Data Structures Recursion

Phil Tayco

Slide version 1.4

Nov. 1, 2021

Recursion

Function calling

- We’re used to function calls appearing as one place in the code calling a function that exists somewhere else
- When the function is called, control goes to that function
- When the function is complete, control goes back to where the function was originally called
- Function calls can be done within functions as well following the same pattern
- Function communication is the heart of how complex programs are broken down into separate manageable parts

Recursion

What is recursion?

- Recursion is designing a function such that within the function, it may call itself
- Normally, a function like main calls function f, f then calls g, and then when g is done, it returns to f which then eventually returns to main
- If a function f uses recursion, then within it is a line (or lines) of code where function f may be called again
- Recursion ends up being a form of repetition like you would code with loops except the loop in this case is through function calling

Recursion

- Here is an example of typical coding where a function called based on a certain condition:
  - Function f has variables a, b, and c. It then says something like if (a>1) c=g(a, b);
  - If a is greater than 1, the values of a and b are sent into g and we wait for g to finish and return a value
  - g takes control. Its input parameter variables are also named a and b, and are local to g which will receive the values of a and b from f
  - g executes and eventually returns a value. Function g is now complete and its own a and b variables disappear
  - f resumes control and the value returned from g gets assigned to c within it. Function f continues from there
  - Note that if a had not originally been greater than 1, function g would not have been called

Recursion

- Now let's say f is the recursive function
- The **process remains the same** but now instead of calling function g within f, we'll call function f
  - Function f has variables a, b, and c. It then says something like if (a>1) c=f(a, b);
  - If a is greater than 1, the values of a and b are sent into f′ and we wait for f′ to finish and return a value
    - For this example, we'll refer to the 2nd time f is called as f′
  - f′ takes control. Its input parameter variables are also named a and b, and are local to f′ which will receive the values of a and b from f
  - f′ executes and eventually returns a value. Function f′ is now complete and its own a and b variables disappear
  - f resumes control and the value returned from f′ gets assigned to c within it. Function f continues from there
  - Note that if a had not originally been greater than 1, function f′ would not have been called

Recursion

Recursive communication

- "f" is the first execution of the function while "f'" is the second call
- It can be confusing, but the key is that **both will execute the same code with their own given values of a and b in their own function execution instances**
- How does the recursive loop stop? Notice that f is only called again when "if (a>1)" results in true
- If at any time a is not > 1, then calling f(a,b) would not happen
- This means that "a" is essentially the loop control variable and the recursion stops when a is not greater than 1 at some point

Recursion

Learning recursion

- Learning the concept is often done through mastering 2 key milestones:
  - Being able to walk through and detail all that is recurring with existing recursive code
  - Being able to design a recursive solution that can be translated into code
- We will next look at some examples to help figure out how to design and write recursive code
- Then, we will apply the code and walk through the execution phases to see the recursion in action

Recursion

Factorials

- Start with a math example
- A factorial is an integer multiplied by the each number in the series from that number descending to one
- Represented with an exclamation point, examples include:
  - 5! = 5 * 4 * 3 * 2 * 1
  - 4! = 4 * 3 * 2 * 1
  - 3! = 3 * 2 * 1
  - 2! = 2 * 1
  - 1! = 1
  - 0! = 1
- No negative number factorials can be performed
- The factorial of 0 is 1

Recursion

Patterns

- Notice those answers can be restated (note the color usage):
  - 0! = 1
  - 1! = 1
  - 2! = 2 * 1 = 2 * 1!
  - 3! = 3 * 2 * 1 = 3 * 2!
  - 4! = 4 * 3 * 2 * 1 = 4 * 3!
- Or restated in general terms (we'll call this the **recursive case**)
  - n! = n * (n – 1)!
- The factorials of 0 and 1 are also 1. They don't fit the general case so we consider these to be special cases (we'll call these **base cases**)
- Put the two cases together and you can state that the factorial for any non-negative number n:
  - If n = 0 or n = 1, the answer is 1
  - Otherwise, the answer is n * (n – 1)!
- This can be stated the same way in program code

Recursion

```
long factorial (int n)
{
  if (n == 0 || n == 1)
     return 1;
  return n * factorial(n - 1);
}
```

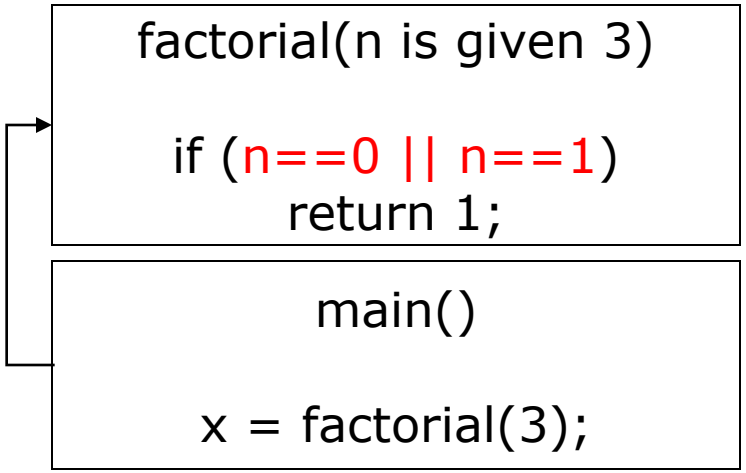
Recursion

Code analysis

- The code is simple in terms of number of lines and following the mathematical model, but can be complex in terms of trying to understand the program flow of control
- The code contains a line that makes a function call to itself. This is the part of the code that is using “recursion”
- Note that each time the recursive function call is made, the value passed in is different than the value that it was given. In this case, we pass into the next call a value of (n-1)
- Eventually, the recursive function calls must stop, this is when the base case is reached. Notice with each recursive function call, the value of n passed in goes down by 1 (n will reach 1 at some point in the recursion)
- When the base case is reached, the simple value of 1 in this case is returned. All the recursive function calls that were made are now “unwound”

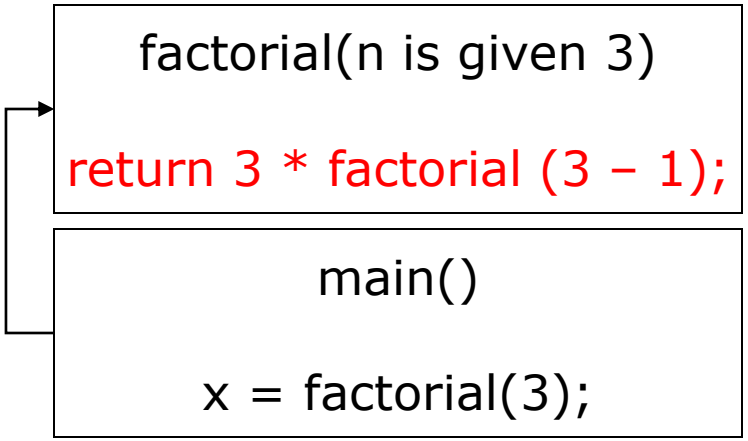
Recursion

- We start with main calling “factorial(3)”
- We enter the function execution of factorial(3)
- In the code, 3 goes into a local value of n
- n is neither 0 nor 1 so the condition for the if statement is false (the base case is not met)



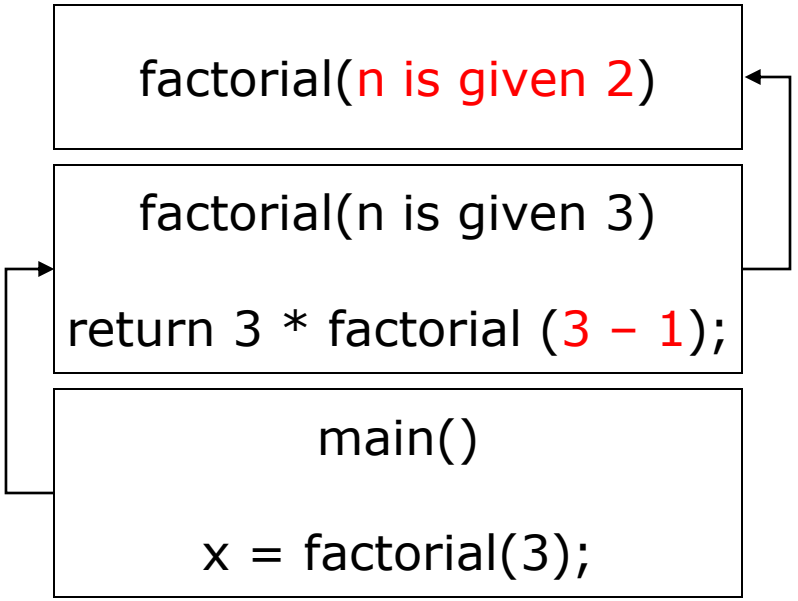
Recursion

- We go to the next line of code which is returning the "n * factorial(n-1);"
- n during this function execution was given 3, so the code being executed is return 3 * factorial(3-1);
- This will now result in a function call to the factorial function with a value of 2…



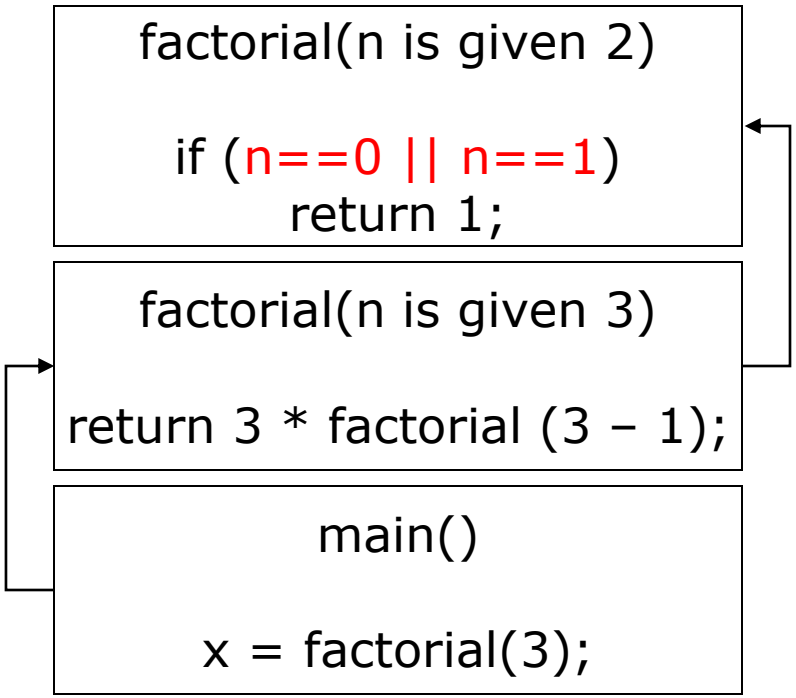
Recursion

- The key thing to note here is that we are now in a completely separate instance of function execution
- In this world, the same factorial function code executes with a starting value of n as 2



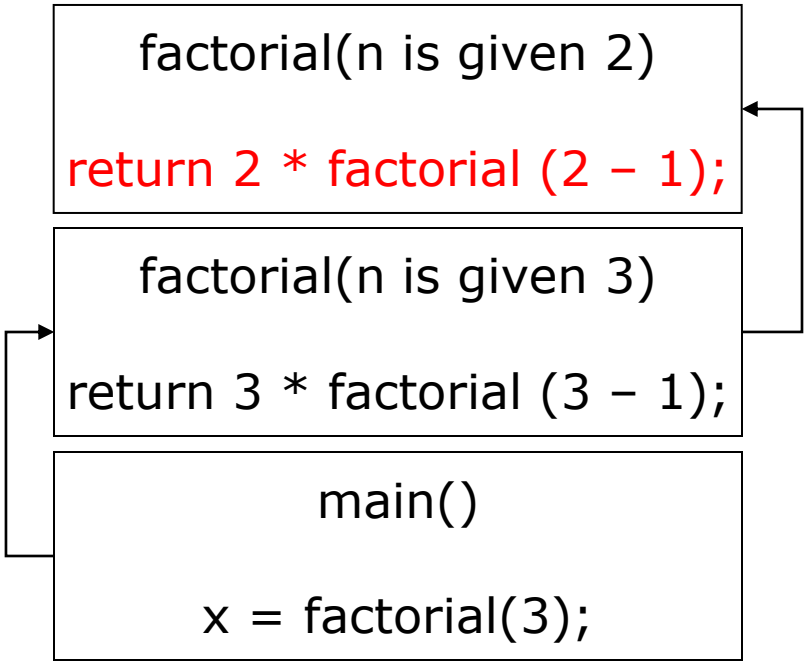
Recursion

- Here, n is checked again and is not 0 nor 1 so we go to the next line of code



Recursion

- "return n * factorial(n-1);" results in another function call, this time calling factorial again with an input parameter value of 1



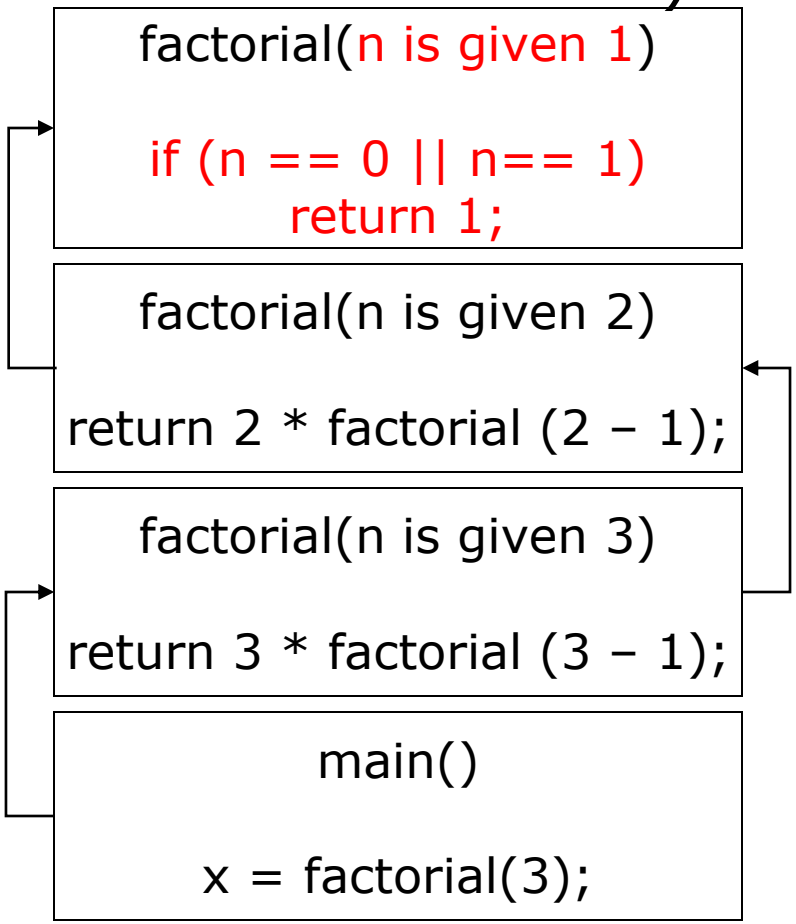
Recursion

Recap

- The main function begins with a standard function call to factorial passing a value of 3
- In the first instance of factorial(3), the base case check is false (n does not equal 0 nor 1)
- Therefore, in factorial(3), the line “return n * factorial(n-1);” is executed
- This temporarily halts execution in factorial(3) as it must wait for the return of factorial(n-1)
- This takes us to the next function instance of factorial(2)
- Note that factorial(3) and factorial(2) are separate execution instances just like any other function call. These function calls will still be using the same code
- The same sequence occurs in factorial(2) where another recursive function call will take place to factorial(1)…

Recursion

- Now n is 1, so the if statement results in true (the "base case" is reached) which returns 1



