ECE 220 Computer Systems & Programming

Lecture 14 – Recursion

October 15, 2019

ECE ILLINOIS

ILLINOIS

Recursion

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

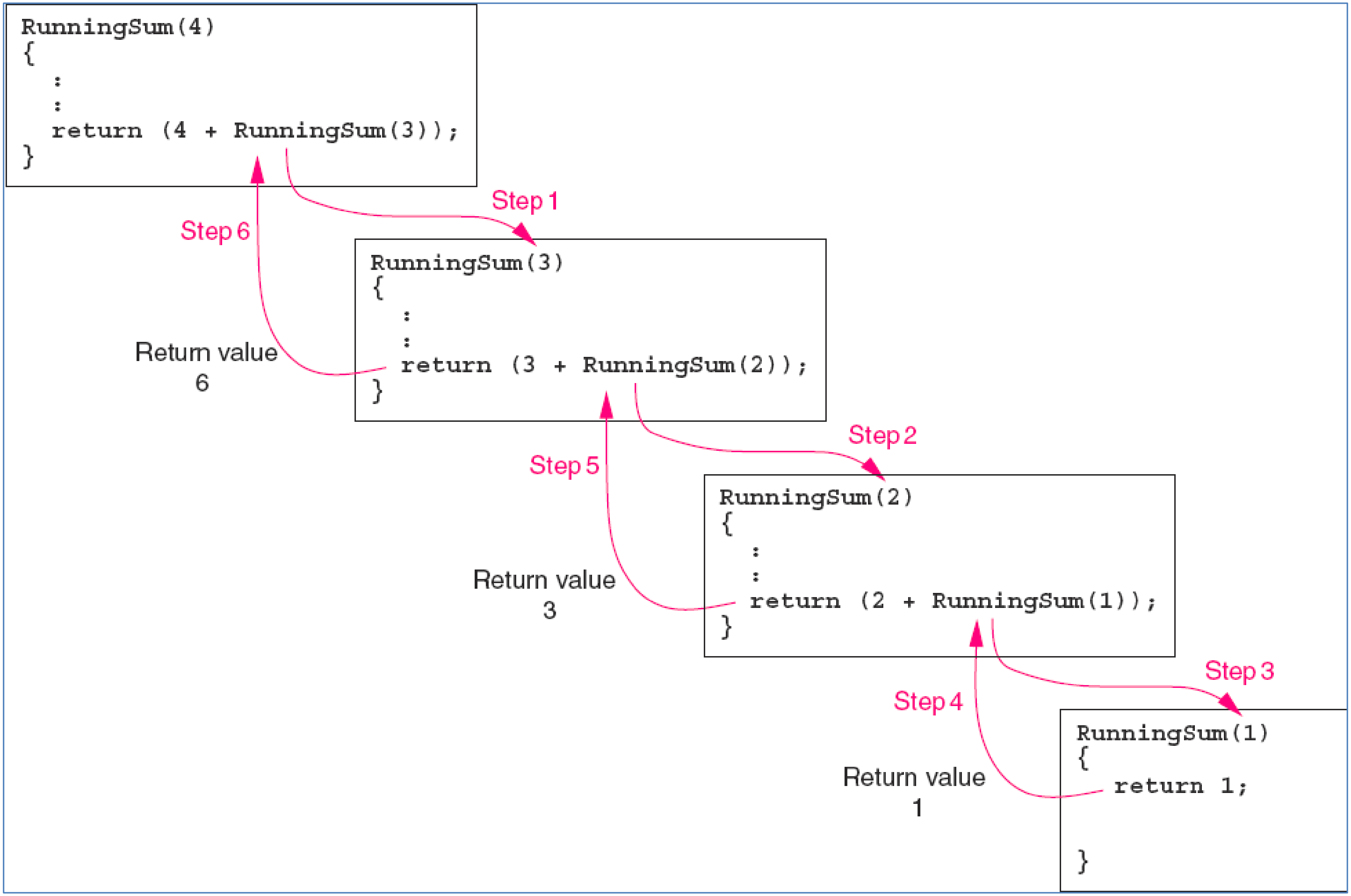
- Similar to recurrence function in mathematics.
- Like iteration -- can be used interchangeably;
  sometimes recursion results in a simpler solution.
- Must have at least 1 **base case** (terminal case) that ends the recursive process.

