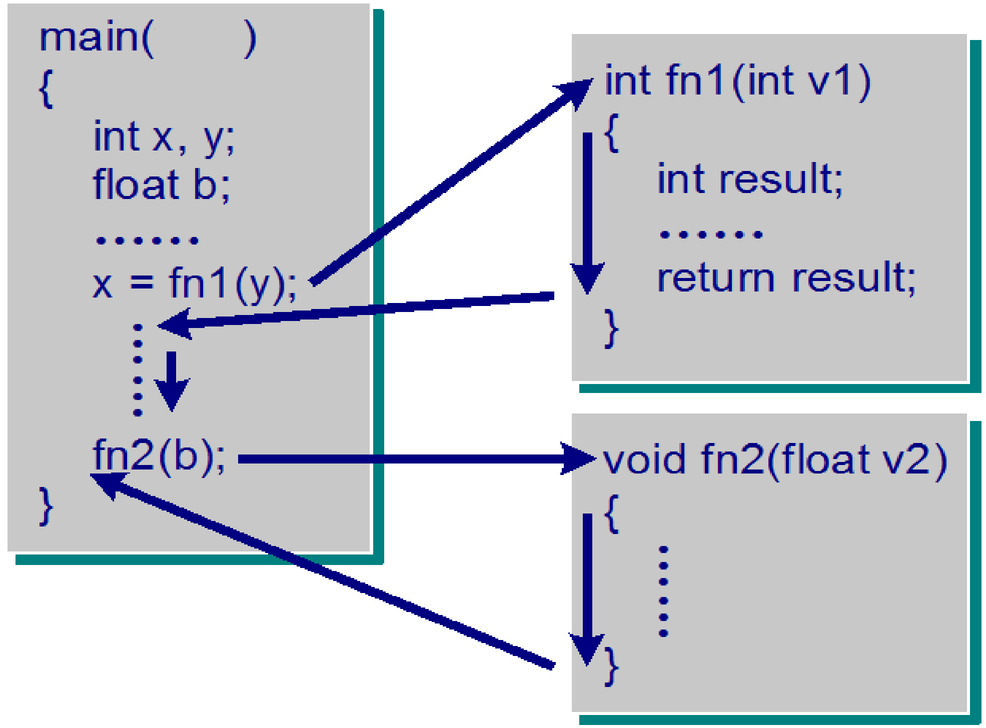
8
Recursion

Recursion

- **What is Recursion?**
- Recursive Functions: Examples
- Recursive Functions: Returning Value
- Recursive Functions: Call by Reference
- Recursion in Arrays
- How to Design Recursive Functions?

Function Execution

- C **Functions** have the following properties:
  - A function, when called, will accomplish a certain job.
  - When a function **fn1()** is called, control is transferred from the calling point to the first statement in **fn1()**. After the function finishes execution, the control will be returned back to the calling point. The next statement after the function call will be executed.
  - **Each call to a function has its <u>own set of values</u> for the actual arguments and local variables.**



What is Recursion?

- Divide and conquer - Recursion is the method in which a problem is solved by reducing it to **smaller cases** of the same problem.

- In recursion, the **function calls itself** or calls a sequence of other functions, one of which eventually calls the first function again.

How Does Recursion Work?

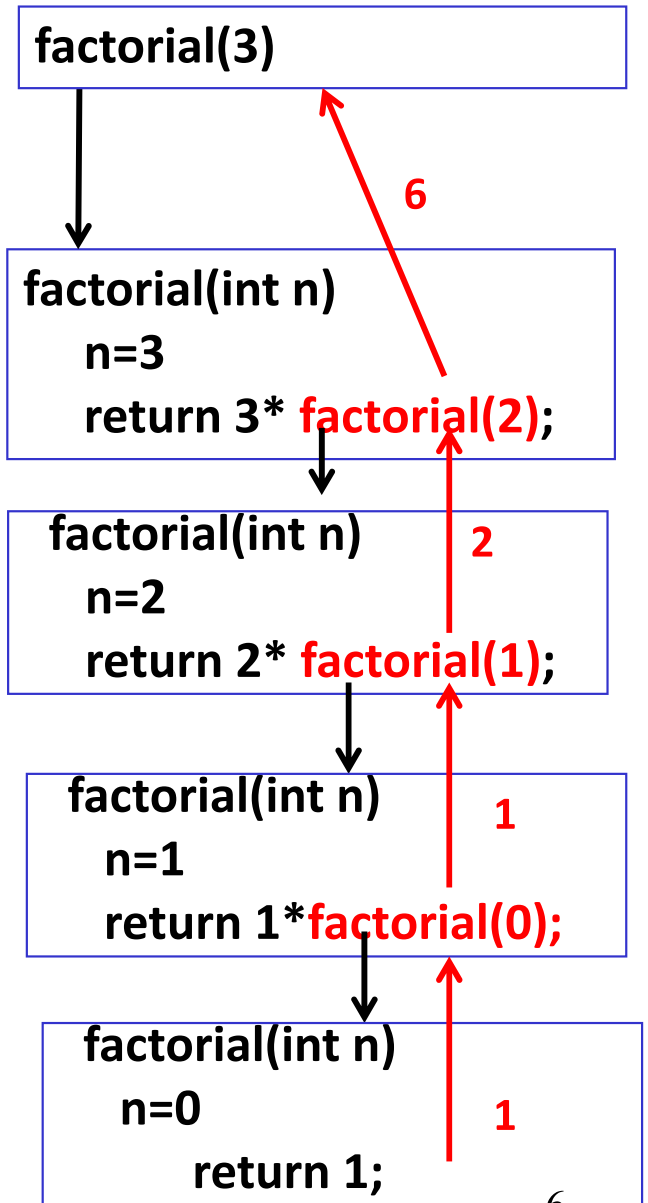
- Recursive function consists of two parts:
  - **Base case** (with **terminating** condition)
  - **Recursive case** (with **recursive** condition)

