RECURSION

Recursion Introduction

- Most mathematical functions are described by a simple formula
  - $C = 5 \times (F - 32) / 9$
  - It is trivial to program the formula
- Mathematical functions are sometimes defined in a less standard form
  - $f(0) = 0$ and $f(x) = 2f(x-1) + x^2$
  - Then, $f(0) = 0$, $f(1) = 1$, $f(2) = 6$, $f(3) = 21$, …
- A function that is defined in terms of itself is *recursive*, and C allows recursive functions

Example

- Factorial: N! = N * (N-1)!
- Fibonacci Number: F(n) = F(n-1)+F(n-2)
- $x^n = x \times x^{n-1}$
- Tower of Hanoi

Example: factorial

- Example: factorials
  - `5! = 5 * 4 * 3 * 2 * 1`
  - Notice that
    - `5! = 5 * 4!`
    - `4! = 4 * 3! ...`
  - Can compute factorials recursively using factorials
  - Solve base case (`1! = 0! = 1`) then plug in
    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`

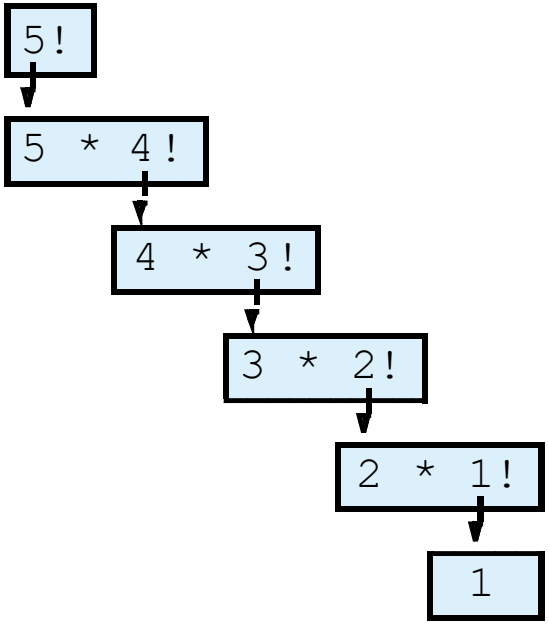
Factorial

- The following function computes $n!$ recursively, using the formula $n! = n \times (n - 1)!$:

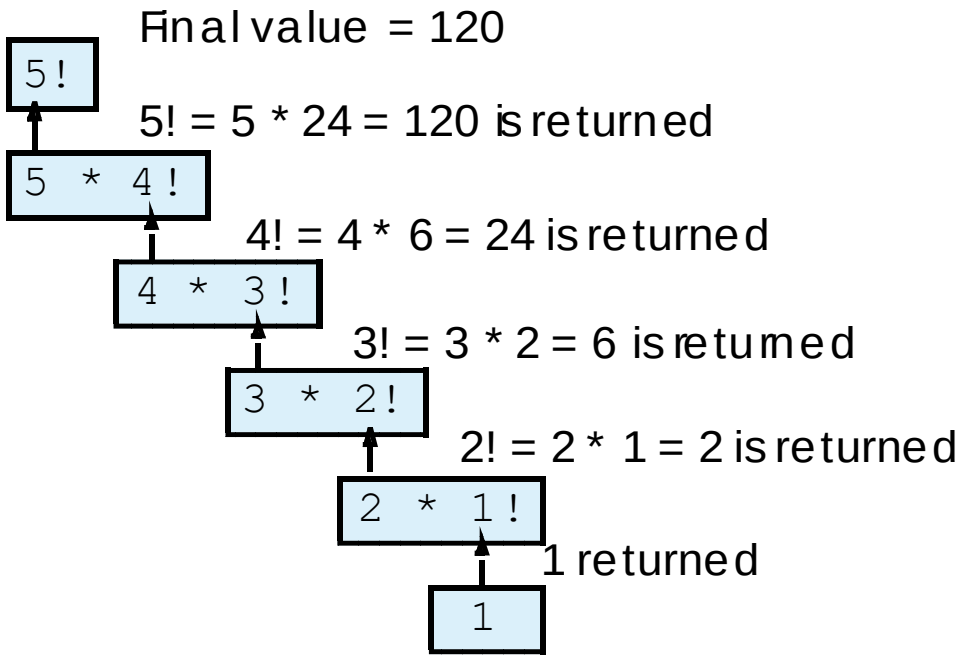
```
int factorial(int n)
{
  int t;
  if (n <= 1)
    t=1;
  else
    t=n * factorial(n-1);
  return t;

}
```

Recursion



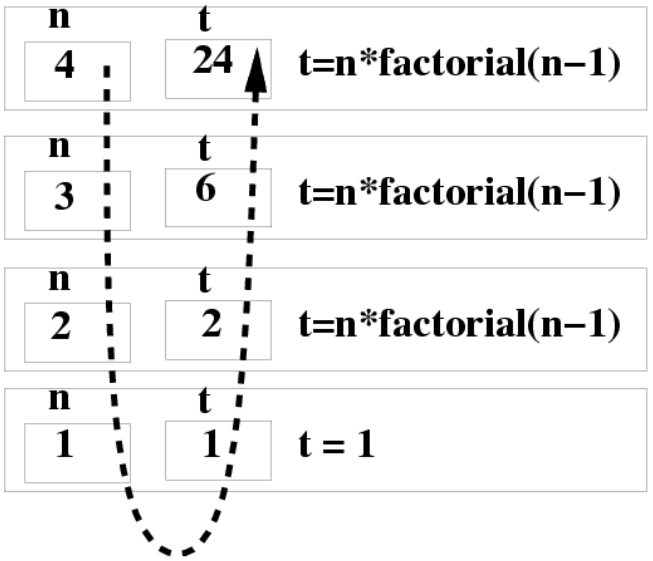
(a) Sequence of recursive calls.



(b) Values returned from each recursive call.

