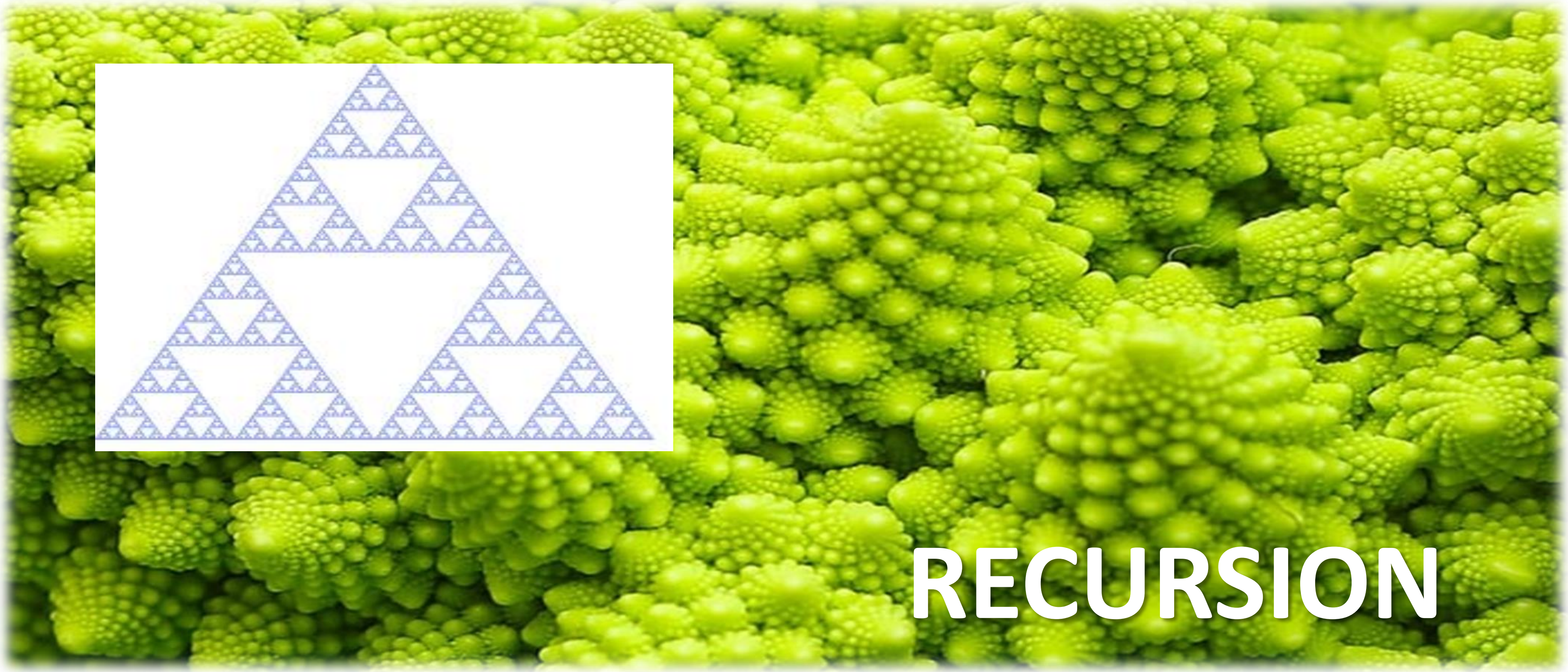


RECURSION

Objectives:

To learn and understand the following concepts:

- ✓ To design a recursive algorithm
- ✓ To solve problems using recursion
- ✓ To understand the relationship and difference between recursion and iteration

Session outcome:

At the end of session one will be able to :

- Understand recursion
- Write simple programs using recursive functions

What is Recursion ?

- Sometimes, the best way to solve a problem is by solving a **smaller version** of the exact same problem first.

- Recursion is a technique that solves a problem by solving a **smaller problem** of the same type.

- A ***recursive function*** *is a function that invokes / calls itself* directly or indirectly.

- In general, code written recursively is shorter and a bit more elegant, once you know how to read it.

- It enables you to develop a natural, straightforward, simple solution to a problem that would otherwise be difficult to solve.

What is Recursion ?

- Mathematical problems that are recursive in nature like factorial, fibonacci, exponentiation, GCD, Tower of Hanoi, etc. can be easily implemented using recursion.
- Every time when we make a call to a function, a stack frame is allocated for that function call where all the local variables in the functions are stored. As recursion makes a function to call itself many times till the base condition is met, many stack frames are allocated and it consumes too much main memory and also makes the program run slower.
- We must use recursion only when it is easy and necessary to use, because it takes more space and time compared to iterative approach.

