# 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 ( $\sum_1^n i$ )

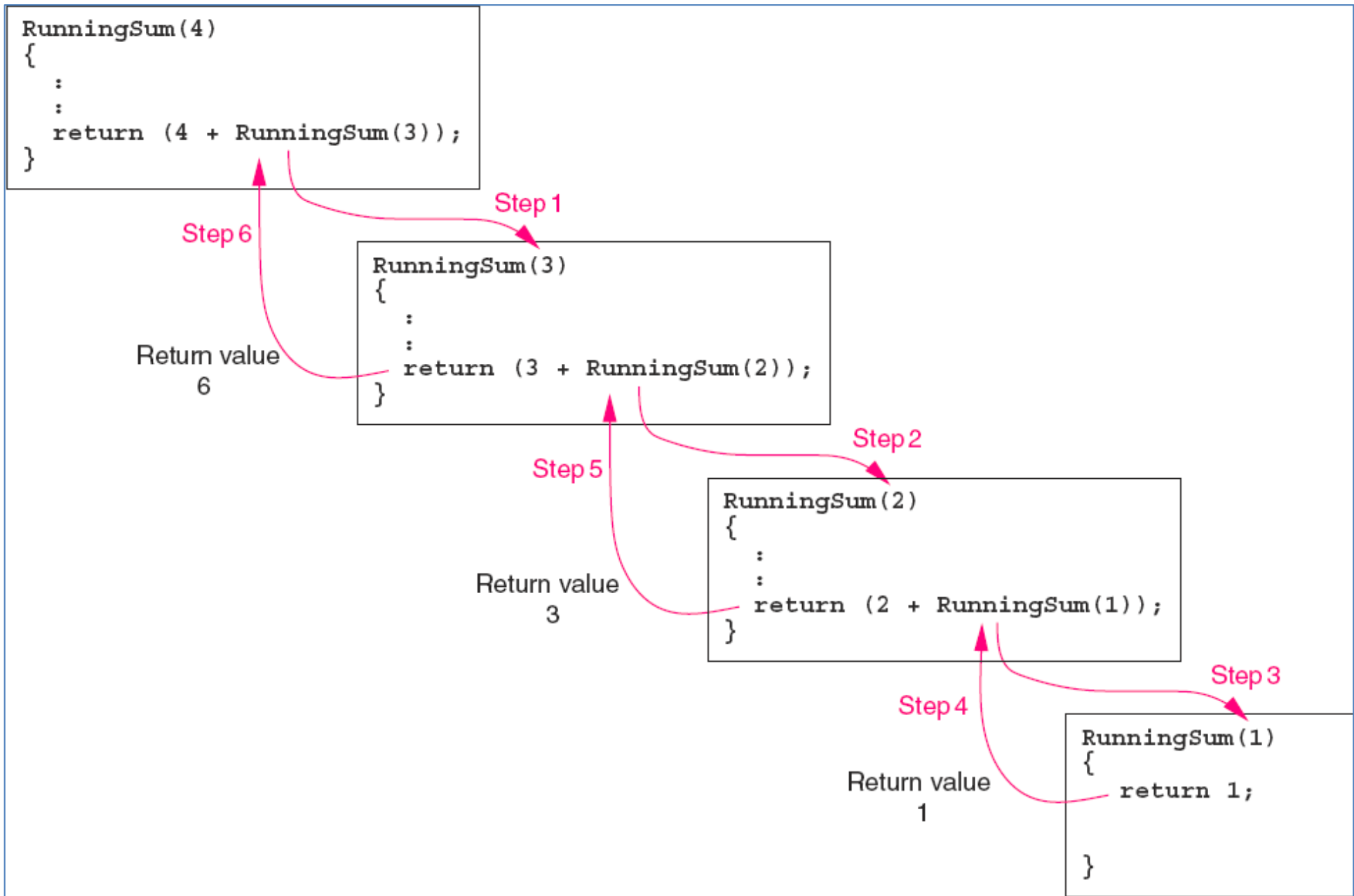**Mathematical Definition:**
RunningSum(1) = 1
RunningSum(n) =
    n + RunningSum(n-1)