FACTORIAL()

```
int factorial (int n) {                    | factorial(4) =
    int t;                                 | 4 * factorial(3) =
    if (n <= 1) t = 1; /* BASE */ | 4 * 3 * factorial(2) =
    else t= n * factorial(n-1);            | 4 * 3 * 2 * factorial(1) =
    return t; /* Make Progress */           | 4 * 3 * 2 * 1 = 24
}
```



Fundamental Rules of Recursion

- **Base Case:** Must have some base cases which can be solved without recursion
- **Making Progress:** for the case that are to be solved recursively, the recursive call always make progress toward the base case
- If you do not follow the above rule, your program would not terminate
- An Example of Non-Terminating Recursion

```
int BAD (int x) {
  if (x==0) return (0);
  else return( BAD( x/3 + 1 ) + x-1 );
}
main() { int y; y = BAD(1) + BAD(4);}
```

Recursion

- The following recursive function computes x^n, using the formula $x^n = x \times x^{n-1}$.

```
int power(int x, int n)
{
   int res;
   if (n == 0)
      return 1;
   else{
      res=x * power(x, n - 1);
      return res;
  }
}
```

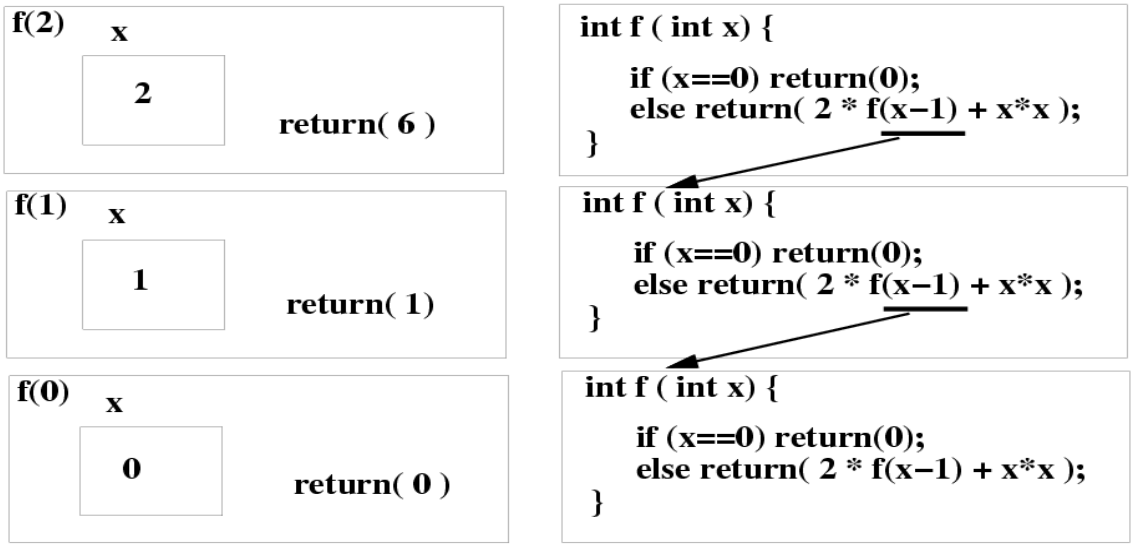
Recursion

- Both `fact` and `power` are careful to test a "termination condition" as soon as they're called.
- All recursive functions need some kind of termination condition in order to prevent infinite recursion.

Recursive Functions: $f(x)=2*f(x-1)+x^2$

- A function that calls itself either directly of indirectly

```
int f (int x)
{
  if( x == 0 ) return (0);                  /* Base Case */
  else return(  2*f(x-1) + x*x  );          /* Make Progress*/
}
```



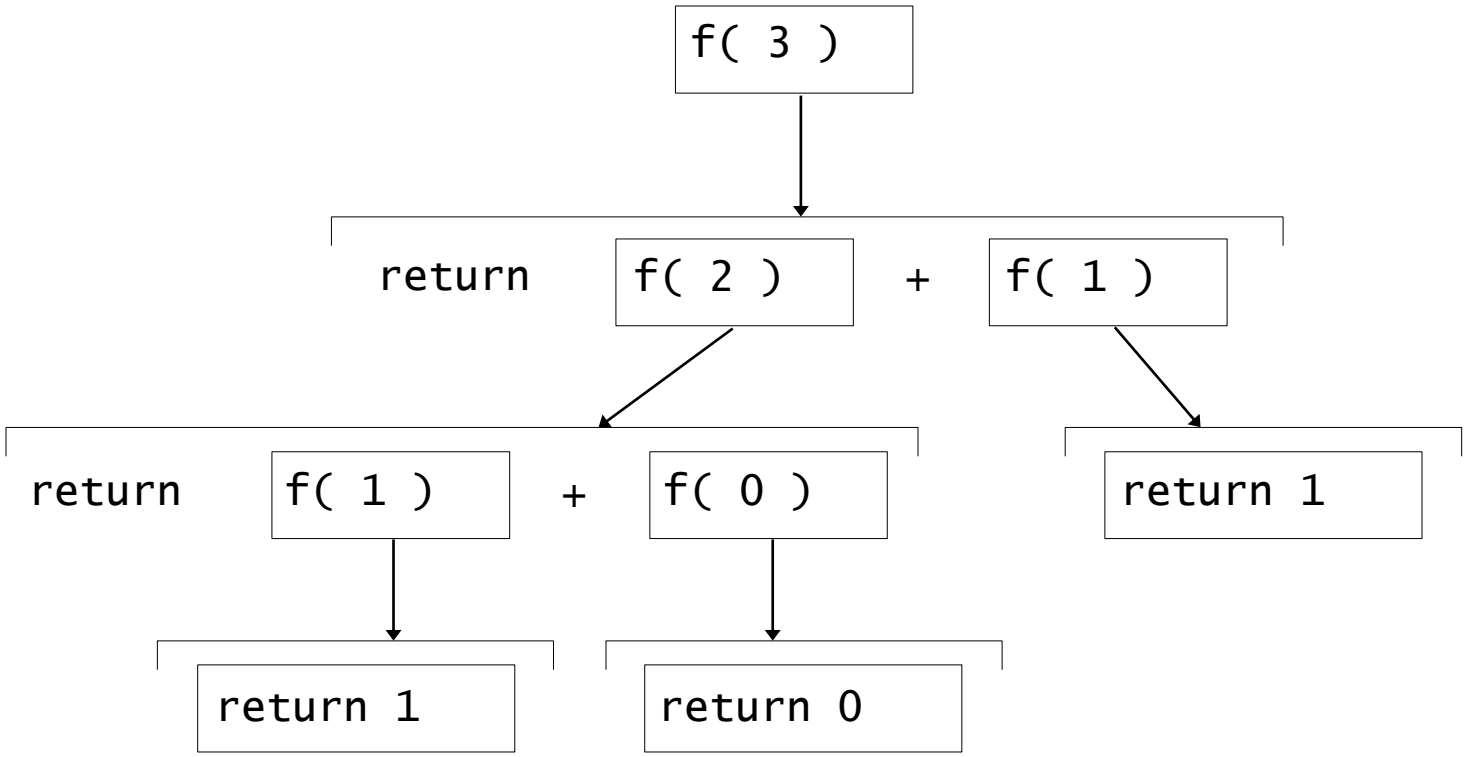
Recursion Example: Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number is the sum of the previous two
  - Can be solved recursively:
    - `fib( n ) = fib( n - 1 ) + fib( n - 2 )`
  - Code for the `fibonacci` function

```
long fibonacci( long n )
{
  if (n == 0 || n == 1)  // base case
     return n;
  else
     return fibonacci(n-1) +fibonacci(n-2);
}
```

