4.5 Recursion

One of the most powerful programming techniques involves a function calling itself; this is called *recursion*. It is not immediately obvious that this is useful; take that on faith for now and concentrate first on how to make recursion work. The utility should become apparent as you see it in action.

Here is a complete program that will serve as our first example of recursion:

```
def Factorial(n):
    if n <= 1:
        return 1
    else:
        return n*Factorial(n-1)

def main():
    print( Factorial(4) )

main()
```

Program 4.5.1: The recursive Factorial function

Before we think about what the program does, let's concentrate on the Factorial () function:

```
def Factorial(n):
    if n == 1:
        return 1
    else:
        return n*Factorial(n-1)
```

Can we see what this function will return for Factorial (1)? Sure; if n is 1 then the first line: **if** n == 1 is True so Factorial (1) returns 1.

What about Factorial (2)? Well, if n is 2 then the test n==1 fails and the function returns 2* Factorial (1). But we just saw that Factorial (1) returns 1, so Factorial (2) must return 2.

Factorial (3)? If n is 3 then again n==1 is False and the function returns 3* Factorial (2). We just saw that Factorial (2) returns 2, so Factorial (3) must return 6.

One more time. If n is 4 then n==1 is False and the function returns 4* Factorial (3). We just saw that Factorial (3) returns 6, so Factorial (4) must return 24. If we now look at our program, in main() it prints Factorial (4), so the program prints the number 24 and halts.

Of course, you have surely noticed that Factorial (2) is 2, which is 2*1, Factorial (3) is 6, which is 3*2*1, and Factorial (4) is 24, which is 4*3*2*1. Our function is indeed computing what mathematicians call the Factorial () function. Factorial (n) is the product of the first n positive integers. This should make sense: if Factorial (n−1) is the product of the first n−1 positive integers, we have defined the function so that Factorial (n) is n times the product of the first n−1 positive integers, and that is the product of the first n positive integers.

Notice that the code for our Factorial () starts with an **if** statement one one of whose branches the function returns without calling itself. The values for which a function returns without recursing are called *base cases*. We computed Factorial (4) by starting at the base case Factorial (1) and working up until we reached the Factorial (4) case. Can we go the other way – computing Factorial(4) the way the computer does?

We start with a definition straight from the function:

```
Factorial(4)=4*Factorial(3)
```

We don't yet know the value of Factorial (3), so we remember where we are and start a new computation:

```
Factorial(4)=4*Factorial(3)
               Factorial(3)=3*Factorial(2)
```

We don't know the value of Factorial (2), so again we remember where we are and start a new computation:

```
Factorial(4)=4*Factorial(3)
               Factorial(3)=3*Factorial(2)
                              Factorial(2)=2*Factorial(1)
```

Finally, we do know the value of Factorial (1):

```
Factorial(4)=4*Factorial(3)
               Factorial(3)=3*Factorial(2)
                              Factorial(2)=2*Factorial(1)
                                             Factorial(1)=1
```

We can now work our way back up. Since we know Factorial (1)=1 we can compute Factorial (2):

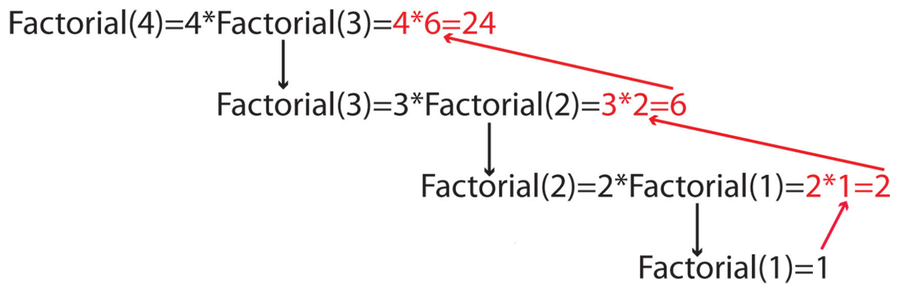
```
Factorial(4)=4*Factorial(3)
               Factorial(3)=3*Factorial(2)
                              Factorial(2)=2*Factorial(1)=2*1=2
                                             Factorial(1)=1
```

