Recursion

Let us consider the code …

```
void main() {
 int i, n, sum=0;
 printf("<<"Enter the limit";
 scanf("%d",&n);
printf("<<"\nThe sum is"<<fnSum(n);
}

int fnSum(int n){
int sum=0;
for(i=1;i<=n;i++) //loop
 sum=sum+i;
return (sum);
}
```

```
void main(){
 int a[10],n,i;
 printf("<<"\nEnter the limit";
 scanf("%d",&n);
 printf("<<"\nThe sum is"<<sumAll(n);}
------------------------------------------------------------
int sumAll(int x) {
  if(x == 1) //base case
    return 1;
  else
    return sumAll(x-1) + x; //recursive case
}
```

Recursion

- Recursion is the property that when a called function calls itself.
- It is useful for many tasks, like sorting or calculate the factorial of numbers.
- For example, to obtain the factorial of a number (n!) the mathematical formula would be:

  n! = n * (n-1) * (n-2) * (n-3) ... * 1//recurrence formula

  more precisely, 5! (factorial of 5) would be:

  5! = 5 * 4 * 3 * 2 * 1 = 120

Factorial of a natural number– a classical recursive example

$$\mathrm{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \mathrm{fact}(n-1) & \text{if } n > 0 \end{cases}$$

So factorial(5)

= 5* factorial(4)

= 4* factorial(3)

= 3*factorial(2)

= 2* factorial(1)

= 1*factorial(0)

= 1

Factorial- recursive procedure

```
long factorial (long a) {
  if (a ==0) //base case
    return (1);
   return (a * factorial (a-1));
 }
```

```
#include <iostream.h>
void main () {
 long number;
 printf("Please type a number: “);
 scanf(“%d”,&number);
 printf(“ number!=%ld “,factorial (number));
}
```

Recursion - How is it doing!

rFact(5) = 120

5 X rFact(4) = 24 = 120

4 X rFact(3) = 6 = 24

3 X rFact(2) = 2 = 6

2 X rFact(1) = 1 = 2

1 X rFact(0) = 1 = 1

Notice that the recursion isn't finished at the bottom --

It must unwind all the way back to the top in order to be done.

factorial(0) = 1
factorial(n) = n * factorial(n-1) [for n>0]

```
long rFact (long a) {
        if (a ==0)
            return (1);
    return (a * rFact (a-1));
}
```

Recursive factorial- Stack trace

Executes factorial(4)

factorial(4)

Recursive factorial

factorial(4)

Step 0: executes factorial(4)

return 4 * factorial(3)

Executes factorial(3)

Recursive factorial

factorial(4)

Step 0: executes factorial(4)

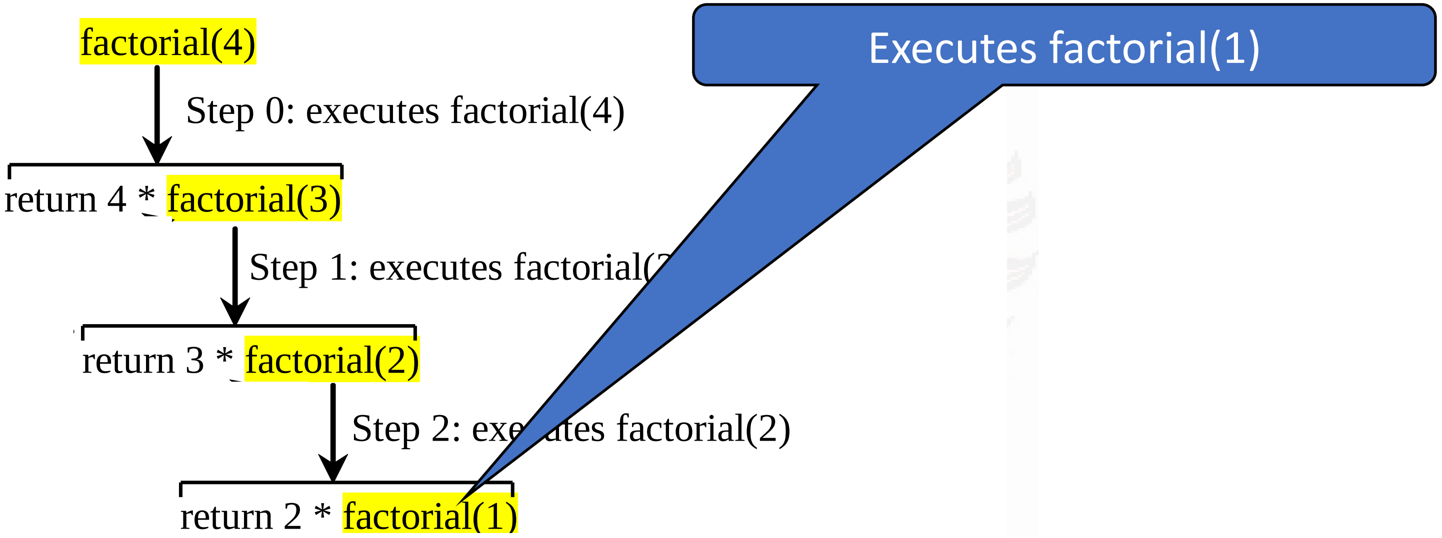
return 4 * factorial(3)

