1) {
        return 1;
    } else {
        // recursive case
        return fibonacci(i-1) + fibonacci(i-2);
    }
}
```

Back to Towers of Hanoi

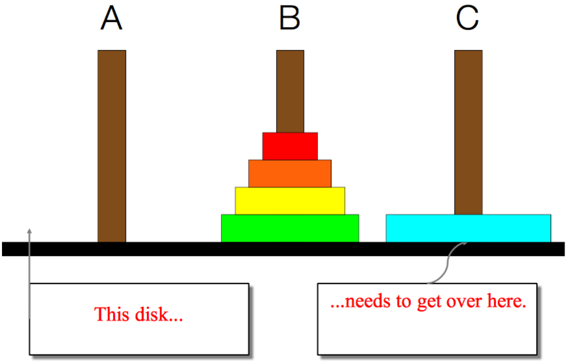
This is a hard problem to solve iteratively, but can be done recursively (though the recursive insight is not trivial to figure out)



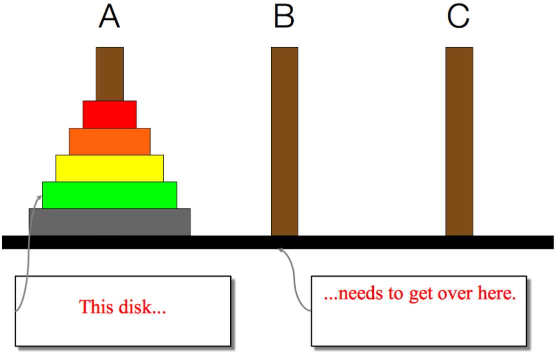
Back to Towers of Hanoi



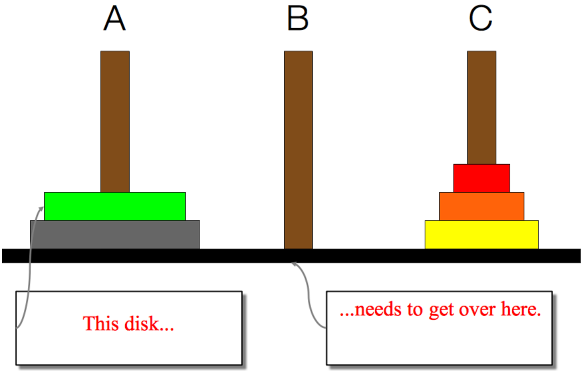
Back to Towers of Hanoi



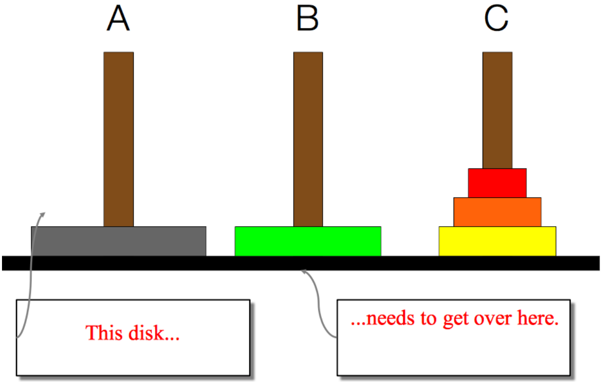
Back to Towers of Hanoi



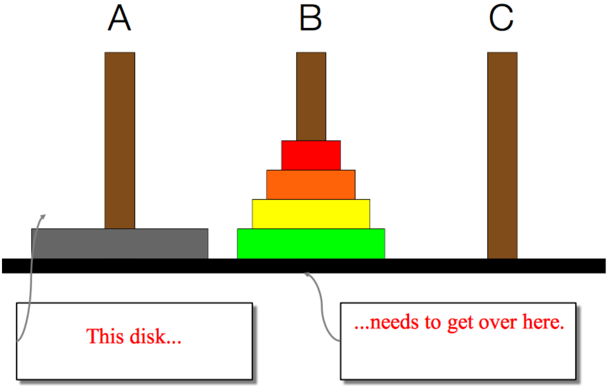
Back to Towers of Hanoi



Back to Towers of Hanoi



Back to Towers of Hanoi



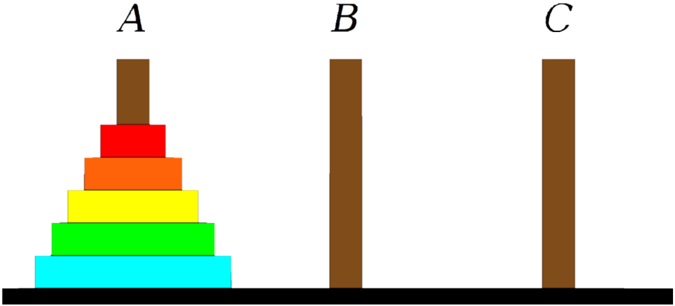
Back to Towers of Hanoi

- We need to find a very simple case that we can solve directly in order for the recursion to work.