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

# Running sum Recursion



```
RunningSum(4)
{
    :
    :
  return (4 + RunningSum(3));
}
```

Step 1

Step 6

Return value
6

```
RunningSum(3)
{
    :
    :
  return (3 + RunningSum(2));
}
```

Step 2

Step 5

Return value
3

```
RunningSum(2)
{
    :
    :
  return (2 + RunningSum(1));
}
```

Step 3

Step 4

Return value
1

```
RunningSum(1)
{
  return 1;
}
```

ILLINOIS

# Running sum (code)

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

```
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


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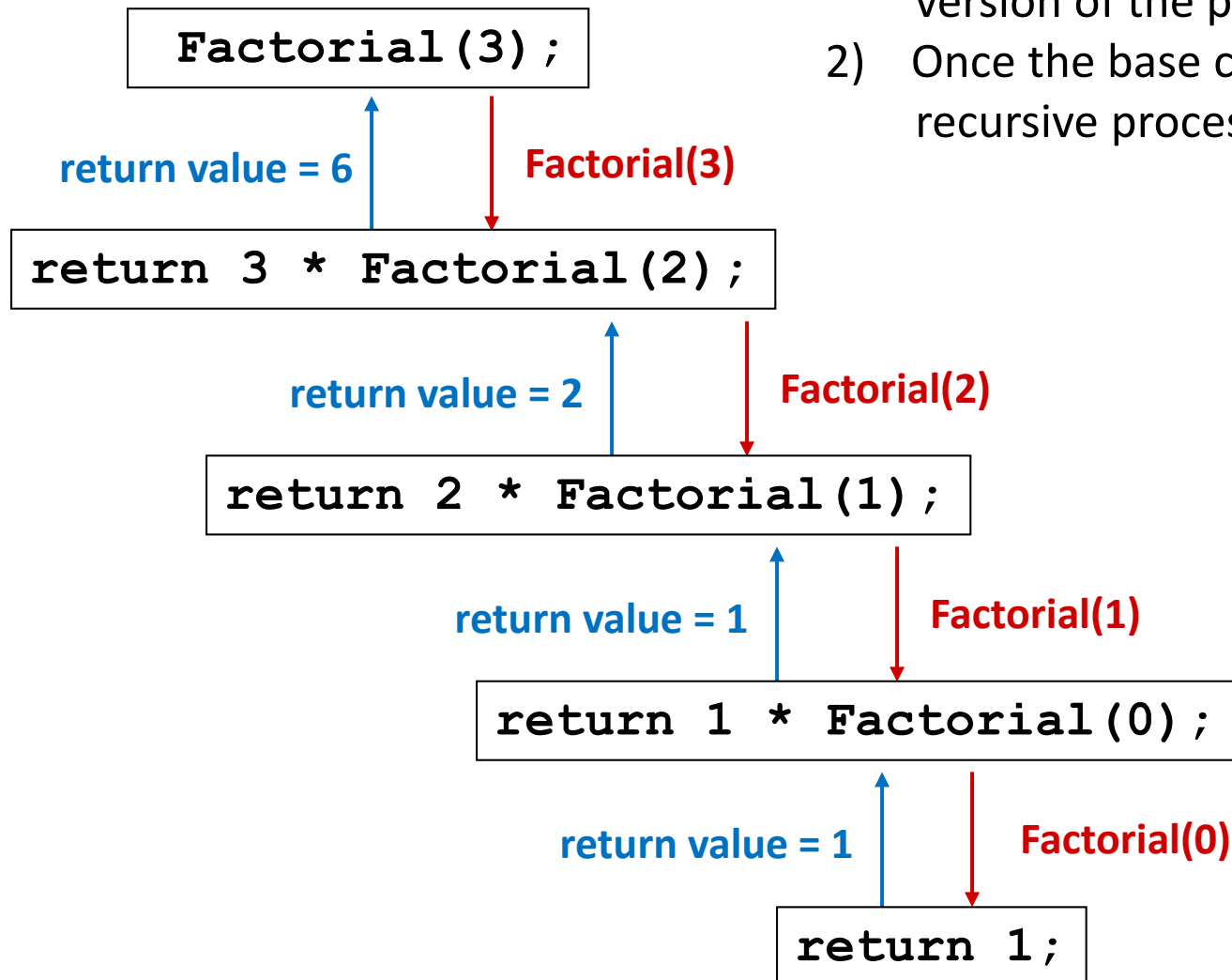
```
1   #include <stdio.h>
2   int Factorial(int n);
3   //assume n is non-negative
4   int Factorial(int n)
5   {
6       if(n == 0)
7           return 1;
8       else
9           return n*Factorial(n-1);
10  }
11
12  int main()
13  {
14      int n=3;
15      printf("Factorial(%d)=%d \n",n,Factorial(n));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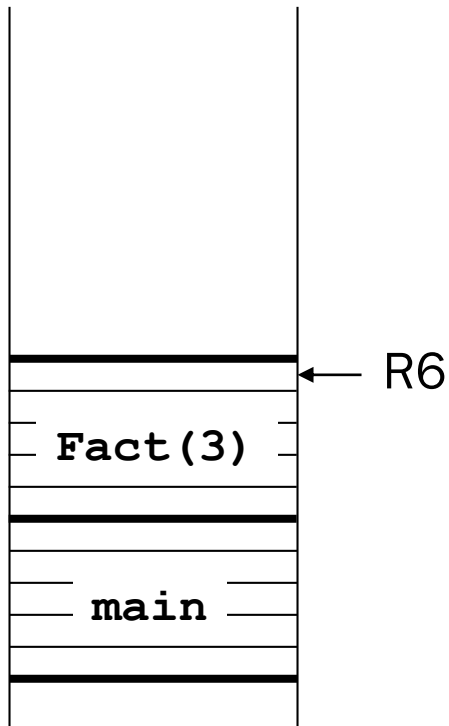
# Executing Factorial