- Example: Factorial - recursive definition:
  - **n! = 1 if n = 0**
  - **n! = n × (n-1)! if n > 0**

```
#include <stdio.h>
int factorial(int n);
int main(){
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    printf("n! = %d\n", factorial(num));
    return 0;
}
int factorial(int n){
  if (n == 0)
      return 1;  /* terminating condition */
  else
      return n*factorial(n – 1);
          /* recursive condition */
}
```

How Does Recursion Work? (Cont'd.)

- Each function makes a **call to itself** with an **argument**
  - which is **closer** to the terminating condition.
- When a recursive call is made
  - **control** is transferred from the calling point to the first statement of the recursive function.
- Each call to a function
  - has its **own set of values/arguments** for the formal arguments and local variables.
- When a call at a certain level is finished
  - **control** is returned to the calling point **one level up**.



Recursion

- What is Recursion?
- **Recursive Functions: Examples**
- Recursive Functions: Returning Value
- Recursive Functions: Call by Reference
- Recursion in Arrays
- How to Design Recursive Functions?

Example 1: Cheers

- What does the following ***recursive*** function print when called with cheers(4)?

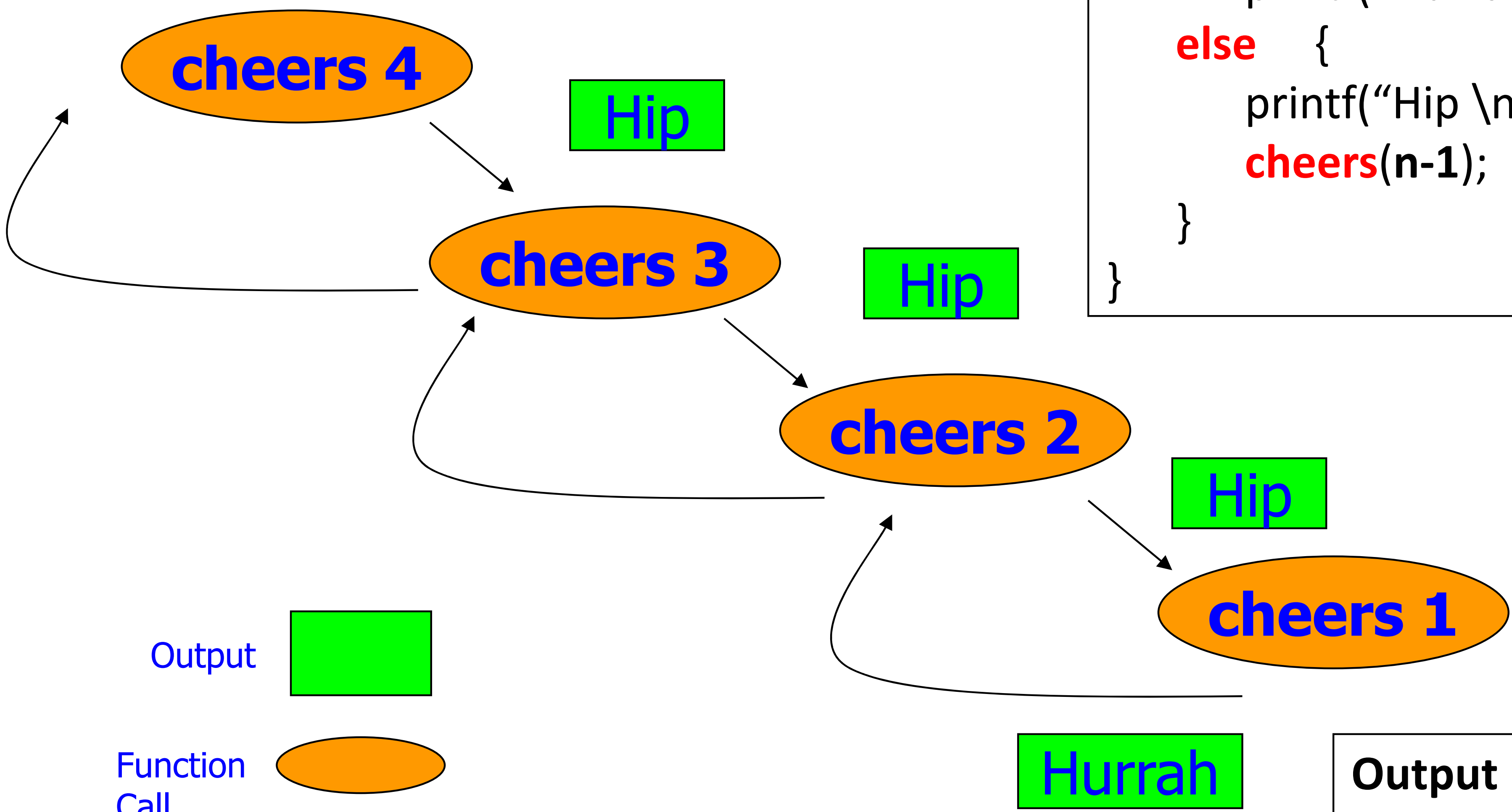
```
void cheers( int n)
{
    if (n <= 1)
        printf("Hurrah \n");
    else
    {
        printf("Hip \n");
        cheers(n-1);
    }
}
```

