ECE 220 Computer Systems & Programming

Lecture 14 – Recursion

March 5, 2019

ECE ILLINOIS

ILLINOIS

Recursion

A **recursive function** is one that solves its task by **calling itself** on smaller pieces of data.

- Similar to recurrence function in mathematics.
- Like iteration -- can be used interchangeably; sometimes recursion results in a simpler solution.
- Must have at least 1 **base case** (terminal case) that ends the recursive process.

Example: Running sum ($\sum_1^n i$)

Mathematical Definition:

RunningSum(1) = 1

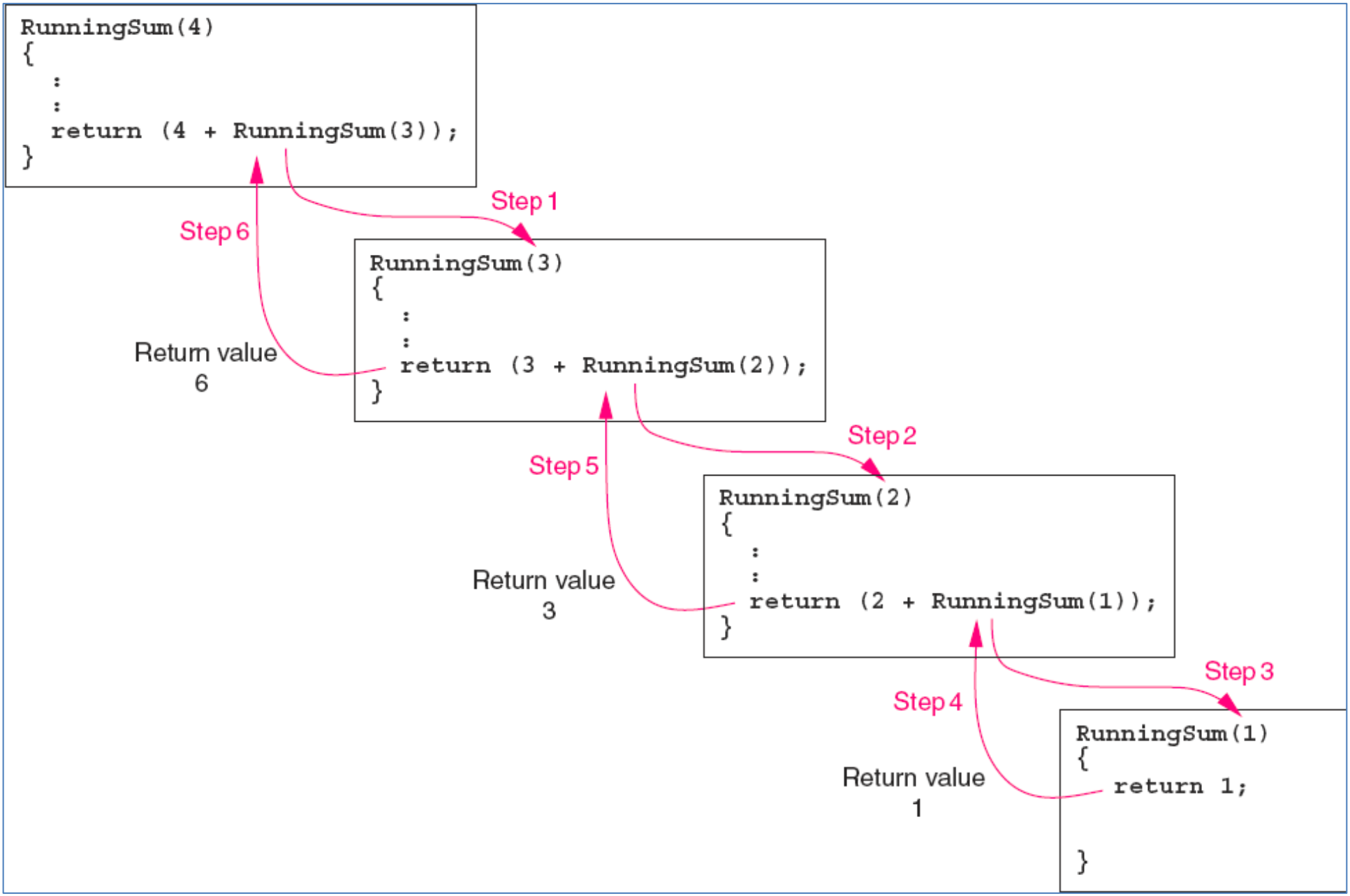
RunningSum(n) =

n + RunningSum(n-1)

