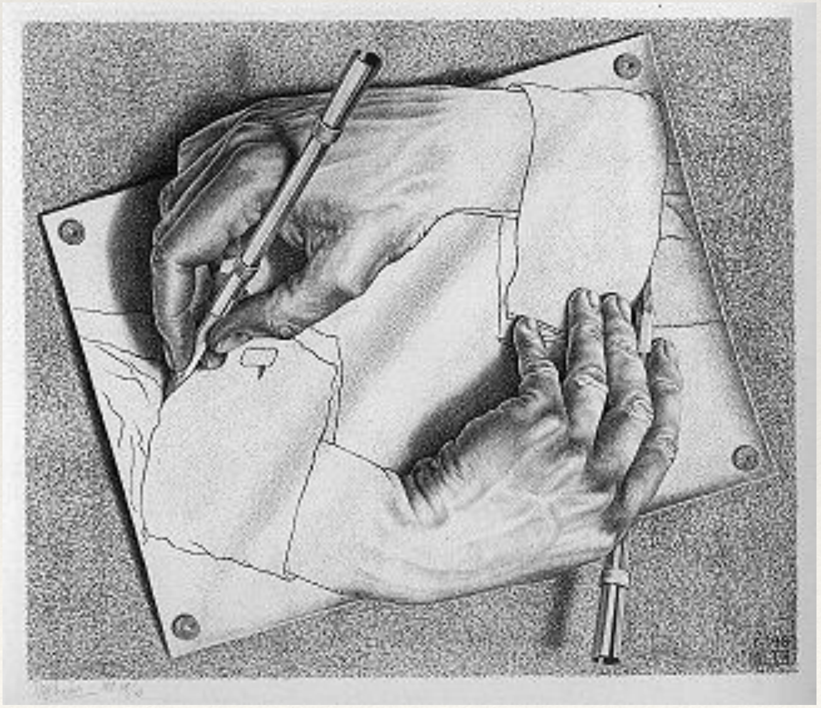
Definition of Recursion … see Recursion

Recursion

Self help

Simply Recursive …

- These notes on recursion are the basic concepts you will need to answer A4 question 2
- More detailed notes on recursion including the concept of “tail recursion as well as animated slides will be coming out next week … stay tuned



Drawing Hands

by M. C. Escher
lithograph
January 1948

Recursion

- *Definition*: When a subroutine **invokes *itself***.
    - Or when a series of subroutines eventually invoke the first subroutine again.
- The intent is to break a large problem into smaller and simpler problems. These smaller solutions are then combined to solve the larger problem.

Example of Recursion

- Factorials can be calculated recursively.

$$n! = n \times (n - 1) \times (n - 2) \times 1$$

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times \underbrace{(n - 1)!} & \text{if } n > 0 \end{cases}$$

- This part suggests recursion. The calculation is done with reference to itself.

Factorial Example

$4! = 4 \times 3!$

$= 4 \times (3 \times 2!)$

$= 4 \times (3 \times (2 \times 1!))$

$= 4 \times (3 \times (2 \times (1 \times 0!)))$

$= 4 \times (3 \times (2 \times (1 \times 1)))$

- Three levels of recursion

Recursion

- Every recursive process requires two things:

1. A **base case** that is processed *without* recursion. This requires an **ending condition** that knows when to apply the base case.

2. A method that reduces a particular case to one or more smaller cases. This requires a **recursive call**.

Recursion

- In the factorial example the base case is
  - n! = 1 if n = 0 *no further recursion is needed*
- and the method to reduce a case to a smaller one is
  - n! = n x (n -1) if n > 0

- Now, let's translate this to C code…

`factorial` Subroutine

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

Base Case

Recursive Call

The call **to** `factorial` **from** `factorial` is *recursive*.

When executed this will continue until the `if` statement is true (*i.e.* n == 0).

```
int factorial ( 4 ) {
   if …
      …
   else                                   4 * 3 * 2 * 1 * 1
      return (4 * factorial(4-1));
         if …
         else                                  3 * 2 * 1 * 1
            return (3 * factorial(3-1));
               if …
               else                                  2 * 1 * 1
                  return (2 * factorial(2-1));
                  if …
                  else                                    1 * 1
                     return (1 * factorial(1-1));
                        if ( n == 0 )
                           return (1);
```

Why Use Recursion?