terminating condition

recursive condition (n > 1)

Recursive Cheers –Tracing

```
void cheers(int n){
    if (n <= 1)
        printf("Hurrah \n");
    else    {
        printf("Hip \n");
        cheers(n-1);
    }
}
```



Output
Hip
Hip
Hip
Hurrah

Example 2: PrintSomething

- What does the following *recursive* function print when called with **printSomething(3)**?

```
void printSomething( int n )
{
    if (n <= 0)
        return;
    else  {
        printf("*");
        printSomething(n-1);
        printf("!");
    }
}
```

terminating condition

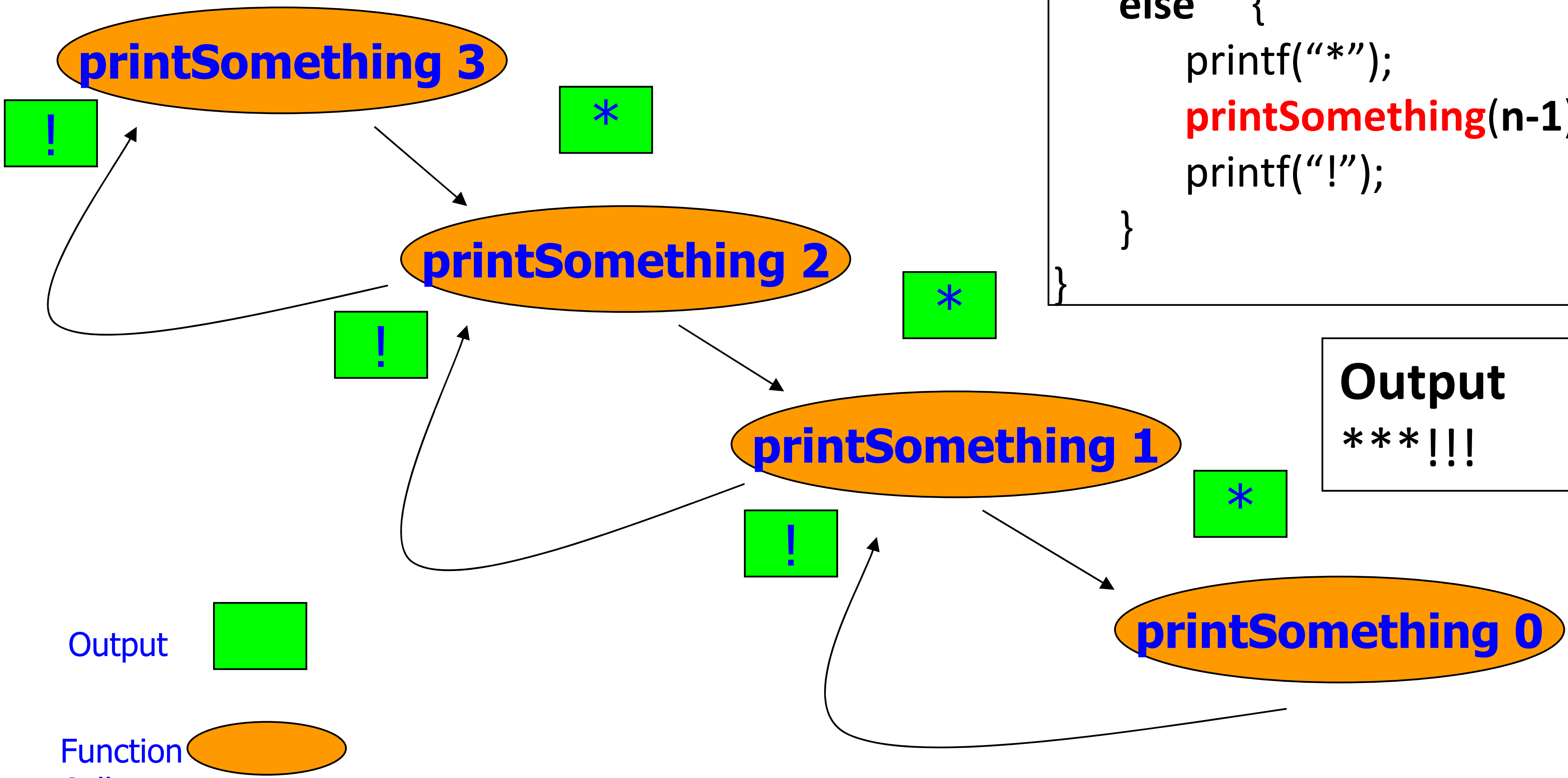
Recursive condition

Recursive PrintSomething - Tracing

```
void printSomething( int n )
{
    if (n <= 0)
        return;
    else    {
        printf("*");
        printSomething(n-1);
        printf("!");
    }
}
```