Recursive Function:

```
int RunningSum(int n) {
  if (n == 1)
    return 1;
  else
    return n + RunningSum(n-1);
}
```

Running sum Recursion



Running sum (code)

```c
#include <stdio.h>
int run_sum(int n);
//assume n is non-negative
int run_sum(int n)
{
    if(n == 1)
     return 1;
    else
        return n+run_sum(n-1);
}

int main()
{
    int n=4;
    printf("run_sum(%d)=%d \n",n,run_sum(n));

    return 0;
}
```

```
run_sum(4)=10
```

Factorial

$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$

$$n! = \begin{cases} n \cdot (n-1)! & , n > 0 \\ 1 & , n = 0 \end{cases}$$

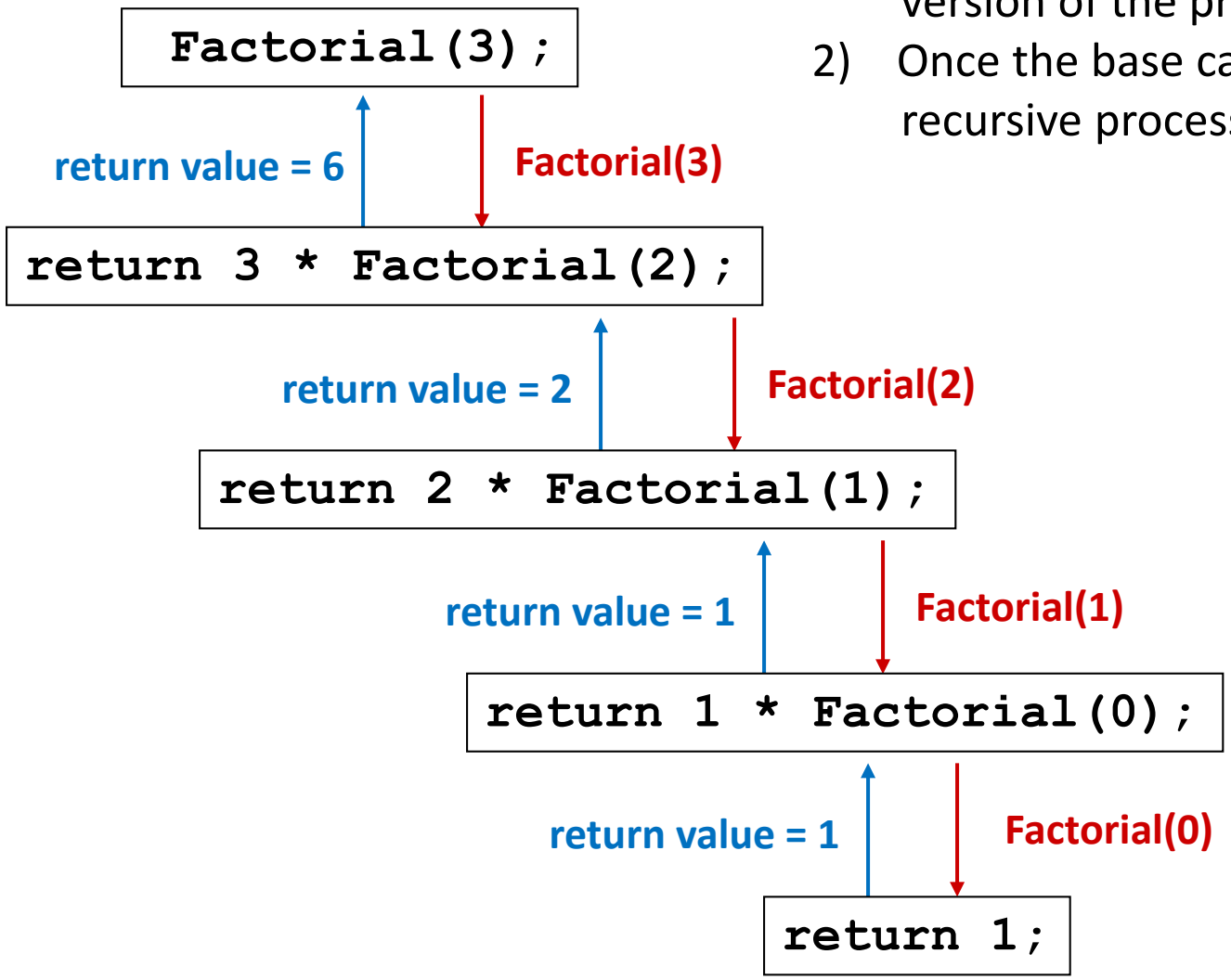
```
int Factorial(int n)
{

      if
       Return ….


 else

 return

}
```

```
1  #include <stdio.h>
2  int Factorial(int n);
3  //assume n is non-negative
4  int Factorial(int n)
5  {
6      if(n == 0)
7          return 1;
8      else
9          return n*Factorial(n-1);
10 }
11
12 int main()
13 {
14     int n=3;
15     printf("Factorial(%d)=%d \n",n,Factorial(n));
16
17     return 0;
18 }
```