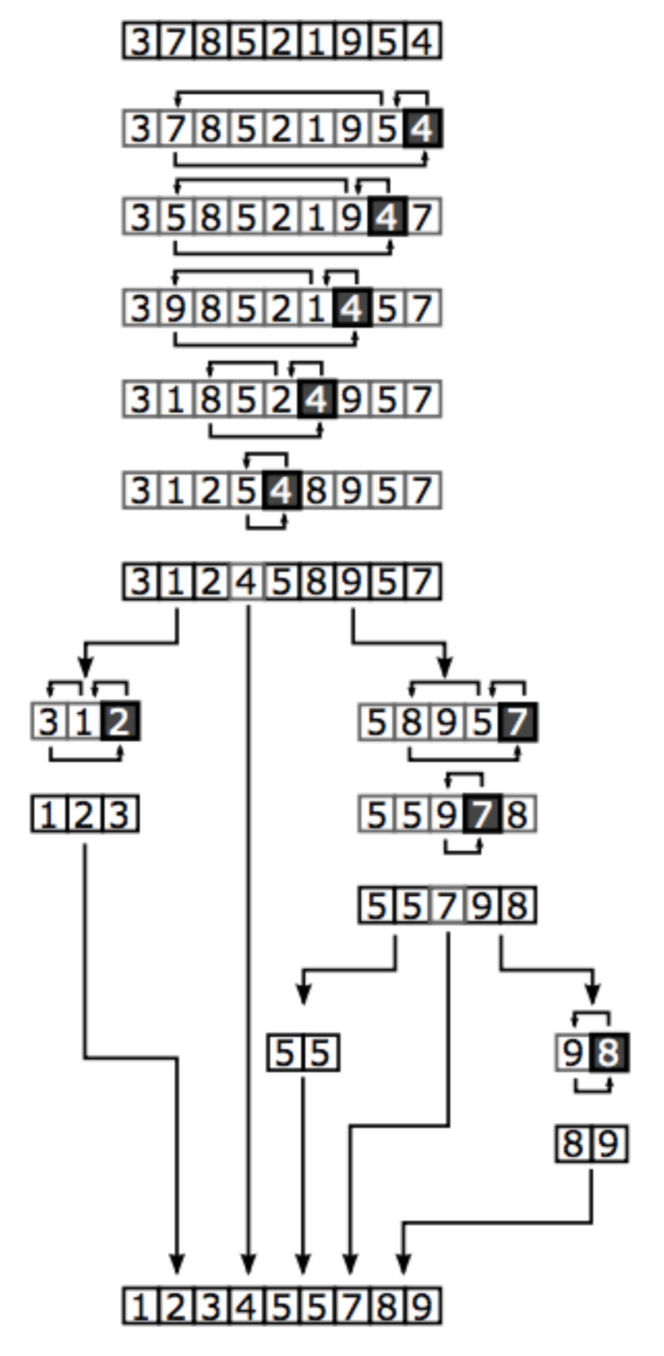
- Recursive solutions can be **concise** and *elegant*.
    - Small amount of code required.
- **But**, it requires that the programmer understand the problem and the recursive solution *very well*.
- What problems are candidates for recursion?
    - Problems that are readily subdivided can be solved recursively in a small amount of code.
    - Problems that have a long chain of partial results can benefit from recursion.

Other Recursive Algorithm Examples

- **Quicksort** is a divide and conquer algorithm.
- A large array is divided into two smaller sub-arrays called low and high.
- The sub-arrays are then recursively sorted.

Quicksort Algorithm

- Pick an element, called a **pivot**, from the array.
- Partition the array by reordering it so that all elements with values **less** than the **pivot** come before the pivot and all elements **greater** than the **pivot** come after it..
- Recursively apply the above steps to the two sub-array created by the pivot: the sub-array of **lesser** values and the sub-array of elements of **greater** values.