Step 1: executes factorial(3)

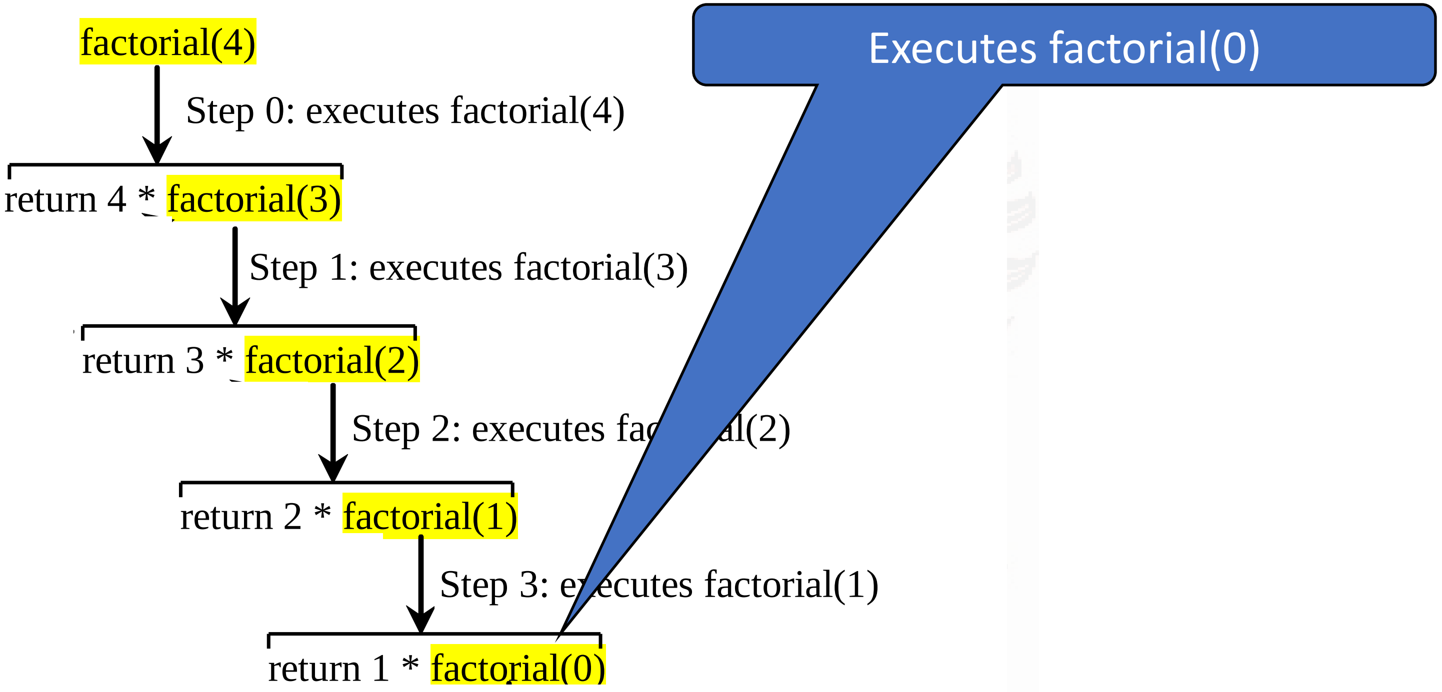
return 3 * factorial(2)

Executes factorial(2)