Executing Factorial

Observation:

1) Each invocation solves a smaller version of the problem;
2) Once the base case is reached, recursive process stops.



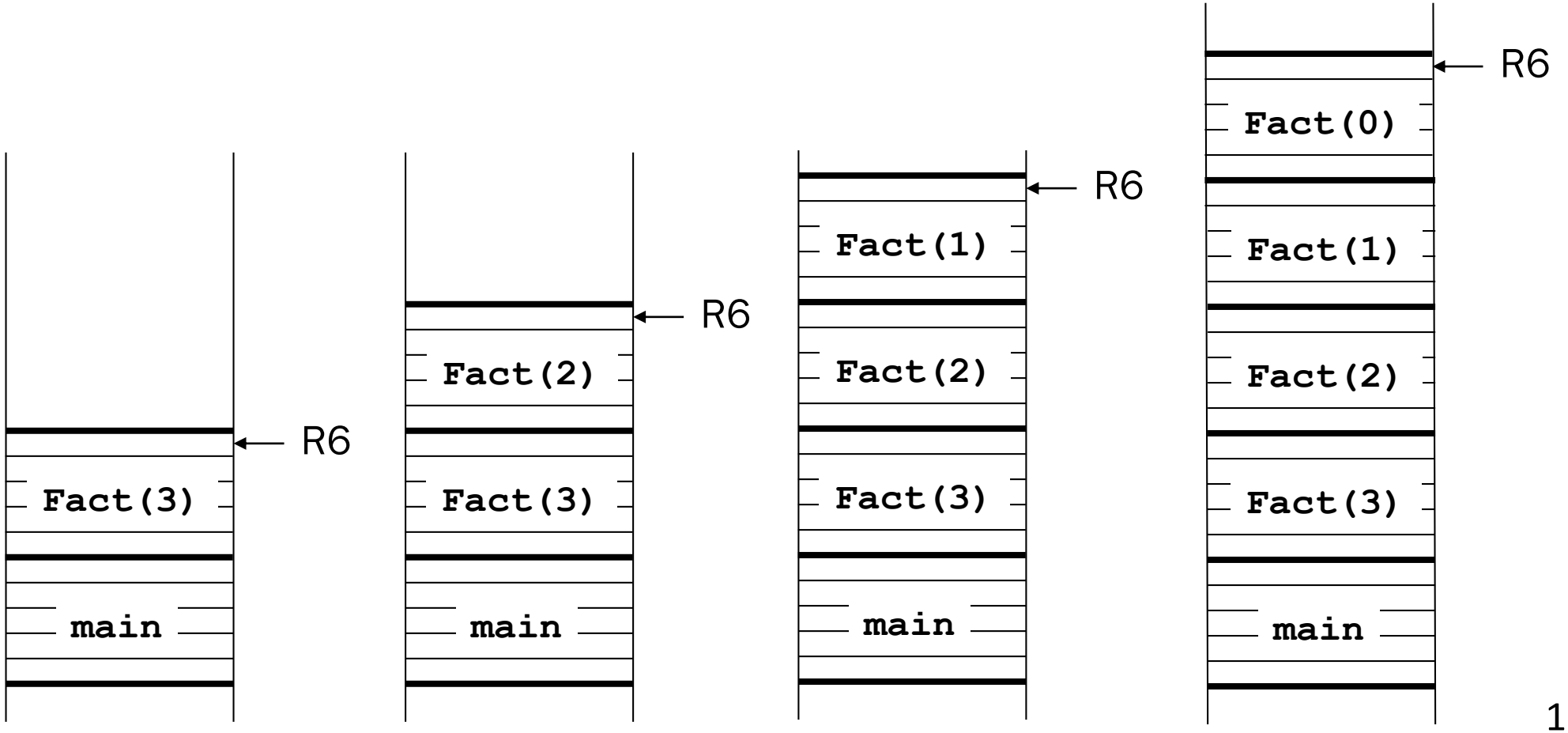
Run-Time Stack During Execution of Factorial

main calls Factorial(3)