Example Using Recursion: The Fibonacci Series

- Set of recursive calls to function `fibonacci`



Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance between performance and software engineering
  - Iteration achieves better performance (e.g., numerical computation)
    - Why? Function call involves lots of overhead
  - Recursion allows simple & efficient programming for complex problems, especially in non-numerical applications
    - e.g., quick sort, depth first traversal, tower of hanoi, ….

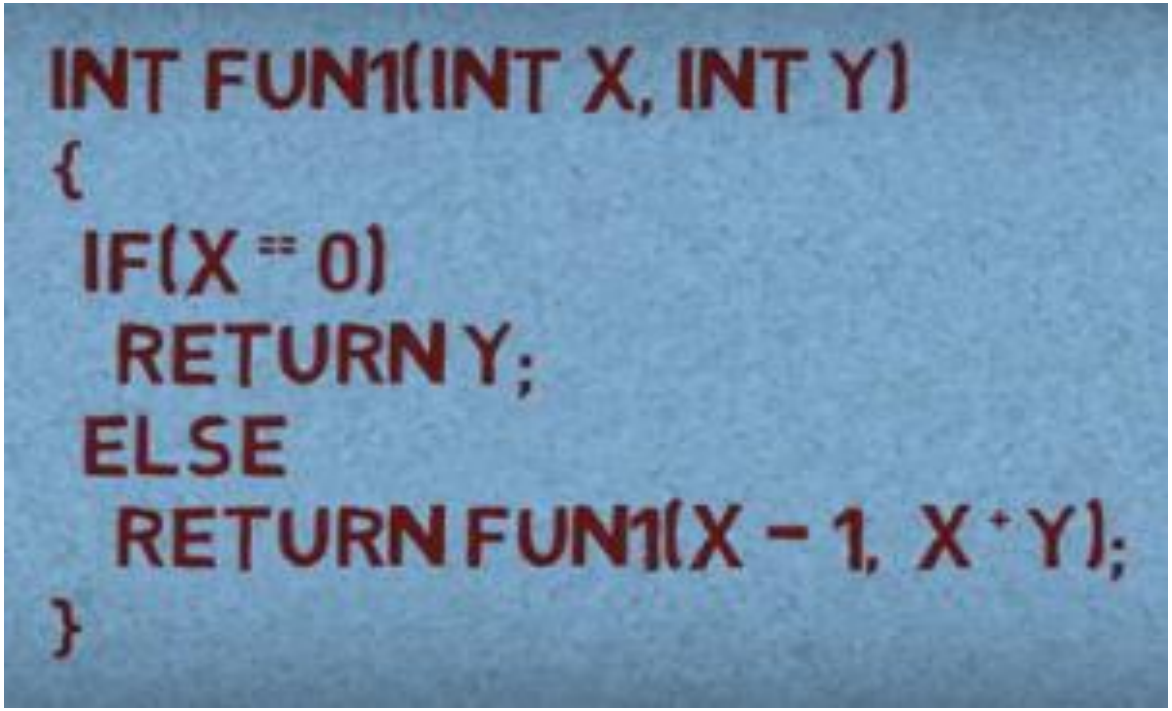
Example-Problem GCD

- Determine the greatest common divisor (GCD) for two numbers.
- Euclidean algorithm: GCD (a,b) can be recursively found from the formula

$$GCD(a,b) = \begin{cases} a & \text{if } b=0 \\ b & \text{if } a=0 \\ GCD(b, a \bmod b) & \text{otherwise} \end{cases}$$

Try Yourself

- FUN1(5, 2)



Try Yourself

- Write a function that will print binary equivalent of n
- For example if n is 21 then binary equivalent is 10101

SOLUTION

```
int convert(int dec)
{
    if (dec == 0)
    {
        return 0;
    }
    else
    {
        return (dec % 2 + 10 * convert(dec / 2));
    }
}
```

Fibonacci

```
int fibonacci(int i) {

   if(i == 0) {
      return 0;
   }

   if(i == 1) {
      return 1;
   }
   return fibonacci(i-1) + fibonacci(i-2);
}
```

The Tower Of Hanoi

Edouard Lucas - 1883

ONCE UPON A TIME!