Output

***!!!



Example 3: Digits Printing

- For example: Given a number, say **2345**, print each digit of the number **one digit per line**.
- The recursive implementation should consist of two parts:
  - **Terminating Condition**: It will happen when the number is a **single** digit. Then, just print that digit.
  - **Recursive Condition**: It **reduces** the problem into a **smaller** but the same problem by using integer division and modulus operators.

Recursive Digits Printing

```
#include <stdio.h>
void printDigit(int num);
int main()
{
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    printDigit(num);
    return 0;
}
void printDigit(int num)
{
    if (num < 10)                  // terminating condition
        printf("%d", num);
    else    {                      // recursive condition
        printDigit(num/10);
        printf( "%d\n", num%10 );
    }
}
```

Recursive Digits Printing: Tracing

printDigit(2345);

```
void printDigit(int num)  {
      if (num < 10)
            printf("%d", num);
      else     {
            printDigit(num / 10);
            printf( "%d\n", num%10 );
      }
}
```

printDigit(2345)
if (num < 10)
…
else
printDigit (2345/10)
output 2345 % 10

num = 2345

5

printDigit(234)
if (num < 10)
…
else
printDigit (234 / 10)
output 234 % 10

num = 234

4

printDigit(23)
if (num < 10)
…
else
printDigit (23 / 10)
output 23 % 10

num = 23

3

printDigit(2)
if (num < 10)
output 2
else
…

num=2

2

14

Output
Enter a number: <u>*2345*</u>
2
3
4
5

Recursion

- What is Recursion?
- Recursive Functions: Examples
- **Recursive Functions: Returning Value**
- Recursive Functions: Call by Reference
- Recursion in Arrays
- How to Design Recursive Functions?

Example 4: Factorial – Iterative Approach

- The factorial function of a positive integer is usually defined by the formula: $n! = n \times (n-1) \times \ldots\ldots \times 1$

Non-recursive (iterative) version:

```
#include  <stdio.h>
int fact(int n);
int main()
{
    int num;
    printf("Enter an integer: ");
    scanf("%d", &num);
    printf("n! = %d\n", fact(num));
    return 0;
}
```

```
int fact(int n)
{
    int i;
    int temp = 1;
    for (i = n; i > 0; i--)
        temp *= i;
    return temp;
}
```

Output

Enter an integer: *4*

n! = 24

Recursive Factorial

- A more precise mathematical definition (recursive definition)

– **n! = 1**	**if n = 0**
– **n! = n × (n-1)!**	**if n > 0**

- Suppose we wish to calculate 4!

  as $4 > 0$, $4! = 4 \times 3!$

- But we do not know what is 3!

  as $3 > 0$, $3! = 3 \times 2!$

  as $2 > 0$, $2! = 2 \times 1!$

  as $1 > 0$, $1! = 1 \times 0!$

- We know what is 0! (i.e. 1)

- Using the factorial recursive definition, we have:

$$
\begin{aligned}
4! &= 4 \times 3! \\
&= 4 \times (3 \times 2!) \\
&= 4 \times (3 \times (2 \times 1!)) \\
&= 4 \times (3 \times (2 \times (1 \times 0!))) \\
\hline
&= 4 \times (3 \times (2 \times (1 \times 1))) \\
&= 4 \times (3 \times (2 \times 1)) \\
&= 4 \times (3 \times 2) \\
&= 4 \times 6 \\
&= 24
\end{aligned}
$$

Recursive Factorial (Cont'd.)

- It is straight-forward to implement recursive function if the **recursive mathematical definition** is available:
  - $n! = 1$ if $n = 0$
  - $n! = n \times (n-1)!$ if $n > 0$

```
int factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n*factorial(n – 1);
}
```