**Observation:**
1) Each invocation solves a smaller version of the problem;
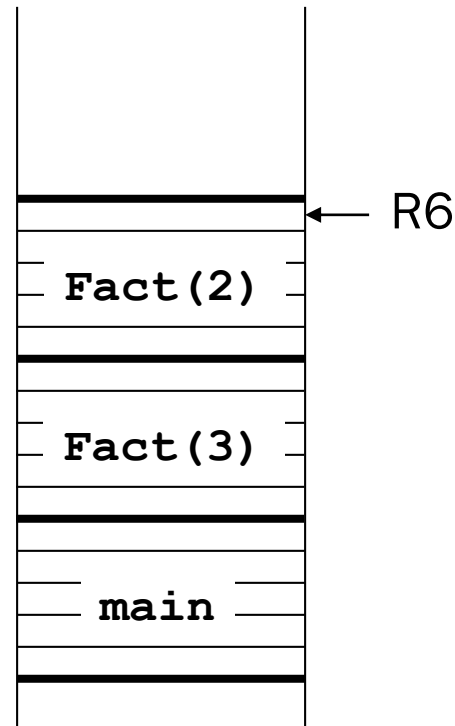2) Once the base case is reached, recursive process stops.
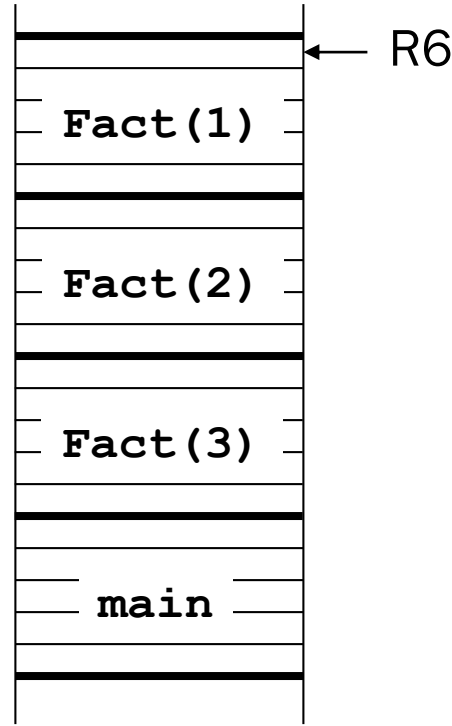
# Run-Time Stack During Execution of Factorial
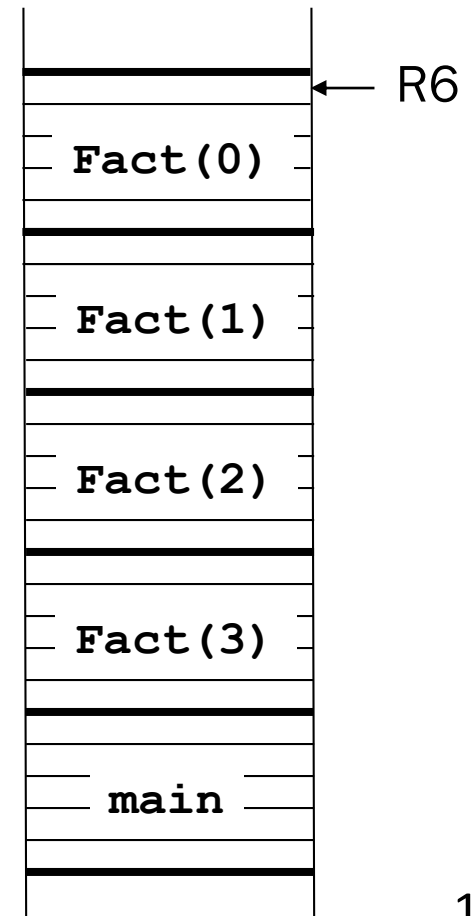
**main calls**
**Factorial(3)**