Select the last element to be the **pivot**.

Compare the **pivot** to other elements and put **greater** element after it.

After the **partition** is finished - *recursively* **partition** each sub-array on either side of the position of the **pivot**.

https://en.wikipedia.org/wiki/Quicksort

```c
#include <stdio.h>
#include <stdlib.h>

void quickSort ( int *arr, int low, int high ) {
   int pivot, i, temp;

   /*
    *  Select a pivot element
    *  - the last element
    */
   pivot = high;

   if ( low < high ) {
      i = low;
      while ( i < pivot ) {
         /*
          *  Go from the lower boundary until you
          *  get a number greater than the pivot index
          */
         while ( arr[i] <= arr[pivot] && i < pivot ) {
           i++;
         }
```

```
            /*
             *  If you find an element that is higher than the pivot
             *  swap with the element in front of the pivot
             */
            temp = arr[i];
            arr[i] = arr[pivot-1];
            arr[pivot-1] = temp;

            /*
             *  Swap the pivot with the element in front of it
             */
            temp = arr[pivot];
            arr[pivot] = arr[pivot-1];
            arr[pivot-1] = temp;
            pivot = pivot - 1;
        }

        /*
         *  Recursion: perform quickSort for the two sub-arrays,
         *  one to the left of pivot and one to the right of the pivot
         */
        quickSort(arr, low, pivot-1);
        quickSort(arr, pivot+1, high);
    }

}
```

```
Generating the numbers to be sorted:
1 45 89 53 33
quicksort ( 0, 4 )
1 53 89 33 45
1 89 33 53 45
1 33 89 53 45
quicksort ( 0, 1 )
1 33
quicksort ( 0, 0 )
quicksort ( 1, 1 )
quicksort ( 2, 4 )
53 45 89
45 53 89
quicksort ( 2, 2 )
quicksort ( 3, 4 )
53 89
quicksort ( 3, 3 )
quicksort ( 4, 4 )
Sorted array:
  1 33 45 53 89
( 1 45 89 53 33 )
```