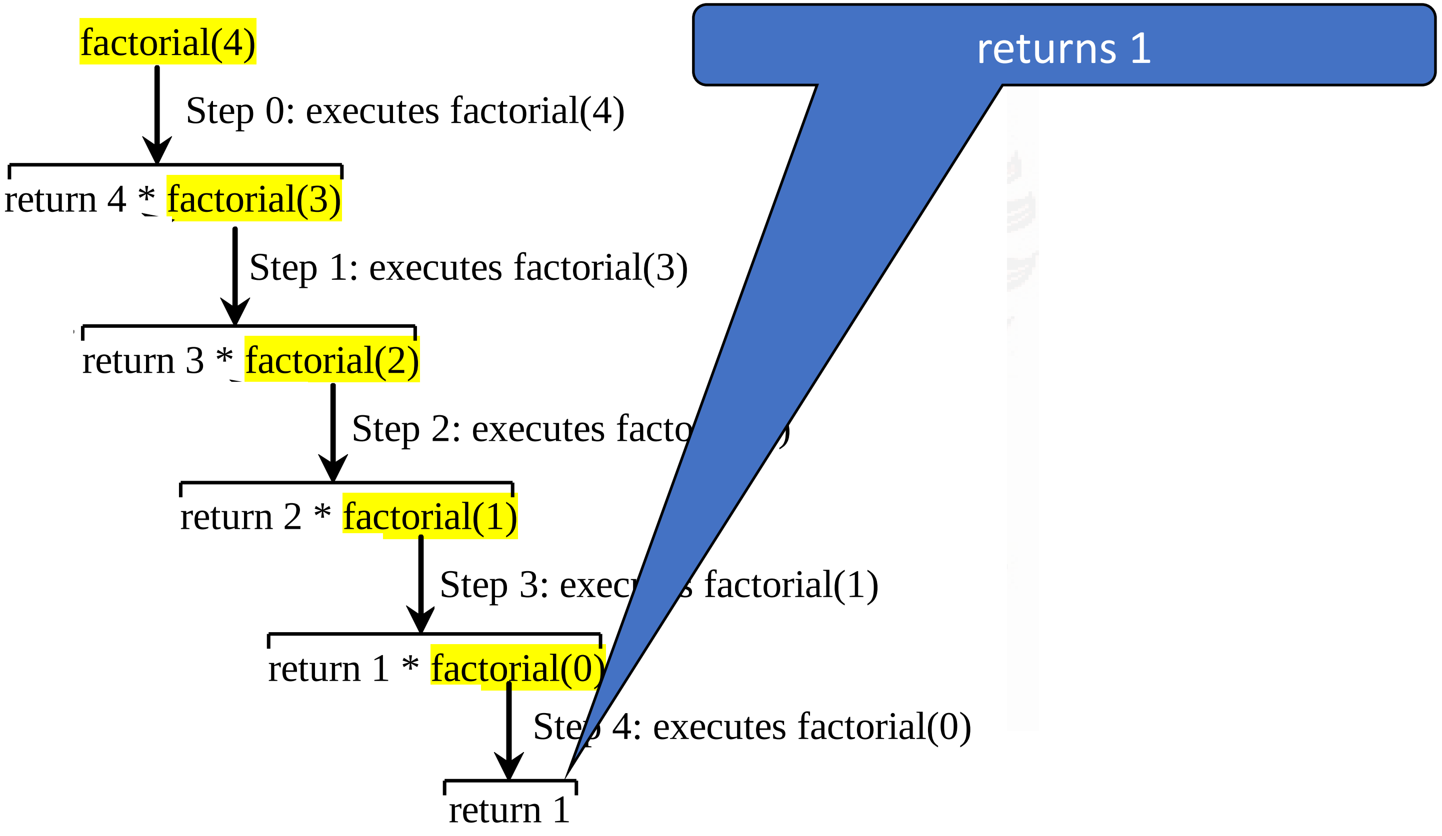
Recursive factorial



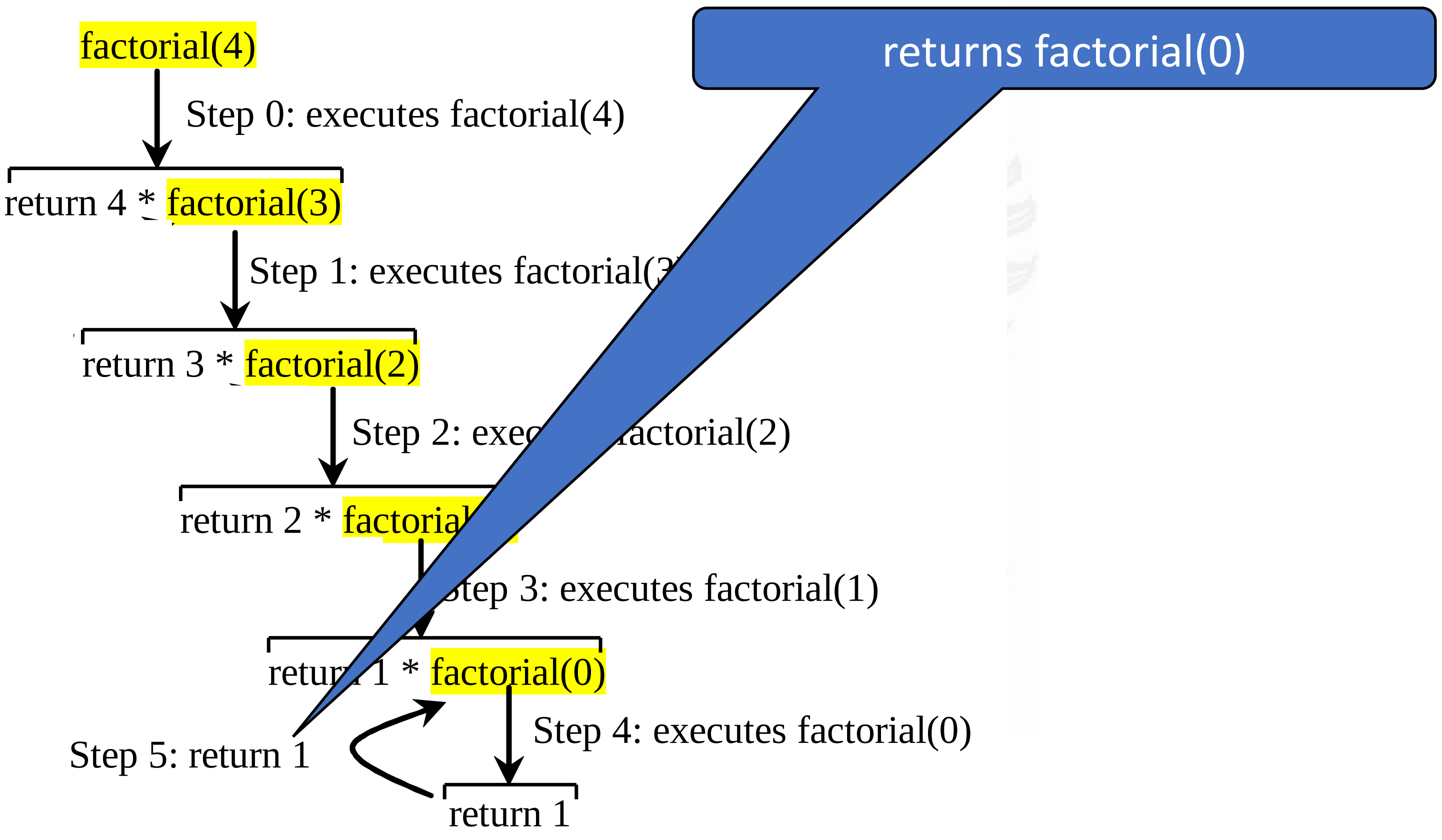
Recursive factorial



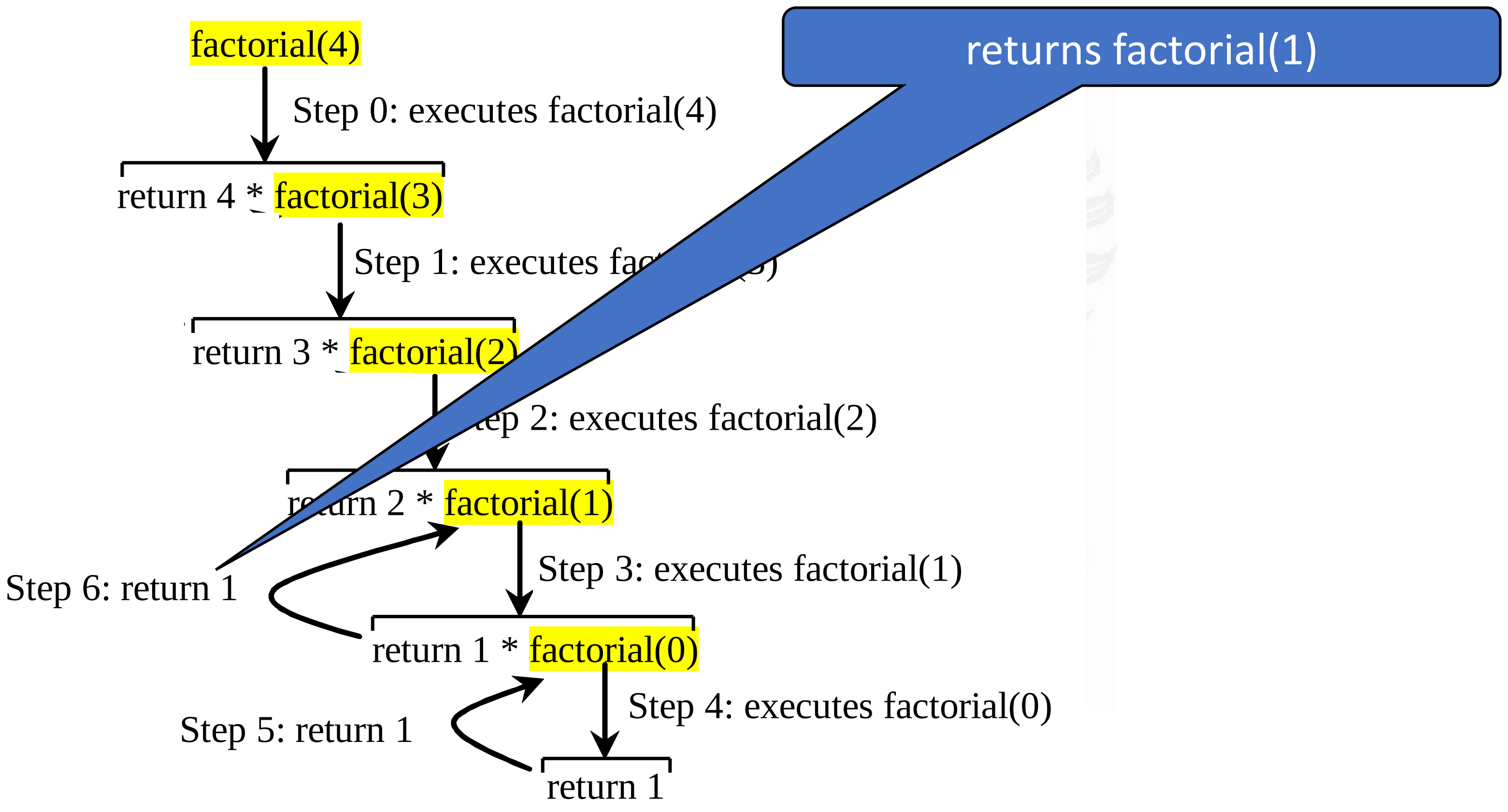