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)

```
#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 · (n-1) · (n-2) · … · 3 · 2 · 1

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

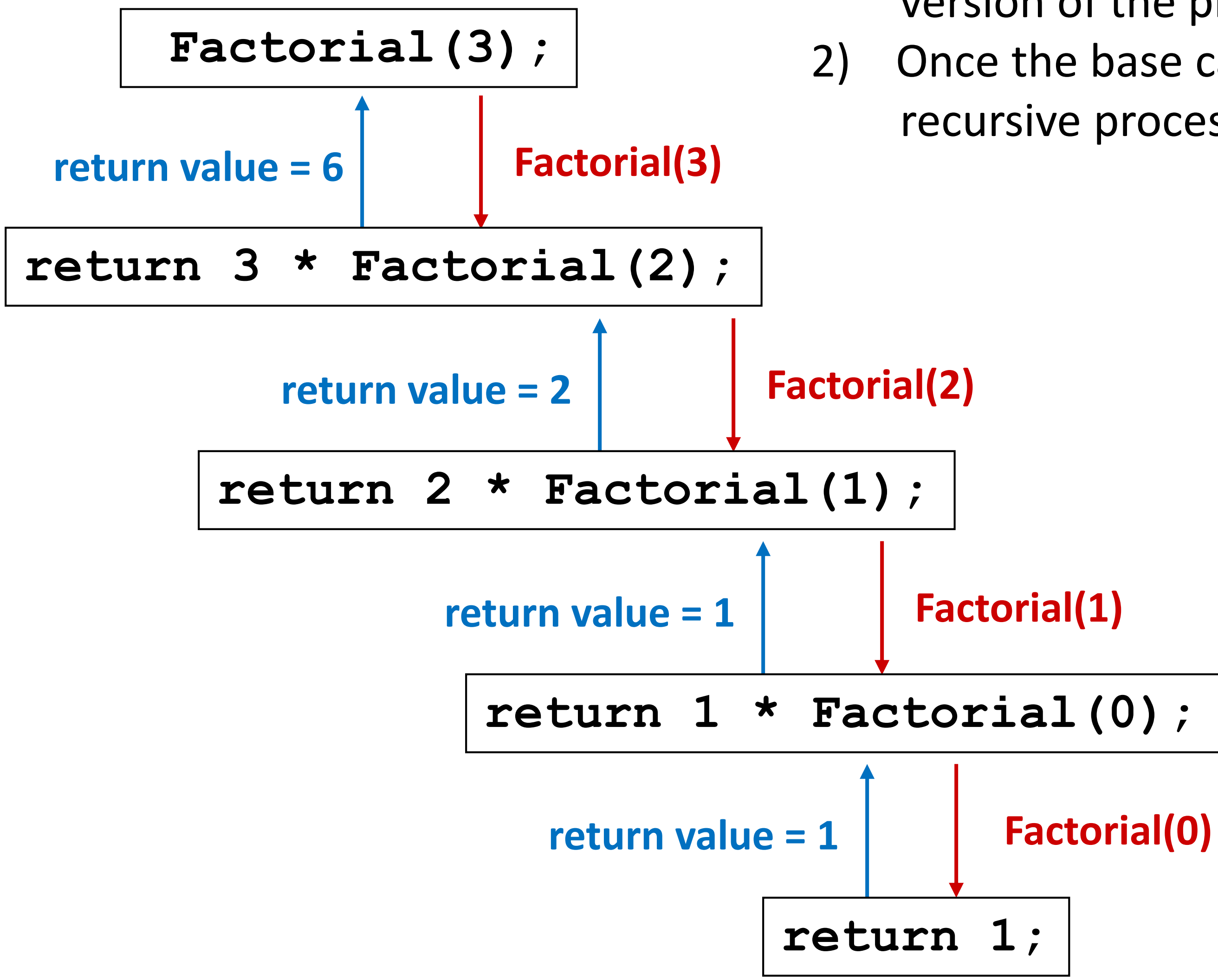
```
int Factorial(int n)
{

      if
       Return  ….