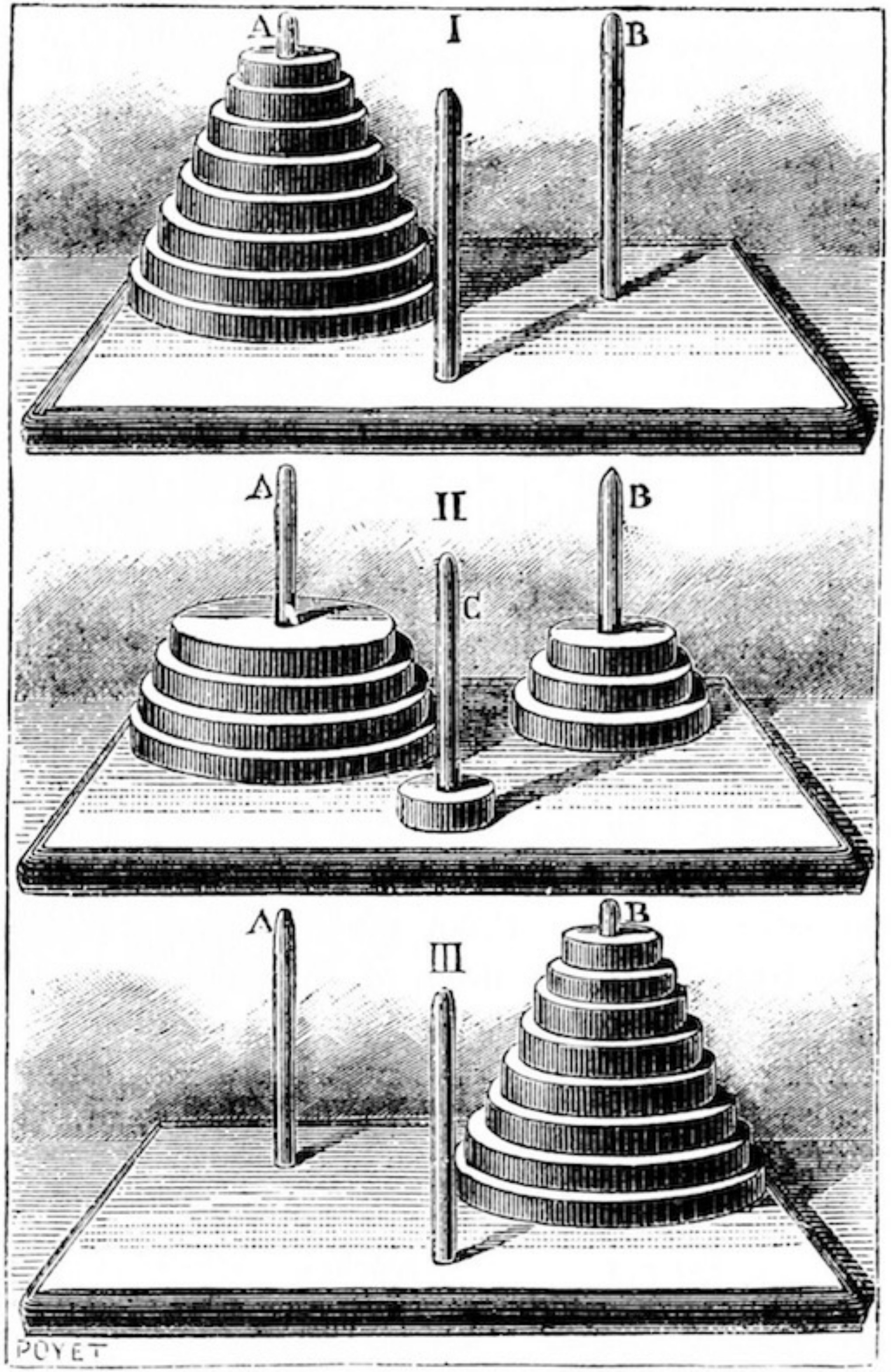
```
Generating the numbers to be sorted:
4 87 32 21 5 25 11 59 4 18
quicksort ( 0, 9 )
4 4 32 21 5 25 11 59 18 87
4 4 59 21 5 25 11 18 32 87
4 4 11 21 5 25 18 59 32 87
4 4 11 25 5 18 21 59 32 87
4 4 11 5 18 25 21 59 32 87
4 4 11 5 18 25 21 59 32 87
quicksort ( 0, 3 )
4 4 5 11
quicksort ( 0, 2 )
4 4 5
quicksort ( 0, 1 )
4 4
quicksort ( 4, 9 )
18 25 21 59 32 87
quicksort ( 4, 8 )
18 25 21 32 59
quicksort ( 4, 7 )
18 25 21 32
quicksort ( 4, 6 )
18 21 25
quicksort ( 4, 5 )
18 21
```

```
Sorted array:
  4 4 5 11 18 21 25 32 59 87
( 4 87 32 21 5 25 11 59 4 18 )
```