**Factorial(3) calls**
**Factorial(2)**



**Factorial(2) calls**
**Factorial(1)**



**Factorial(1) calls**
**Factorial(0)**



12

```
 1  .ORIG x3000
 2  ; push argument
 3      LD R6, STACK_TOP
 4      AND R0,R0,#0
 5      ADD R0,R0,#3
 6      STR R0,R6,#0
 7  ; call subroutine
 8      JSR FACTORIAL
 9  ; pop return value from run-time stack (to R0)
10      LDR R0,R6,#0
11      ADD R6, R6, #-1
12      HALT
13
14  FACTORIAL:
15  ; push callee's bookkeeping info onto the run-time stack
16  ; allocate space in the run-time stack for return value
17      ADD R6, R6, #-1
18  ; store caller's return address and frame pointer
19      ADD R6, R6, #-1
20      STR R7, R6, #0
21      ADD R6, R6, #-1
22      STR R5, R6, #0
23  ; allocate memory for local variable fn
24      ADD R6, R6, #-1
25      ADD R5, R6, 0
26  ; if (n>0)
27      LDR R1, R5, #4
28      ADD R2, R1, #-1
29      BRn ELSE
```

```
30  ;  compute fn = n * factorial(n-1)
31  ;  caller-built stack for factorial(n-1) function call
32  ;  push n-1 onto run-time stack
33      ADD R6, R6, #-1
34      STR R2, R6, #0
35  ;  call factorial subroutine
36      JSR FACTORIAL
37  ;  pop return value from run-time stack (to R0)
38      LDR R0, R6, #0
39      ADD R6, R6, #1
40  ;  pop function argument from the run-time stack
41      ADD R6, R6, #1
42  ;  multiply n by the return value (already in R0)
43      LDR R1, R5, #4
44      ;MUL R0, R0, R1 ; R0 <- n * factorial(n-1)
45      ST R7, SAVE_R7
46      JSR MULT
47      LD R7, SAVE_R7
48      ADD R0, R2, #0
49  ;  store result in memory for fn
50      STR R0, R5, #0
51  ;  done with this branch
52      BRnzp RETURN
```

```
53  ELSE:
54  ; store value of 1 in memory for fn
55       AND R2, R2, #0
56       ADD R2, R2, #1
57       STR R2, R5, #0
58  ; tear down the run-time stack and return
59  RETURN:
60  ; write return value to the return entry
61       LDR R0, R5, #0
62       STR R0, R5, #3
63  ; pop local variable(s) from the run-time stack
64       ADD R6, R6, #1
65  ; restore caller's frame pointer and return address
66       LDR R5, R6, #0
67       ADD R6, R6, #1
68       LDR R7, R6, #0
69       ADD R6, R6, #1
70  ; return control to the caller function
71       RET
```

```
72  ; multiply subroutine
73  ; input should be in R0 and R1
74  ; output should be in R2
75  MULT
76      ; save R3
77      ST R3, SAVE_R3
78      ; reset R2 and initialize R3
79      AND R2, R2, #0
80      ADD R3, R0, #0
81      ; perform multiplication
82      MULT_LOOP
83      ADD R3, R3, #-1
84      BRn MULT_DONE
85      ADD R2, R2, R1
86      BRnzp MULT_LOOP
87      MULT_DONE
88      ; restore R0
89      LD R3, SAVE_R3
90      RET
91
92  SAVE_R3              .BLKW #1
93  SAVE_R7              .BLKW #1
94  STACK_TOP            .FILL x4000
95  .END
```