 else

 return

}
```

```c
#include <stdio.h>
int Factorial(int n);
//assume n is non-negative
int Factorial(int n)
{
    int fn;
    if(n == 0)
        fn=1;
    else
        fn= n*Factorial(n-1);
    return fn
}

int main()
{
    int n=3;
    int result = Factorial(n);
    printf("Factorial(%d)=%d \n",n,result);

    return 0;
}
```

Executing Factorial

Observation:

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



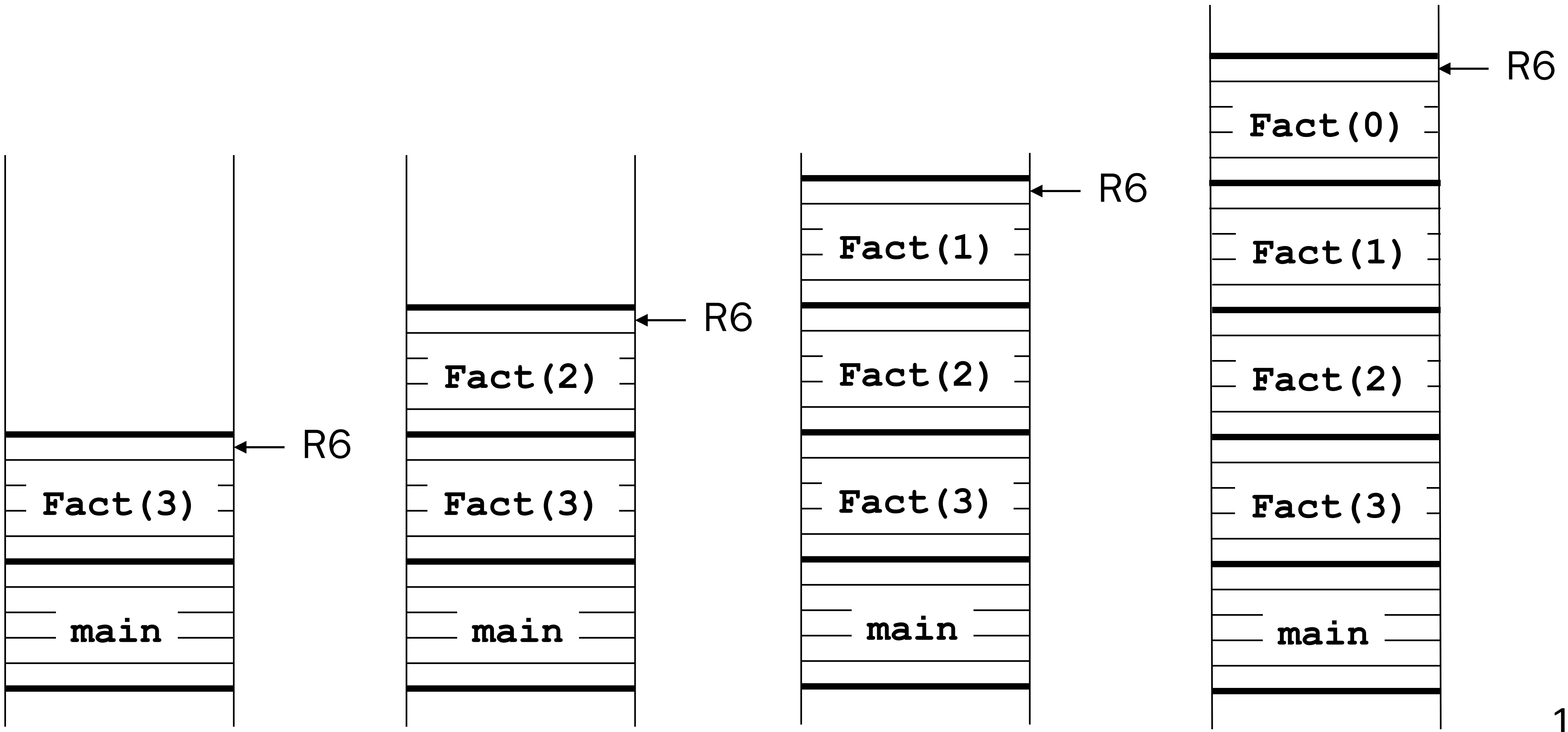
Run-Time Stack During Execution of Factorial

main calls Factorial(3)

Factorial(3) calls Factorial(2)

Factorial(2) calls Factorial(1)

Factorial(1) calls Factorial(0)



Recursion- C code to LC3

(See the code factorial.asm)

```
.ORIG x3000
; push argument
    LD R6, STACK_TOP
    AND R0,R0,#0
    ADD R0,R0,#3
    STR R0,R6,#0
; call subroutine
    JSR FACTORIAL
; pop return value from run-time stack (to R0)
    LDR R0,R6,#0
    ADD R6, R6, #+1
    HALT