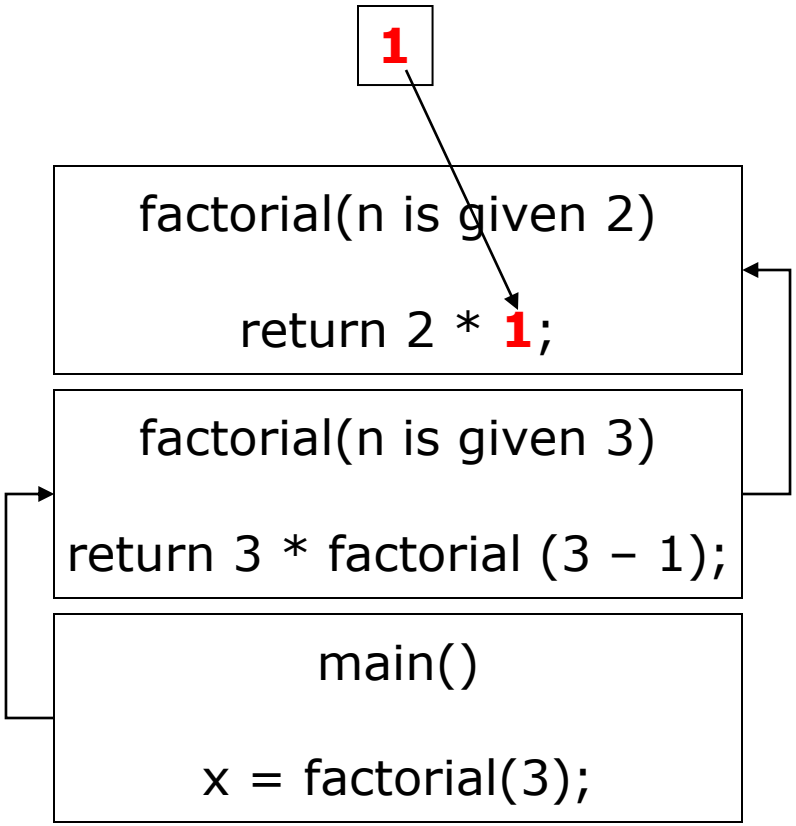
Recursion

Recursion ends at the base case

- In the function instance of factorial(1), the base case is now reached
- No further recursive function calls occur, and a value of 1 is returned
- Like any other function that completes, the return value goes back to the original function call for use
- In this case the original function call is the previous instance of itself and the “unwinding” begins

Recursion

- "return n;" results in returning a value of 1



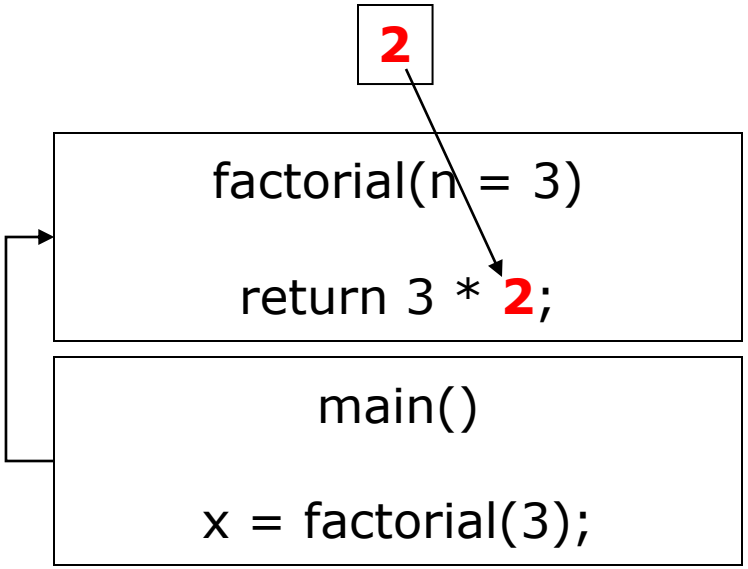
Recursion

Recursive function call unwinding

- factorial(2) was the instance that made the function call to factorial(1)
- factorial(1) reached the base case and simply returns a value of 1
- That value comes back to the factorial(2) execution instance at the point where the function call was made
- That point is in the line “return n * factorial(n-1);”
- In this instance, then, the line of code in factorial(2) is now “return 2 * 1;” because the recursive function call is replaced with that function’s return value
- This results in a “return 2;” which now continues the unwinding by returning that value to factorial(2)’s original function caller which was factorial(3)…

Recursion

- "return n * factorial(n-1);" results in returning a value of 2 (because n is 2 in this function instance and factorial(n-1) resulted in 1)

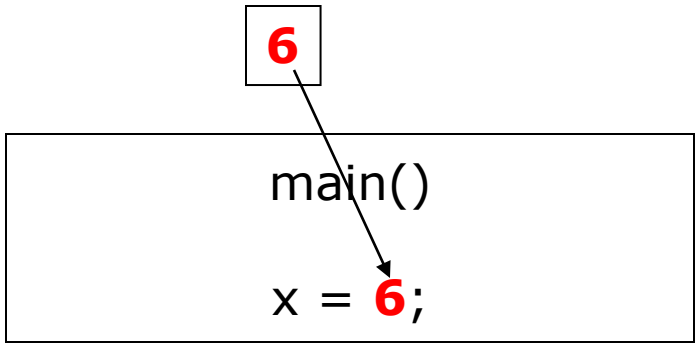


Recursion

Almost done

- factorial(3) now receives the return value of 2 exactly like factorial(2) received the return value of 1 from factorial(1)
- That value of 2 is applied to the line it was called from which will result in “return 3 * 2;” in the factorial(3) function
- The unwinding now completes with factorial(3) returning a value of 6 back to its original function caller. In this example, that function is main
- main called factorial(3) and is assigning that function’s return value into x and thus completing the recursive line of execution

Recursion



Recursion

Function call stack

- Recall in the stacks and queues discussion that one of the examples of using a stack is function call management
- When a function is called, in instance of that function is pushed onto the stack and executes as coded. When the function completes, the instance is popped from the stack any value returned is passed back into the next instance on top at the point where it made its function call
- The same process is happening with recursion. The key difference is that the function instances created are using the same function code
- The recursive functions are using the same code, but the logical design of the base and inductive cases set it up so that the recursive loop will eventually end when the base case is reached

Recursion

Theory and efficiency

- Can this factorial example be written without recursion? Of course! In previous classes, you probably did it with a for or while loop
- In theory, any recursive function can be written using loops and potentially other data structures like a stack
- This suggests performance does not improve with using recursion
- Recursive functions, though, simplify code for more complex algorithms

Recursion

Practice, practice, practice

- Understanding recursion is not trivial
- Just like other programming concepts, understanding the theory and the code starts with practicing different algorithms and walking through the code, line by line
- In this case, following the function call stack is necessary as well. It is very easy to get lost in the recursion without the visualization
- Another famous example and mathematical sequence: Fibonacci numbers

Recursion

Fibonacci numbers

- A Fibonacci number is a number in a sequence where its value equals the sum of the 2 previous Fibonacci numbers
- The Fibonacci number sequence is:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- What is the pattern in this sequence? Starting at Fibonacci number 3:
  - Fib at 3 = 1 which equals 0 + 1 which equals (Fib at 1 + Fib at 2)
  - Fib at 4 = 2 which equals 1 + 1 which equals (Fib at 2 + Fib at 3)
  - Fib at 5 = 3 which equals 1 + 2 which equals (Fib at 3 + Fib at 4)
  - Fib at 6 = 5 which equals 2 + 3 which equals (Fib at 4 + Fib at 5)
- Like we did with the factorial, we can restate this in general terms:
  - Fib at location n = (Fib at location (n – 1) + Fib at location (n – 2))
- The Fibonacci numbers at locations 1 and 2 don't fit the general case which shows that these are the base cases
- Put the two together and you can state that finding the Fibonacci number for a given location number n (assuming n > 0) is:
  - If n = 1, the answer is 0
  - If n = 2, the answer is 1
  - If n > 2, the answer is Fib(n-2) + Fib(n-1)
- Given this and our current understanding of recursive programming, the code is a near direct translation of the formula

Recursion

```
int fib(int n)
{
  if (n == 1)
     return 0;
  if (n == 2)
     return 1;
  return fib(n-2) + fib(n-1);
}
```

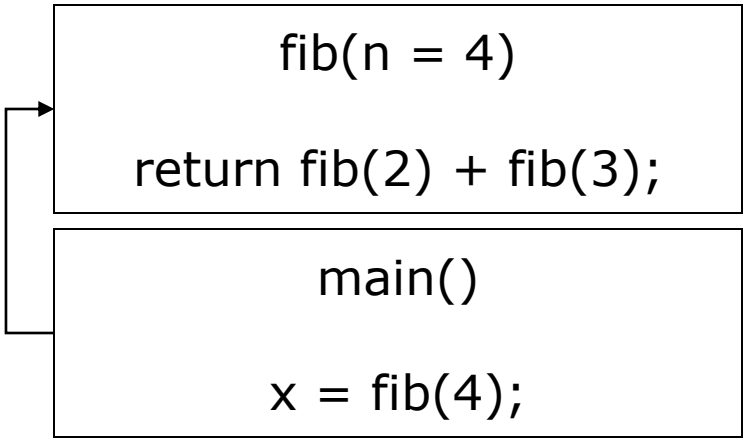
Recursion

Code analysis

- The base cases and inductive case follows a similar pattern as the factorial
- Notice here that the line of code with the recursion is making 2 recursive functions calls on the same line
- This means if the recursive line is reached, 2 recursive calls are handled before that value is returned
- Practice understanding this by drawing the function call stack and following the execution with “x = fib(4);”
- If you can do this on your own and feel comfortable with it, you have a nice start to understanding recursion

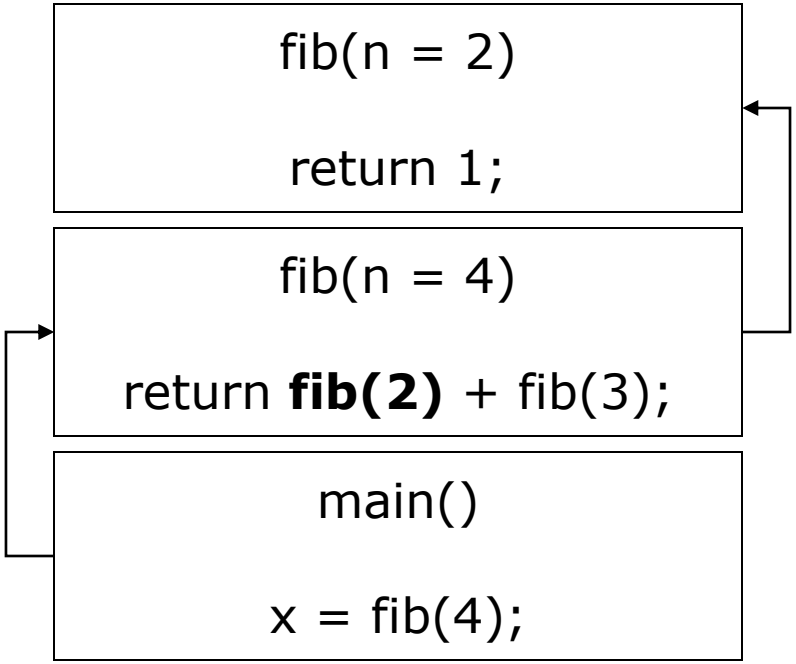
Recursion

Main calls fib(4). In fib(4), base cases are not true.
Thus, we call return fib(2) + fib(3);



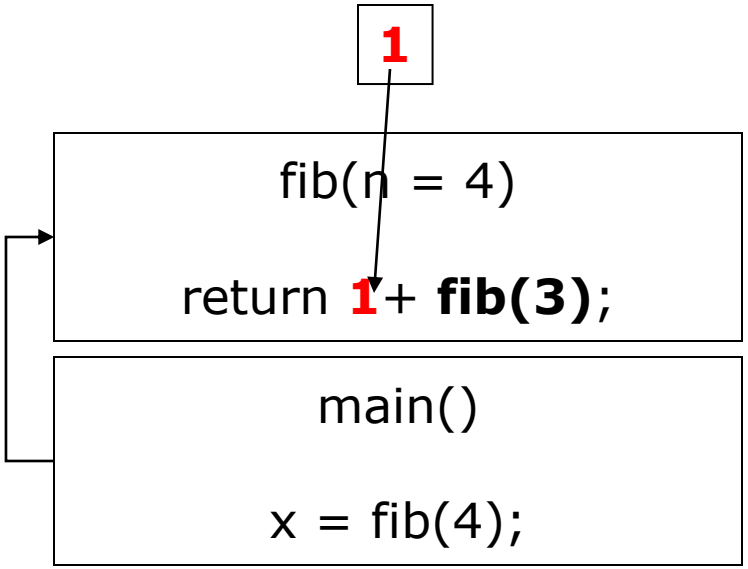
Recursion

fib(2) is handled next and is a base case, so it returns 1



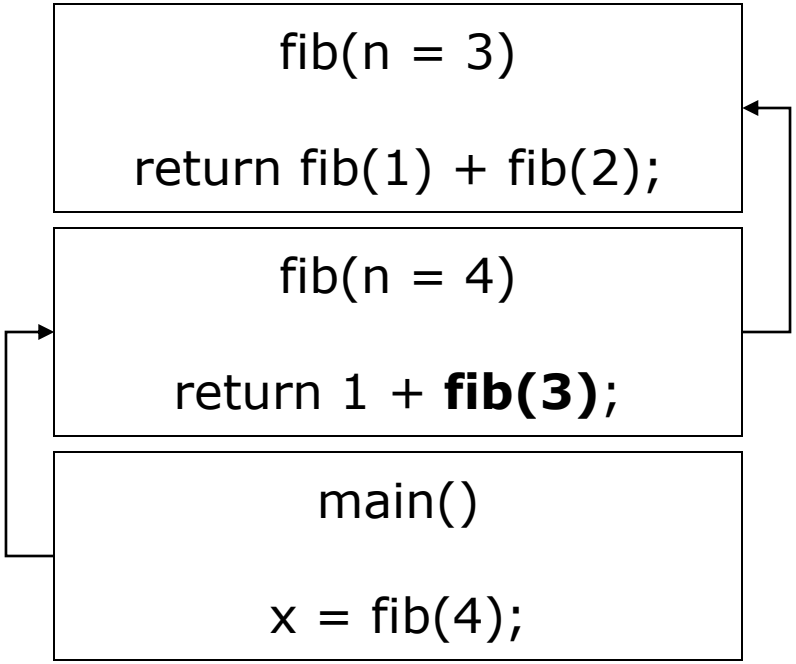
Recursion

1 is returned back to fib(4) where fib(2) was called. Now the fib(3) part of the code must be executed