Factorial(3) calls Factorial(2)

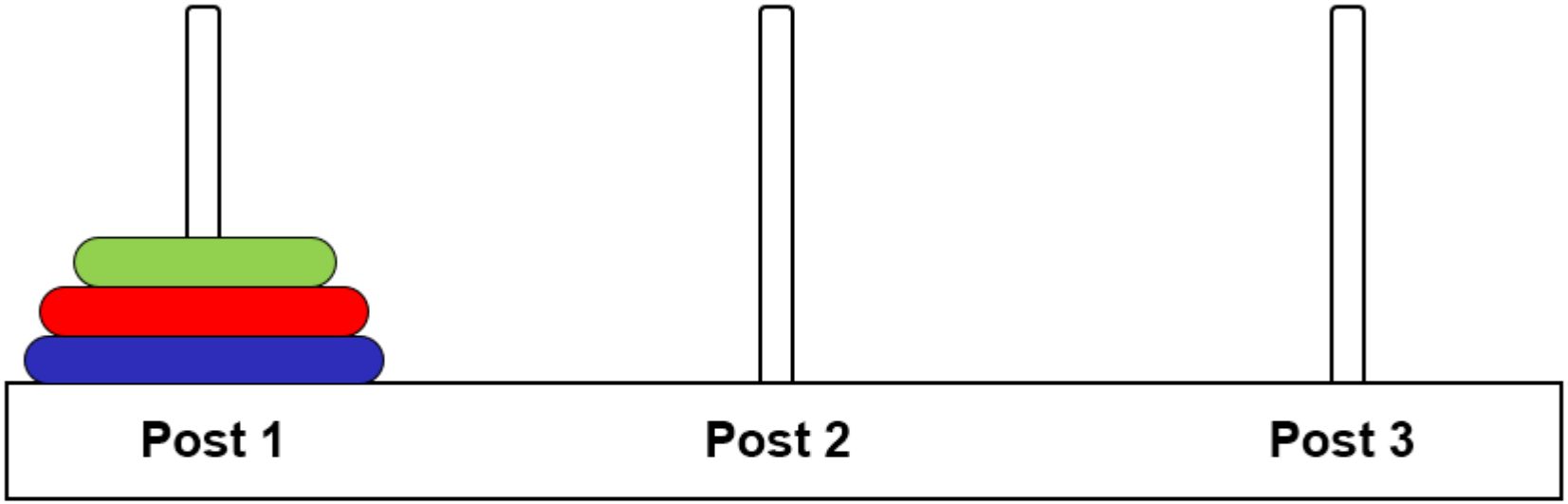
Factorial(2) calls Factorial(1)

Factorial(1) calls Factorial(0)



Towers of Hanoi Problem

Task: Move all disks from current post to another post.



Rules:

(1) Can only move one disk at a time.

(2) A larger disk can never be placed on top of a smaller disk.

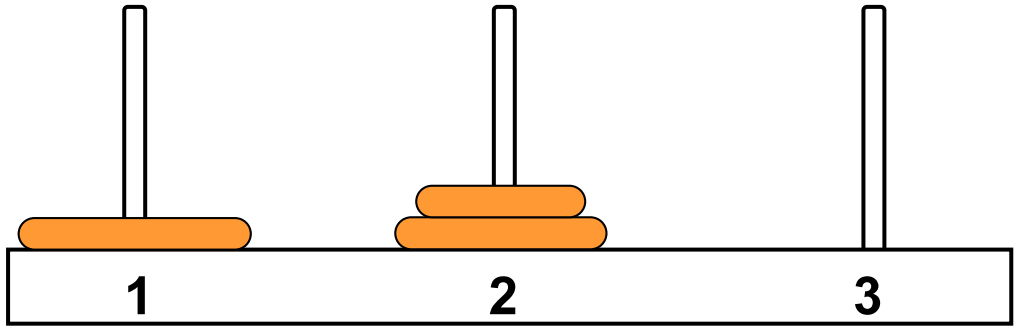
(3) May use third post for temporary storage.