FACTORIAL:
; push callee's bookkeeping info onto the run-time stack
; allocate space in the run-time stack for return value
    ADD R6, R6, #-1
; store caller's return address and frame pointer
    ADD R6, R6, #-1
    STR R7, R6, #0
    ADD R6, R6, #-1
    STR R5, R6, #0
; allocate memory for local variable fn
    ADD R6, R6, #-1
    ADD R5, R6, 0
; if (n>0)
    LDR R1, R5, #4
    ADD R2, R1, #-1
    BRn ELSE
```

Recursion- C code to LC3 (cont.)

(See the code factorial.asm)

```
; compute fn = n * factorial(n-1)
; caller-built stack for factorial(n-1) function call
; push n-1 onto run-time stack
    ADD R6, R6, #-1
    STR R2, R6, #0
; call factorial subroutine
    JSR FACTORIAL
; pop return value from run-time stack (to R0)
    LDR R0, R6, #0
    ADD R6, R6, #1
; pop function argument from the run-time stack
    ADD R6, R6, #1
; multiply n by the return value (already in R0)
    LDR R1, R5, #4
    ;MUL R2, R0, R1 ; R2 <- n * factorial(n-1)
    ST R7, SAVE_R7
    JSR MULT
    LD R7, SAVE_R7
    ADD R0, R2, #0
; store result in memory for fn
    STR R0, R5, #0
; done with this branch
    BRnzp RETURN
```

Recursion- C code to LC3 (cont.)

```
ELSE:
; store value of 1 in memory for fn
    AND R2, R2, #0
    ADD R2, R2, #1
    STR R2, R5, #0
; tear down the run-time stack and return
RETURN:
; write return value to the return entry
    LDR R0, R5, #0
    STR R0, R5, #3
; pop local variable(s) from the run-time stack
    ADD R6, R6, #1
; restore caller's frame pointer and return address
    LDR R5, R6, #0
    ADD R6, R6, #1
    LDR R7, R6, #0
    ADD R6, R6, #1
; return control to the caller function
    RET
```

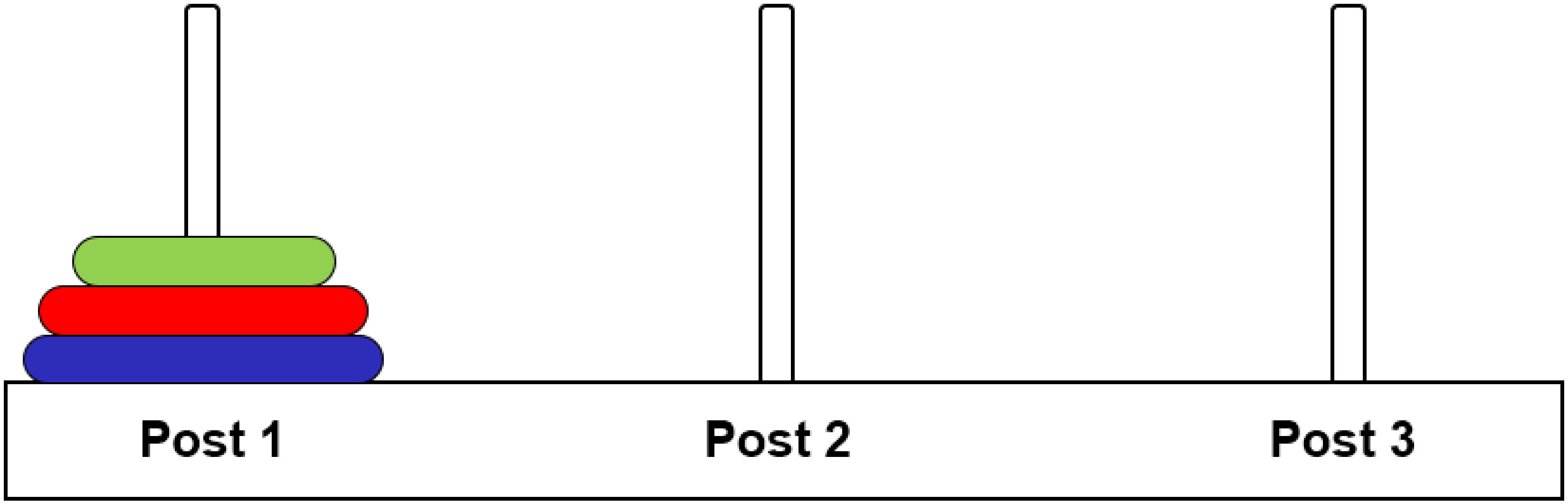
Recursion- C code to LC3 (cont.)

(See the code factorial.asm)