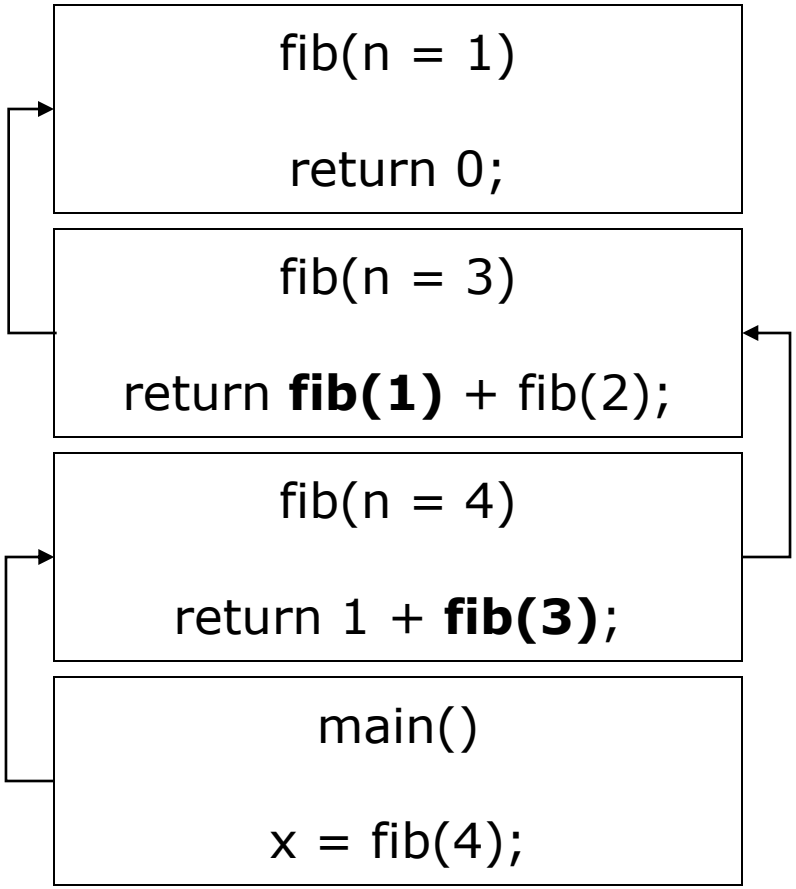
Recursion

fib(3) is next. This is not a base case, so yet another set of recursion occurs



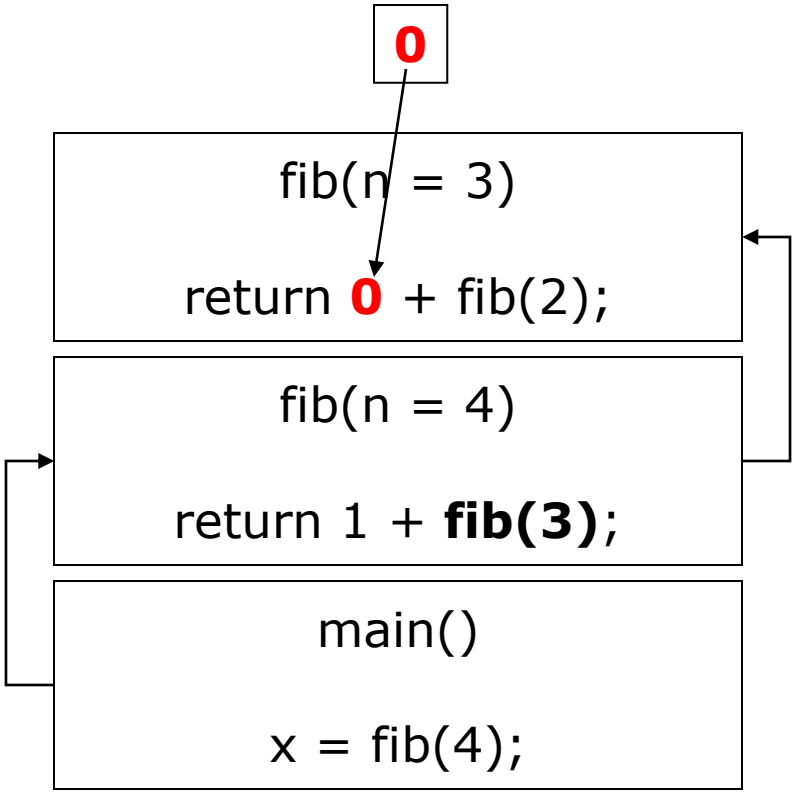
Recursion

fib(1) goes first and is a base case



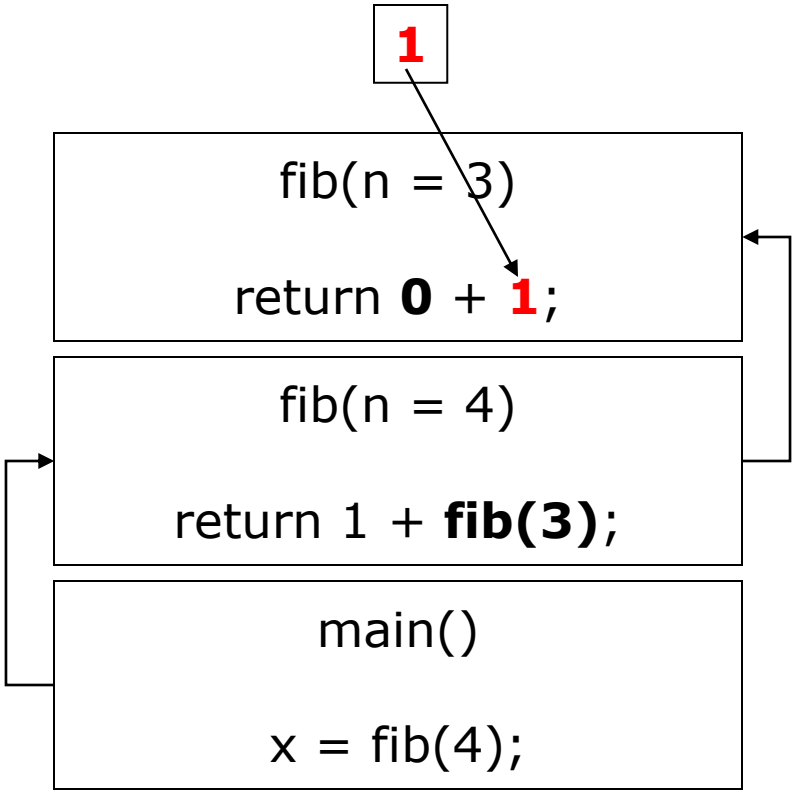
Recursion

0 is returned from fib(1). We next call fib(2)…



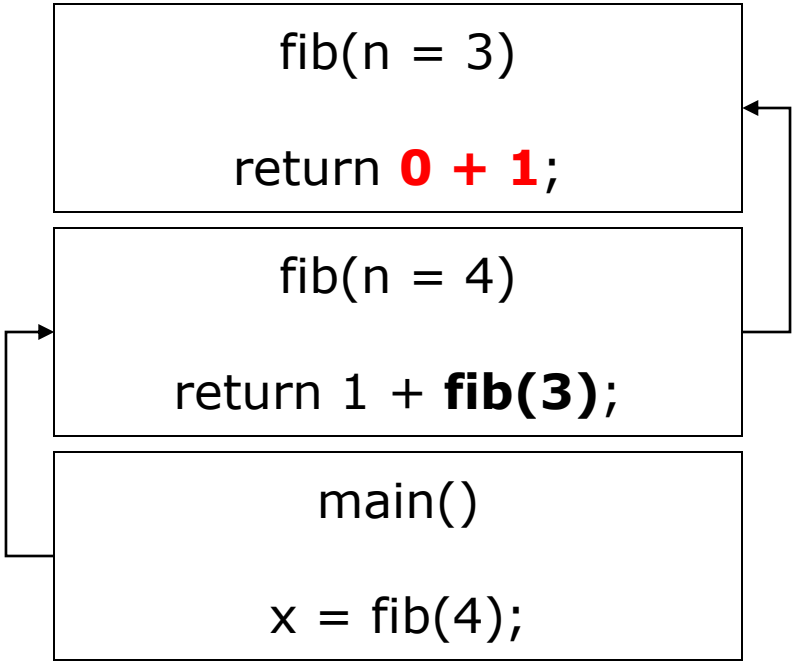
Recursion

fib(2) is another base case situation that will return 1



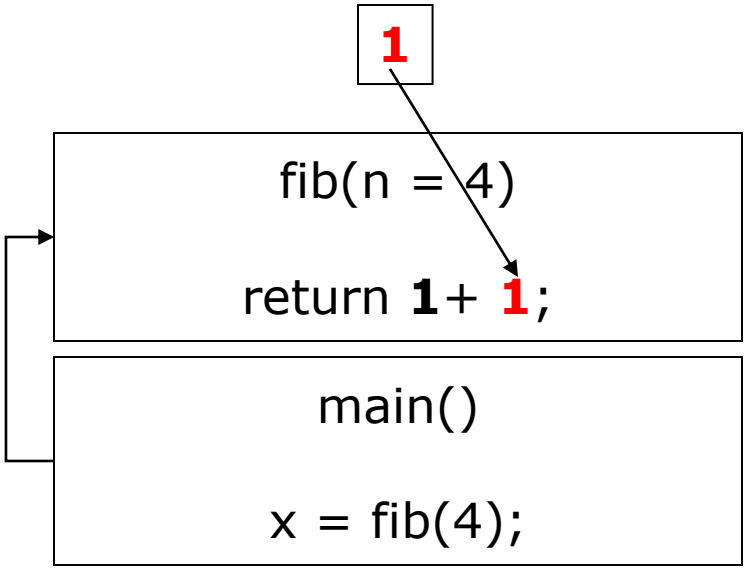
Recursion

fib(3) is now complete and will return 0+1 to fib(4)



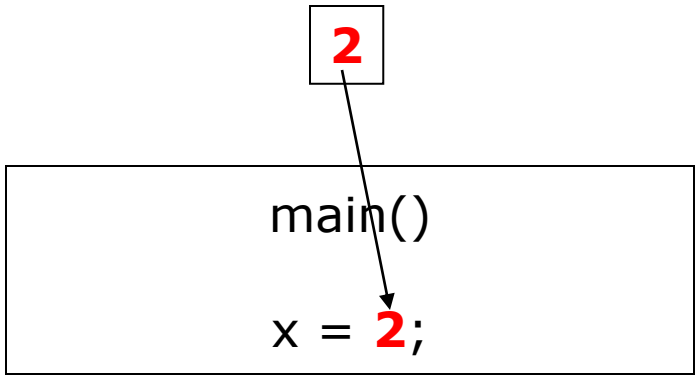
Recursion

With fib(3) complete for fib(4), fib(4)'s recursion is now complete and will return 2 to main and complete all the recursion



Recursion

All done!



Recursion

Recursive Fibonacci analysis

- Note again that the process of calling a function, pushing the new function onto the stack for processing and returning to the point of the function call is consistent whether it's a call to another function or a recursive call
- Understanding the recursion call process requires drawing out the different function instances and tracing the flow of control
- Simple mathematical series that can be stated inductively are classic cases for recursion, but are not the only ones
- A famous recursive function example is the Towers of Hanoi

Recursion

The Legend of the Towers of Hanoi

- An Asian monk is tasked with transferring 64 disks from one pillar to another
- Each disk is different in size with a smaller disk always on top of a larger disk (making the 64th disk on the bottom the largest of them all)
- The are three pillars total (call them A, B and C) and all 64 disks are on pillar A
- The goal is to get them all to C following 2 rules:
  - Only one disk can move at a time
  - A larger disk cannot rest on top of a smaller disk
- When all 64 disks are transferred, the world ends

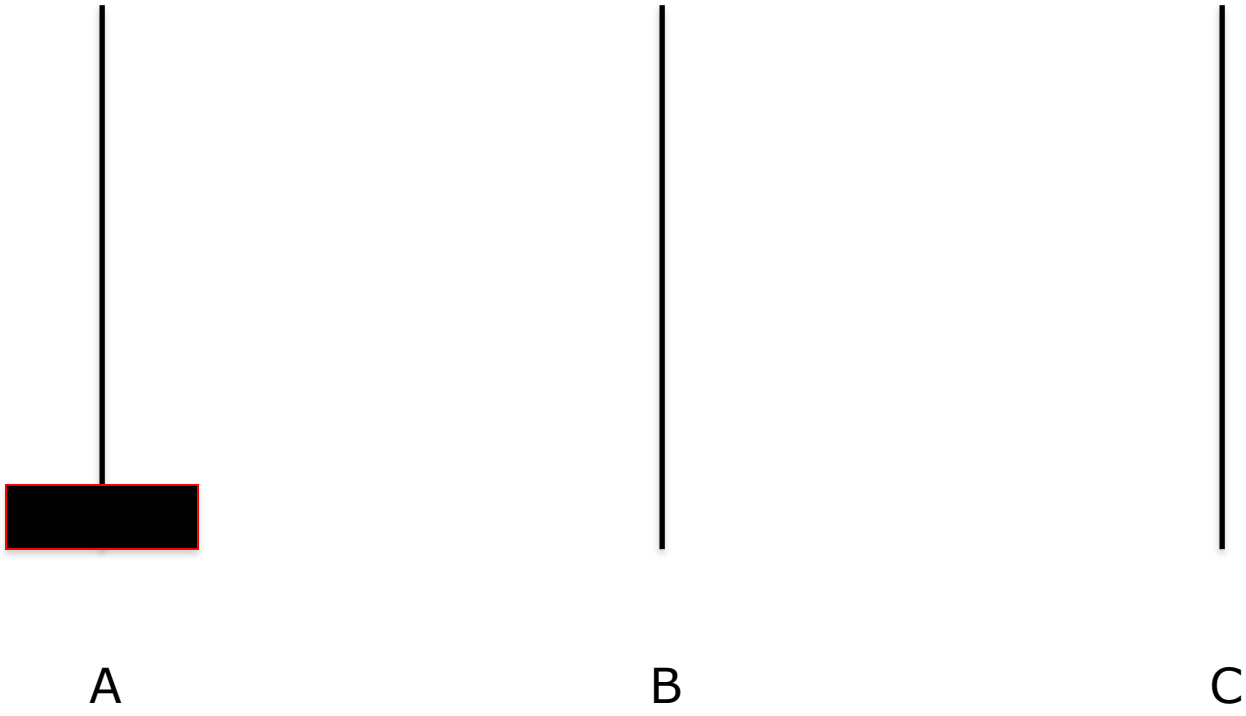
Recursion

Algorithm

- Assuming it takes one second to move a disk and he started right now, how long before the world ends?
- More importantly for us, what is the algorithm to do this? How could we do this without recursion?
- Obviously, in the current context, recursion is involved, but developing this algorithm is not as intuitive as the previous examples
- As with other situations, work out solutions with smaller values to derive the base and inductive cases
- Let’s start with 1 disk instead of 64

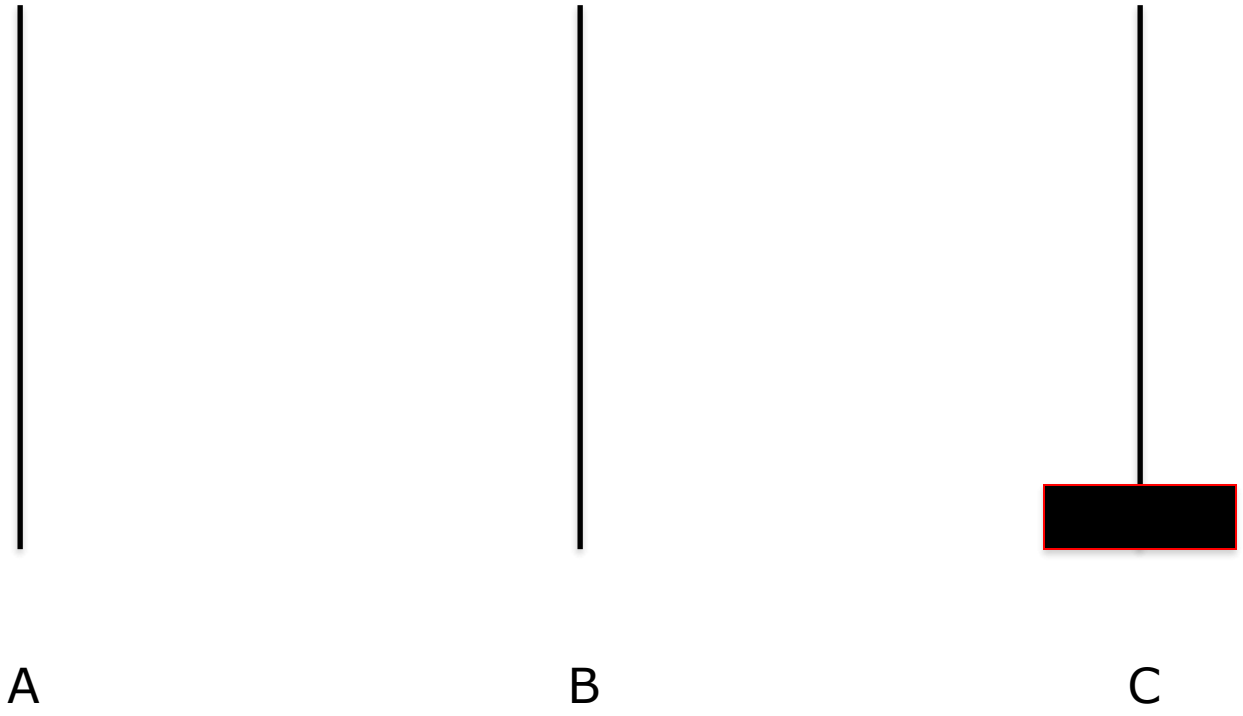
Recursion

With one disk, the move is obvious. Move disk from A to C.