and now we can compute Factorial (3):

```
Factorial(4)=4*Factorial(3)
               Factorial(3)=3*Factorial(2)=3*2=6
                              Factorial(2)=2*Factorial(1)=2*1=2
                                             Factorial(1)=1
```

and finally

```
Factorial(4)=4*Factorial(3)=4*6=24
               Factorial(3)=3*Factorial(2)=3*2=6
                              Factorial(2)=2*Factorial(1)=2*1=2
                                             Factorial(1)=1
```


Factorial(4)=4*Factorial(3)=4*6=24
Factorial(3)=3*Factorial(2)=3*2=6
Factorial(2)=2*Factorial(1)=2*1=2
Factorial(1)=1

Follow the black arrows down into the recursion, then the red arrows back out.

You can imitate the way a computer executes any recursive function in this way. It becomes a bookkeeping problem – the difficulty is just in keeping track of what has been evaluated and where you are in the computation.

Notice that our recursive factorial function started with an if–statement. This is true of almost all recursive functions. The first question to ask is whether the argument to the function is one of the base cases. If so we can just return the answer; it is only when the argument is not a base case that we need to recurse. Notice also that in order for the recursive function call to terminate, the argument in the recursive call must be closer to a base can than the initial argument was. To compute `Factorial (n)` we recurse on `Factorial (n−1)`. If `n` is larger than 1, `n−1` is closer to the base case of 1 than `n` is. On the other hand, if `n` is not positive then `n−1` moves farther away from 1, and as a result the `Factorial ()` function never halts if you give it a non-positive argument.

The aspect of recursion that students find hardest is writing recursive functions. This almost seems like magic. Here is a way to think about the process:

- Find the base case. This usually tells you what the parameter of the function should be. Start the recursive function with an if-statement that describes how to perform the computation for the base case.
- Handle the recursive case. Describe how to perform the computation if the parameter is not the base case, in terms of parameters that would be closer to the base case.
- Be sure that the recursion always works down to one of the base cases.

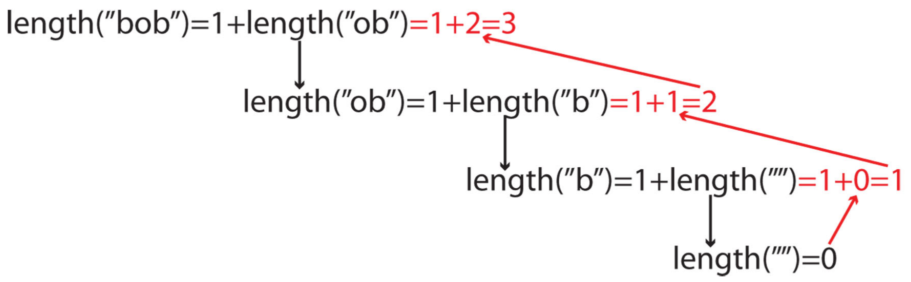
Example 1: *Write a recursive function that returns the length of a string.* Strings can be long: "Marvin Krislov is President of Oberlin College", or short: "bob". Short ones seem easier to deal with than long ones. The shortest possible string is "", the *empty* string. We'll take that as our base case. If *the string is empty* is our base case, our recursive parameter must be the string and our function starts

```
def length(string):
    if string == "":
        return 0
    else:
```

Now we need to handle the recursive cases: the cases where the base case does not apply. This means we have a string whose length is not 0. The length can't be negative, so our string must have positive length. To make use of recursion, we want to find its length in terms of the lengths of strings closer to the base case: i.e., string with shorter length. With a little thought you can see that the length of the string is 1 more than it would be if we removed one letter. One way to remove a letter is to make use of our string indexing operations: string [1:] consists of all of the letters of string after the first. We add this to our code with one line:

```
def length(string):
    if string == "":
        return 0
    else:
        return 1 + length(string[1:]
```