```
; multiply subroutine
; input should be in R0 and R1
; output should be in R2
MULT
    ; save R3
    ST R3, SAVE_R3
    ; reset R2 and initialize R3
    AND R2, R2, #0
    ADD R3, R0, #0
    ; perform multiplication
    MULT_LOOP
    ADD R3, R3, #-1
    BRn MULT_DONE
    ADD R2, R2, R1
    BRnzp MULT_LOOP
    MULT_DONE
    ; restore R0
    LD R3, SAVE_R3
    RET

SAVE_R3                 .BLKW #1
SAVE_R7                 .BLKW #1
STACK_TOP               .FILL x4000
.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.

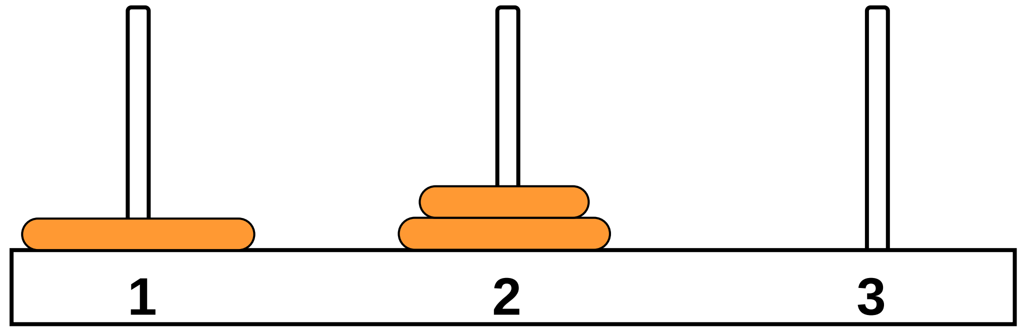
(3) May use third post for temporary storage.

Tower of Hanoi (Animation)

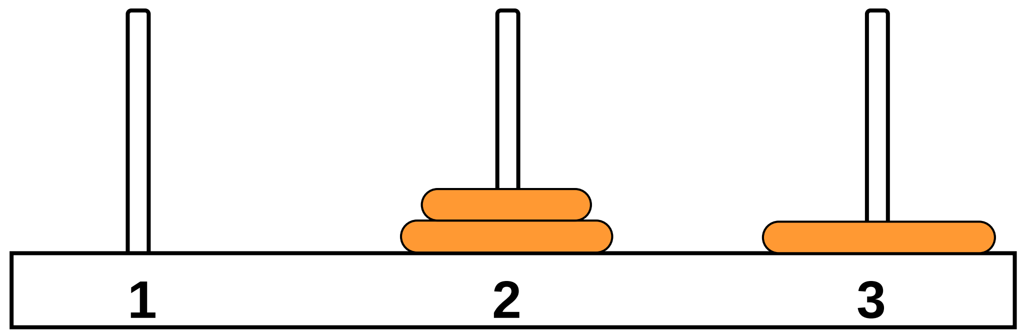
Task Decomposition

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

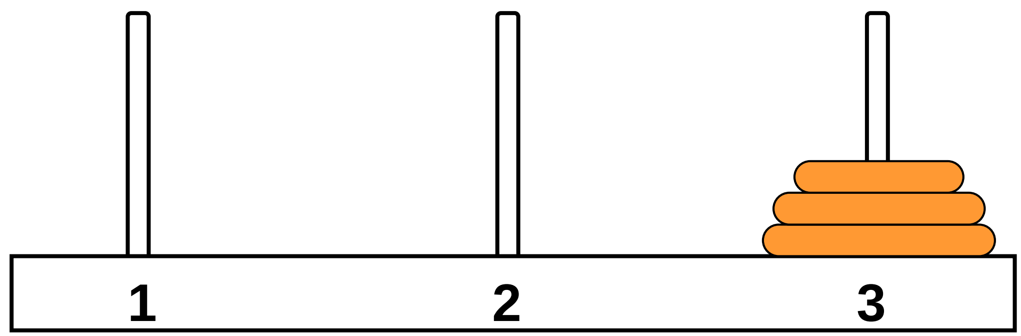
1. Move top n-1 disks to Post 2.