Steps to Design a Recursive Algorithm

- Base case:
  - It prevents the recursive algorithm from running forever.

- Recursive steps:
  - Identify the base case for the algorithm.
  - Call the same function recursively with the parameter having slightly modified value during each call.
  - This makes the algorithm move towards the base case and finally stop the recursion.

Let us consider the code …

```
int main() {
  int i, n, sum=0;
  printf("Enter the limit“);
  scanf(“%d”,n);
  printf("The sum is %d“,fnSum(n));
  return 0;
}
```

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

Let us consider same code again …

```
int main() {
   int i, n, sum=0;
   printf("Enter the limit“);
   scanf(“%d”, n);
   printf(“The sum is %d“,fnSum(n));
   return 0;
}
```

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

```
int fnSum(int x) {
  if (x == 1) //base case
    return 1;
  else
    return fnSum(x-1) + x; //recursive case
}
```

Factorial of a natural number–

a classical recursive example

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

So factorial(5)

= 5* factorial(4)

= 4* factorial(3)

= 3*factorial(2)

= 2* factorial(1)

= 1*factorial(0)

= 1

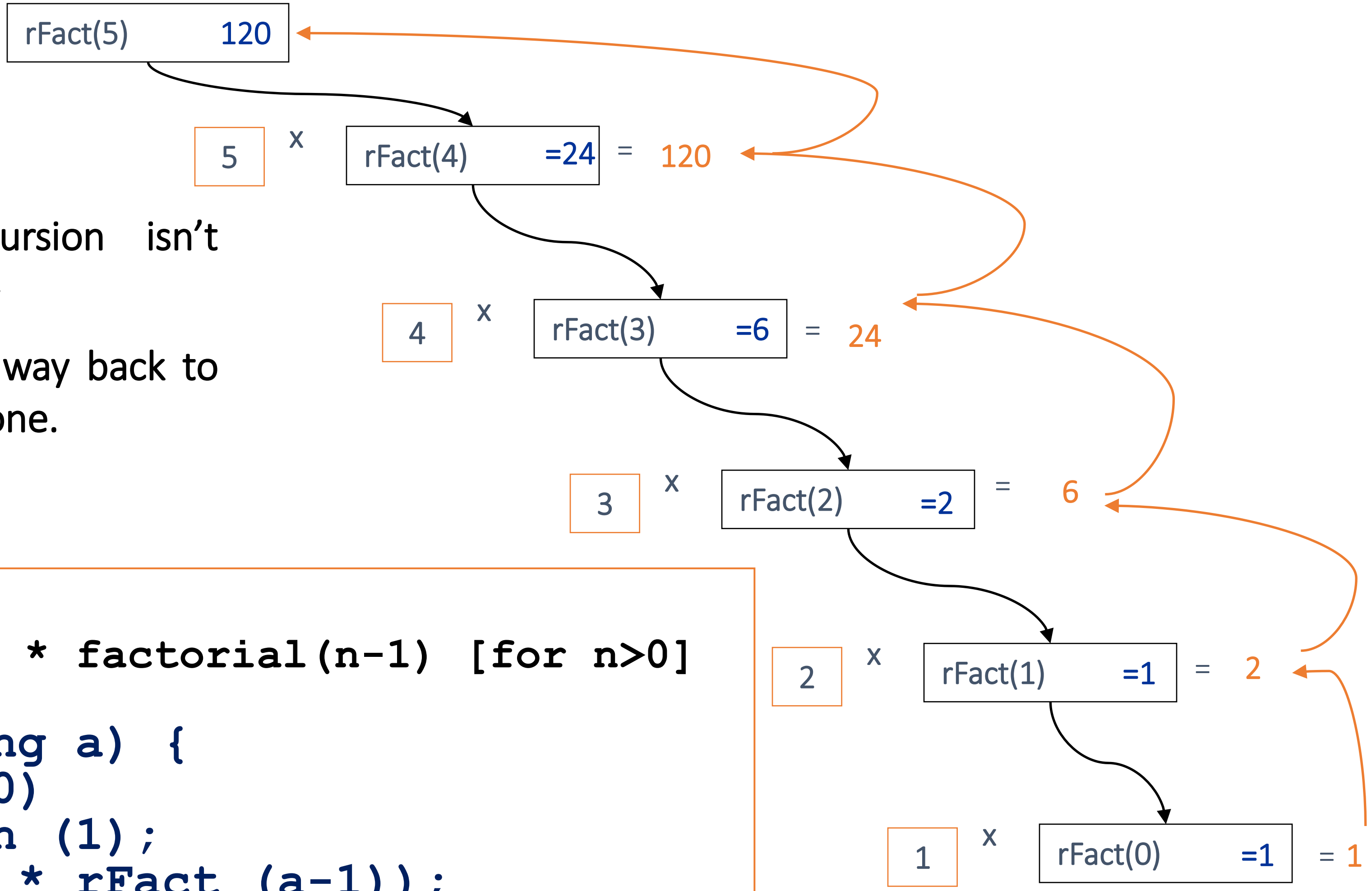
Factorial- recursive procedure

```
#include <stdio.h>

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

          int main () {

               long number;
               printf("Please enter the number: “);
               scanf(“%d”, &number);
               printf(“%ld! = %ld”, number, factorial (number));
               return 0;
          }
```