That is our function. Does it always lead us to a base case? Sure; for any starting string we remove one letter at a time until we get to the empty string:


length("bob")=1+length("ob")=1+2=3
length("ob")=1+length("b")=1+1=2
length("b")=1+length("")=1+0=1
length("")=0

Example 2: *Write a recursive function to make the Fibonacci sequence 0,1,1,2,3,5,8,13.....* The Fibonacci sequence is easy to describe in English: it starts with 0 and 1, and then every subsequent number is the sum of the two previous numbers. Since "starts with 0 and 1" is a statement about position in the sequence, we will write a function Fib(n), where n refers to position in the sequence: Fib(0) is the first number in the sequence, Fib(1) is the second, and so forth. Our English description of the problem gives us two base cases:

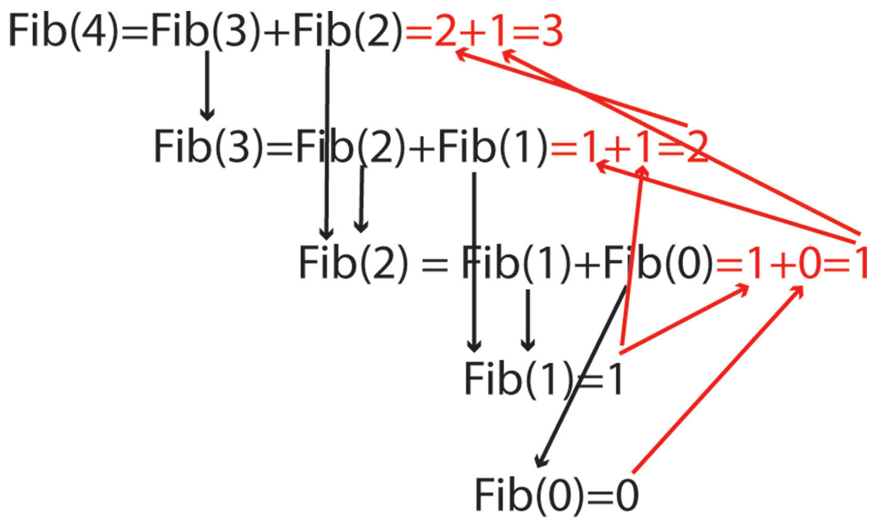
```
def Fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
```

We will look later at whether we actually need both of the base cases. Our English description is easy to translate into the recursive case of our function. The description says "every subsequent number is the sum of the two previous numbers." If n is the index of the number we are currently calculating, the indexes of the two previous numbers are n−1 and n−2. In other words, our description says Fib(n)=Fib(n−1)+Fib(n−2). We translate this into code by making the right hand side the value that we return:

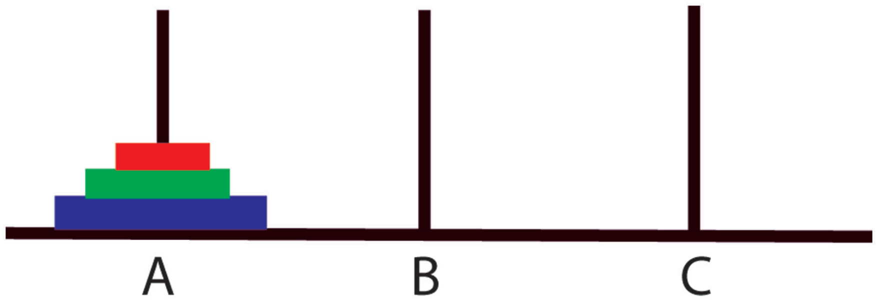
```
def Fib(n):
    if n==0:
        return 0
    elif n==1:
        return 1
    else:
        return Fib(n-1)+Fib(n-2}
```

It is easy to see that we need both of our base cases. The calculation of Fib(2 uses both of them: Fib(2)=Fib(1)+Fib(0). If we omitted one of them as a base case and tried to calculate it recursively we would get into trouble. For example, if we tried to calculate Fib(1) recursively we would get Fib(1)=Fib(0)+Fib(−1)=0+Fib(−2)+Fib(−3)=... and this clearly will never terminate. Are our two base cases enough? Yes; the calculation works down through even numbers and odd numbers and our base cases of 0 and 1 terminate both the evens and the odds.