Recursion

Another way to say it is that we have 1 disk and will move it to its target (destination) pillar



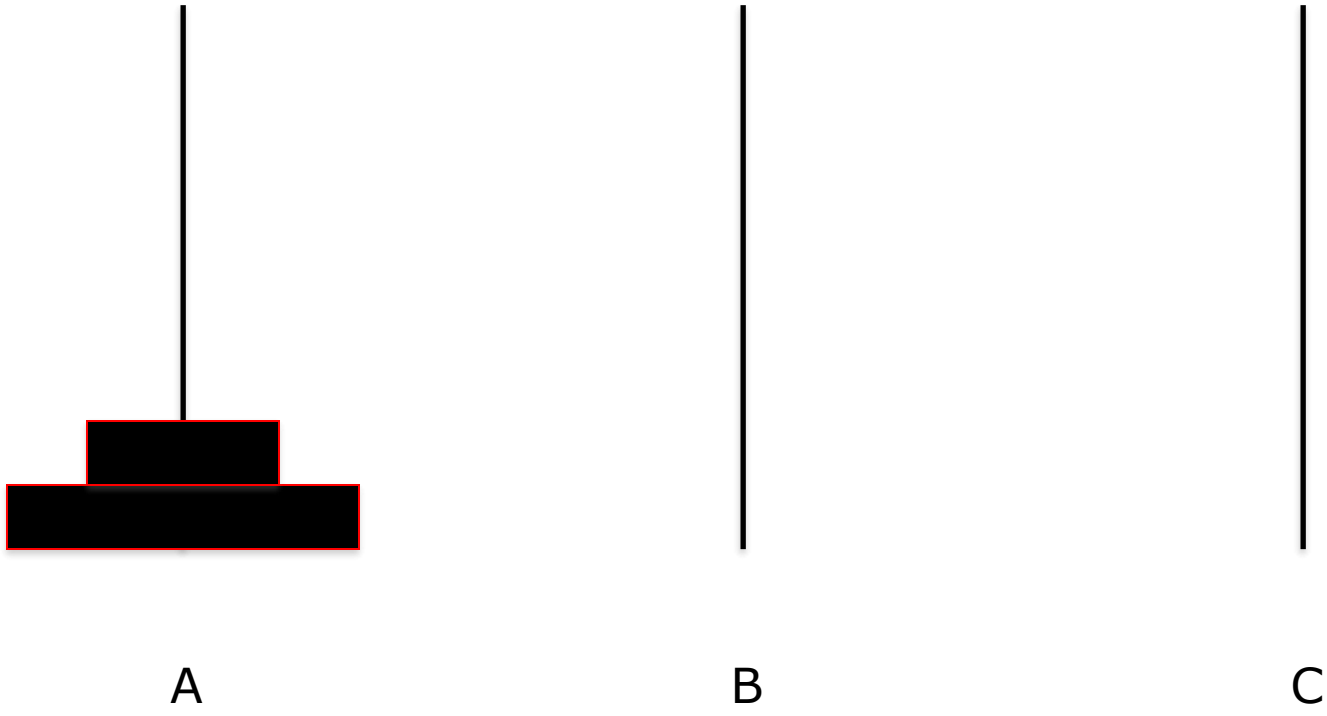
Recursion

Establish our base

- The move from start to destination is occurring with 1 disk
- The key action is **if we are looking at one disk, move it to where you want it to go**
- This sounds like a base case, but may seem peculiar given that the overall rules for the problem is that you can only move one disk at a time anyway
- Let’s do the 2 disk situation. First, let’s look at how we would do it ourselves

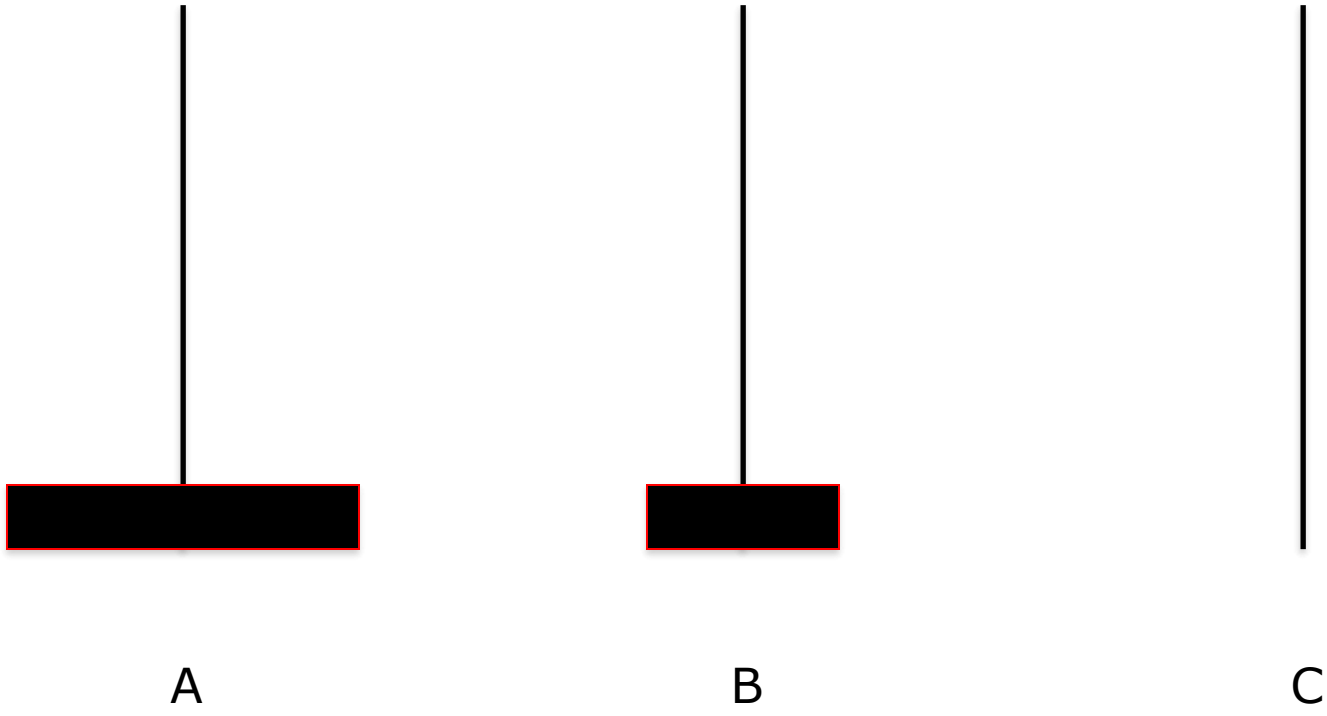
Recursion

Here, if we move the first disk on A to C, the next disk on A won’t be able to go to C without the smaller one out of the way



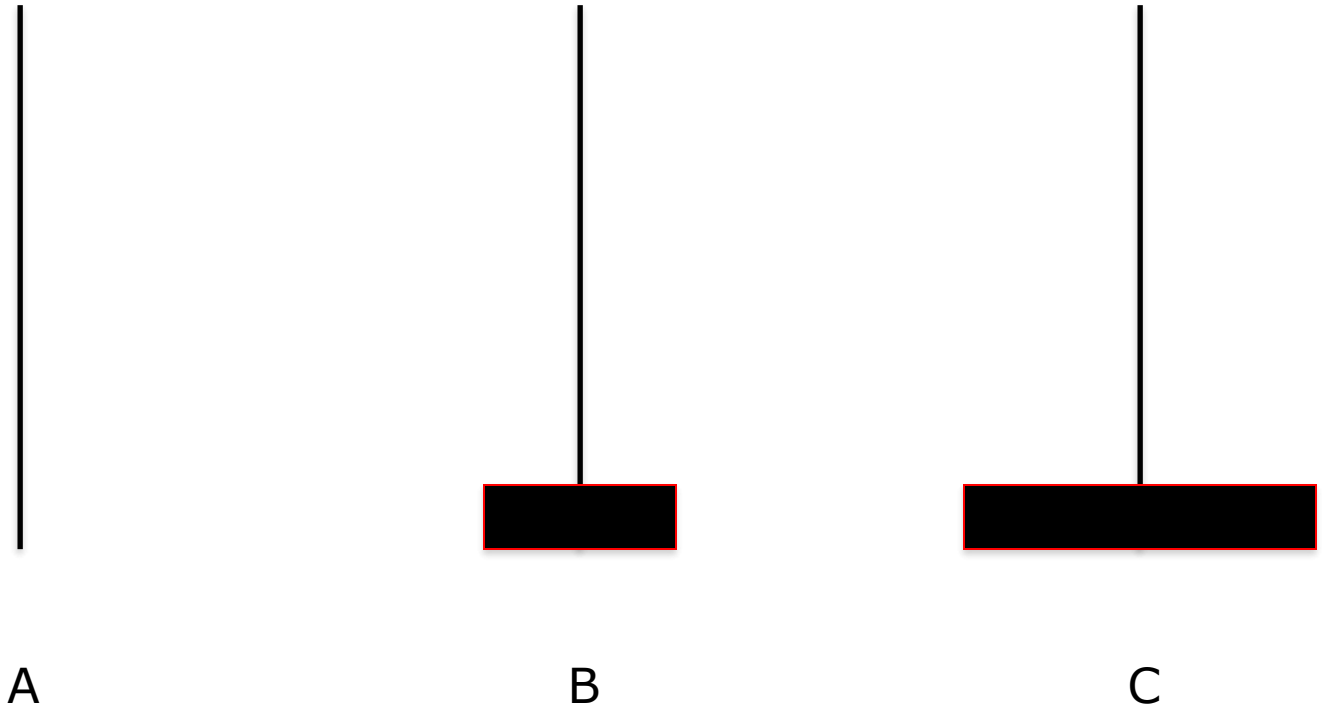
Recursion

Step 1: Move disk from A to B. B acts as a "temporary" pillar, while A and C are "start" and "destination" pillars respectively



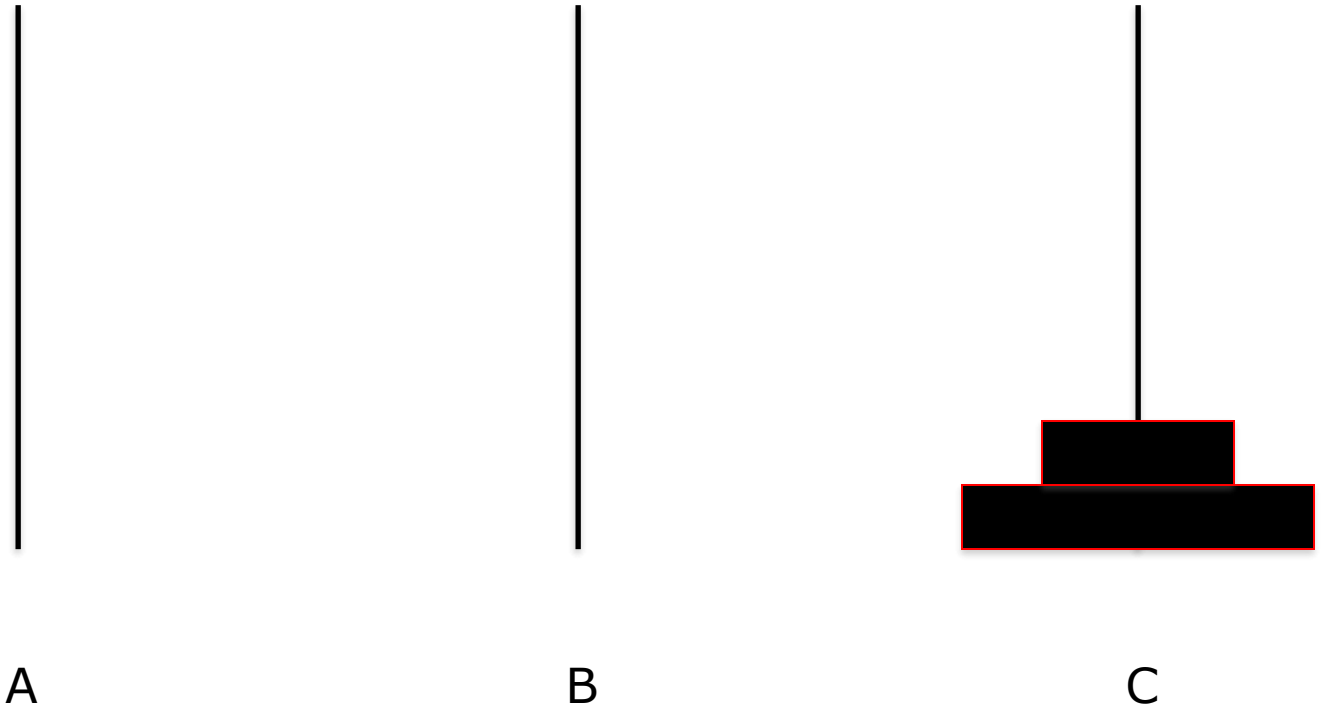
Recursion

Step 2: Now we can move the disk from A (start) to C (destination)



Recursion

Step 3: Last move is simple. From disk from B (temp) to C (destination)



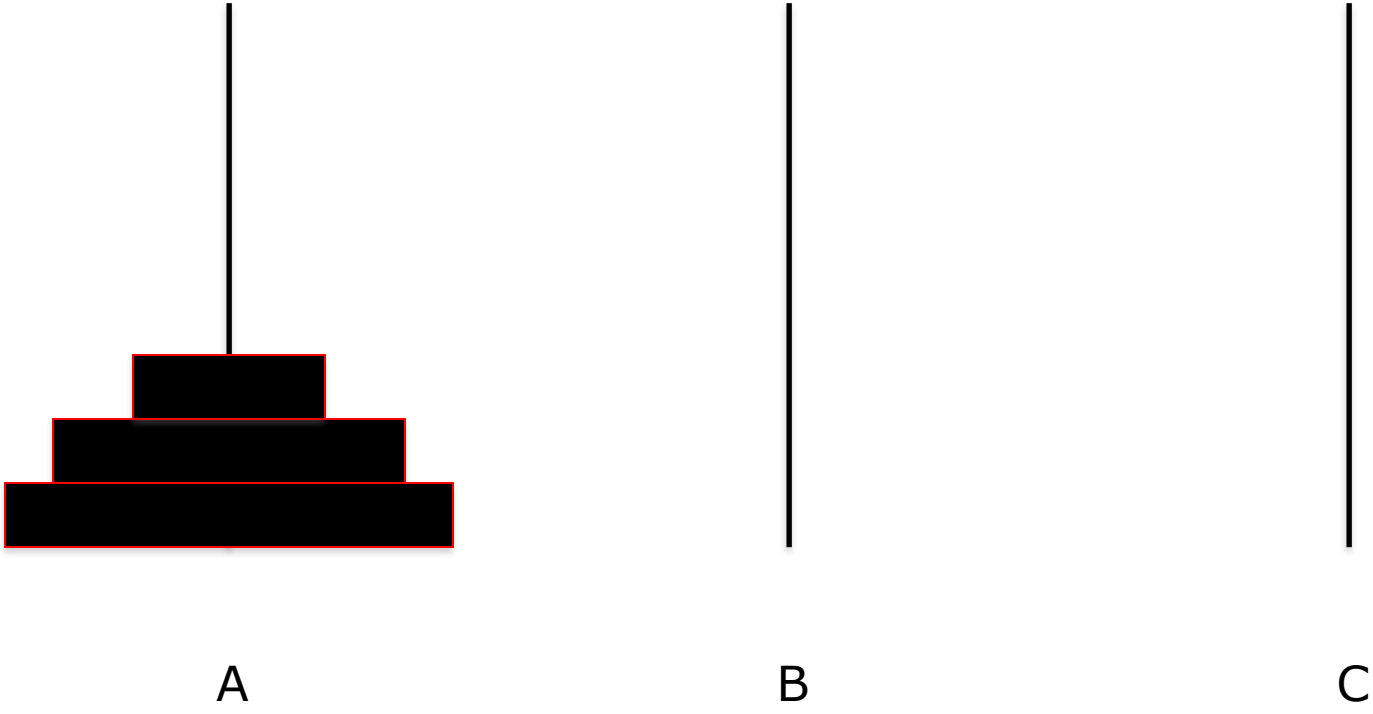
Recursion

2 disk case

- The 2 disk situation shows the key 3 step process
  - Move 1 disk from start to temp
  - Move 1 disk from start to destination
  - Move 1 disk from temp to destination
- The idea that one pillar serves as a “temporary” one while the other two are start and destination is critical to understanding the solution
- The ultimate goal is A to C for all disks, but along the way, what “start”, “temporary” and “destination” are will differ depending on your situation
- Now let’s bump it up to 3 disks keeping in mind the reasoning behind the simple steps for 2 disks

