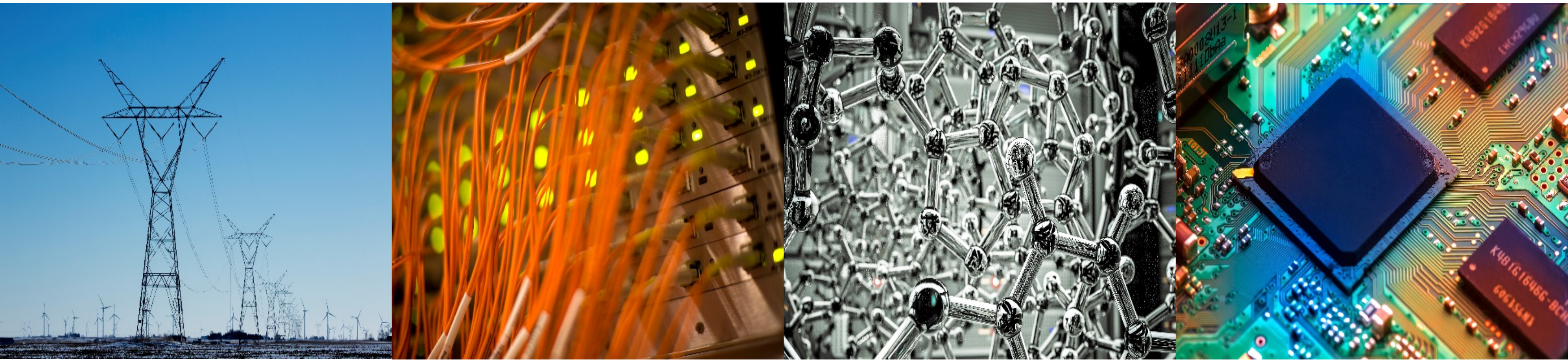
ECE 220 Computer Systems & Programming

Lecture 20 – Recursion

July 14, 2020



ILLINOIS

Electrical & Computer Engineering

GRAINGER COLLEGE OF ENGINEERING

- MP4 due today
- MT2 past exam & practice questions posted
- Informal Early Feedback

Recursion

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

- **At least** 1 ______________ case and 1 _______________ case

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);
}
```

✓ Recursive Fibonacci

✓ Recursive Fibonacci with Look-up Table

Recursive Binary Search

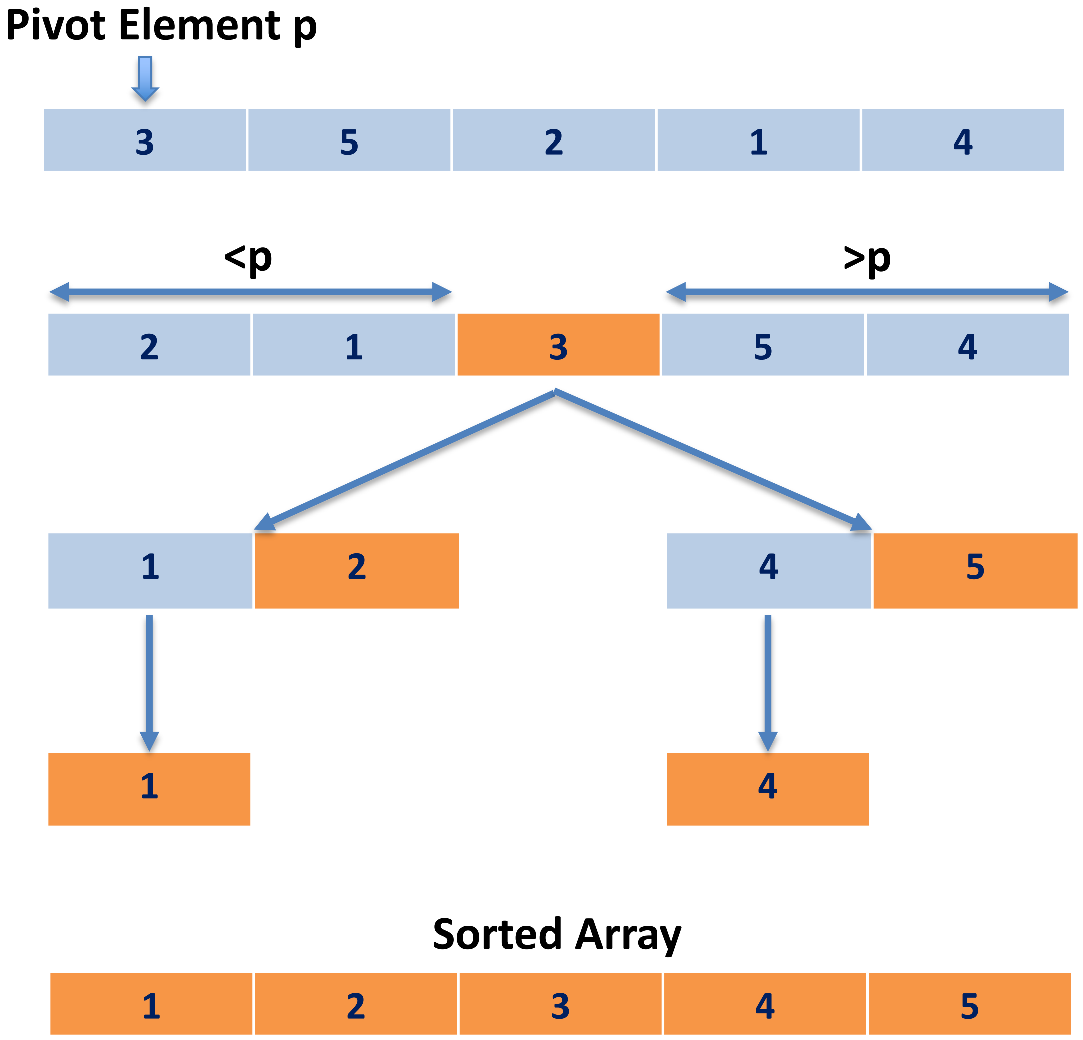
```
/* This function takes four arguments: pointer to a sorted array
in ascending order, the search item, the start index and the end
index of the array. If the search item is found, the function
returns its index in the array. Otherwise, it returns -1. */
int binary(int array[], int item, int start,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