Here is a sample calculation with our function


Fib(4)=Fib(3)+Fib(2)=2+1=3
Fib(3)=Fib(2)+Fib(1)=1+1=2
Fib(2) = Fib(1)+Fib(0)=1+0=1
Fib(1)=1
Fib(0)=0

Example 3: *Write a recursive function to solve the Towers of Hanoi puzzle* *Towers of Hanoi* is a puzzle game invented in1883 by the French mathematician Edouard Lucas. In this game there are three ”towers” or vertical sticks that can hold a set of concentric disks. At the start of the game the disks are all on one tower, in order of decreasing size with the largest disk at the bottom and the smallest at the top:



The goal of the game is to move all of the disks from their starting tower to another tower, using the following three rules:

1. Only one disk can be moved at a time.

2. Only the top disk on a tower can be moved.

3. A disk can only be placed on an empty tower or on a larger disk, never on a smaller disk.

Lucas's game included a story, supposedly taken from the writings of an eminent Chinese monk who lived in Hanoi, that there is a temple in India where the Brahmin monks are working on a version of this puzzle with 64 golden disks. A prophecy claims that when the monks finish solving the puzzle the world will come to an end.

Just for fun, here is the cover of the box containing Lucas's first version of the puzzle:

Now we need to find a solution. As usual we'll start thinking about base cases. An easy setup to solve is when we have only one disk on the starting tower; we can move that to either of the other towers in one step. However, an even easier setup is when we have no disks at all; we can solve that without doing anything! Both of these cases refer to the number of disks on the starting tower. If you think about it, the only difference in two distinct starting states is the number of disks they contain. So we'll include the number of disks as a parameter on our Hanoi() function. Since our case with 0 disks requires no work, we can start the function as follows

```
def Hanoi(n):
    #Solves Towers of Hanoi with n disks on the starting tower
    if n>0:
```

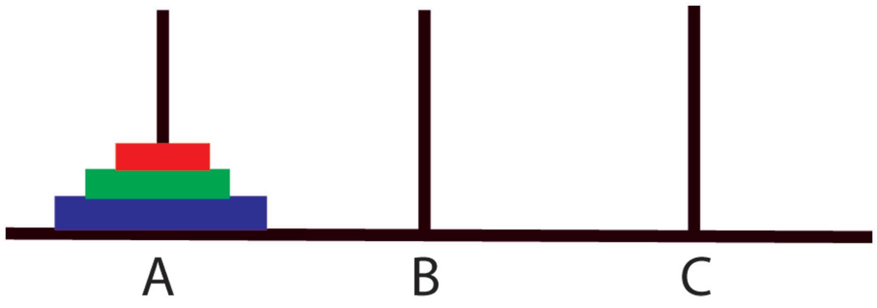
This, however, is not enough. We need to know not just how many disks are on the starting tower, but the situations with the other towers as well. Once we are into the recursion we need to move disks not just from the starting tower, but from all three towers. So we will include all three towers as parameters:

```
def Hanoi(n, source, destination, free):
    # Moves n disks from the "source" tower to the
    # "destination" tower, making use of the "free" tower.
    if n>0:
```

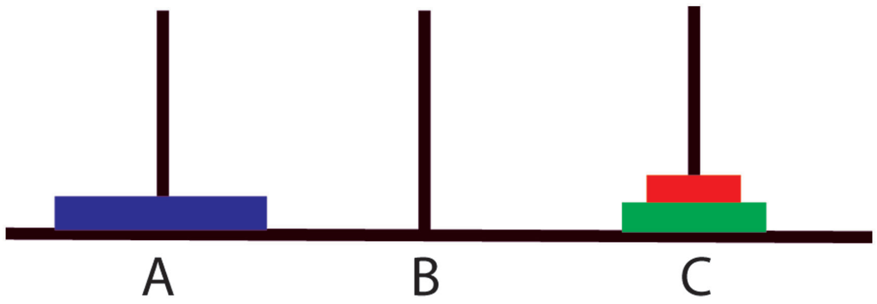
Now we are rolling. We need to express how to move `n` disks from the source to the destination in terms of moving small numbers of disks from one tower to another. This is easy:

1. Move `n−1` disks from the source tower to the free tower.
2. Move 1 disk, the largest, from the source tower to the destination.
3. Move `n−1` disks from the free tower to the destination.