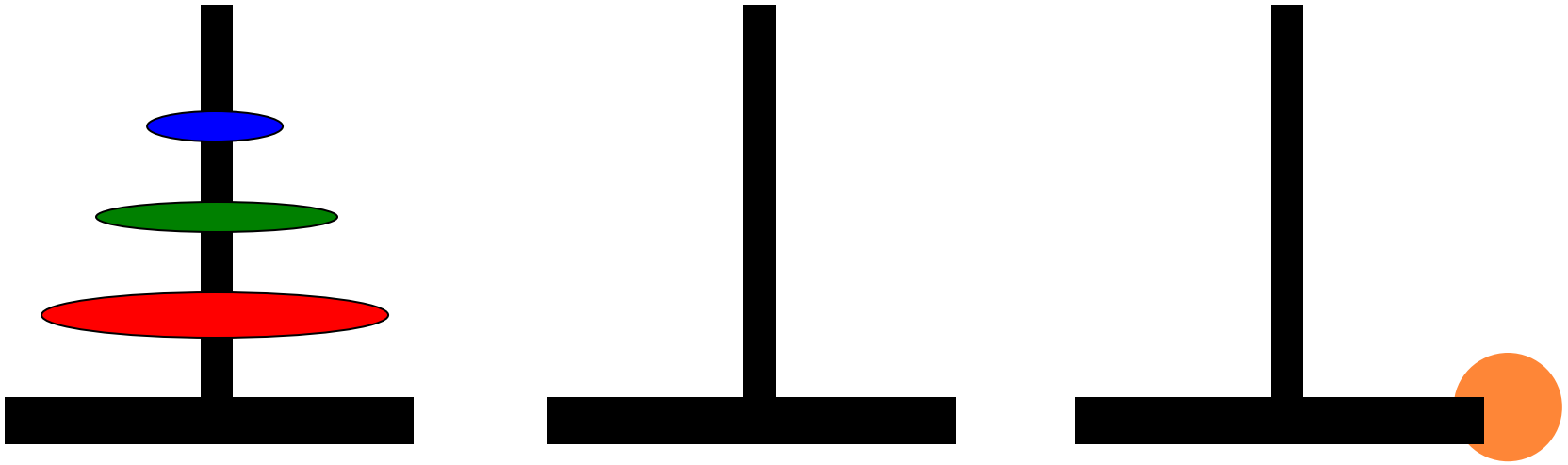
- The Tower of Hanoi (sometimes referred to as the Tower of Brahma or the End of the World Puzzle) was invented by the French mathematician, Edouard Lucas, in 1883. He was inspired by a legend that tells of a Hindu temple where the pyramid puzzle might have been used for the mental discipline of young priests.

WHAT DOES LEGEND SAY??

- Legend says that at the beginning of time the priests in the temple were given a stack of 64 gold disks, each one a little smaller than the one beneath it. Their assignment was to transfer the 64 disks from one of three poles to another, with one important proviso a large disk could never be placed on top of a smaller one. The priests worked very efficiently, day and night. When they finished their work, the myth said, the temple would crumble into dust and the world would vanish. How many moves (& how long) *would* the priests need to take to complete the task??

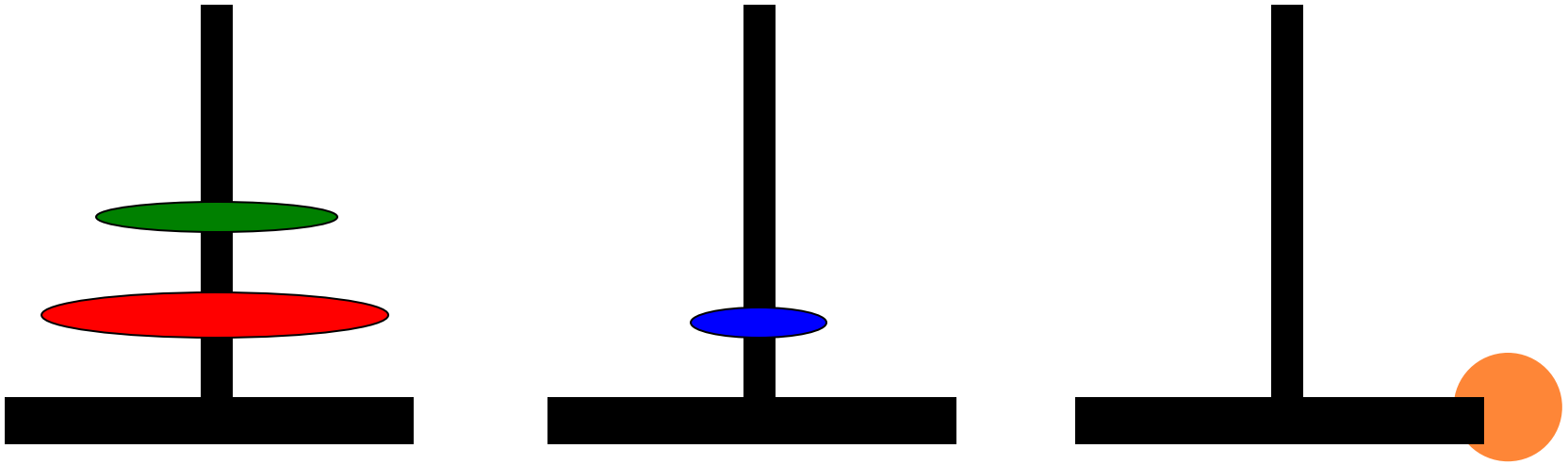
Tower of Hanoi

(0,0,0)