Tower of Hanoi (Animation)

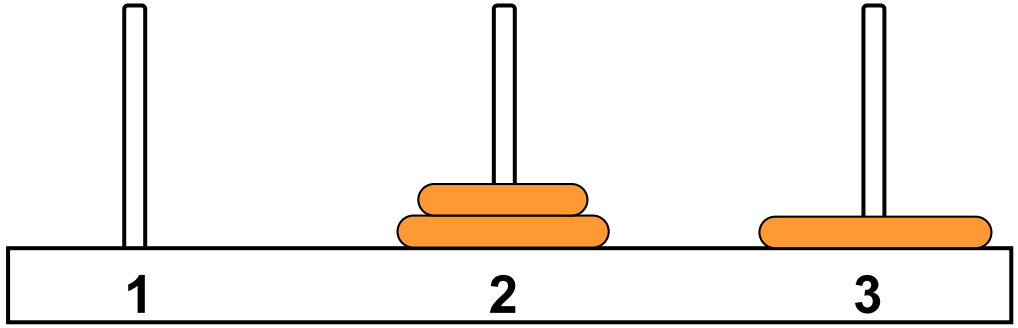
Task Decomposition

Suppose disks start on Post 1, and target is Post 3.

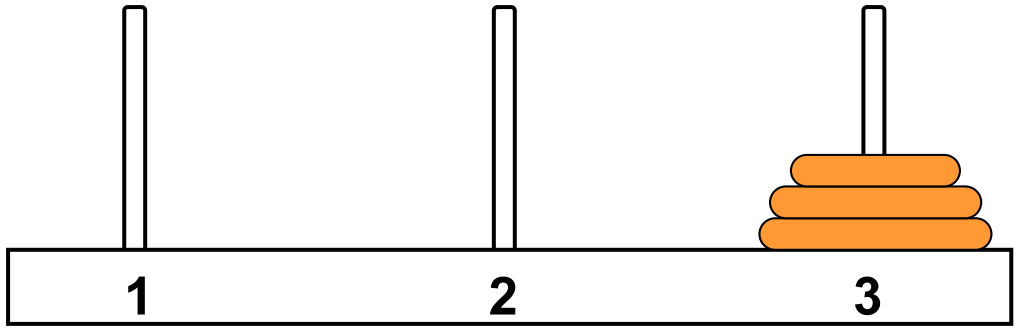
1. **Move top n-1 disks to Post 2.**



2. **Move largest disk to Post 3.**



3. **Move n-1 disks from Post 2 to Post 3.**



Task Decomposition (cont.)

Task 1 is really the same problem,
with fewer disks and a different target post.

- **"Move n-1 disks from Post 1 to Post 2."**

And Task 3 is also the same problem,
with fewer disks and different starting and target posts.

- **"Move n-1 disks from Post 2 to Post 3."**

So this is a recursive algorithm.

- **The terminal case is moving the smallest disk -- can move directly without using third post.**
- **Number disks from 1 (smallest) to n (largest).**

Towers of Hanoi: Pseudocode

```
MoveDisk(diskNumber, startPost, endPost, midPost)
{
  if (diskNumber > 1) {
    /* Move top n-1 disks to mid post */
    MoveDisk(diskNumber-1, startPost, midPost, endPost);

    printf("Move disk number %d from %d to %d.\n",
           diskNumber, startPost, endPost);

    /* Move n-1 disks from mid post to end post */
    MoveDisk(diskNumber-1, midPost, endPost, startPost);
  }
  else
    printf("Move disk number 1 from %d to %d.\n",
           startPost, endPost);
}
```

Fibonacci Number

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

$$\begin{cases} F_n = F_{n-1} + F_{n-2} \\ F_0 = 0 \\ F_1 = 1 \end{cases}$$

```
int Fibonacci(int n)
{

}
```

Fibonacci with Look-up Table

```
int table[100];
/* each element will be initialized to -1 in main */

int fibonacci(int n){
   /*if fibonacci(n) has been calculated, return it*/


   /*otherwise, perform the calculation, save it to table
     and then return it*/




}
```