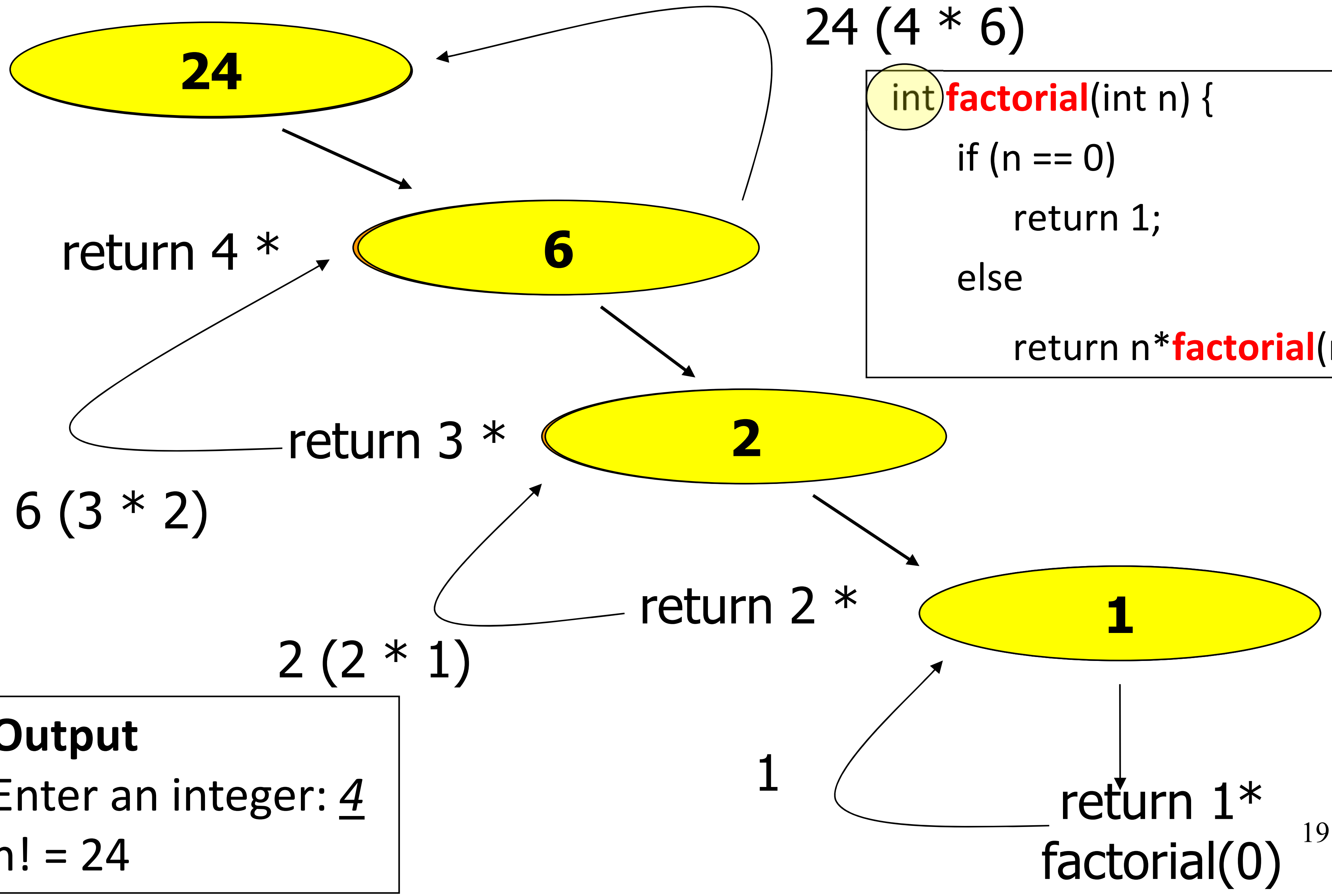
NB: By returning value

/* terminating condition*/

/* recursive condition*/

Recursive Factorial: Tracing

```
int factorial(int n) {
    if (n == 0)
        return 1;
    else
        return n*factorial(n – 1); }
```



Output
Enter an integer: *4*
n! = 24

Example 5: Multiplication by Addition

- The multiplication operation:

  multi(m,n) = m x n [e.g. 3x2]

can be defined recursively mathematically as follows:

multi(m,n) = m if n = 1
multi(m,n) = m + multi(m,n-1) if n > 1

- For example,

multi(3,2) = 3 + multi(3,1) [using recursive condition]
(i.e. 3x2) = 3 + (3) [using terminating condition]
= 6

Recursive Multiplication

```
/*Using pass by value  by returning the result*/
#include  <stdio.h>
int  multi(int, int);
int main()
{
      printf("5 * 3 = %d\n", multi(5, 3));
      return 0;
}
int multi(int m, int n)
{
      if (n == 1)        /* terminating condition */
         return m;
      else {             /* recursive condition */
         return m + multi(m, n-1);
      }
}
```

Note that:
Each function call will maintain its local variables.

NB: By returning value

Recursive Definition:

multi(m,n) = m if n = 1

multi(m,n) = m + multi(m,n-1) if n>1

Recursive Multiplication: Tracing

```
/*Using pass by value  by returning the result*/
#include  <stdio.h>
int  multi(int, int);
int main()
{
     printf("5 * 3 = %d\n", multi(5, 3));
     return 0;
}
```

```
int multi(int m, int n)
{
     if (n == 1)    /* terminating condition */
        return m;
     else {         /* recursive condition */
        return m + multi(m, n-1);
     }
}
```

Note that:
Each function call will maintain its local variables.

multi(5, 3)

multi(int m, int n)
m=5, n=3
return 5+ multi(5,2);

multi(int m, int n)
m=5, n=2
return 5+ multi(5,1);

multi(int m, int n)
m=5, n=1
return 5;

Recursion Multiplication: Tracing (Cont'd.)

```
/*Using pass by value  by returning the result*/
#include  <stdio.h>
int  multi(int, int);
int main()
{
     printf("5 * 3 = %d\n", multi(5, 3));
     return 0;
}
int multi(int m, int n)
{
     if (n == 1)    /* terminating condition */
        return m;
     else {          /* recursive condition */
        return m + multi(m, n-1);
     }
}
```

Output
5 * 3 = 15

multi(5, 3)

multi(int m, int n)
m=5, n=3
return 5+ multi(5,2);

5+10=15

multi(int m, int n)
m=5, n=2
return 5+ multi(5,1);

5+5=10

5

multi(int m, int n)
m=5, n=1
return 5;

Recursion

– What is Recursion?