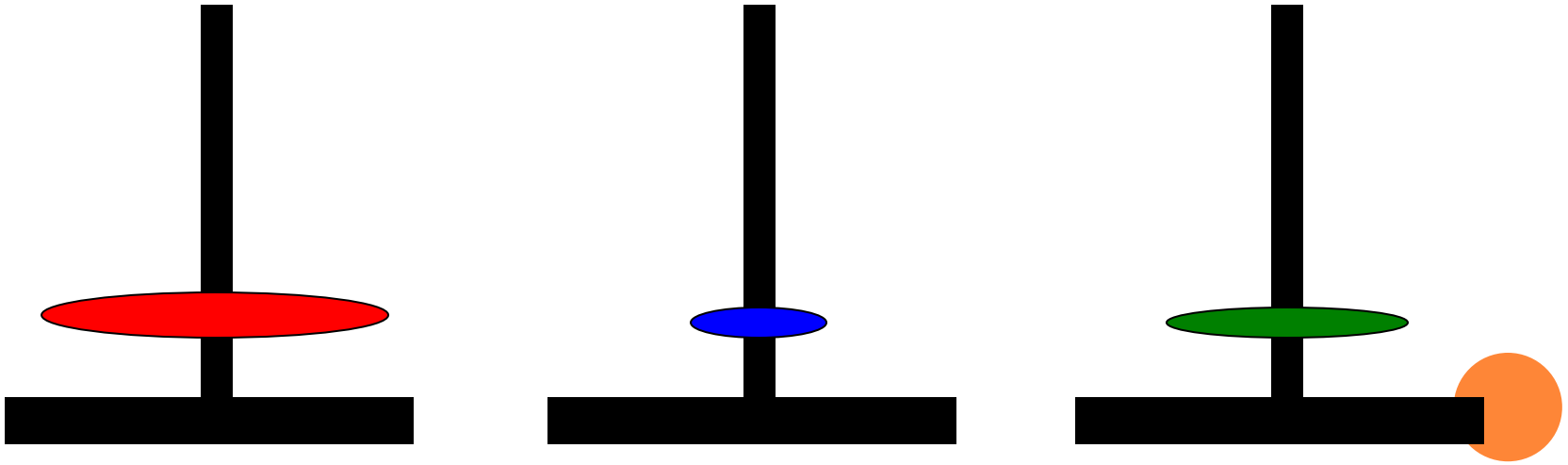
Tower of Hanoi

(0,0,1)



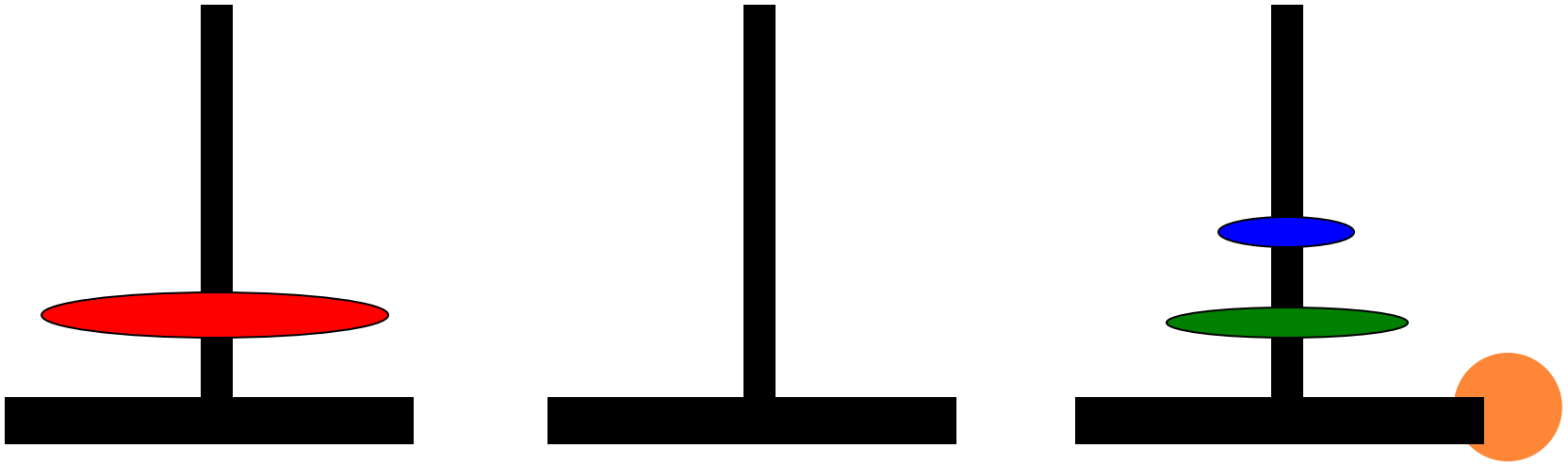
Tower of Hanoi

(0,1,1)



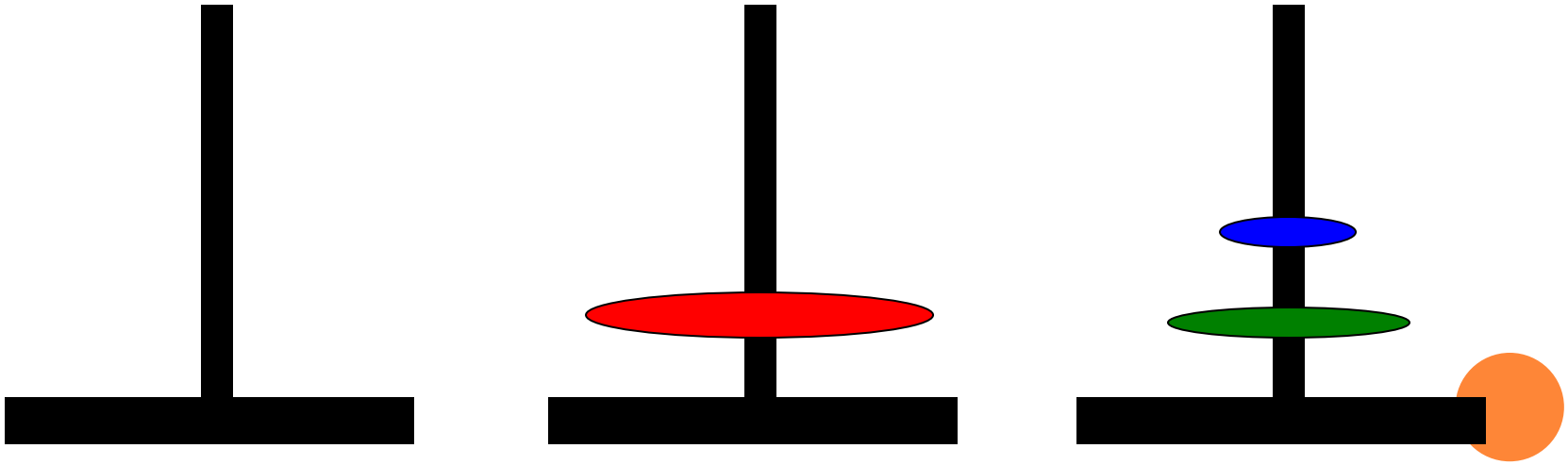
Tower of Hanoi

(0,1,0)