Recursive factorial



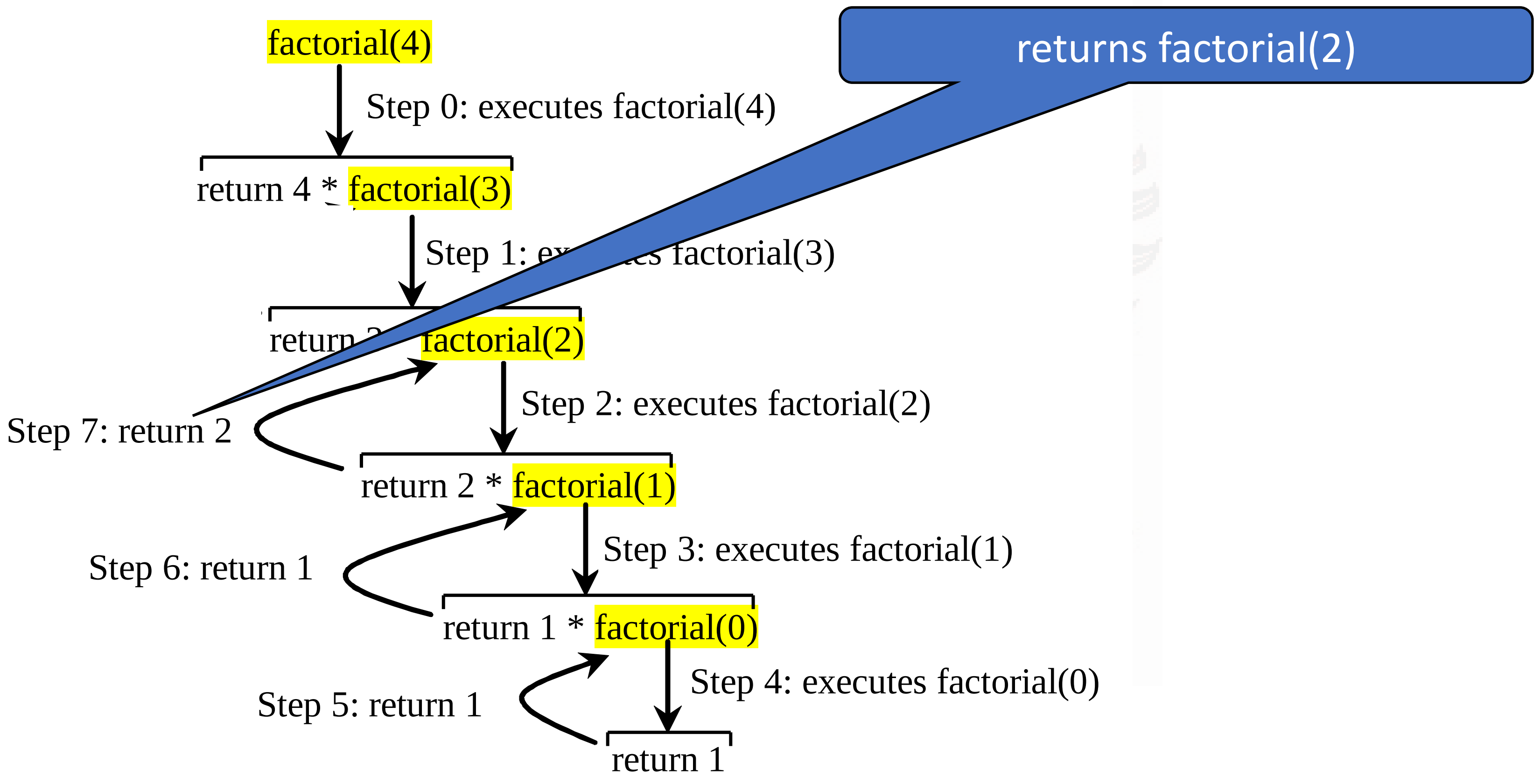
Recursive factorial



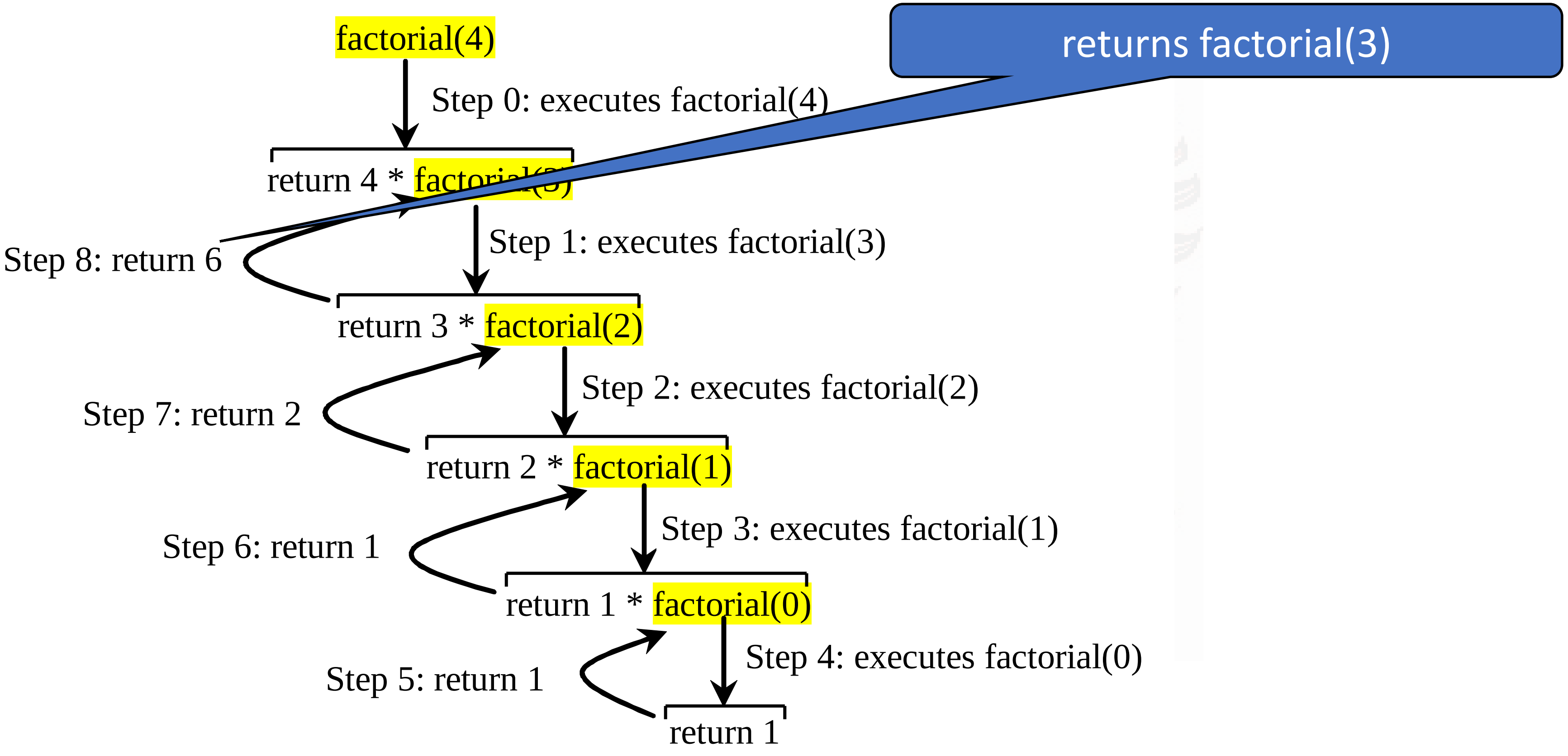
Recursive factorial



