ECE 220 Computer Systems & Programming

Lecture 12 – Recursion

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: n!

Factorial:

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

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

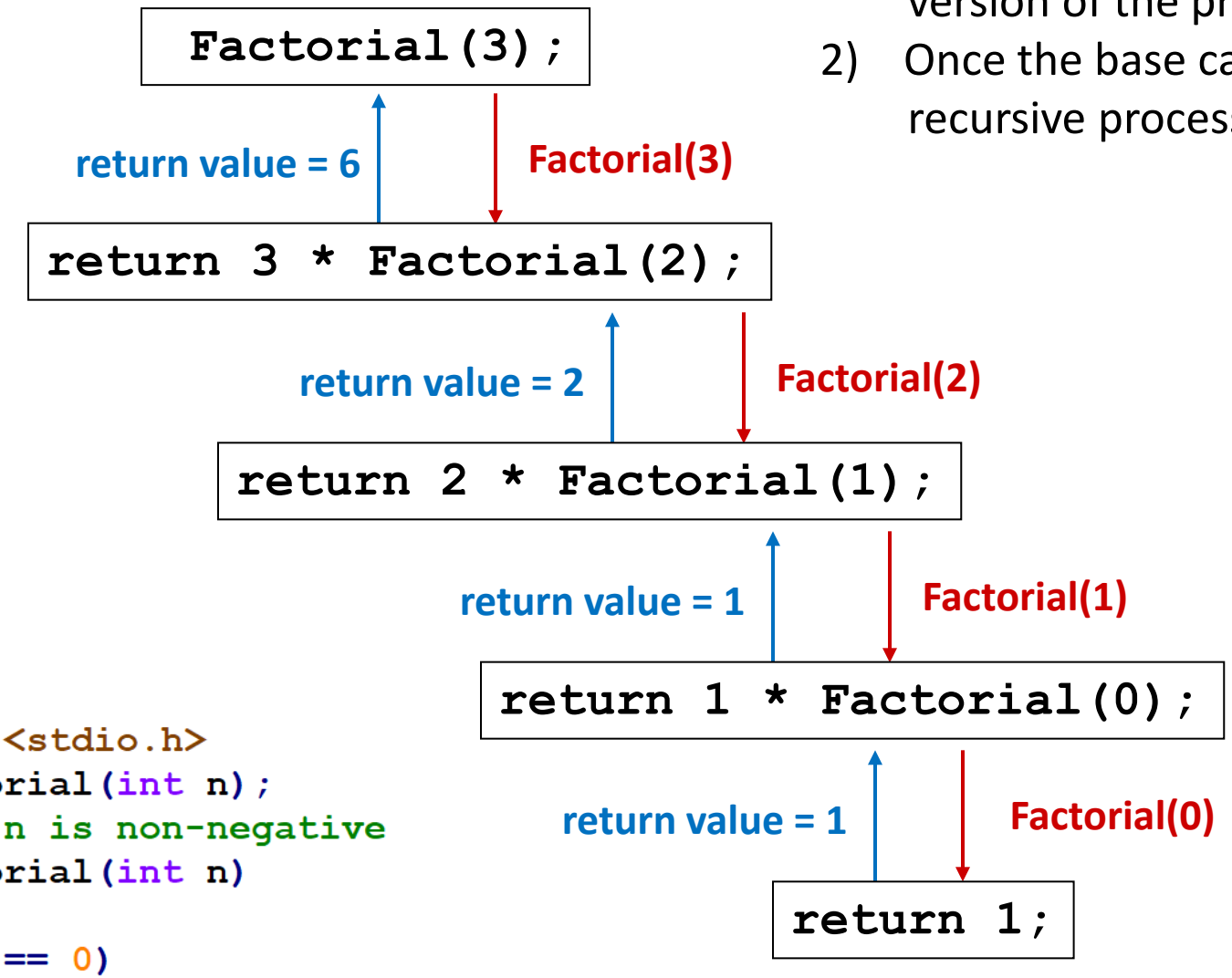
```
int Factorial(int n)
{

      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     int result = Factorial(n);
16     printf("Factorial(%d)=%d \n",n,result);
17
18     return 0;
19 }
```

Executing Factorial

Observation:

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




```
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  }
```