# Towers of Hanoi Problem

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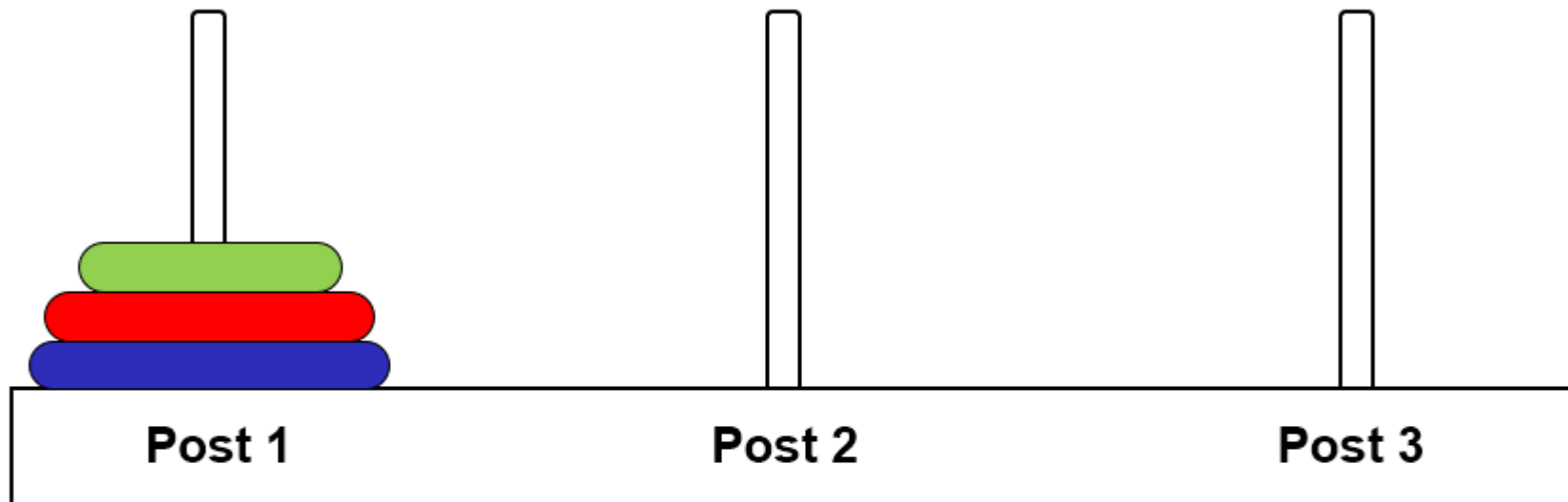