– Recursive Functions: Examples

– Recursive Functions: Returning Value

– **Recursive Functions: Call by Reference**

– Recursion in Arrays

– How to Design Recursive Functions?

Recursive Multiplication: Call by Reference

```
/* Using pass by reference and result is passed via pointer */
 #include <stdio.h>
void multi2(int, int, int*);
 int main()
{
     int result=0;
     multi2(5, 3, &result);
     printf("5 * 3 = %d\n", result);
     return 0;
}
void multi2(int m, int n, int *product)
{
     if (n == 1)  /* terminating condition*/
          *product = m;
     else {      /* recursive condition */
          multi2(m, n-1, product);
          *product = *product + m;
     }
}
```

NB: By call by reference

Recursive Definition:

multi(m,n) = m if n = 1

multi(m,n) = m + multi(m,n-1) if n>1

Recursive Multiplication: Tracing

```c
/* Using pass by reference and result is passed via pointer */
 #include  <stdio.h>
void multi2(int, int, int*);
 int main()
{
     int result=0;
     multi2(5, 3, &result);
     printf("5 * 3 = %d\n", result);
     return 0;
}
```

```c
void multi2(int m, int n, int *product)
{
     if (n == 1)  /* terminating condition*/
          *product = m;
     else {      /* recursive condition */
          multi2(m, n-1, product);
          *product = *product + m;
     }
}
```

result: 0

multi2(5, 3, &result)

product

multi2(int m, int n, int *product)
m=5, n=3
multi2(5,2,product);

product

multi2(int m, int n, int *product)
m=5, n=2
multi2(5,1,product);