Output:

```
n = 5
5! = 120
```

Recursion - How is it doing!



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));
 }
```

Fibonacci Numbers: Recursion

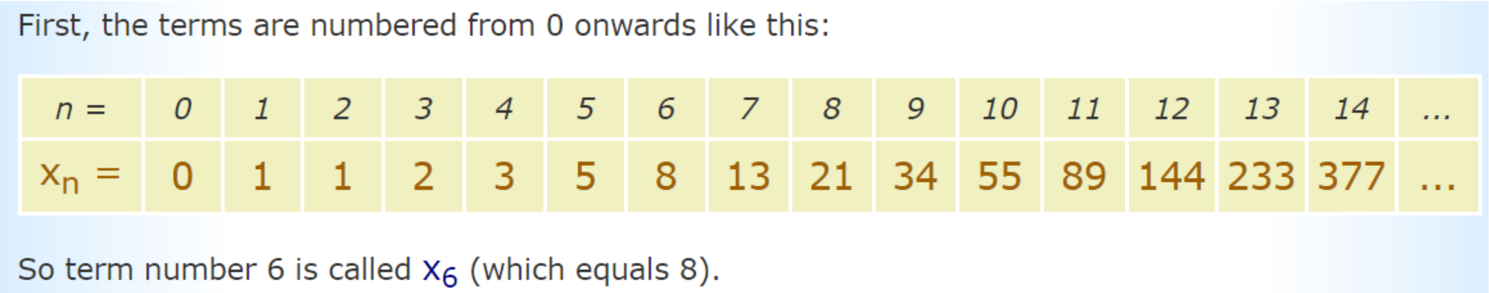
Fibonacci series is 0,1, 1, 2, 3, 5, 8 …

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

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

Output:

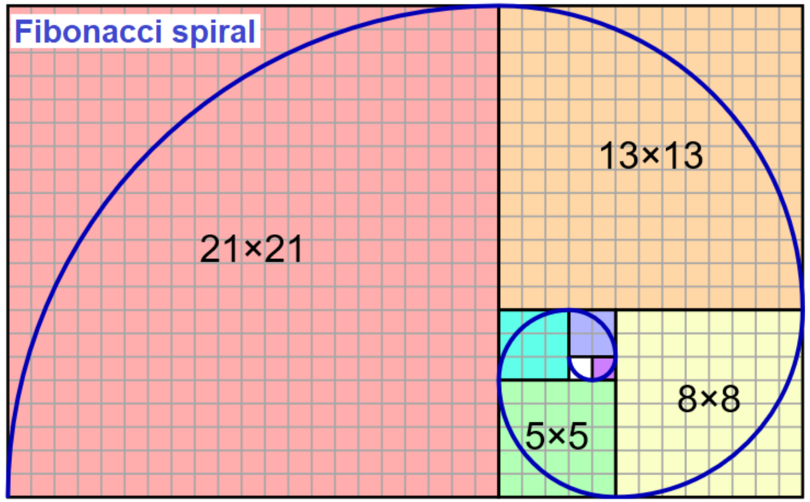