**Rules:**

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

(2) A larger disk can never be placed on top of a smaller disk.
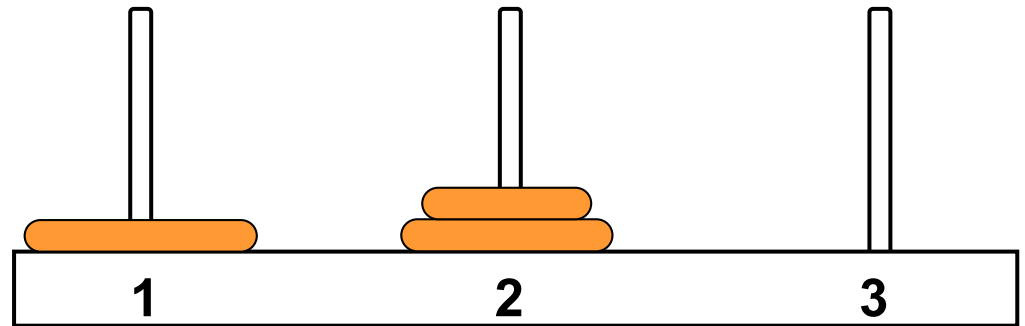
(3) May use third post for temporary storage.
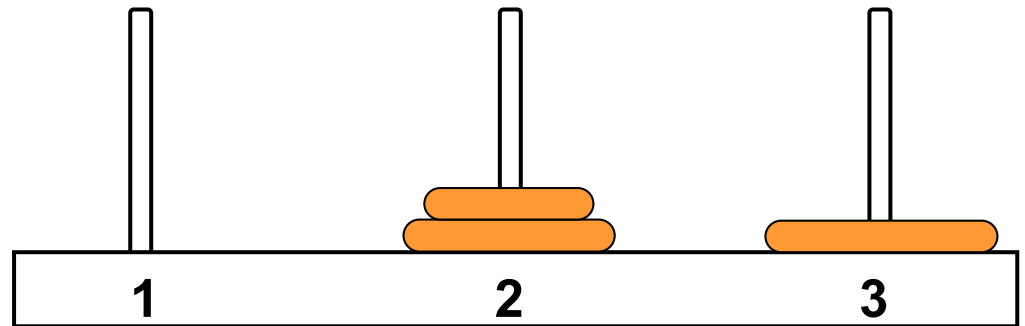
# Tower of Hanoi (Animation)

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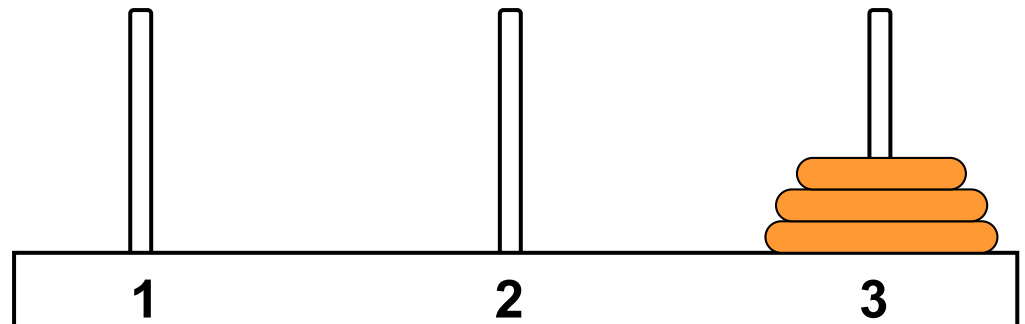
## Suppose disks start on Post 1, and target is Post 3.