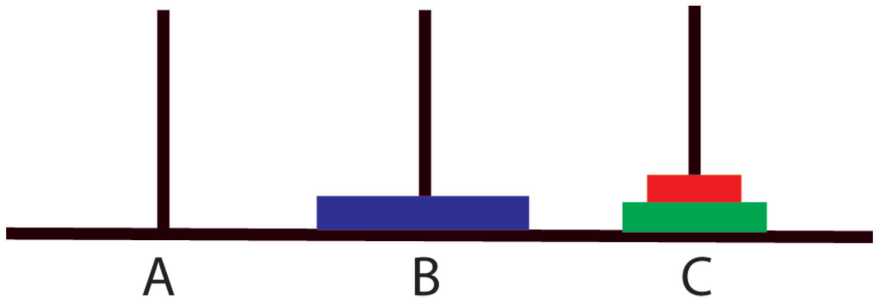
For example, suppose we start with 3 disks on tower A, the other towers empty:



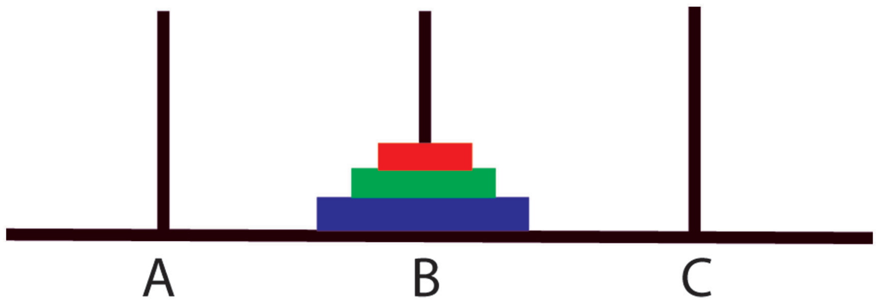
We will think of A as our source tower, B as our destination and C as free. We first recursively move 2 disks from A to C:



Next we move the largest disk from A to B:



Finally, we move those 2 disks that we placed onto the "free" tower C to the destination tower:



To turn this into code, we just need to realize that the only actual disk moves are occurring on step (2), when we move one disk from the source tower to the destination tower. Everything else is just organization for the recursive calls. We will handle step (2) with a print statement. Here is the complete code for our solution, along with a typical main() function that solves the puzzle by moving 3 disks from tower "A" to tower "B":

```
def Hanoi(n, source, destination, free):
    # Moves n disks from the "source" tower to the
    # "destination" tower, making use of the "free" tower.
    if n > 0:
        Hanoi(n-1, source, free, destination)
        print( "%s ---> %s"%(source, destination))
        Hanoi(n-1, free, destination, source)

def main():
    Hanoi(3, "A", "B", "C")

main()
```

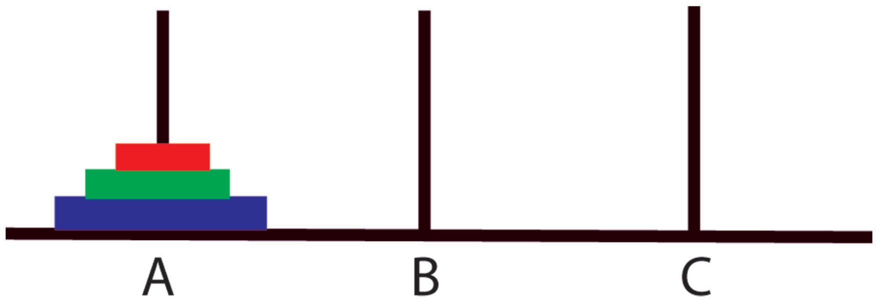
Here is a trace of this program. Because the Hanoi() function doesn't return anything, we need to use a slightly different style of trace. This time we indent to show levels of recursion. To make it a little easier to read we'll leave the quotes off the strings. At the top level Hanoi(3,A,B,C) is performed with 3 calls:

```
Hanoi(2,A,C,B)
print A ---> B
Hanoi(2,C,B,A)
```