Towers of Hanoi

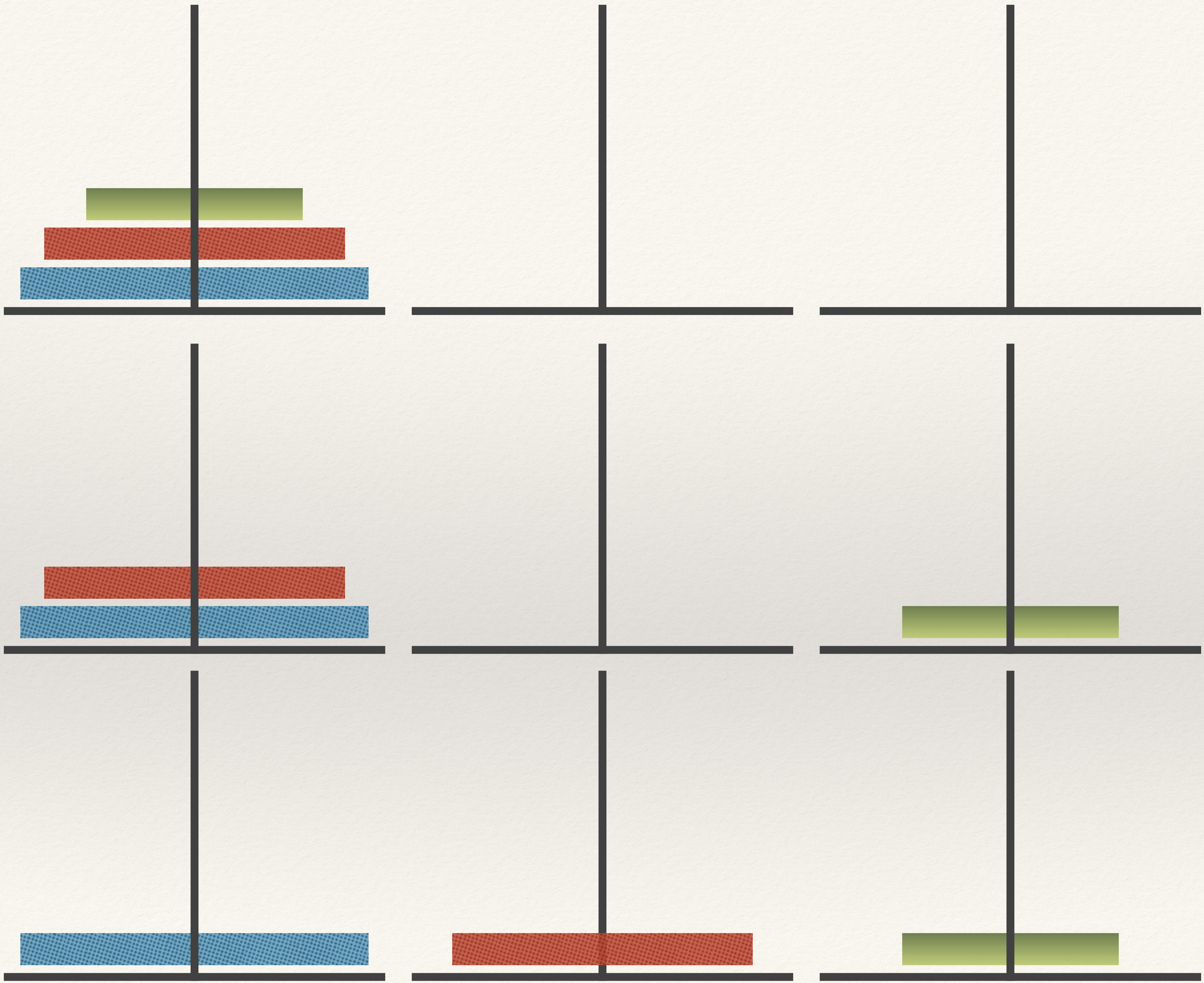
- Towers of Hanoi is a puzzle that consists of three posts, and a set of disks of different sizes that can be stacked on the posts.
- At the start, all the disks are stacked on one post by size (largest on bottom).
- The challenge is to transfer the stack from the first post to the third, using the second post for temporary storage.
- Only one disk can be moved at a time, and a larger disk can never be put on top of a smaller disk.