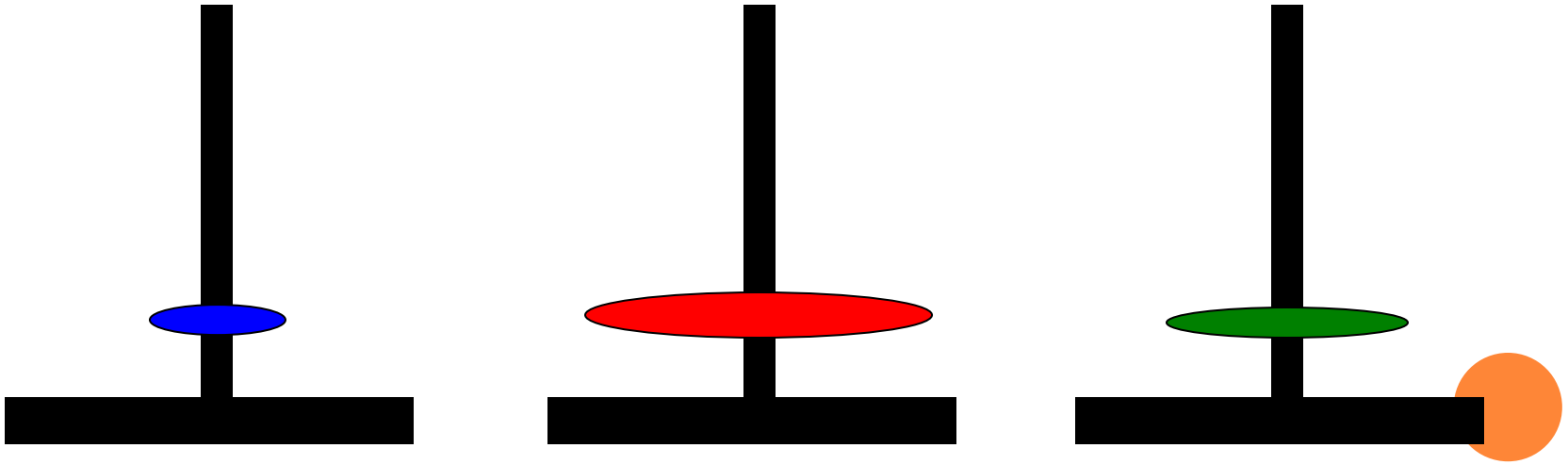
Tower of Hanoi

(1,1,0)



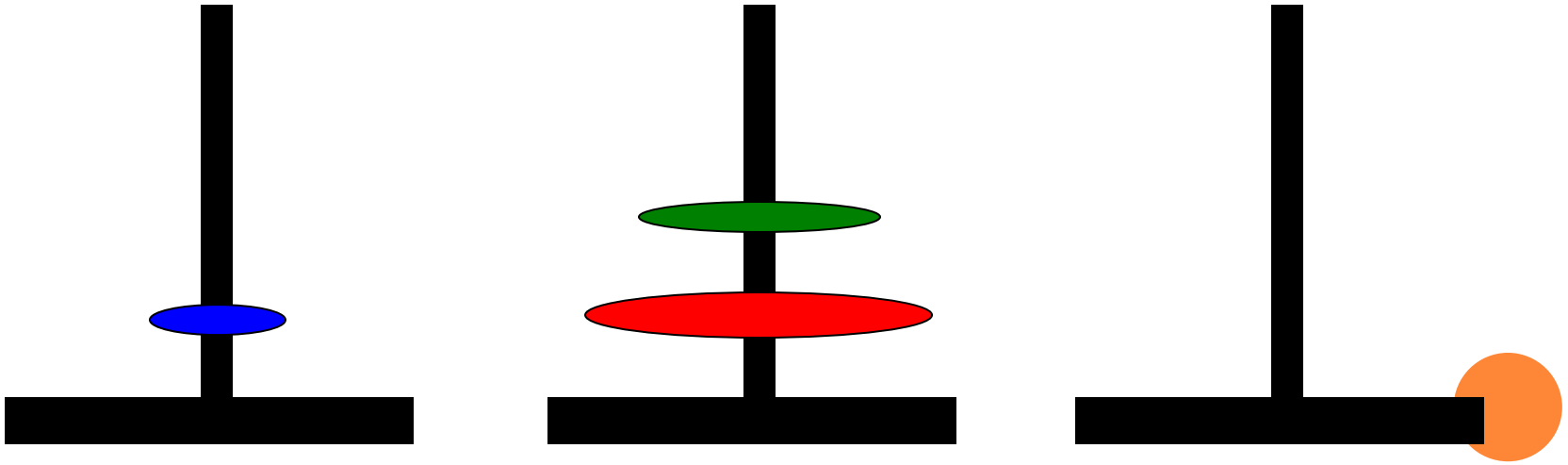
Tower of Hanoi

(1,1,1)