2. Move largest disk to Post 3.



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



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

int binary(int array[], int num, int start, int end);

int main()
{
    int index;
    int array[]={1,3,5,7,9,11,13,15,17};

    index = binary(array, 13, 0, LENGTH-1);
    printf("the value of the matched index: %d\n",index);
    return 0;
}

int binary(int array[], int item, int start, int end)
{
    if(start>end)
        return -1;

    int middle = (start+end)/2;

    if(item == array[middle]){
        return middle;}
    else if(item > array[middle])
        return binary(array, item, middle+1, end);
    else //item < array[middle]
        return binary(array, item, start, middle-1);

}
```

Quick Sort: also called divide-and-conquer

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

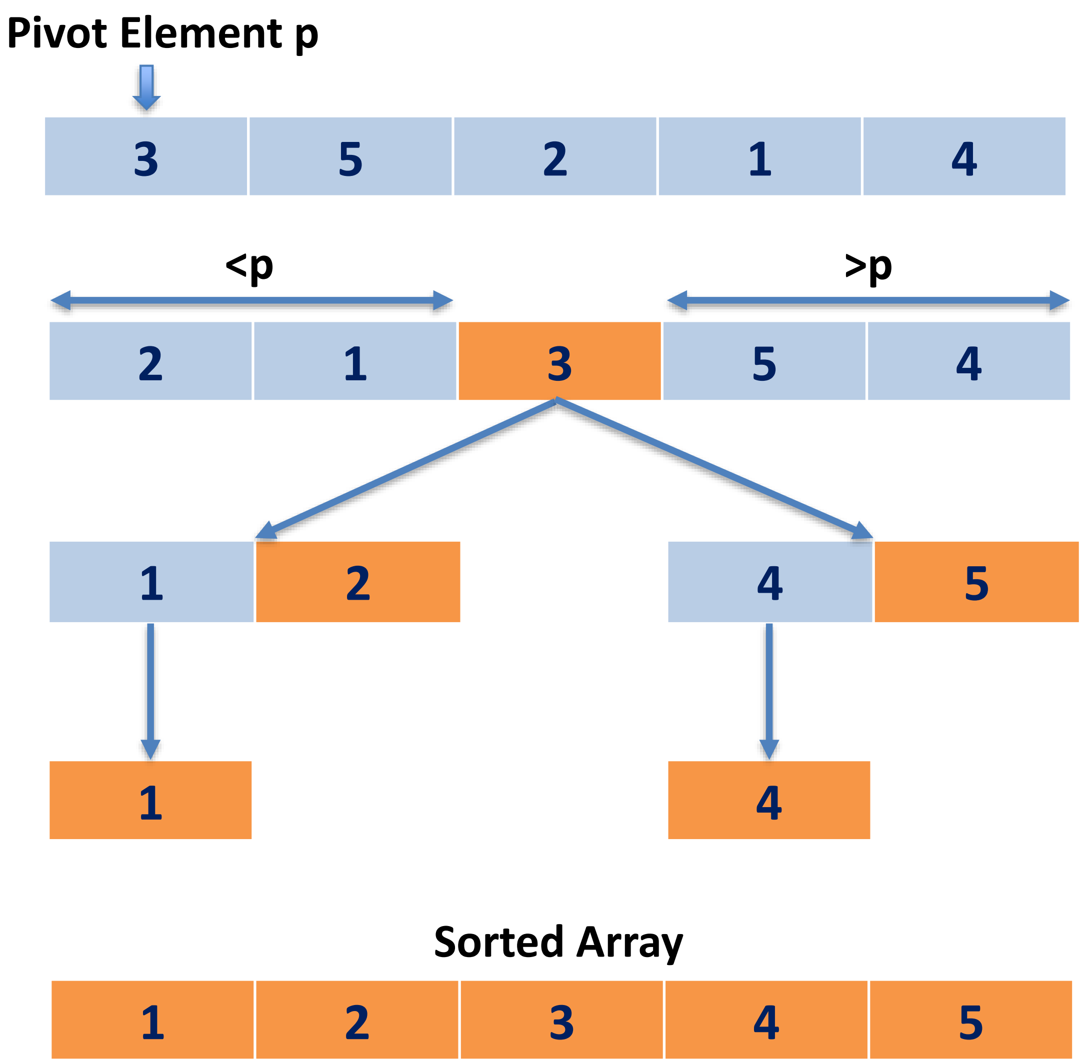
2) then sort subarrays using the same method.

```
/*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.



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);
    }
}
```