n = 4
fib = 3



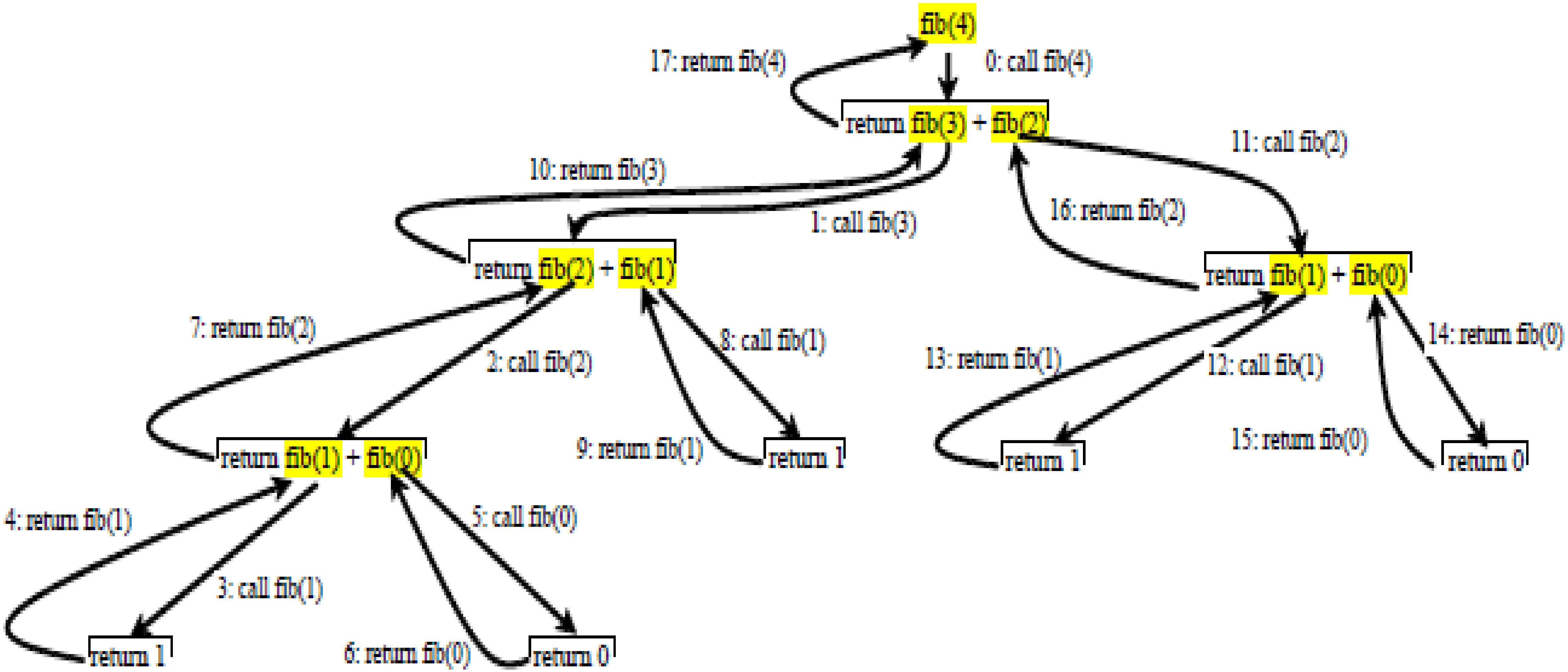
First, the terms are numbered from 0 onwards like this:

n =	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*...*
x_n =	0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	...

So term number 6 is called x_6 (which equals 8).



Recursive Calls initiated by Fib(4)



Fibonacci Series using Recursion