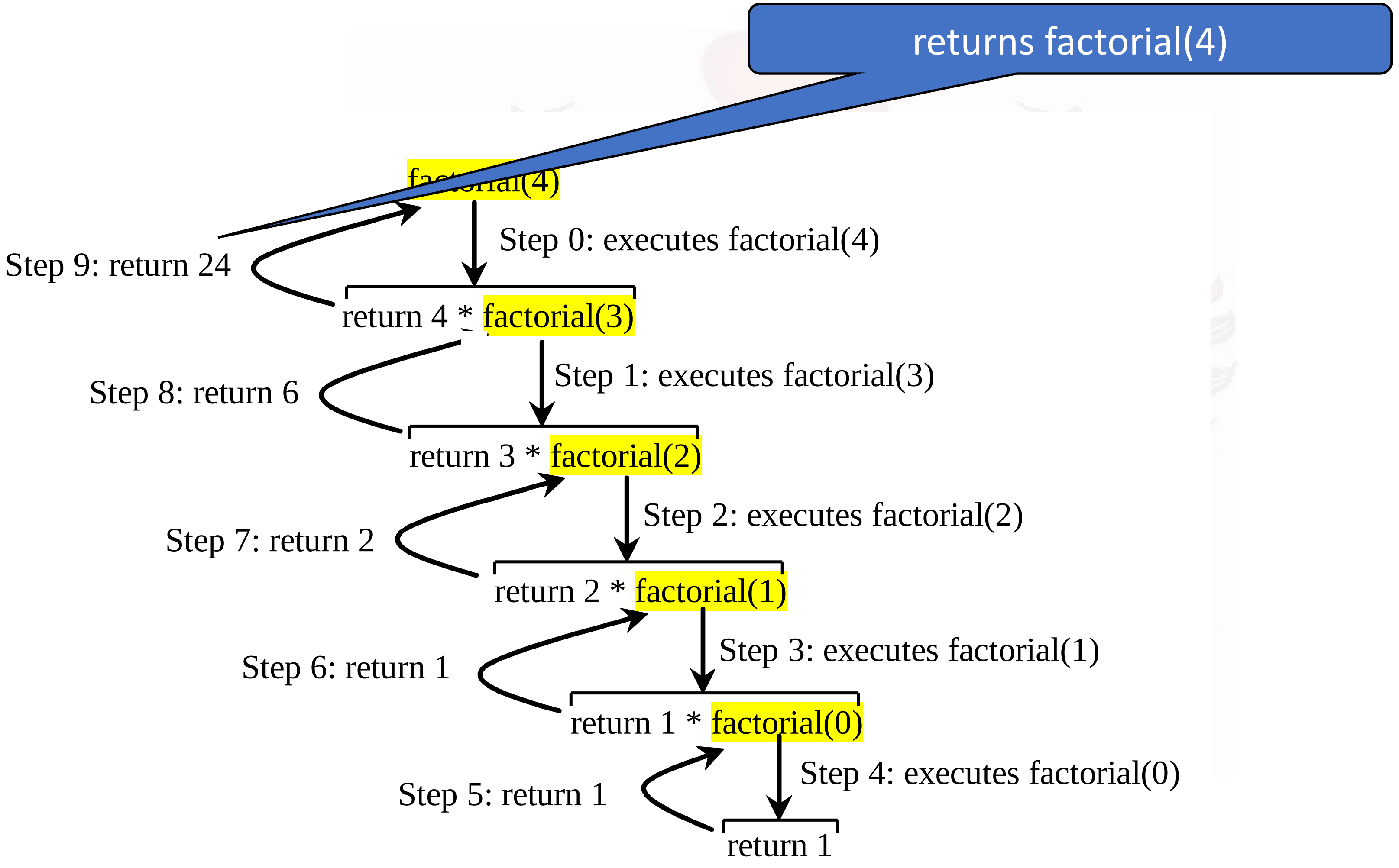
Recursive factorial



Recursive factorial



Recursive factorial



To be solved using recursive fns…

- Write a recursive function to generate n^{th} Fibonacci term. Print first N Fibonacci terms using this function.

  [Hint: Fibonacci series is 0,1, 1, 2, 3, 5, 8 …]
- Write a recursive function to reverse a number.
- Find GCD of two numbers.

  (Ex: GCD of 9,24 is 3)
- Write a function to sort a list of number.