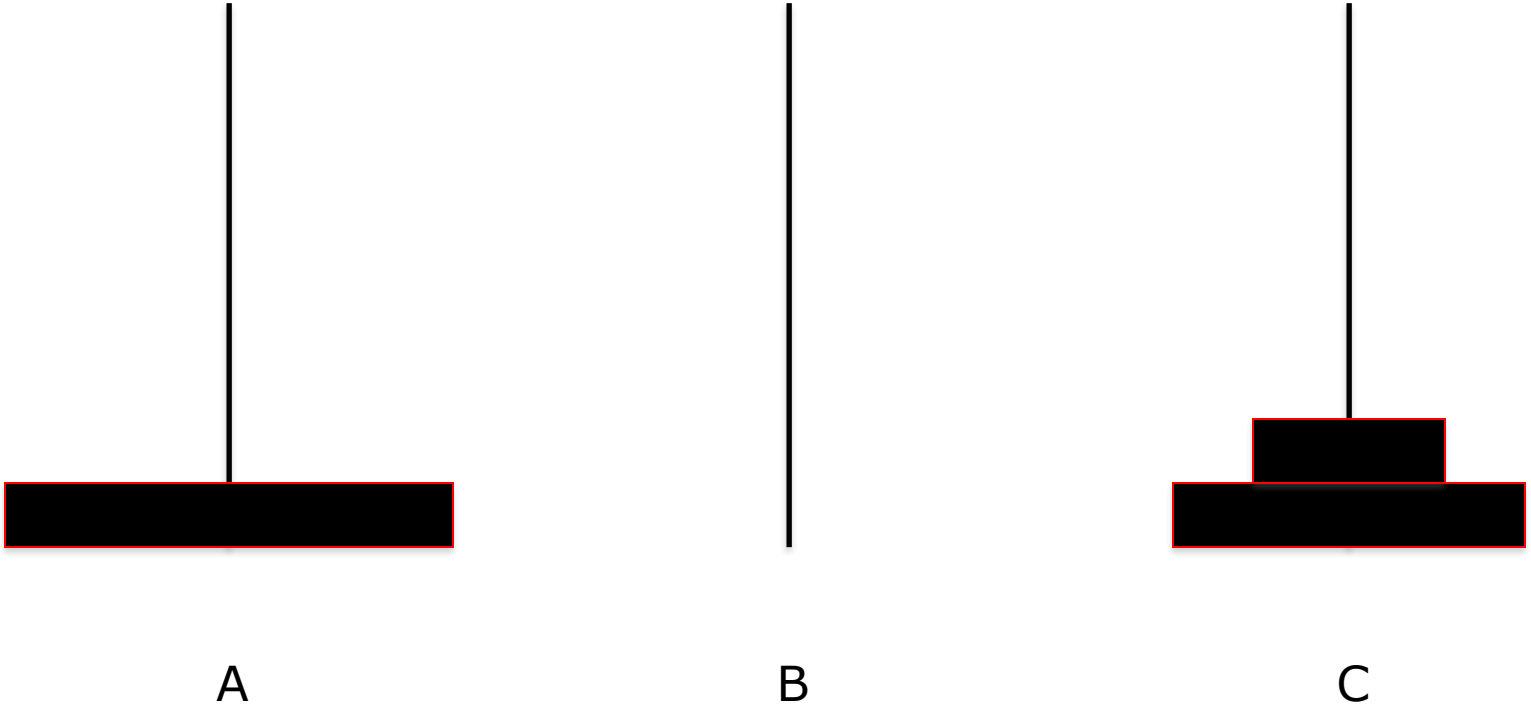
Recursion

Start. Where do we go from here?



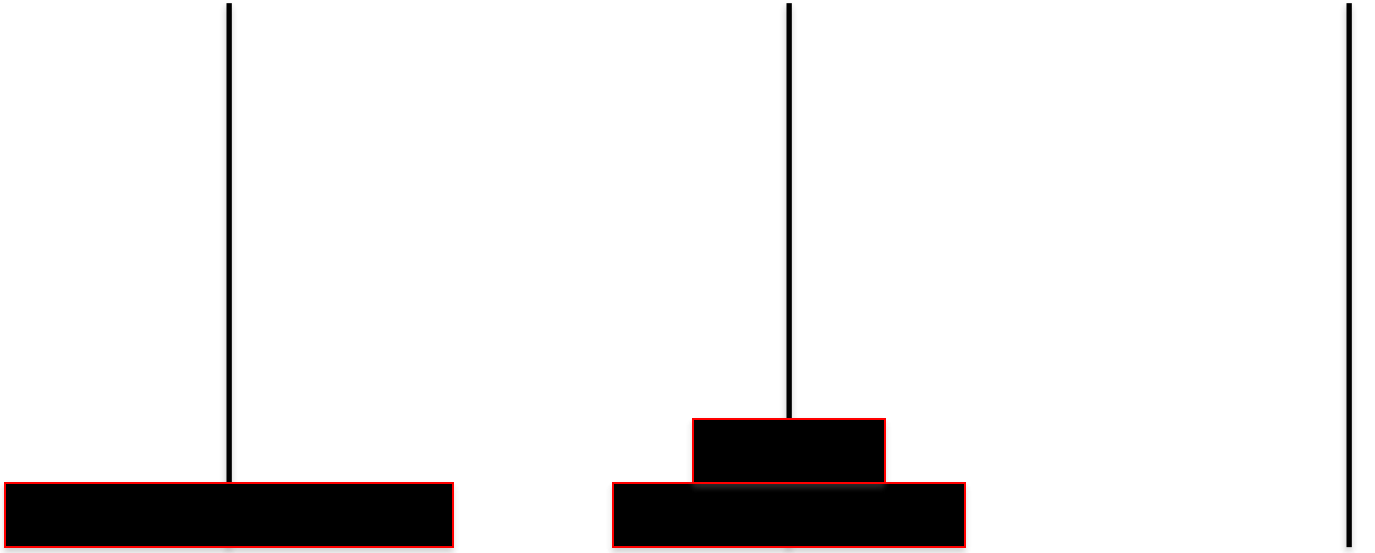
Recursion

If we follow the same moves (A to B, A to C, B to C), we would have 2 disks at C, but the one big disk still at A. Thus, we should not end up with these 2 disks on C



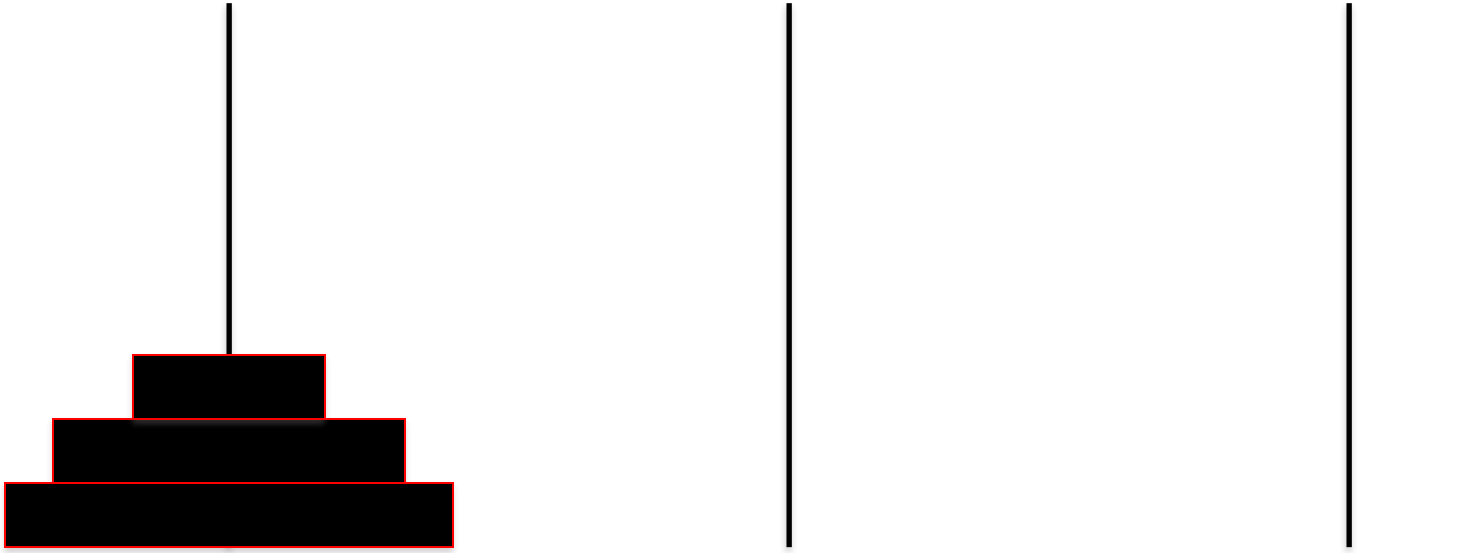
Recursion

We should then try to eventually get the 2 disks to a position where they are both on B. Then, the big disk on A can get to C



Recursion

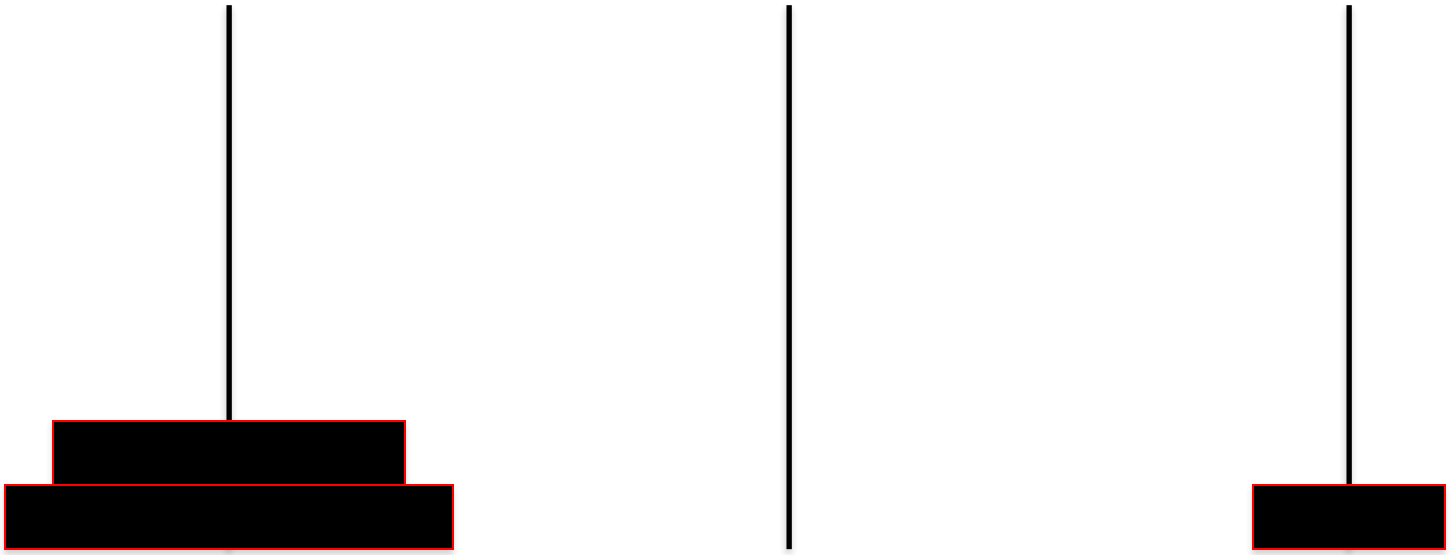
How do we get to that point? Note that for the top 2 disks, the steps in the 2 disk solution apply, but B would be our "destination" and C would be our "temp"



A (start) B (dest) C (temp)

Recursion

Given these labels, the 3 moves are the same as before. Step 1: Move A to C (start to temp)



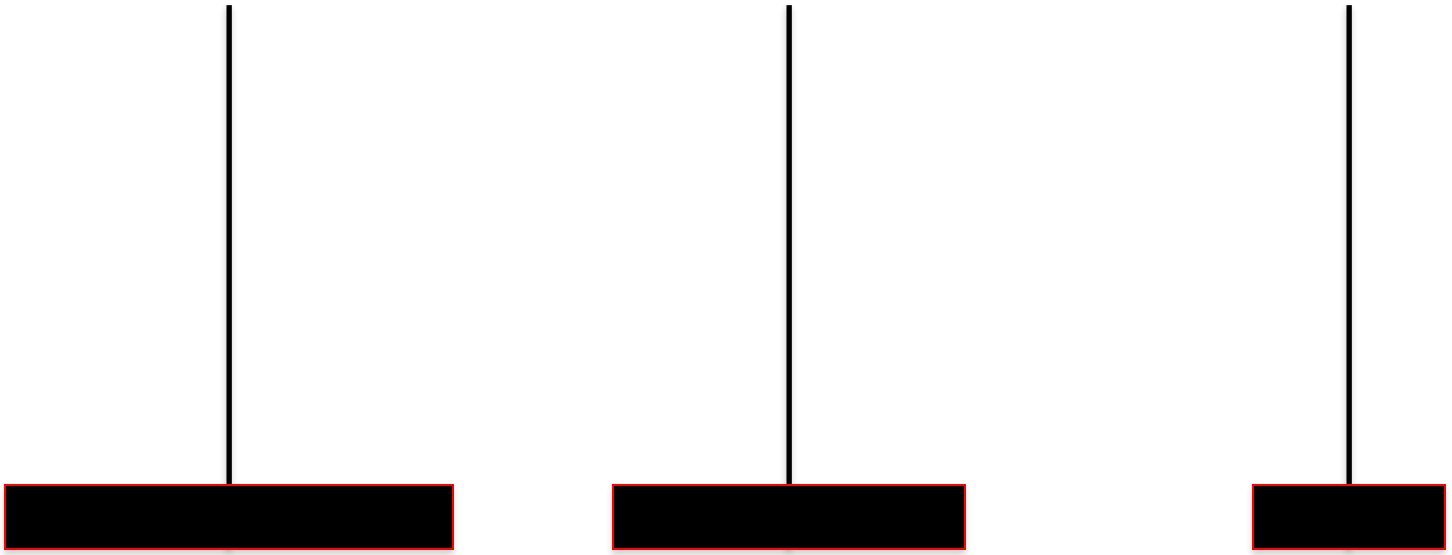
A (start)

B (dest)

C (temp)

Recursion

Step 2: Move A to B (start to dest)



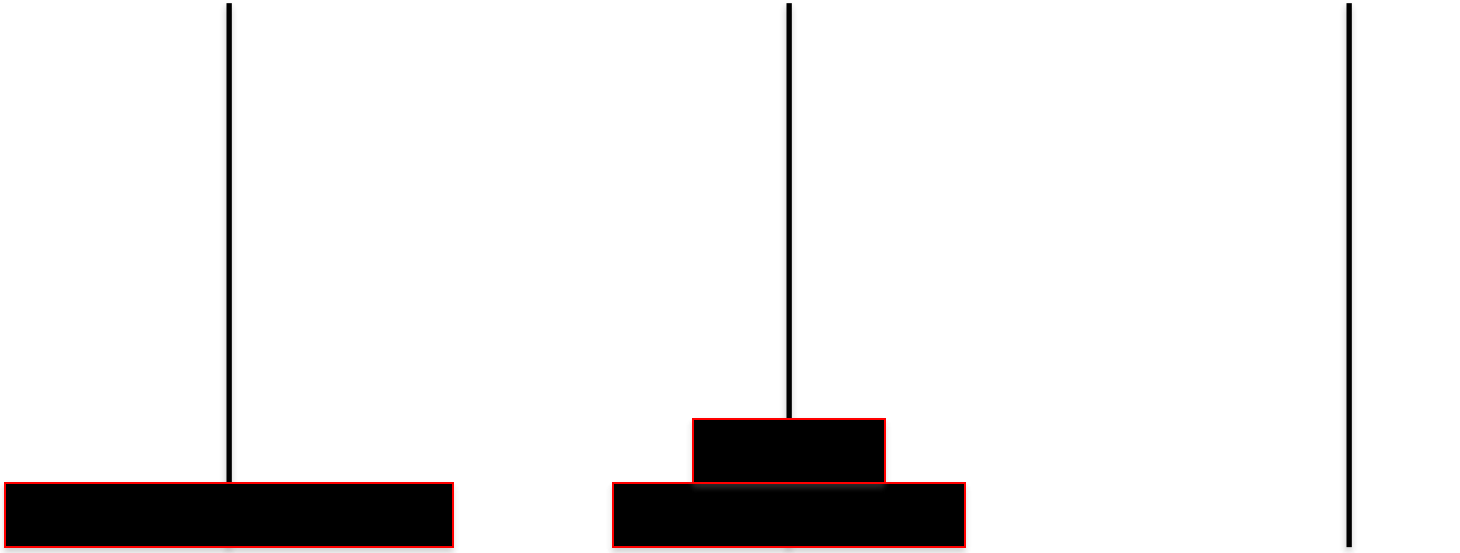
A (start)