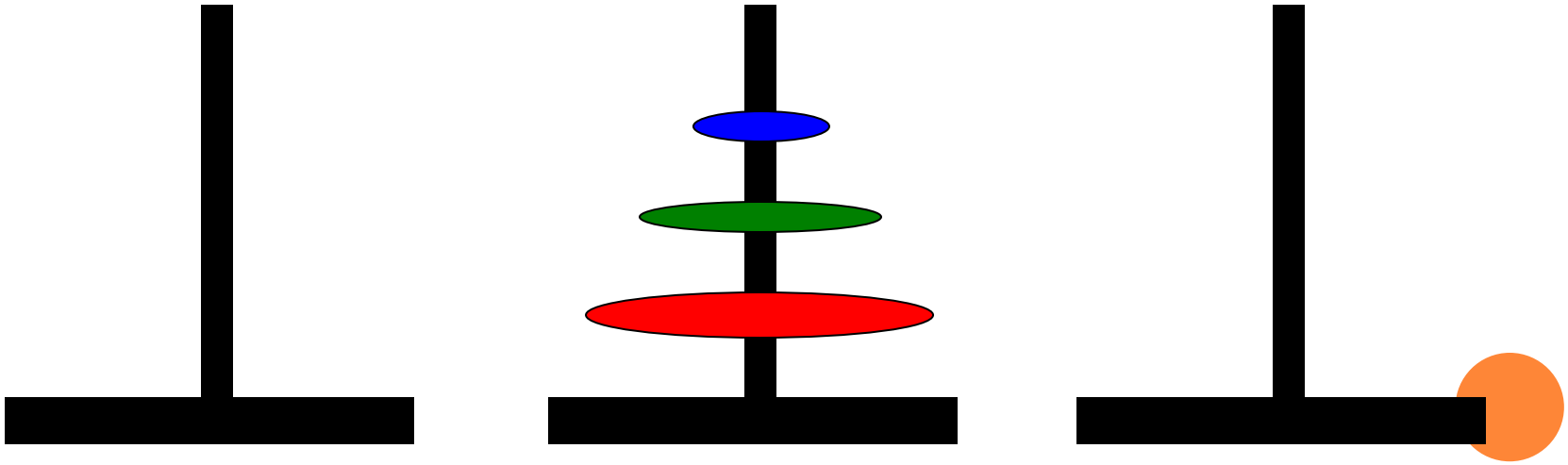
Tower of Hanoi

(1,0,1)



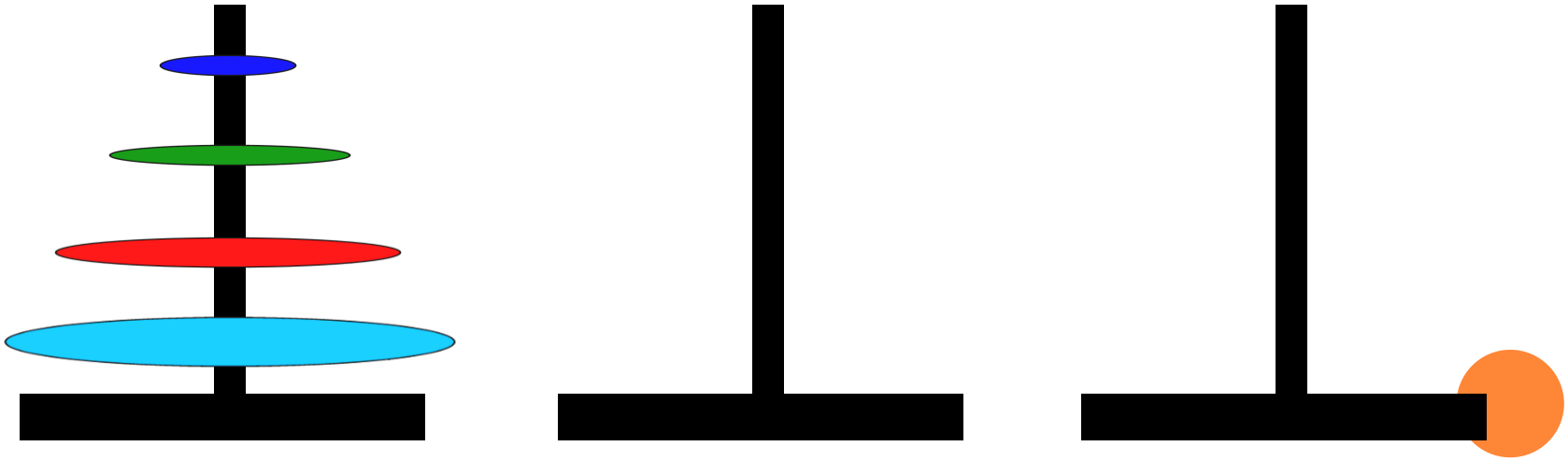
Tower of Hanoi

(1,0,0)



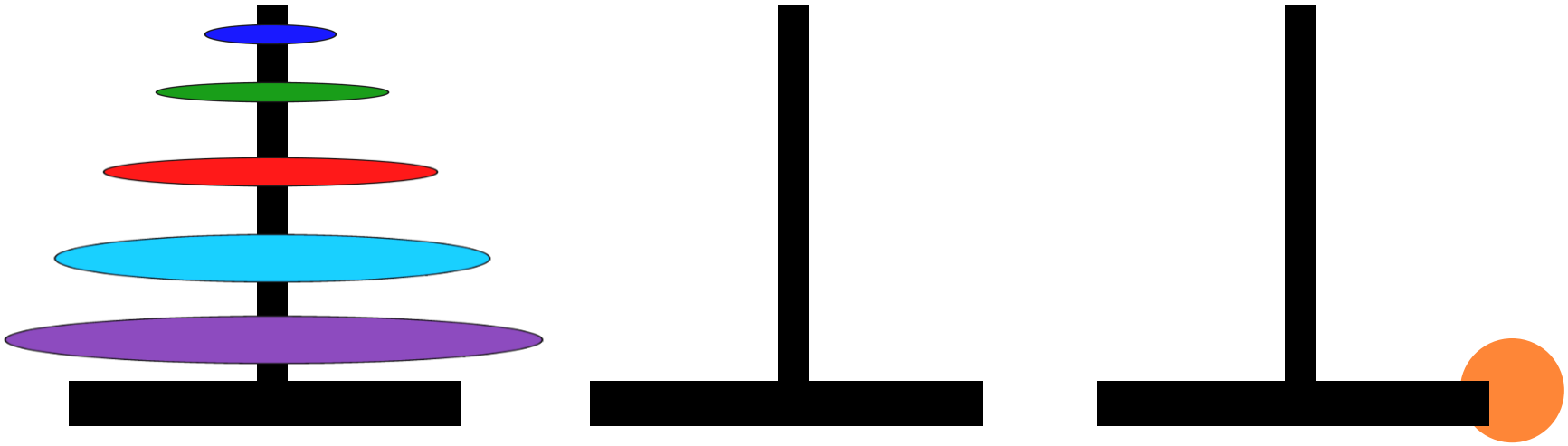
Tower of Hanoi

Try it



Tower of Hanoi

Try it



Tracing

Hanoi(3, A, B, C)