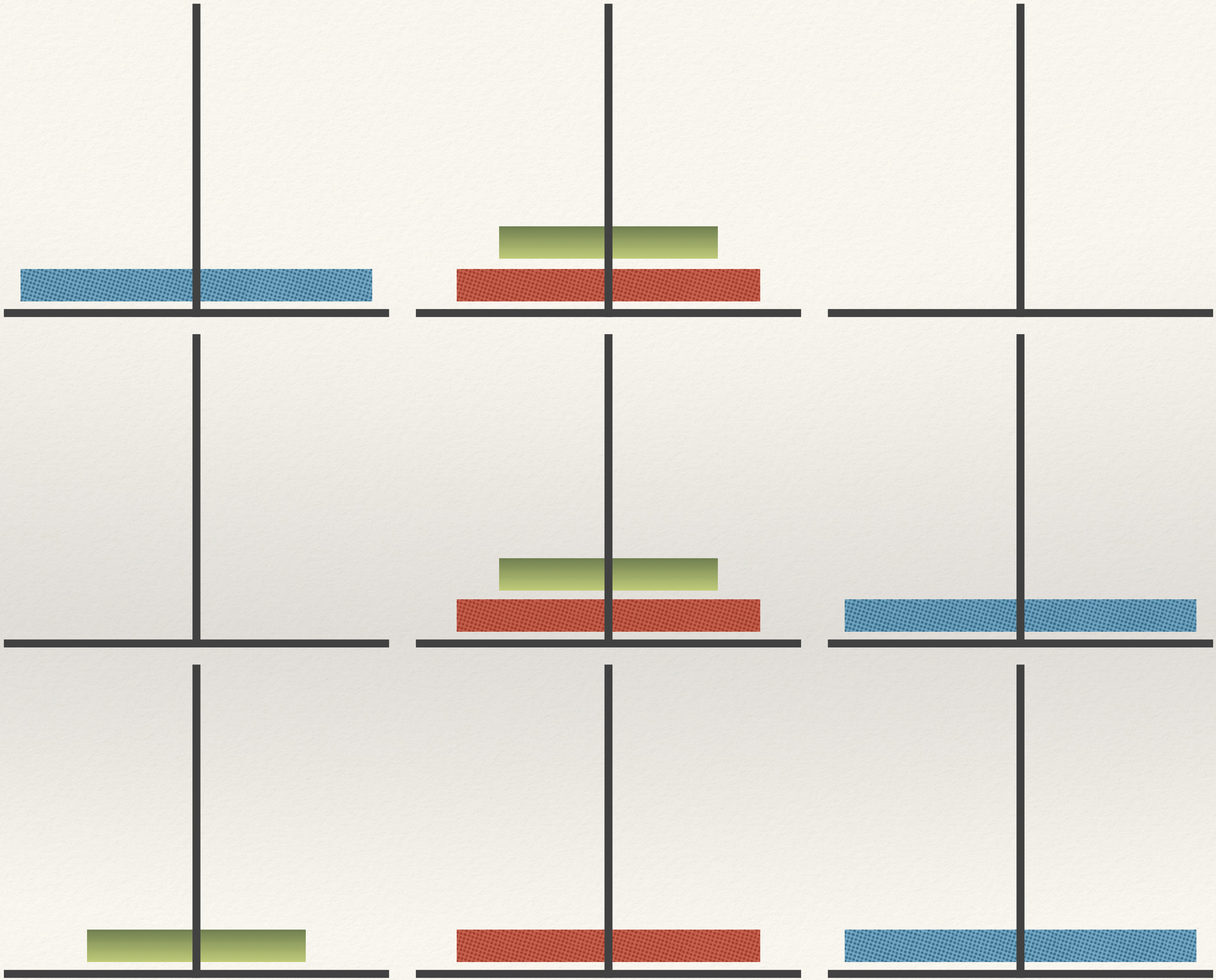
https://sleeplessafternoon.wordpress.com/2013/03/26/examples-of-recursion-the-good-the-bad-and-the-silly/