- If the tower has size one, we can just move that single disk from the source to the destination.
- If the tower has more than one, we have to use the auxiliary spindle.

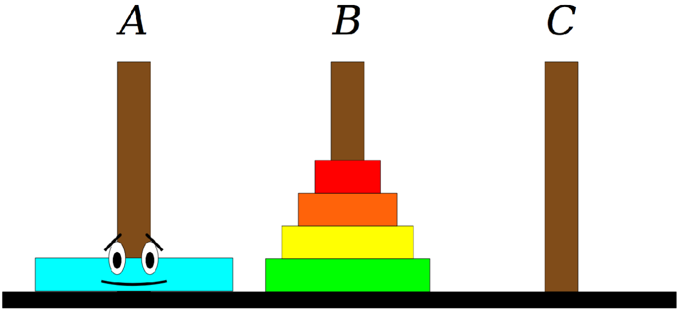
Back to Towers of Hanoi

- We can break the entire process down into very simple steps -- not necessarily easy to think of steps, but simple ones!

Back to Towers of Hanoi

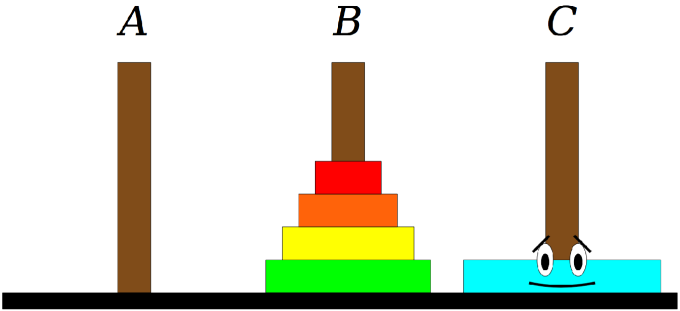


Back to Towers of Hanoi



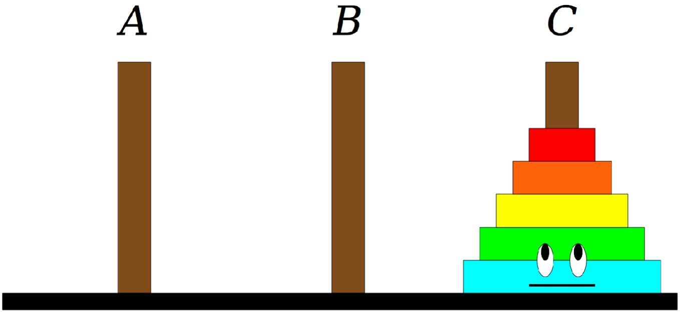
Step One: Move the four smaller disks from Spindle A to Spindle B.

Back to Towers of Hanoi



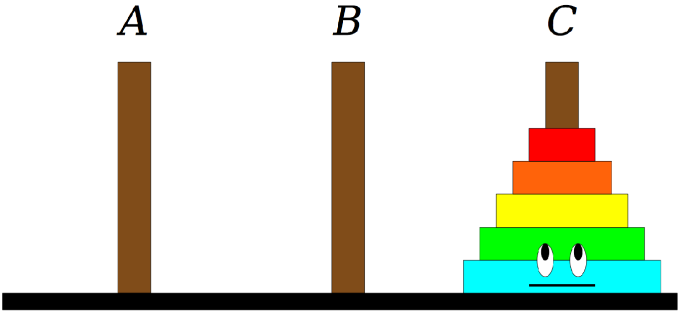
Step One: Move the four smaller disks from Spindle A to Spindle B.
Step Two: Move the blue disk from Spindle A to Spindle C.

Back to Towers of Hanoi



Step One: Move the four smaller disks from Spindle A to Spindle B.
Step Two: Move the blue disk from Spindle A to Spindle C.
Step Three: Move the four smaller disks from Spindle B to Spindle C.

Back to Towers of Hanoi



Repeat these steps at each stage!

Step One: Move the four smaller disks from Spindle A to Spindle B.
Step Two: Move the blue disk from Spindle A to Spindle C.
Step Three: Move the four smaller disks from Spindle B to Spindle C.