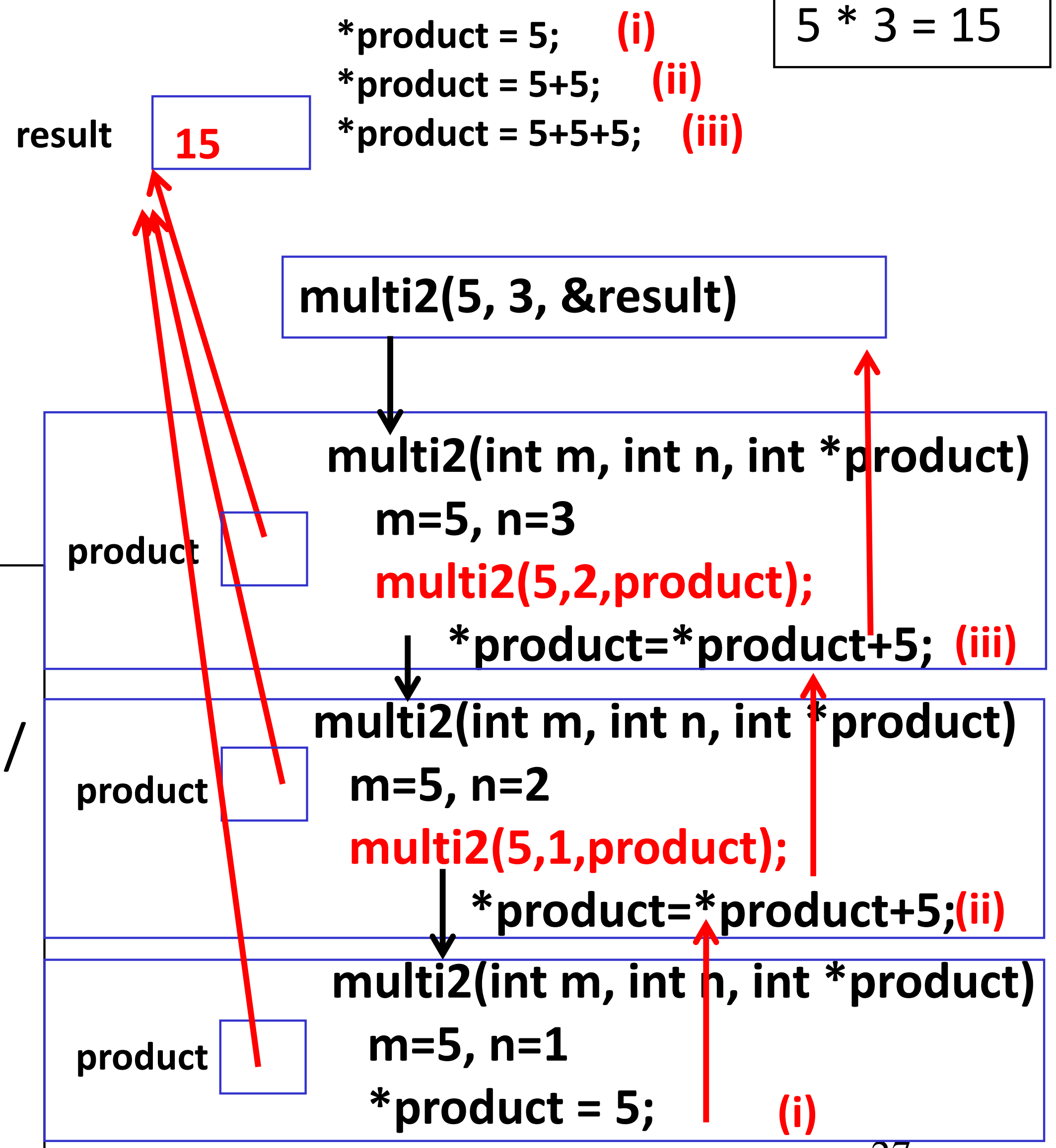
product

multi2(int m, int n, int *product)
m=5, n=1
*product = 5;

Recursive Multiplication: Tracing (Cont'd.)

```
/* Using pass by reference and result is passed via pointer */
 #include <stdio.h>
void multi2(int, int, int*);
 int main()
{
     int result;
     multi2(5, 3, &result);
     printf("5 * 3 = %d\n", result);
     return 0;
}
void multi2(int m, int n, int *product)
{
     if (n == 1)  /* terminating condition*/
           *product = m;
     else {      /* recursive condition */
           multi2(m, n-1, product);
           *product = *product + m;
     }
}
```

Output
5 * 3 = 15



Recursion

- What is Recursion?
- Recursive Functions: Examples
- Recursive Functions: Returning Value
- Recursive Functions: Call by Reference
- **Recursion in Arrays**
- How to Design Recursive Functions?

Recursion in Arrays
Example 6: Summing Array of Integers

```
#include <stdio.h>
int sumArray(int a[], int size);
int main()
{
  int a[10] = {1, 2, 3, 4};
  int sum;

  sum = sumArray(a, 4);
  printf("Sum = %d", sum);
  return 0;
}
```

```
int sumArray(int a[], int size)
{   // iterative version
    int sum = 0,i;
    for (i = 0; i < size; i++)
          sum = sum + a[i];
    return sum;
}
```

- Given an array of integers, write a method to calculate the **sum** of all the integers.

a	1	2	3	4
	[0]	**[1]**	**[2]**	**[3]**

Output
Sum = 10

Recursive Array Sum

```
#include <stdio.h>
int recursiveSum(int a[], int size);
int main()
{
   int a[10] = {1, 2, 3, 4};
   int sum;
   sum = recursiveSum(a, 4);
   printf("Sum = %d", sum);
   return 0;
}
```

- Given an array of integers, write a method to calculate the **sum** of all the integers.