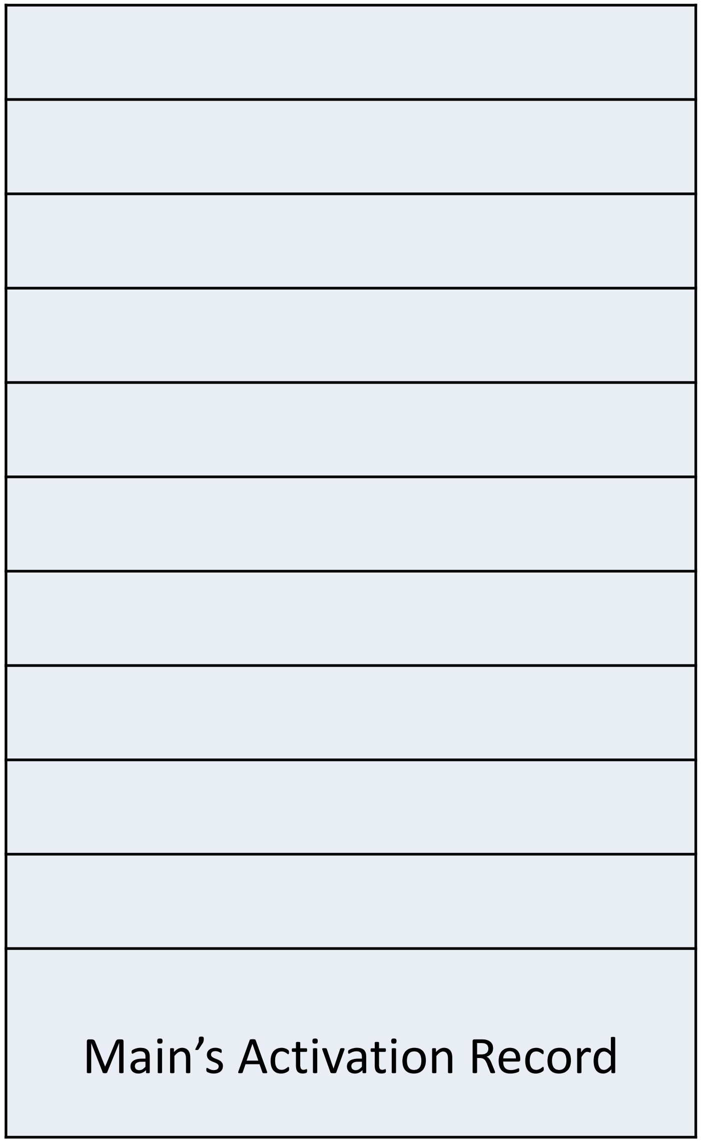
```c
/* Assume partition() function is given and it returns the index of the
pivot after partitioning the array within start and end indices. */
int partition(int array[], int start, int end);

/* This function takes 3 arguments: a pointer to the array, the start
index of the array and the end index of the array. The array should be
sorted in ascending order after the function call. */
void quicksort(int array[], int start, int end){















}
```