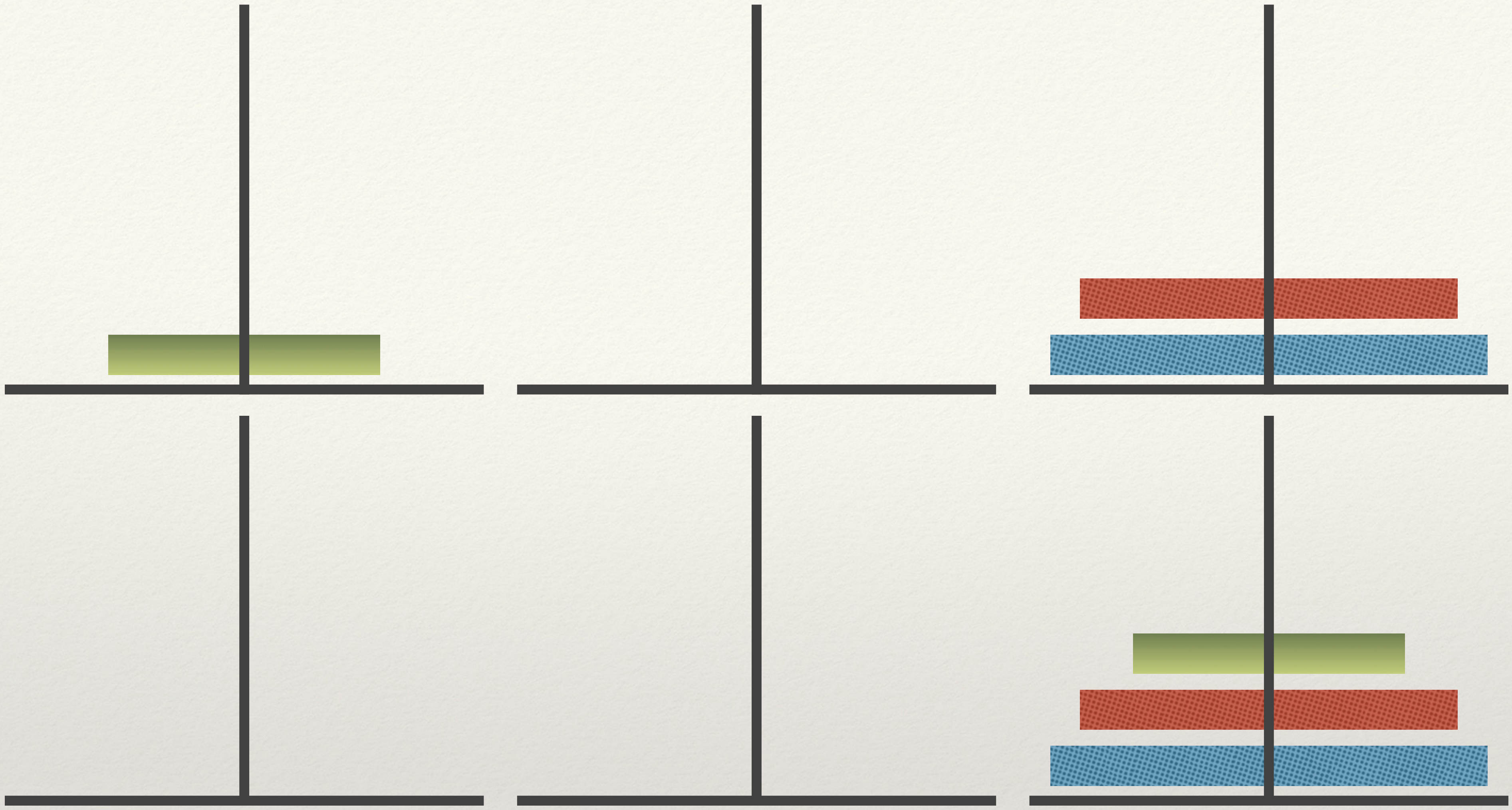

A
I
B
A
II
B
C
A
III
B
POYET

The Recursive Algorithm

- Step 1. Move the stack of all but the largest disk from the first to the second post.
- Step 2. Then move the largest disk from the first post to the third post.
- Step 3. Then move the stack from the second to the third post.
- Steps 1 and 3 are *recursive*.







```
void hanoi (int height, int one, int two, int three)
{
    if (height <= 0) {
      return;
    }

    hanoi ( height-1, one, three, two );

    printf ("Move disk %d from %d to %d\n", height, one, three);

    hanoi ( height-1, two, one, three);
}
```

```
Solution to Towers of Hanoi with 3 disks:   hanoi ( 3, 1, 2, 3 )
Move disk 1 from 1 to 3
Move disk 2 from 1 to 2
Move disk 1 from 3 to 2
Move disk 3 from 1 to 3
Move disk 1 from 2 to 1
Move disk 2 from 2 to 3
Move disk 1 from 1 to 3
```

But when is recursion not the answer?