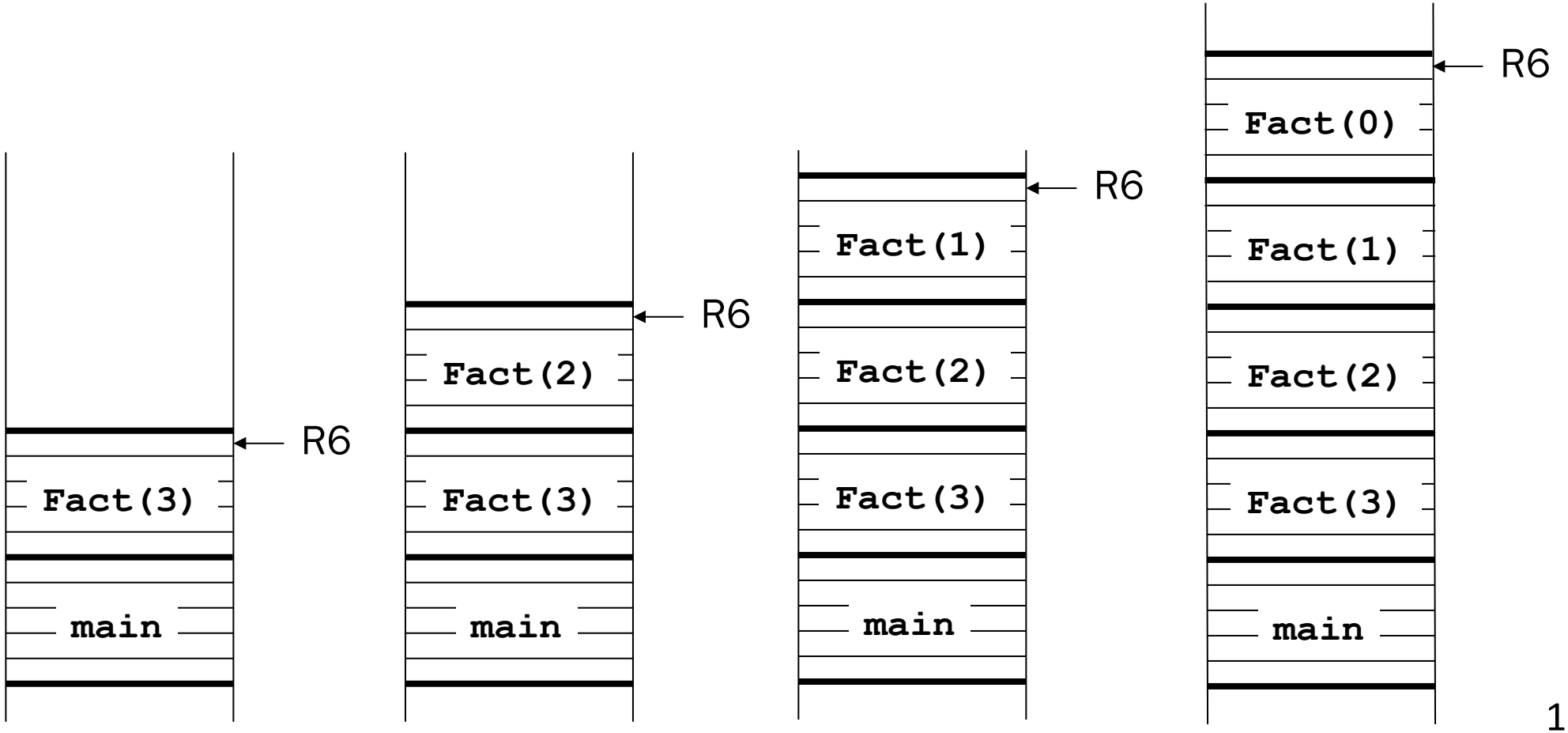
Run-Time Stack During Execution of Factorial

main calls Factorial(3)

Factorial(3) calls Factorial(2)

Factorial(2) calls Factorial(1)

Factorial(1) calls Factorial(0)



C to LC3 implementation of n!

(test case n=3)

```
1  .ORIG x3000
2  ; push argument
3      LD R6, STACK_TOP
4      AND R0,R0,#0
5      ADD R0,R0,#3
6      ADD R6,R6,#-1 ;R6 <- R6-1;
7      STR R0,R6,#0  ;push argument n
8  ; call subroutine
9      JSR FACTORIAL
10 ; pop return value from run-time stack (to R0)
11     LDR R0,R6,#0
12     ADD R6, R6, #2
13 ;Store the result
14     STR R0,R6,#0  ;dump the result at STACK_TOP
15     HALT
16
```

```
18 FACTORIAL:
19 ; push callee's bookkeeping info onto the run-time stack
20 ; allocate space in the run-time stack for return value
21     ADD R6, R6, #-1
22 ; store caller's return address and frame pointer
23     ADD R6, R6, #-1
24     STR R7, R6, #0
25     ADD R6, R6, #-1
26     STR R5, R6, #0
27 ; Update frame pointer for the callee
28     ADD R5, R6, #-1
29
30 ; if (n>0)
31     LDR R1, R5, #4
32     ADD R2, R1, #-1
33     BRn ELSE
34 ; compute fn = n * factorial(n-1)
35 ; caller-built stack for factorial(n-1) function call
36 ; push n-1 onto run-time stack
37     ADD R6, R6, #-1
38     STR R2, R6, #0
39 ; call factorial subroutine
40     JSR FACTORIAL
41 ; pop return value from run-time stack (to R0)
42     LDR R0, R6, #0
43     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
    BRnzp RETURN
ELSE:
; store value of 1 in to the memory of return value
    AND R0, R0, #0
    ADD R0, R0, #1
; tear down the run-time stack and return
RETURN:
; write return value to the return entry
    STR R0, R5, #3
; pop local variable(s) from the run-time stack
    ;no local variable for this implementation
; restore caller's frame pointer and return address
    LDR R5, R6, #0
    ADD R6, R6, #1
    LDR R7, R6, #0
    ADD R6, R6, #1 ;stack pointer is at the return value location
; return control to the caller function
    RET
```