Recursive Factorial

n! = n · (n-1) · (n-2) · … · 3 · 2 · 1

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

```
/* assume n is non-negative */
int Factorial(int n){
      int fn;










      return fn;
}
```

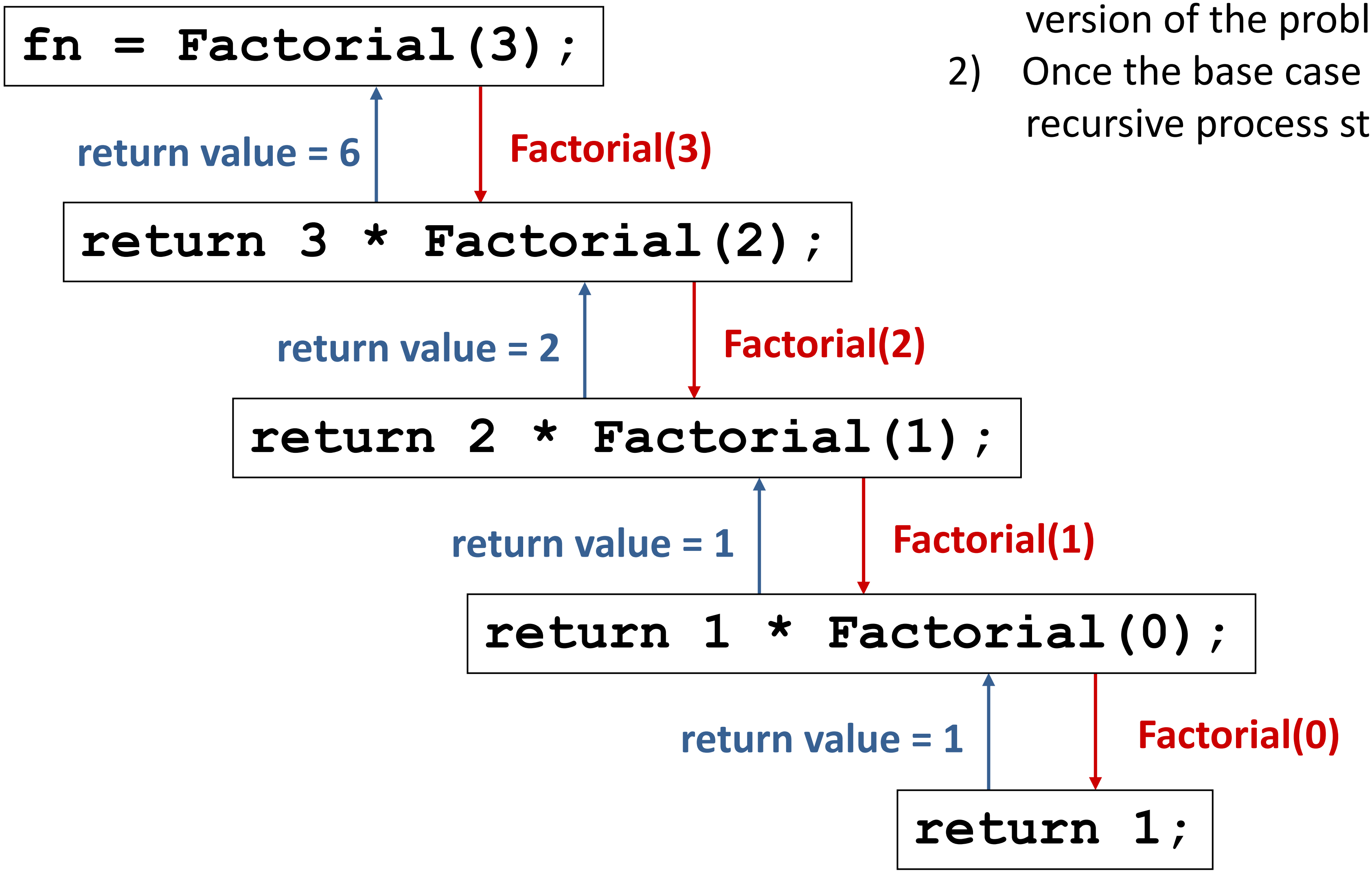


Main’s Activation Record

Executing Recursive Factorial

Observation:

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



```
fn = Factorial(3);
```

return value = 6

Factorial(3)

```
return 3 * Factorial(2);
```

return value = 2

Factorial(2)

```
return 2 * Factorial(1);
```

return value = 1

Factorial(1)

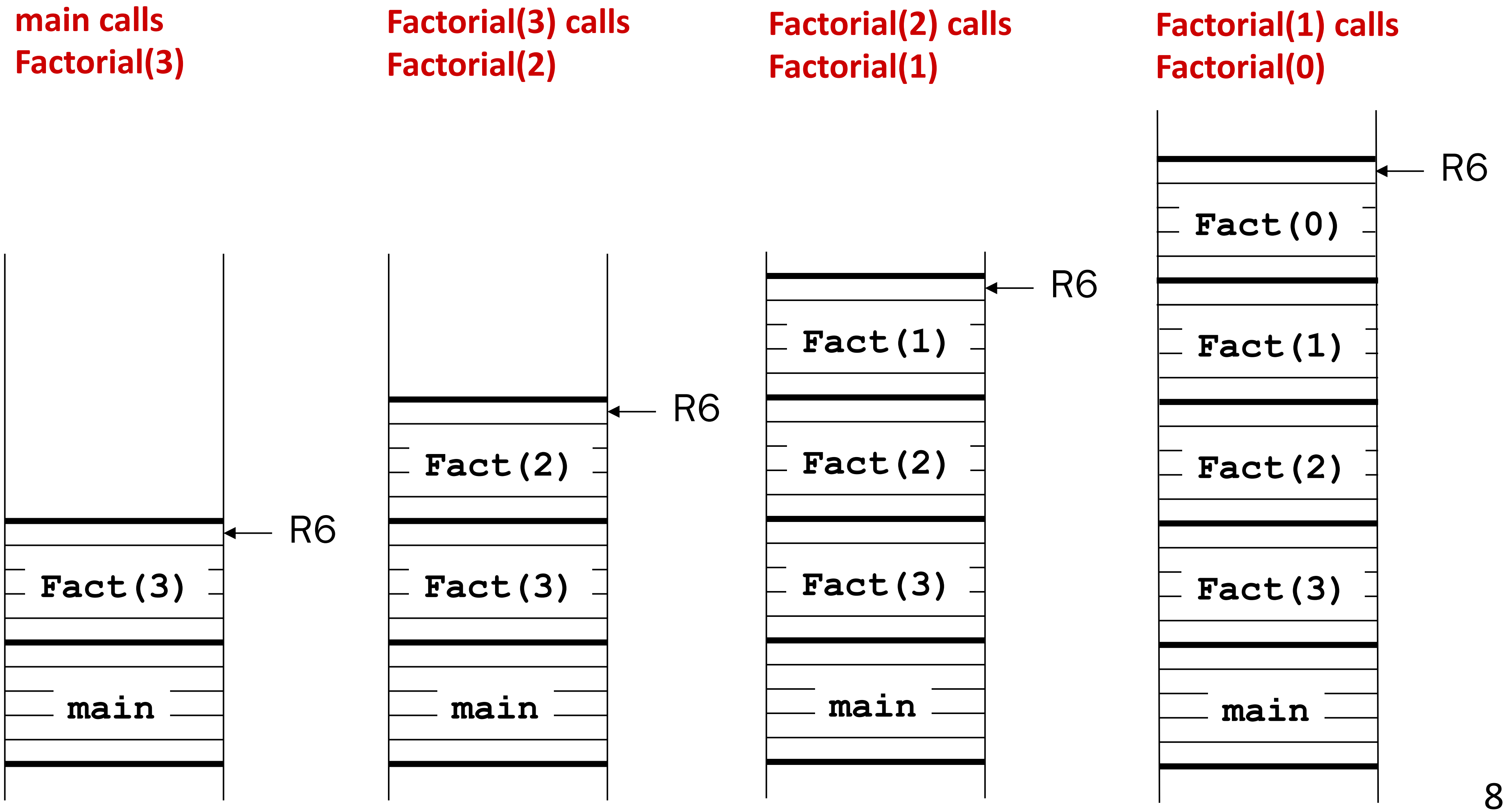
```
return 1 * Factorial(0);
```

return value = 1

Factorial(0)

```
return 1;
```

RTS During Execution of Recursive Factorial



main calls Factorial(3)

Factorial(3) calls Factorial(2)

Factorial(2) calls Factorial(1)

Factorial(1) calls Factorial(0)