B (dest)

C (temp)

Recursion

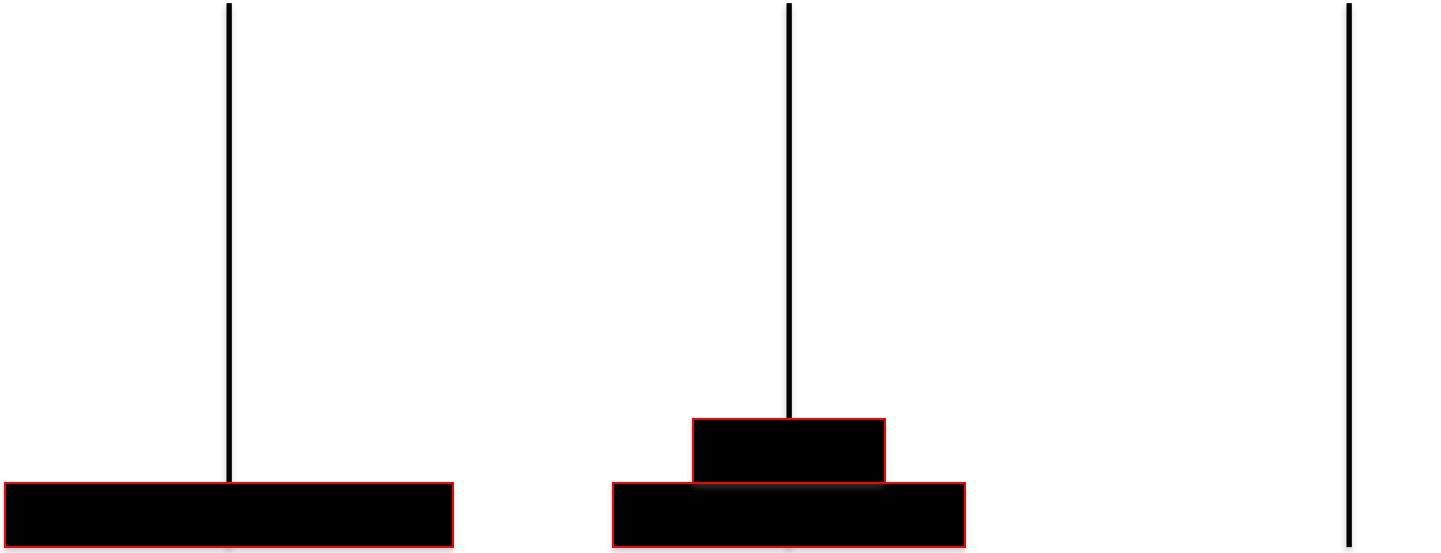
Step 3: Move C (temp) to B (dest). Note now that this temporary 2 disk goal of getting them to destination B is complete



A (start) B (dest) C (temp)

Recursion

In the overall picture for 3 disks, our destination is C and at this point, we have successfully moved 2 disks off A to the temporary pillar B



A (start)

B (temp)

C (dest)

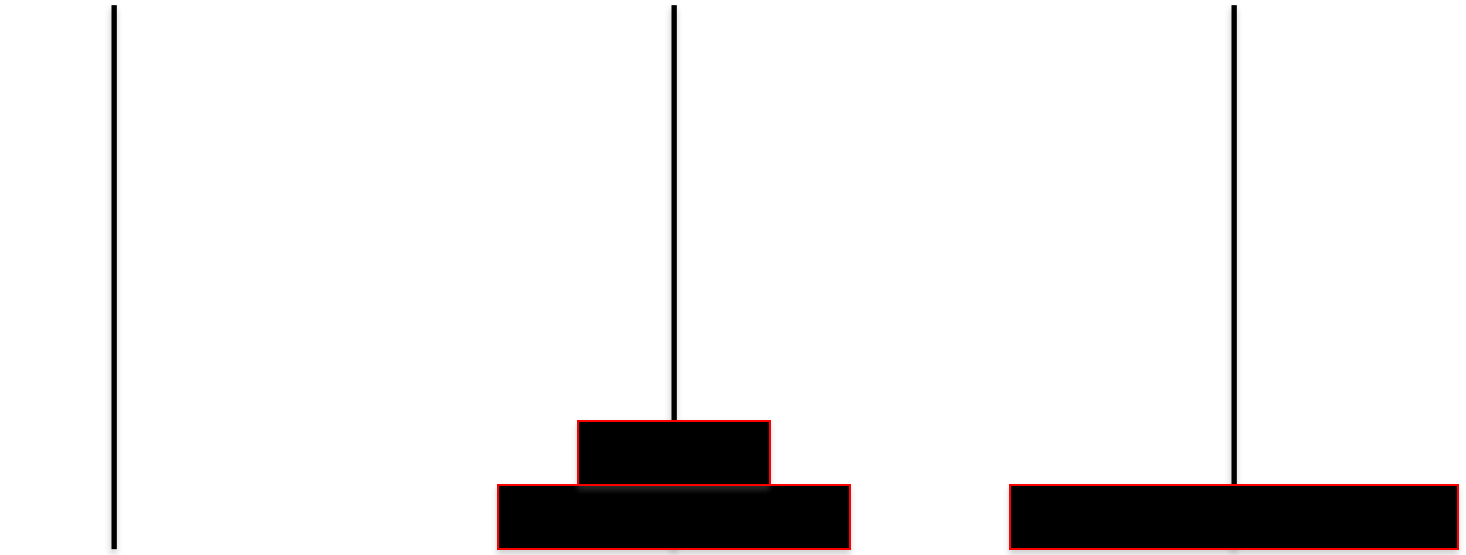
Recursion

3 disk case so far

- Recall the steps when there are only 2 disks
  - Move 1 disk from start to temp
  - Move 1 disk from start to destination
  - Move 1 disk from temp to destination
- We got the same effect with the top 2 disks for the 3 disk situation with temp in the middle
- If we generalize that a bit, we could also say "for n disk situation, first move (n-1) disks to temp"
  - Move **(n-1)** disks from start to temp
  - Move 1 disk from start to destination
  - Move 1 disk from temp to destination
- Now let's see if the 2nd step still applies

Recursion

Step 4: Move A to C (start to dest)



A (start) B (dest) C (temp)

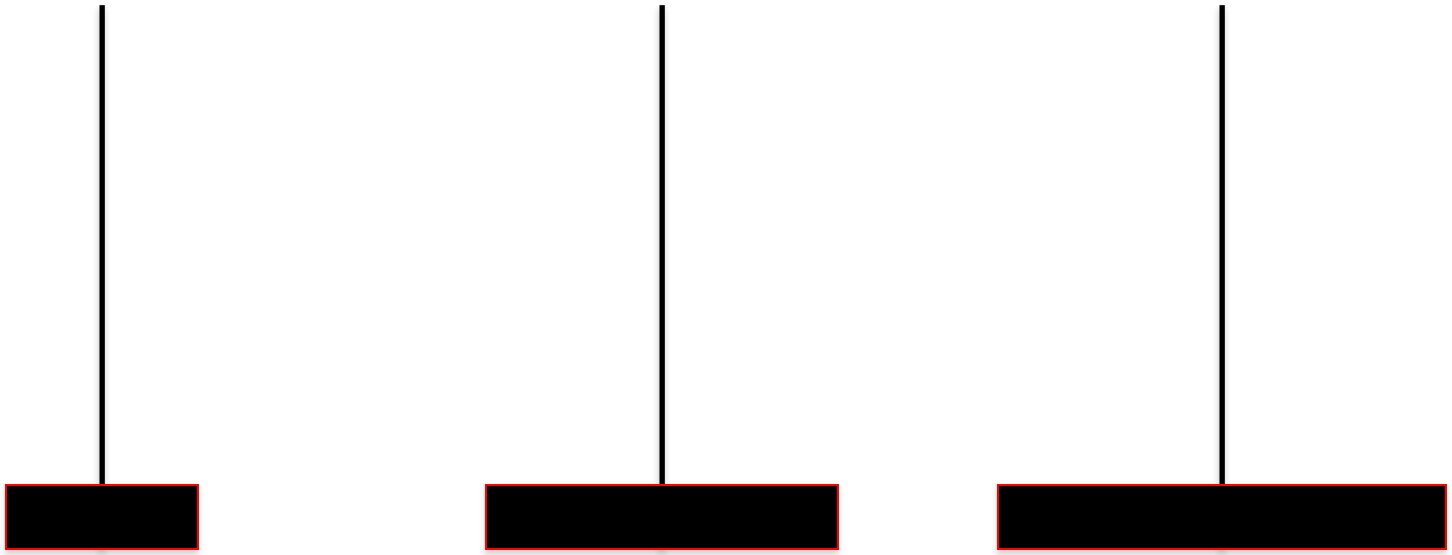
Recursion

Almost there!

- This is an easy move leaving only the last 2 disks from temp to move to dest:
  - **Move (n-1) disks from start to temp**
  - **Move 1 disk from start to destination**
  - Move 1 disk from temp to destination
- Like we did in generalizing the first step, we can do the same thing here in generalizing the last step:
  - Move (n-1) disks from start to temp
  - Move 1 disk from start to destination
  - Move **(n-1)** disks from temp to destination
- How do you move these disks? Note that B is start and A is temp

Recursion

Step 5: Move B to A (start to temp)



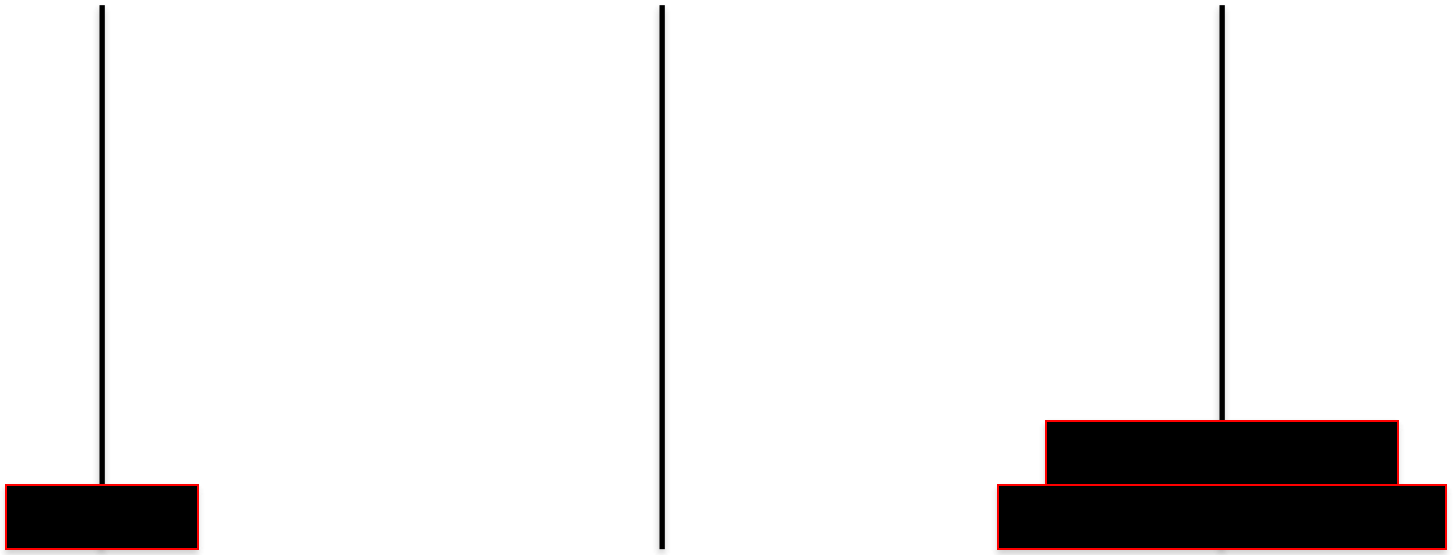
A (temp)

B (start)

C (dest)

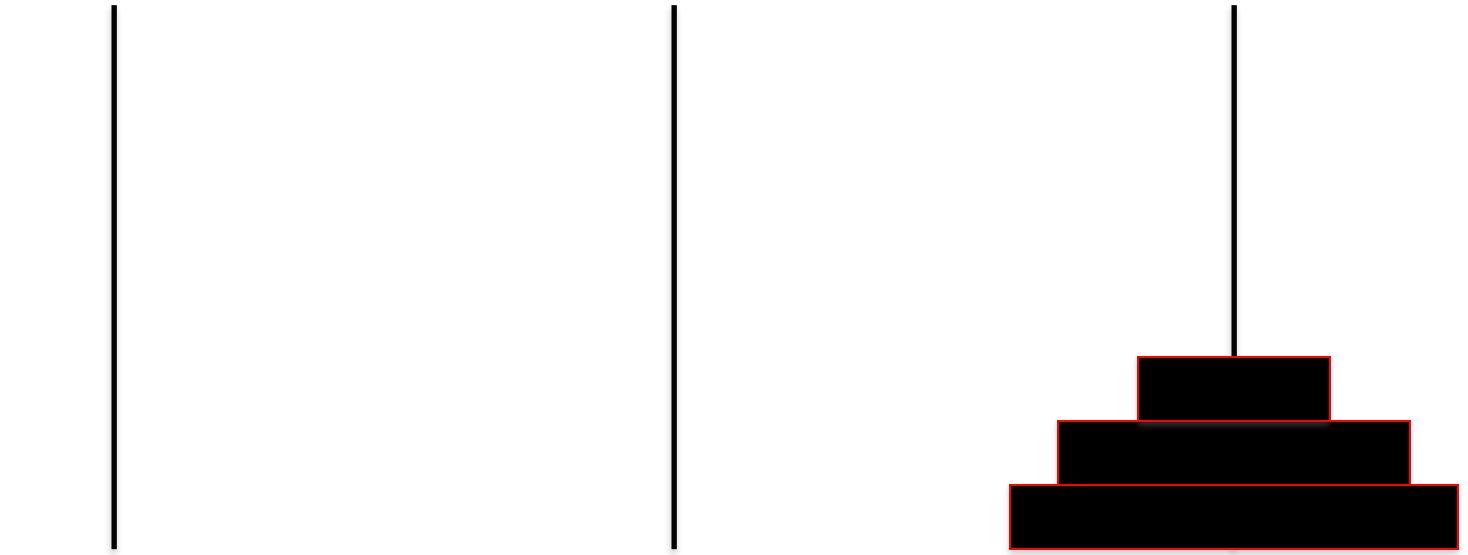
Recursion

Step 6: Move B to C (start to dest)



Recursion

Step 7: Move A to C (temp to dest)



A (temp) B (start) C (dest)

Recursion

Done! So what's the formula?

- The general case is complete:
  - Move (n-1) disks from start to temp
  - Move 1 disk from start to destination
  - Move (n-1) disks from temp to destination
- The base case is simply to move the disk using the same terminology:
  - If n=1, move the disk from start to destination
- Now to program this in recursive code, we need key information of the number of disks and where the "start", "temp" and "destination" pillars are
- We can start with a function signature

Recursion

```
void hanoi(int n, char start, char temp, char dest)
{
}

main()
{
   hanoi (3, 'A', 'B', 'C');
}
```

- The main function calls hanoi saying let's move 3 disks from A to C with B as our temp
- The hanoi function signature matches it
- Now let's do the base case

Recursion

```
void hanoi(int n, char start, char temp, char dest)
{
   if (n == 1)
       System.out.printf("Move disk from %c to %c\n",
   start, dest);
}
```

This is the base case code from our analysis saying that if there is only one disk to move, move it from the given start to the given dest

Now for the recursive case code, which will be tricky because of how we will use start, temp and dest…

Recursion

```
void hanoi(int n, char start, char temp, char dest)
{
   if (n == 1)
       System.out.printf("Move disk from %c to %c\n",
   start, dest);
   else
   {
       hanoi(n-1, start, dest, temp);
       System.out.printf("Move disk from %c to %c\n",
   start, dest);
       hanoi(n-1, temp, start, dest);
   }
}
```

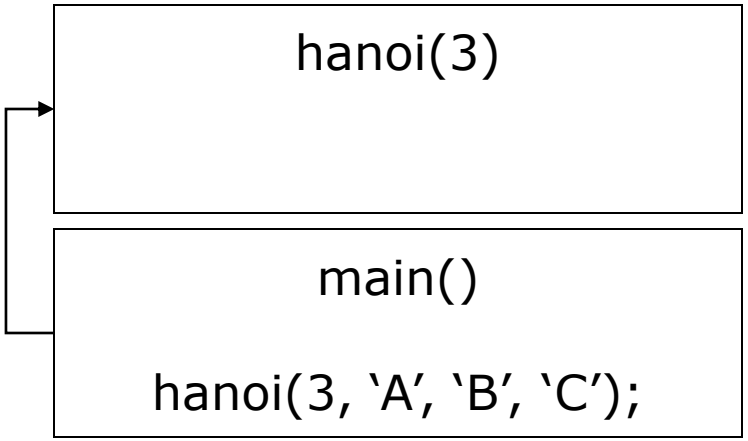
Recursion

That’s it?!