**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.**

17-14

# 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)
{



}
```

```c
 4
 5    int binary(int array[], int num, int start, int end);
 6
 7    int main()
 8    {
 9        int index;
10        int array[]={1,3,5,7,9,11,13,15,17};
11
12        index = binary(array, 13, 0, LENGTH-1);
13        printf("the value of the matched index: %d\n",index);
14        return 0;
15    }
16
17    int binary(int array[], int item, int start, int end)
18    {
19        if(start>end)
20            return -1;
21
22        int middle = (start+end)/2;
23
24        if(item == array[middle]){
25            return middle;}
26        else if(item > array[middle])
27            return binary(array, item, middle+1, end);
28        else //item < array[middle]
29            return binary(array, item, start, middle-1);
30
31    }
```

**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.
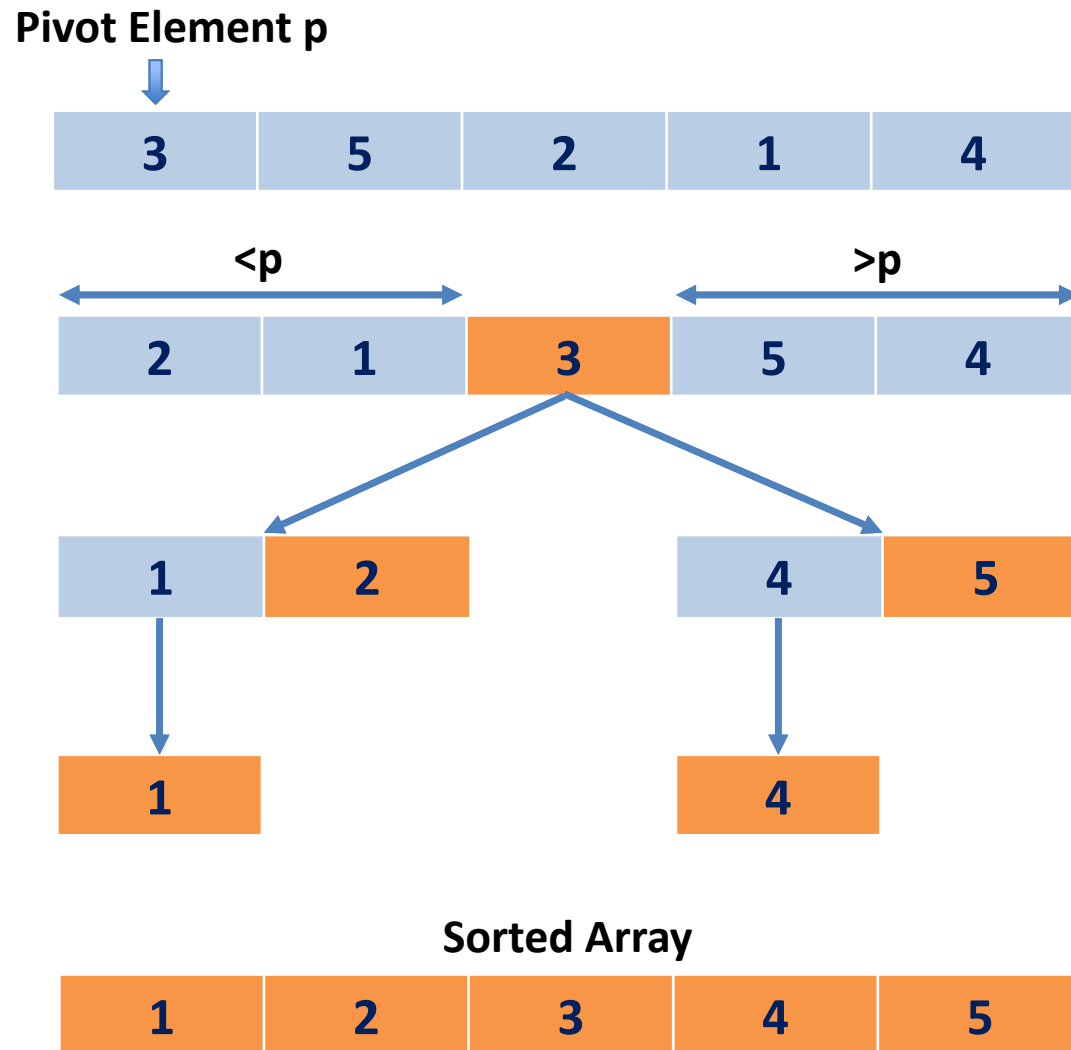
```
/*Quicksort is a divide and conquer algorithm.
The steps are:
1) Pick an element from the array,
this element is called as pivot element.
2) Divide the unsorted array of elements in
two arrays with values less than the pivot come
in the first sub array, while all elements with
values greater than the pivot come in the second
sub-array (equal values can go either way).
This step is called the partition operation.
3) Recursively repeat the step 2
(until the sub-arrays are sorted) to the
sub-array of elements with smaller values
and separately to the sub-array of elements
with greater values.
*/
```

**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.

**Pivot Element p**

| 3 | 5 | 2 | 1 | 4 |
|---|---|---|---|---|

<p                                    >p

| 2 | 1 | **3** | 5 | 4 |
|---|---|---|---|---|

| 1 | **2** | | 4 | **5** |
|---|---|---|---|---|

| **1** | | | **4** | |
|---|---|---|---|---|

**Sorted Array**

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|

## Quick Sort

```c
#include<stdio.h>
void quicksort(int number[],int first,int last){
    int i, j, pivot, temp;

    if(first<last){
        pivot=first;
        i=first;
        j=last;
// Partitioning (using while loop)
        while(i<j){




        }
// Move the Pivot



// Recursive Calling
        quicksort(number,first,j-1);
        quicksort(number,j+1,last);
    }
}
```

9