Binary Search

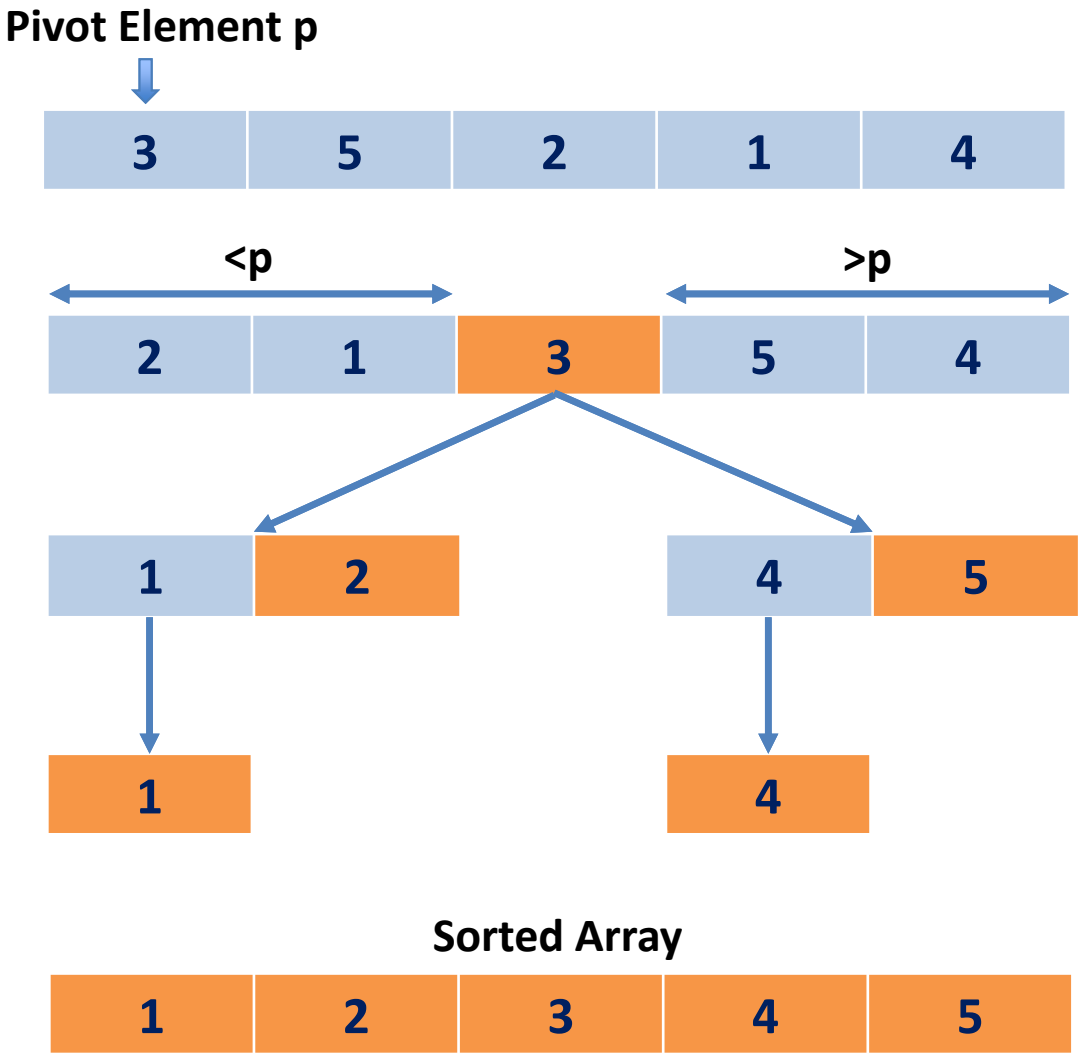
/*This function takes four arguments: pointer to a sorted array, the search item, the start index and the end index of the array. If the search item is found, the function returns its index in the array. Otherwise, it returns -1.*/

```
int binary(int array[], int item, int start, int end)
{

}
```

Quick Sort: also called divide-and-conquer

1) pick a pivot and partition array into 2 subarrays;

2) then sort subarrays using the same method.



Quick Sort

```
/* Assume partition() function is given and it returns the index of
the pivot after partitioning. */
int partition(int array[], int start, int end);

/* This function takes 3 arguments: a pointer to the array, the
start index of the array and the end index of the array. */
void quicksort(int array[], int start, int end){

}
```