- That’s it! The only way to truly internalize this is to walk through the code
- Like with factorial and Fibonacci, tracing through the function call stack is important
- Let’s do that again here with attempting to move 3 disks from A to C with B as our temp

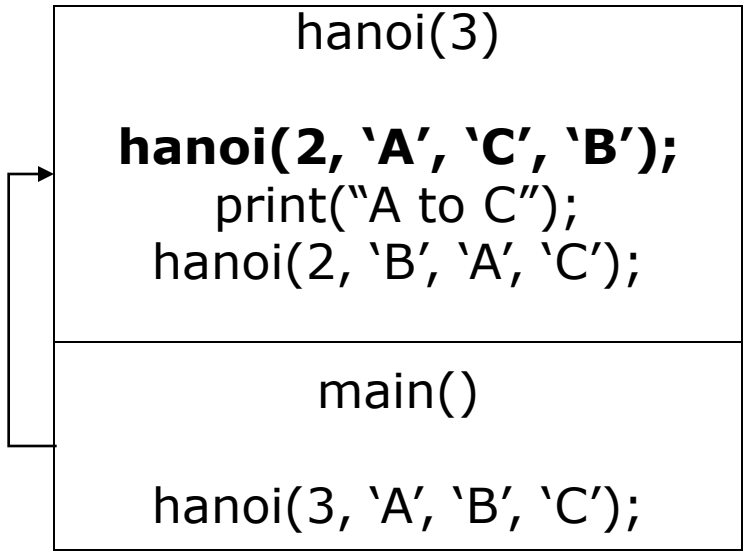
Recursion

Main calls hanoi (3, ‘A’, ‘B’, ‘C’);



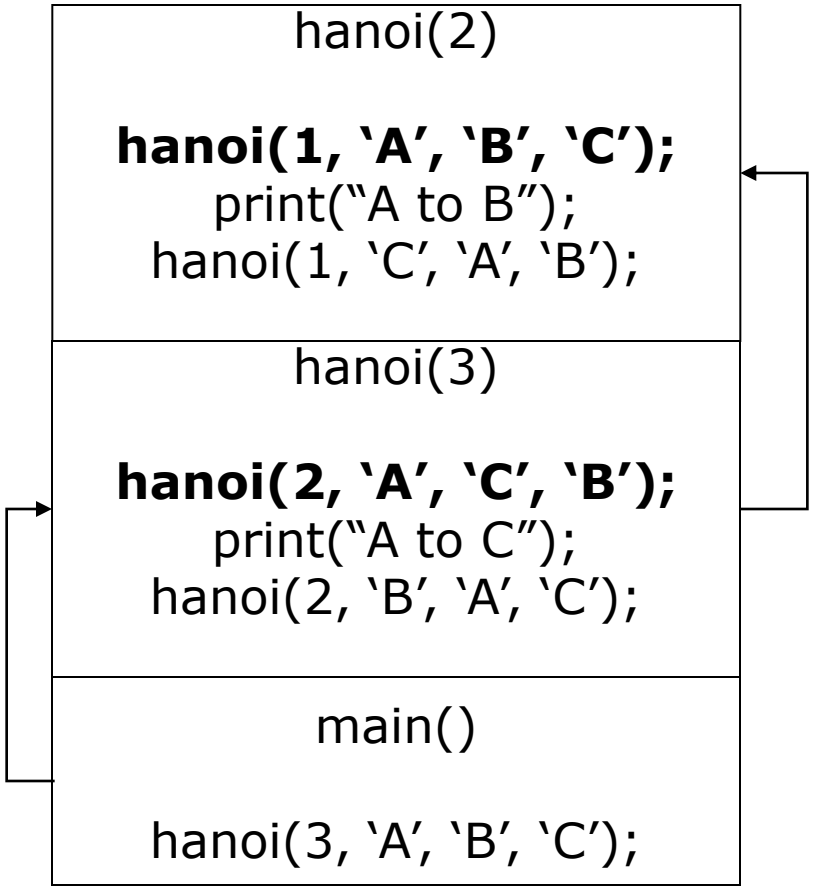
Recursion

hanoi(3) is not a base case, so we go to the inductive case starting with hanoi(3-1, ‘A’, ‘C’, ‘B’); Note the function call value orders whenever the hanoi function is called

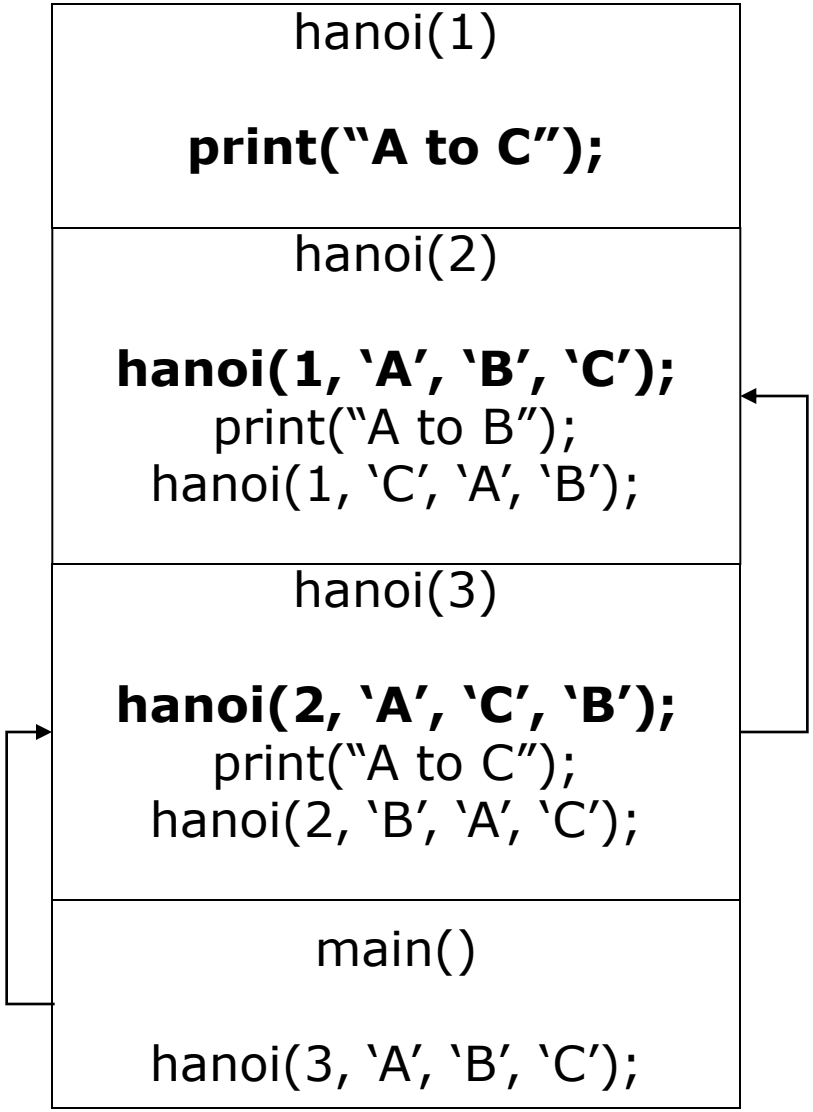


Recursion

hanoi(2) follows suit



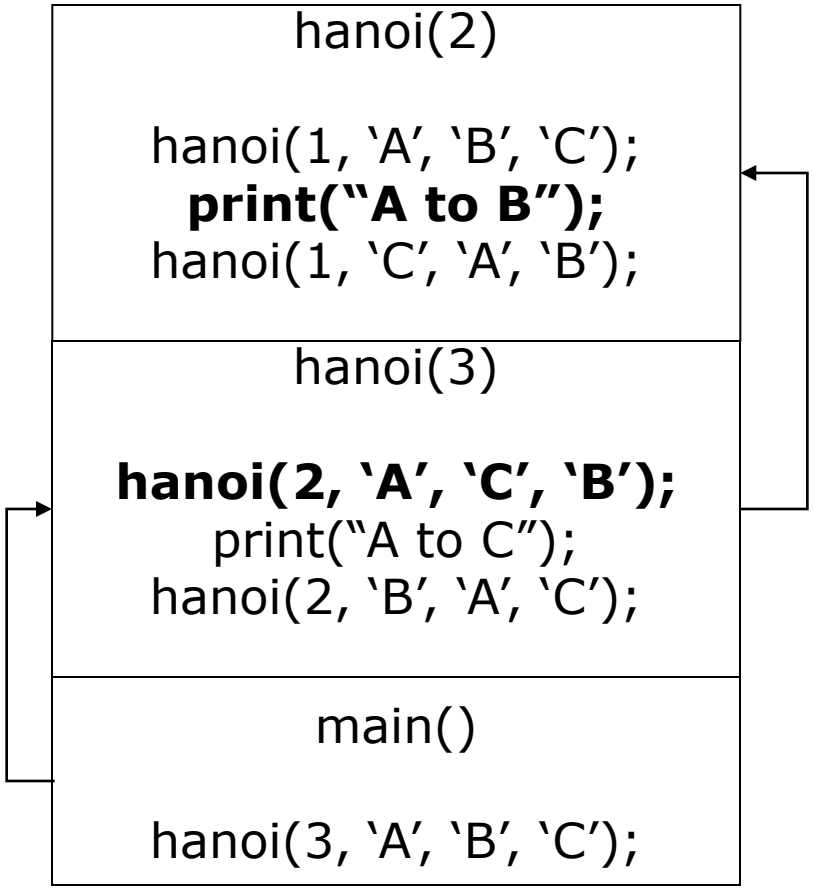
Recursion



hanoi(1) is a base case. See why it prints A to C?

Move A to C

Recursion

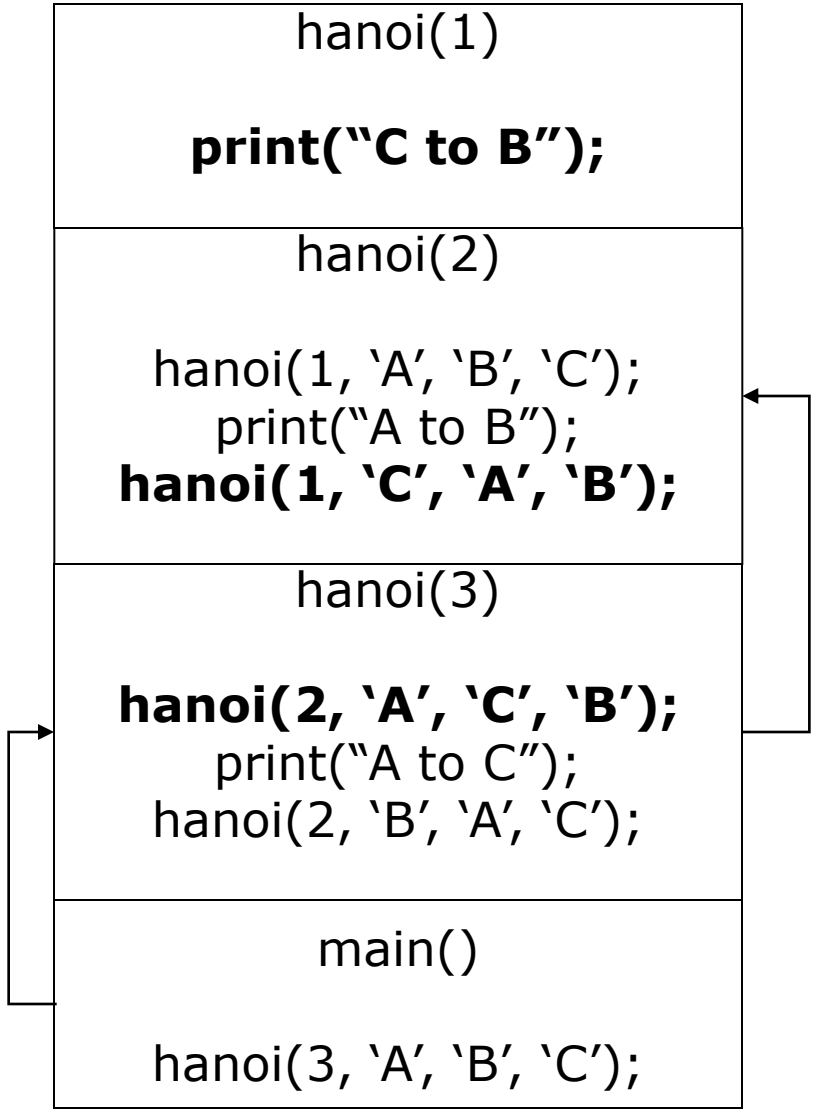


Returning to hanoi(2), we reach the next line of code which prints A to B?

Move A to C

Move A to B

Recursion



Last line in hanoi(2) makes another recursive call of hanoi(1). Note the change in start, temp and dest

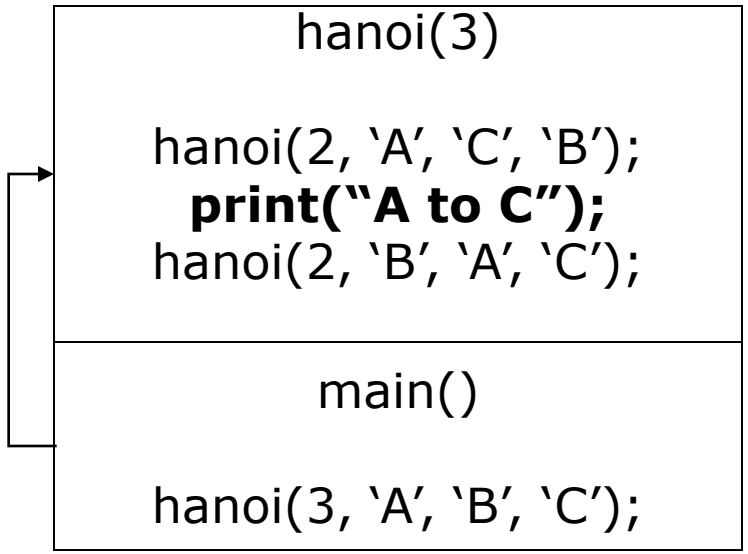
Move A to C

Move A to B

Move C to B

Recursion

When hanoi(1) finishes, it returns to hanoi(2) which also finishes and we're back to hanoi(3). Next line there does a print