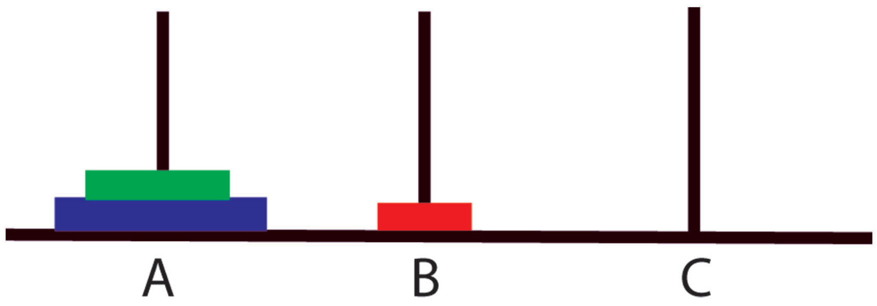
We write those three at the same indentation level, then go in a further level for the recursive calls for Hanoi(2,A,C,B) and Hanoi(2,C,B,A). Here is the resulting trace:

```
Hanoi(3,A, B, C)
    Hanoi(2,A,C,B)
        Hanoi(1,A,B,C)
            print A--->B
        print A--->C
        Hanoi(1,B,C,A)
            print B--->C
    print A--->B
    Hanoi(2,C,B,A)
        Hanoi(1,C,A,B)
            print C--->
        print C--->B
        Hanoi(1,A,B,C)
            print A--->B
```

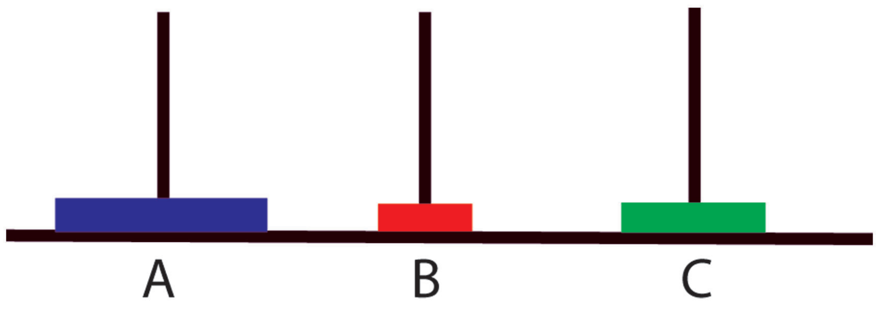
We will walk through the 7 steps. We start with



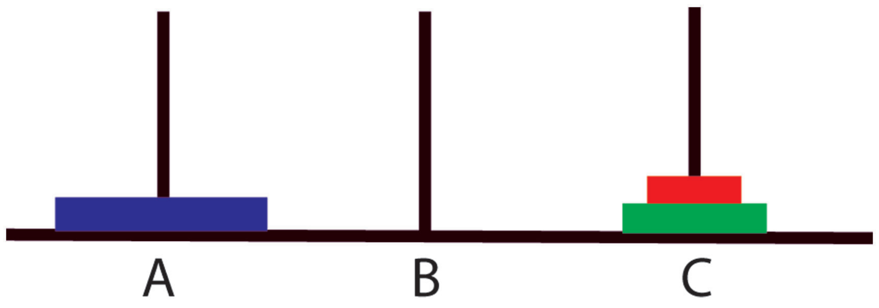
The first move is A—>B:



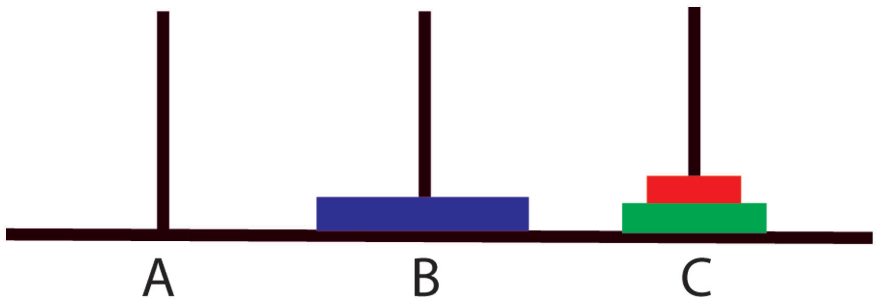
Then A—>C:



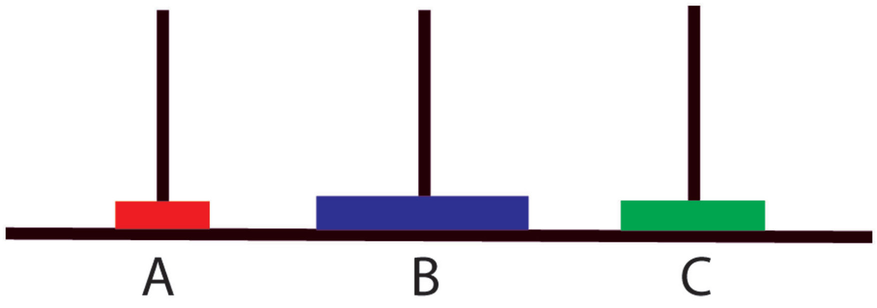
B—>C:



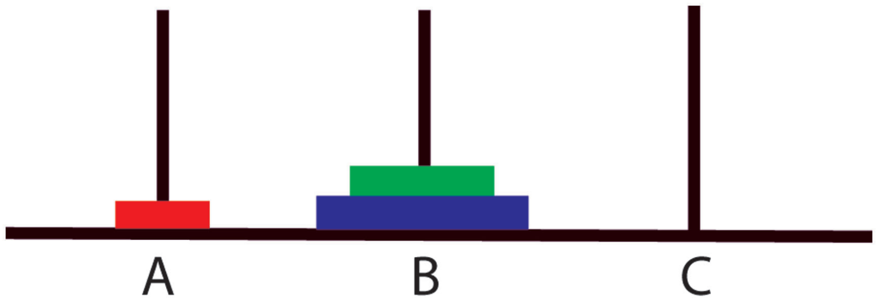
A–>B:



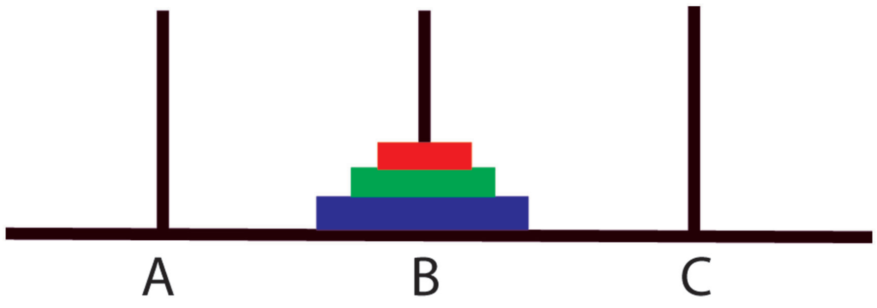
C—>A:



C—>B:



And finally A–>B:



It is not hard to show that solving the puzzle with `n` disks takes $2^n - 1$ moves. Those 64 disks the monks are supposedly moving need about 10^{19} individual moves. There are about $3 * 10^7$ seconds in a year, so if the monks were moving one disk per second it would take roughly $3 * 10^{11}$ years to solve the puzzle. Sadly, astrophysicists currently estimate that the sun will burn out in on $5 * 10^9$ years, so the monks really need to speed up.