a	1	2	3	4
	[0]	**[1]**	**[2]**	**[3]**

NB: By returning value

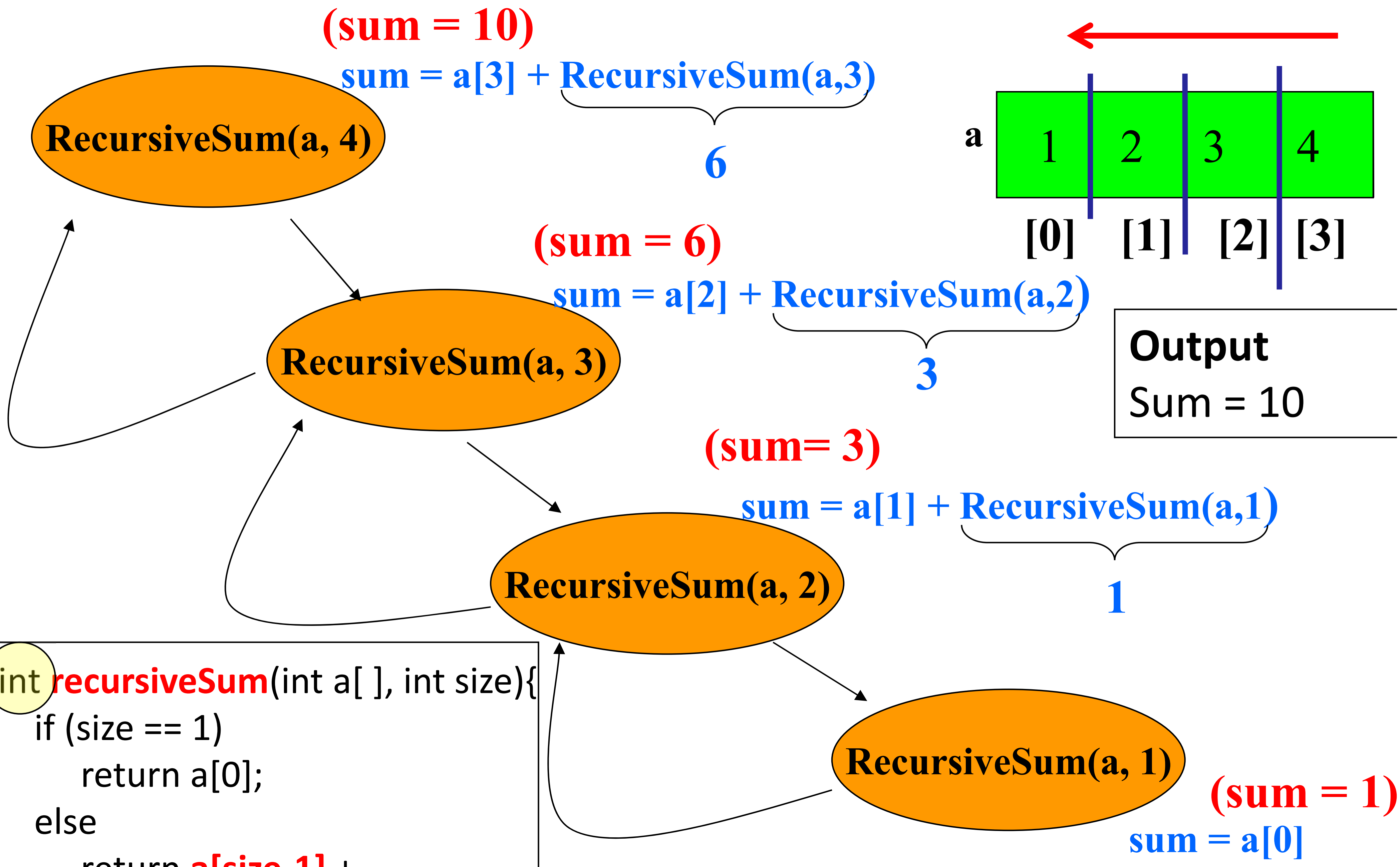
```
int recursiveSum(int a[ ], int size)
{   // recursive version
    if (size == 1)    /* terminating condition*/
        return a[0];
    else              /* recursive condition*/
        return a[size-1] + recursiveSum(a, size-1);
}
```

Recursive Definition:

rSum(a,size) = a[0] if size = 1

rSum(a,size) = a[size-1] + rSum(a,size-1)   if size>1

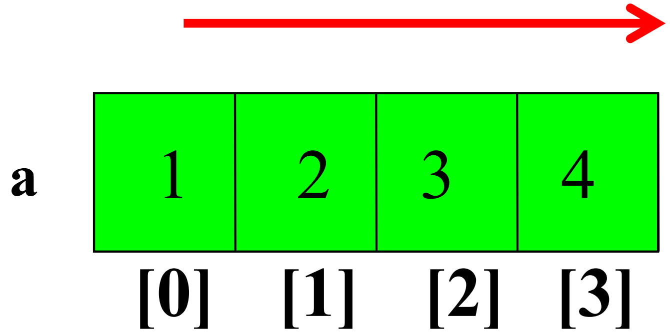
Recursive Array Sum –Tracing



Output
Sum = 10

```
int recursiveSum(int a[ ], int size){
   if (size == 1)
      return a[0];
   else
      return a[size-1] +
         recursiveSum(a, size-1);}
```

Another Version




```
#include <stdio.h>
int recursiveSum(int a[], int size);
int main()
{
  int a[10] = {1, 2, 3, 4};
  int sum;
  sum = recursiveSum(a, 4);
  printf("Sum = %d", sum);
  return 0;
}
```

Is this working?

```
int recursiveSum(int a[ ], int size)
{
    if (size == 1)    /* terminating condition*/
        return a[0];
    else              /* recursive condition*/
        return a[0] + recursiveSum(a+1, size-1);
}
```

Recursive Definition:

rSum(a,size) = a[0] if size = 1

rSum(a,size) = a[0] +
    rSum(a+1,size-1)   if size>1

Recursion

- What is Recursion?
- Recursive Functions: Examples
- Recursive Functions: Returning Value
- Recursive Functions: Call by Reference
- Recursion in Arrays
- **How to Design Recursive Functions?**

How to Design Recursive Functions?

- Find the **key step** (**recursive condition**)
  - How can the problem be divided into parts?
  - How will the key step in the middle be done?
- Find a **stopping rule** (**terminating condition**)
  - Small, special case that is trivial or easy to handle without recursion
- Outline your algorithm
  - **Combine** the stopping rule and the key step, using an **if-else** statement to select between them
- Check termination
  - **Verify** recursion always **terminates** (it is necessary to make sure that the function will also terminate)

Recursion or Iteration ?

- Advantage of using recursive functions:
  - when the problem is recursive in nature, a recursive function results in **short**, **clear** code.
- Disadvantage of using recursive functions:
  - recursion is more **expensive** (computationally) than iteration.

- Any problem that can be solved recursively can also be solved iteratively (by using **loops**).

- A recursive approach is normally chosen in preference to an iterative approach when the recursive approach more **naturally** **mirrors the problem** and the results in a program that is easier to understand and debug.

Thank you !!!