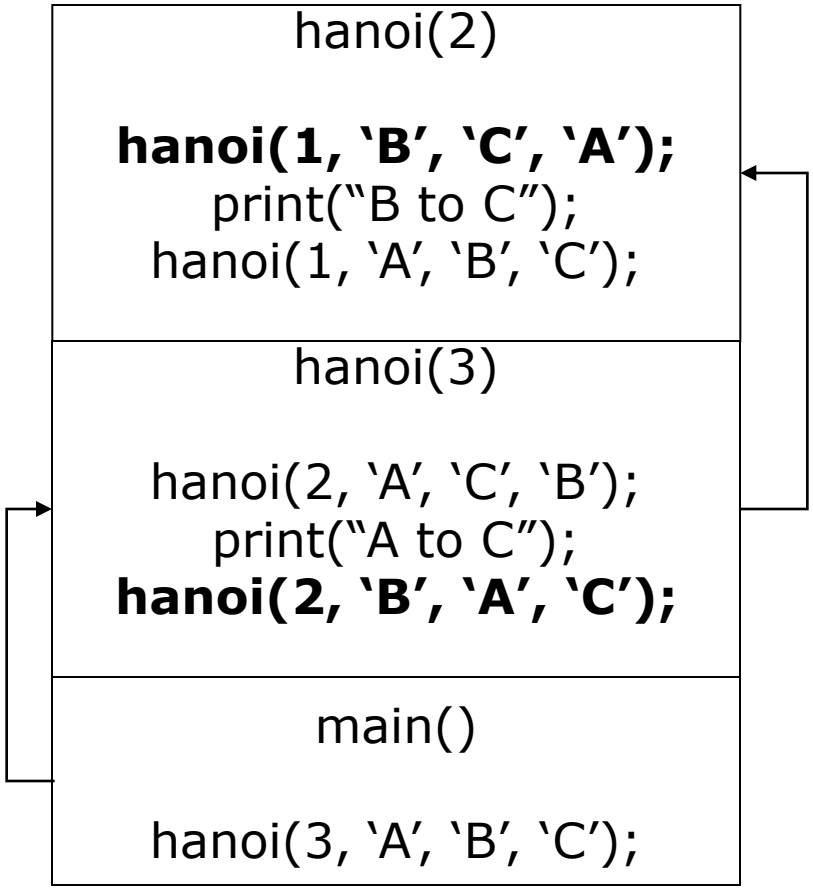
Move A to C

Move A to B

Move C to B

Move A to C

Recursion



The last line in hanoi(3) recursively calls hanoi(2). Note the changes in start, temp and dest

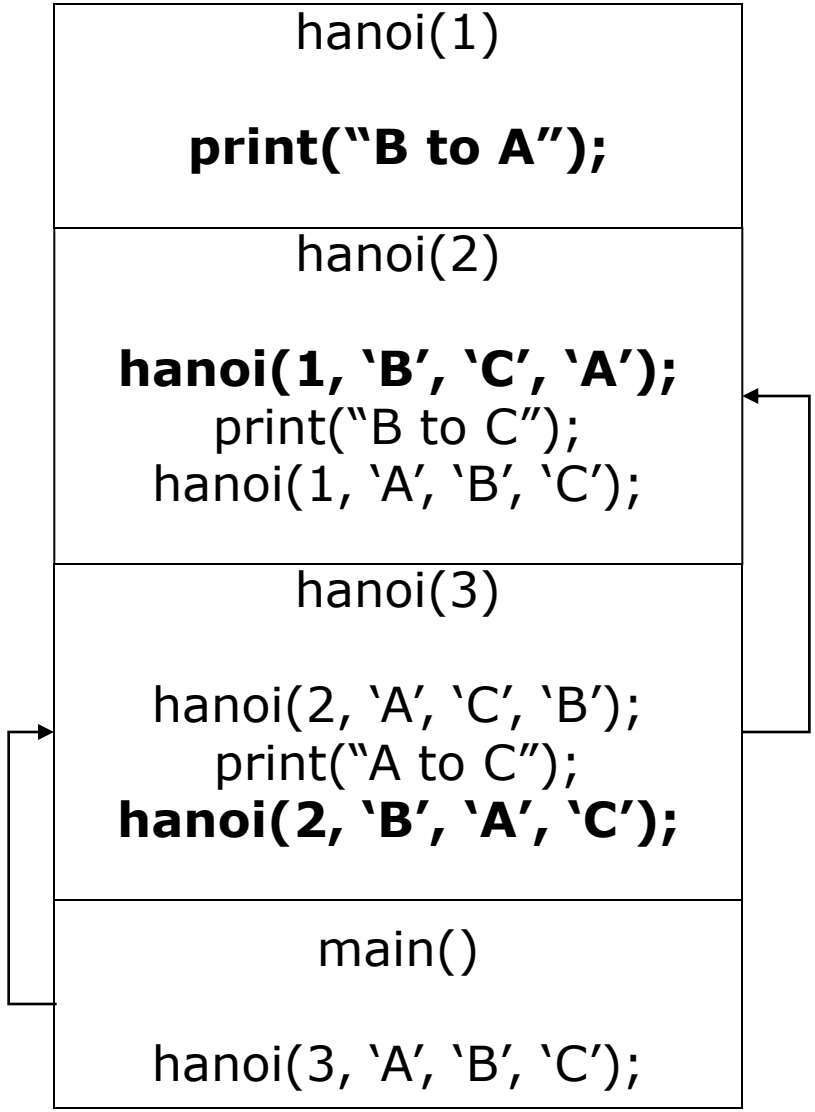
Move A to C

Move A to B

Move C to B

Move A to C

Recursion



hanoi(2) starts with a call to hanoi(1) which will be a base case printing B to A

Move A to C

Move A to B

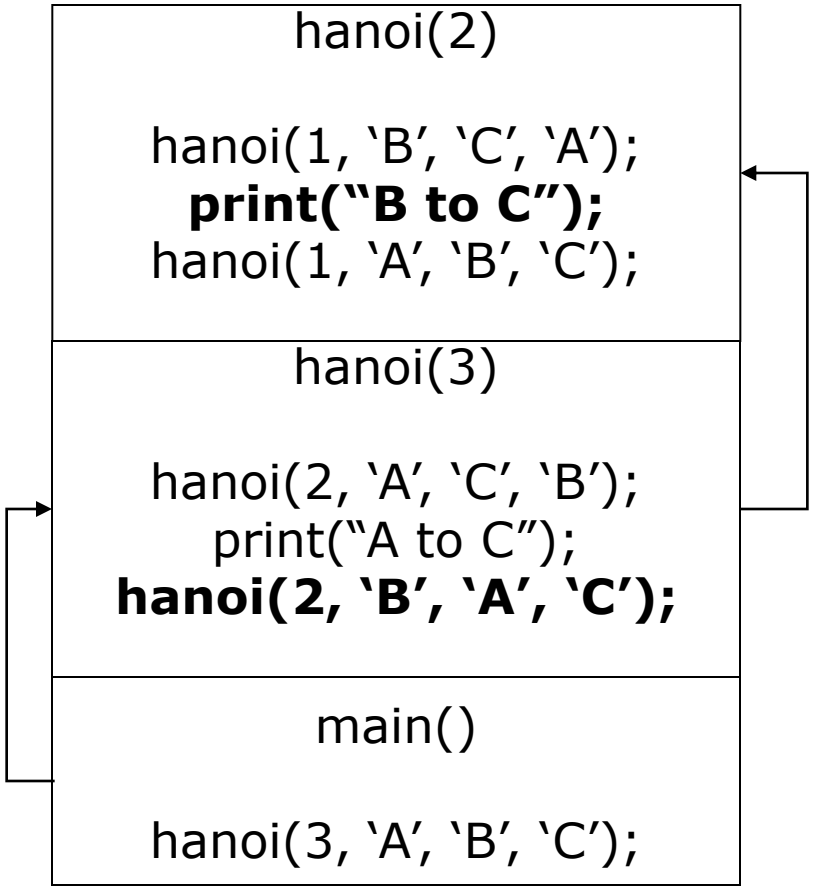
Move C to B

Move A to C

Move B to A

Recursion

Returning to hanoi(2), we now print B to C



Move A to C

Move A to B

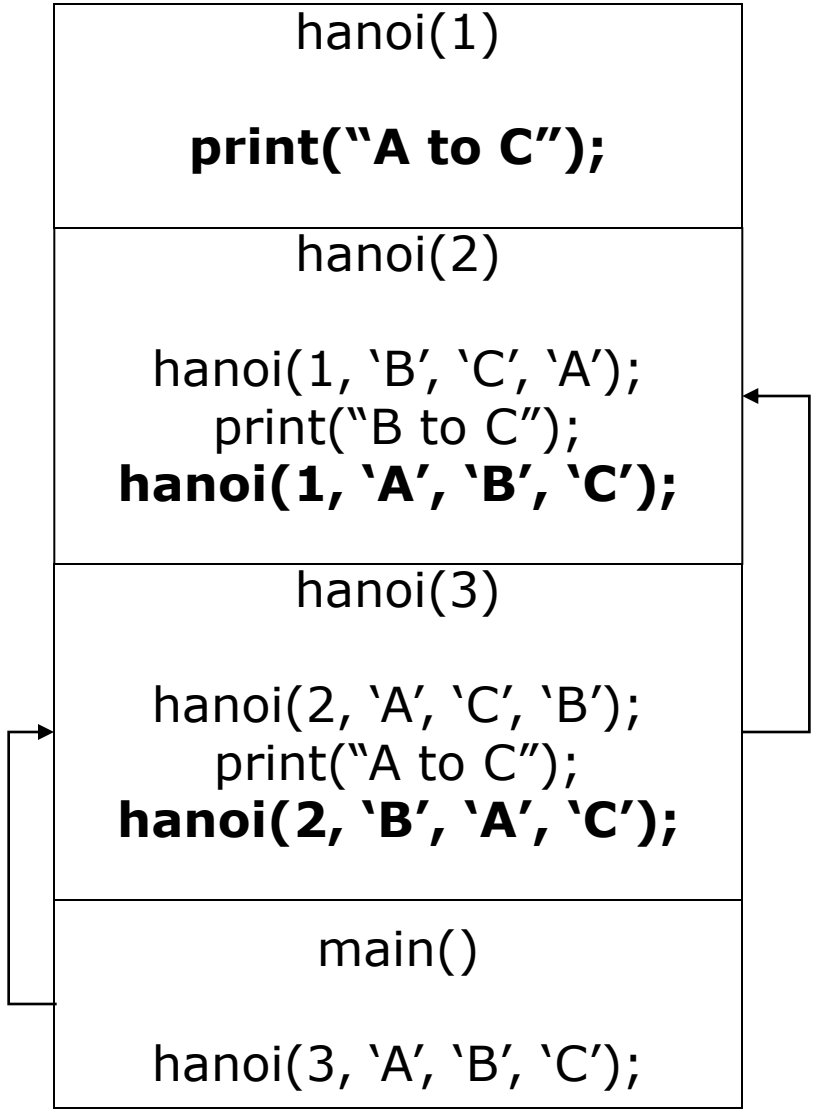
Move C to B

Move A to C

Move B to A

Move B to C

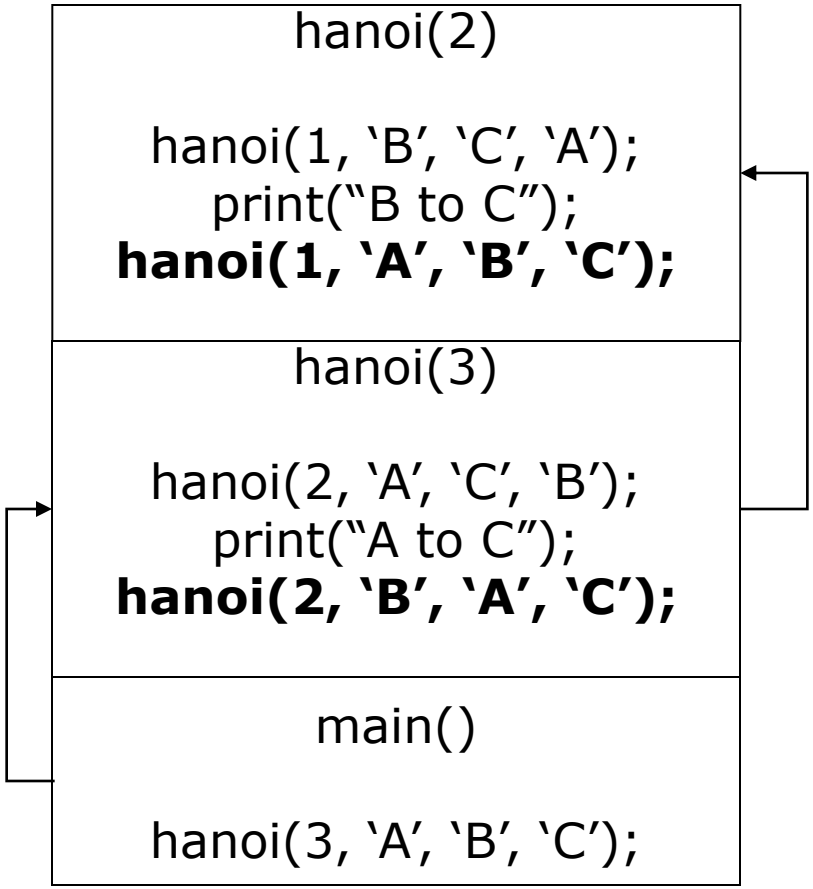
Recursion



Last line in Hanoi(2) is a final call to Hanoi(1), another base case

Move A to C
Move A to B
Move C to B
Move A to C
Move B to A
Move B to C
Move A to C

Recursion

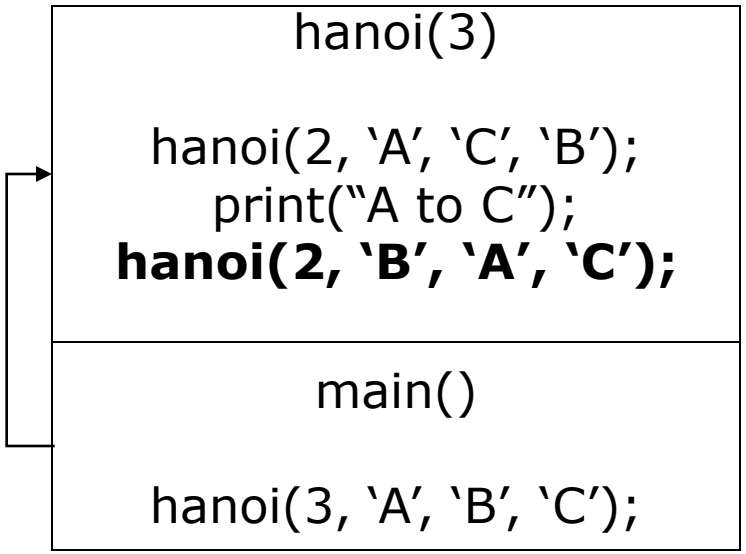


Everything finishes and unwinds at this point eventually getting back to main

Move A to C
Move A to B
Move C to B
Move A to C
Move B to A
Move B to C
Move A to C

Recursion

Everything finishes and unwinds at this point eventually getting back to main



Move A to C
Move A to B
Move C to B
Move A to C
Move B to A
Move B to C
Move A to C

Recursion

Everything finishes and unwinds at this point eventually getting back to main

```
main()

hanoi(3, 'A', 'B', 'C');
```

Move A to C
Move A to B
Move C to B
Move A to C
Move B to A
Move B to C
Move A to C

Recursion

Code is beautiful

- This is a key (and historic) example of the Towers of Hanoi solution
- Tracing through with only 3 disks may be interesting, but to truly appreciate it in action, trace through the code with 4 or more disks. This is an excellent way to practice recursion

Recursion

Analysis

- How many steps will we see with 3 disks? How many with 4? Can you generalize it to a formula?
  - 1 disk = 1 move
  - 2 disks = 3 moves
  - 3 disks = 7 moves
  - 4 = 15 moves
  - $N = 2^n - 1$
- Subsequently, 64 disks equals $2^{64} - 1$
- This equals an astounding 1.84×10^{19}
- At one move per second, this works out to about 585 billion years, with no restroom breaks. We have time before the end of the world ☺

Recursion

Performance

- Analyzing the Big-O for comparisons with recursive solutions is noteworthy, but often not seen as an improvement to a solution
- Moreover, the function call stack is heavily utilized making recursion higher in memory usage
- Functions like factorial and Fibonacci can probably perform faster and use memory better using standard loops iteratively
- In theory, an iterative function can be written recursively, and vice versa
- The iterative approach is often easier to visualize

Recursion

Recursion uses stack methodology

- Hanoi and some other solutions we'll see, though, could be a challenge to do iteratively versus with recursion
- Hanoi can be written iteratively, but to do so would require incorporating some use of a stack that recursion naturally does
- If you can understand the recursive approach and application, it can simplify code as we will see with more complex algorithms

Recursion

Summary

- Practice, practice, practice. Understanding recursive code by tracing is the first step
- There are many problems that have potential recursive solutions. Once you are able to read and trace recursive code, the next big challenge is learning how to develop a recursive algorithm
- The key is learning how to identify base and recursive cases. They are not easy to do, but very gratifying when developed
- Fun problems:
  - Checking if a given String is a palindrome
  - Checking if a given integer is a prime number
  - The classic “knapsack” problem
- Examples of other solutions always help too. We will see more as we look at the advanced sorting algorithms next…