Hanoi(2, A, C, B)

Hanoi(1, A, B, C)

A => C

A => B

Hanoi(1, C, A, B)

C => B

A => C

Hanoi(2, B, A, C)

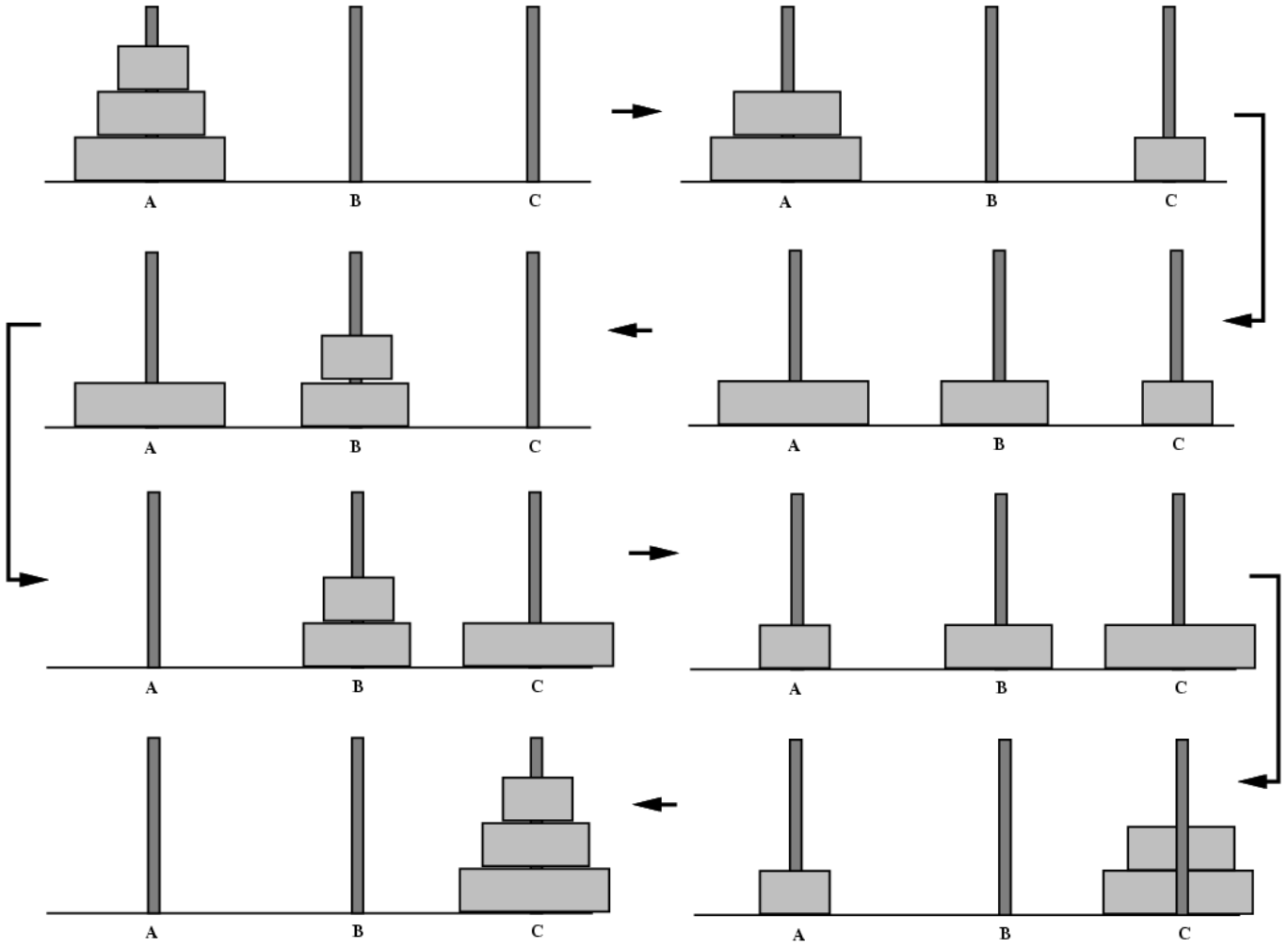
Hanoi(1, B, C, A)

B => A

B => C

Hanoi(1, A, B, C)

A => C



Example: The Tower of Hanoi

- Recursive Solution: Divide and Conquer
  - The problem of moving ***n*** smallest disks from A to C can be thought of two subproblems of size ***n* – 1**
    - First, move the smallest *n -1* disks from A to B
    - Then, move the *n*th disk from A to C
    - Finally, move the smallest *n – 1* disks from B to C
  - The underlined subproblems can also be solved recursively
  - The base case is when $n \leq 0$.

Sketch of the Algorithm

```
int i=1;
void hanoi(int n, char a, char b, char c)
{
    if (n > 0) {
            hanoi(n-1, a, c, b);
            printf("%c -> %c\n", a, c);
            i++;
            hanoi(n-1, b, a, c);
    }
}
main() {
    int number;
    printf("Enter number of disks : ");
    scanf("%d", &number);
    hanoi(number, 'A', 'B', 'C');
}
```