- The Fibonacci sequence can be defined as follows:

  $F_0 = 0$

  $F_1 = 1$

  $F_n = F_{n-1} + F_{n-2}$ for $n >= 2$

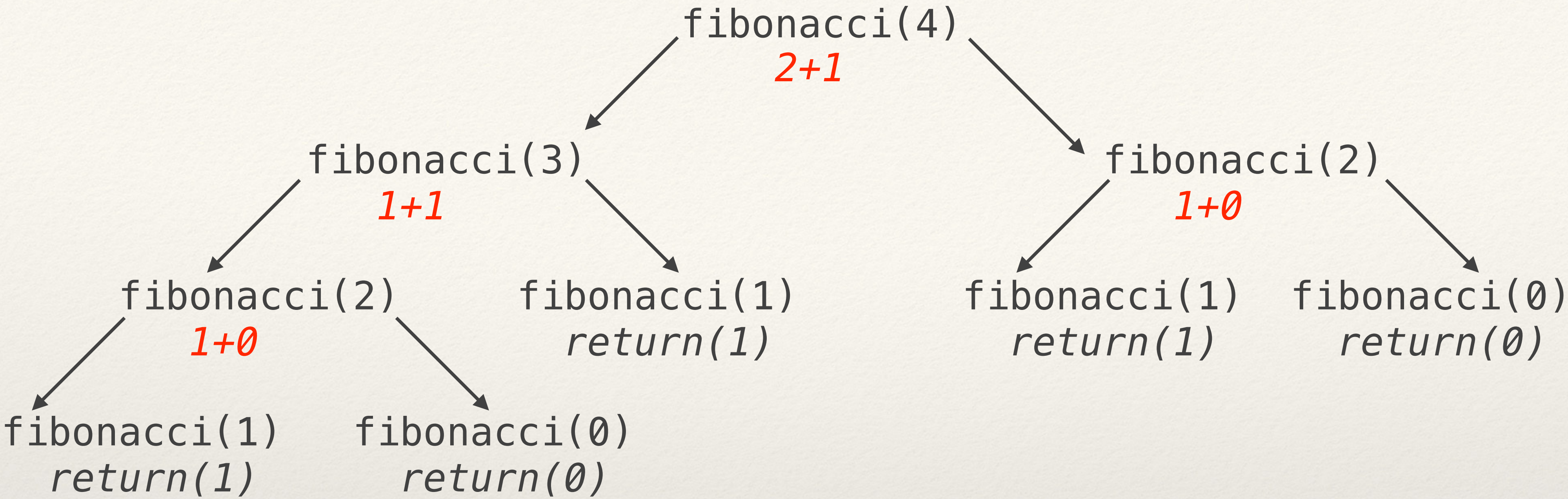
- The sequence looks like:

  0, 1, 1, 2, 3, 5, 8, 13, 21, …

  F_0 F_1 F_2 F_3 F_4 F_5 F_6 F_7 F_8

Recursive Fibonacci

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




```
int main ( int argc, char *argv[] )
{
   int i, n;
   n = atoi ( argv[1] );
   for ( i=0; i<=n; i++ ) {
      printf ( "%d ", fibonacci(i) );
   }
   printf ( "\n" );
   return (0);
}
```

```
$ ./fib 8
0 1 1 2 3 5 8 13 21
```

But…

- The recursive solution for the Fibonacci sequence is very inefficient.
  - It requires a large number of function calls.
- The problem is that we have stated the problem incorrectly. We are not looking to find the *n*th term but are instead looking at a way to construct a sequence.
- This is better done as an iterative (loop-based) function.

```c
int fibonacci ( int n ) {
   int i;
   int oneBack, twoBack, current;

   if ( n <= 0 ) {
      return (0);
   } else if ( n == 1 ) {
      return (1);
   } else {
      twoBack = 0;
      oneBack = 1;
      for ( i=2; i<=n; i++ ) {
         current = twoBack + oneBack;
         twoBack = oneBack;
         oneBack = current;
      }
      return (current);
   }
}
```