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

Fibonacci Series

$$f(n) = f(n-1) + f(n-2)$$

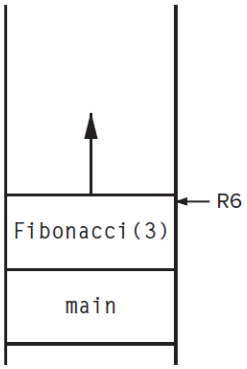
$$f(1) = 1$$

$$f(0) = 1$$

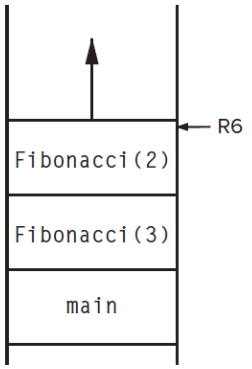
```
1   #include <stdio.h>
2
3   int Fibonacci(int n);
4
5   int main(void)
6   {
7      int in;
8      int number;
9
10     printf("Which Fibonacci number? ");
11     scanf("%d", &in);
12
13     number = Fibonacci(in);
14     printf("That Fibonacci number is %d\n", number);
15  }
16
17  int Fibonacci(int n)
18  {
19     int sum;
20
21     if (n == 0 || n == 1)
22        return 1;
23     else {
24        sum = (Fibonacci(n-1) + Fibonacci(n-2));
25        return sum;
26  }
```

```c
#include <stdio.h>

int Fibonacci(int n);

int main(void)
{
   int in;
   int number;

   printf("Which Fibonacci number? ");
   scanf("%d", &in);

   number = Fibonacci(in);
   printf("That Fibonacci number is %d\n", number);
}

int Fibonacci(int n)
{
   int sum;

   if (n == 0 || n == 1)
      return 1;
   else {
      sum = (Fibonacci(n-1) + Fibonacci(n-2));
      return sum;
}
```

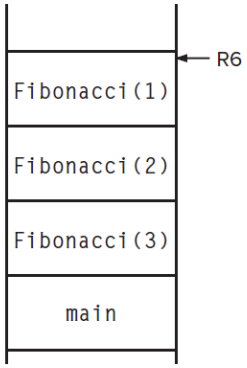
Consider, n=3



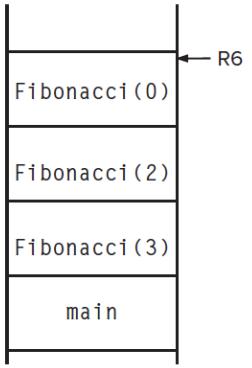
Step 1: Initial call



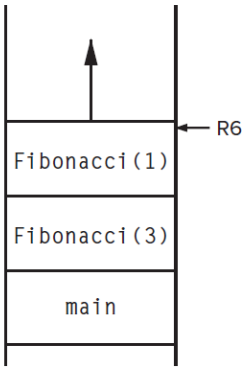
Step 2: Fibonacci(3) calls Fibonacci(2)



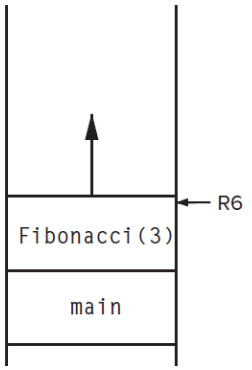
Step 3: Fibonacci(2) calls Fibonacci(1)



Step 4: Fibonacci(2) calls Fibonacci(0)



Step 5: Fibonacci(3) calls Fibonacci(1)



Step 6: Back to the starting point

Recursive Binary Search

```c
// C program to implement binary search using recursion
#include <stdio.h>

// A recursive binary search function. It returns location
// of x in given array arr[l..r] if present, otherwise -1
int binarySearch(int arr[], int l, int r, int x)
{
    // checking if there are elements in the subarray
    if (r >= l) {

        // calculating mid point
        int mid = (l + r) / 2;

        // If the element is present at the middle itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then it can only
        // be present in left subarray
        if (arr[mid] > x) {
            return binarySearch(arr, l, mid - 1, x);
        }

        // Else the element can only be present in right
        // subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not present in array
    return -1;
}
```

Recursive Quick Sort

```c
// The QuickSort function implementation
void quickSort(int arr[], int low, int high) {
    if (low < high) {

        // pi is the partition return index of pivot
        int pi = partition(arr, low, high);

        // Recursion calls for smaller elements
        // and greater or equals elements
        quickSort(arr, pi + 1, high);
        quickSort(arr, low, pi - 1);
    }
}
```

```c
int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
   printf("\n");
    return 0;
}
```

Helper Function: partition(….)

```c
// Partition function
int partition(int arr[], int low, int high) {

    // Choose the pivot
    int pivot = arr[high];

    // Index of smaller element and indicates
    // the right position of pivot found so far
    int i = low - 1;

    // Traverse arr[low..high] and move all smaller
    // elements to the left side. Elements from low to
    // i are smaller after every iteration
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    // Move pivot after smaller elements and
    // return its position
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}
```

```c
void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```