Fibonacci Numbers: Recursion

Fibonacci series is 0,1, 1, 2, 3, 5, 8 …

```
int rfibo(int n)
{
  if (n <= 1)
      return n;
  else
      return (rfibo(n-1) + rfibo(n-2));
}
```

$$\mathrm{fib}(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{if } n >= 2 \end{cases}$$

Output:

n = 4

fib = 3

Fibonacci Series using Recursive fn

```
int rfibo(int);

void main(void){
  int n,i, a[20], fibo;
  printf("enter any num to n\n");
  scanf("%d",&n);
  for (i=1; i<=n; i++)
  a[i]= rfibo(i);
  printf(" Fibonacci Series\n");
   for(i=1; i<=n; i++)
   printf("%d\n",a[i]);
}
```

```
for (i=1; i<=n; i++){
  fibo = rfib(i);
  printf("%d\n",fibo);}
```

GCD: Recursion

- The *greatest common divisor* (gcd) of two positive integers is the largest integer that divides evenly into both of them.
- For example, the greatest common divisor of 24 and 9 is 3 since both 24 and 9 are multiples of 3, but no integer larger than 3 divides evenly into 24 and 9.

- We can efficiently compute the gcd using the following property, which holds for positive integers *p* and *q*:

  If $p > q$,

  the gcd of p and q is the same as

  the gcd of q and p % q

```
int gcd(int x, int y)
 {
    if (x == 0)
       return (y);
   if (y==0)
       return (x);
   return gcd(y, x % y);
  }
```

GCD: Recursion

$$\gcd(x,y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x,y)) & \text{if } x \geq y \text{ and } y > 0 \end{cases}$$

```
int gcd(int x, int y)
 {
  if (x == 0)
      return (y);
  if (y==0)
      return (x);
  return gcd(y, x % y);
 }
```

gcd(24,9) ← Control In gcd fn on call

gcd(9,24%9)   gcd(9, 6)
gcd(6,9%6)   gcd(6, 3)
gcd(3,6%3)   gcd(3, 0)
return values   return 3
return 3
return 3
return 3

Output:
x= 24 , y = 9
gcd = 3

Sorting: Recursion

sort(list, n);// call of fn & display of sorted array in main()

```
void sort(int list[], int ln){
int i, tmp, min;
if (ln == 1)
   return;
/* find index of smallest no */
min = 0;
for(i = 1; i < ln; i++)
if (list[i] < list[min])
min = i;
/* move smallest element to 0-th
   element */
tmp = list[0];
list[0] = list[min];
list[min] = tmp;
/* recursion */
sort(&list[1], ln-1); }
```

Output:

Orign. array-: 33 -2 0 2 4

Sorted array -: -2 0 2 4 33

Recursion - Should I or Shouldn't I?

- Pros
  - Recursion is a natural fit for some types of problems
- Cons
  - Recursive programs typically use a large amount of computer memory and the greater the recursion, the more memory used
  - Recursive programs can be confusing to develop and extremely complicated to debug

Stack Program (1)

```
#define SIZE 4
int top = -1, A[SIZE];
void push(int);
int pop();
int topele();
void show();

void push(int ele)
{    if (top == SIZE - 1)
        printf("\nOverflow!!");
    else
    {
        top = top + 1;
        A[top] = ele;  //A[++top]=ele;
    }
}
```

```
int pop()
{ int ele;
    if (top == -1)
    {
        printf("\nUnderflow!!"); return(999);
    }
    else
    {   ele=A[top];
        top = top - 1;
        return(ele);   //return(a[top--])
    }
}
```

Stack Program (2)

```
void show()
{
   if (top==-1) printf("\nUnderflow!!");

   else
   {     printf("\nElements present in the stack: \n");
       for (int i = top; i >= 0; --i) printf("%d ", A[i]);
       printf("\n");
   }
}
int topele()
{
   if (top == -1)
   {    printf("\nUnderflow!!"); return(999);    }
   else return(A[top]);
}
```

```
int main()
{   int choice,x;
    while (1)
    {printf("\n1.Push the element\n2.Pop the element\n3.Show\n4.peek\n5.End\n6.Exit ");scanf("%d", &choice);
        switch (choice)
        {
case 1: printf("\nEnter the element to be added onto the stack: ");
            scanf("%d", &x);     push(x);
            break;
        case 2:      x=pop();
            if(x!=999)  printf("Popped element: %d\n",x);
            break;
        case 3: show();         break;
        case 4:
              x=topele();
              if(x!=999)  printf("stack top element is: %d\n",x);
              break;
        case 5:  exit(0);
        default:  printf("\nInvalid choice!!");
        }
    }
}
```