```
int rfibo(int);

int main(void){
 int n,i, a[20], fibo;
 printf("enter any num to n\n“);
 scanf(“%d”, n);
 printf(“Fibonacci series “);
 for (i=1; i<=n; i++) {
  fibo = rfibo(i);
  printf(“%d”, fibo);
 }
 return 0;
}
```

Factorial- recursive procedure – A revisit

$$n! = (n-0) \times (n-1) \times (n-2) \times (n-3) \times \cdots \times 3 \times 2 \times 1$$

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

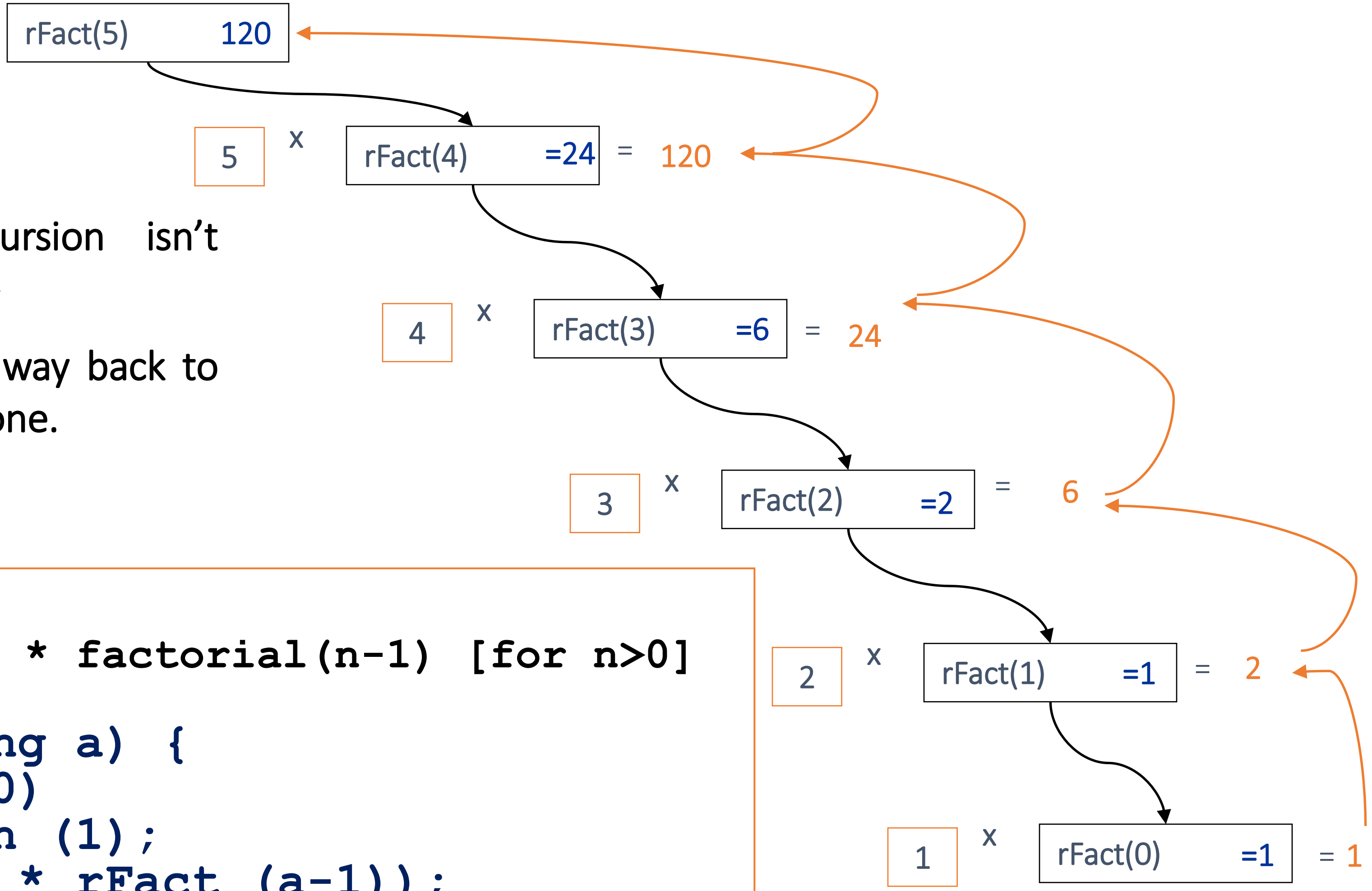
```
long factorial (long a) {
     if (a ==0) //base case
        return (1);
     return (a * factorial (a-1));
 }
```

```
int main () {

    long number;
    printf("Please enter the number: “);
    scanf(“%d”, &number);
    printf(“%ld! = %ld”, number, factorial (number));
    return 0;
}
```

Output:

```
n = 5
5! = 120
```

Recursion - How is it doing!



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));
  }
```

Static Variable:

The value of static variable persists until the end of the program.

Static variables can be declared as

```
static int x;
```

A static variable can be either an internal or external type depending on the place of declaration.

```
void fnStat( );
int main() {
int  i;
for( i= 1; i<=3; i++)
fnStat( );
 return 0;
}
```

```
void fnStat( ){
static int x = 0;
x = x + 1;
printf("x=%d", x);
}
```

Output:

```
        x = 1
        x = 2
        x = 3
```

Reversing a Number

```
#include <stdio.h>
int rev(int);
int main() {
  int  num;
  printf(“enter number)”;
  scanf(“%d”,num);
  printf(“%d”, rev(num));
  return 0;
}
```

```
int  rev(int num){
    static int n = 0;
    if(num > 0)
      n = (n* 10) + (num%10) ;
    else
      return n;
    return  rev(num/10);
}
```

Output:

```
        num = 234
        rev = 432
```

Sorting a list – Recursive

Base Case:

if length of the list (n) = 1

No sorting, return

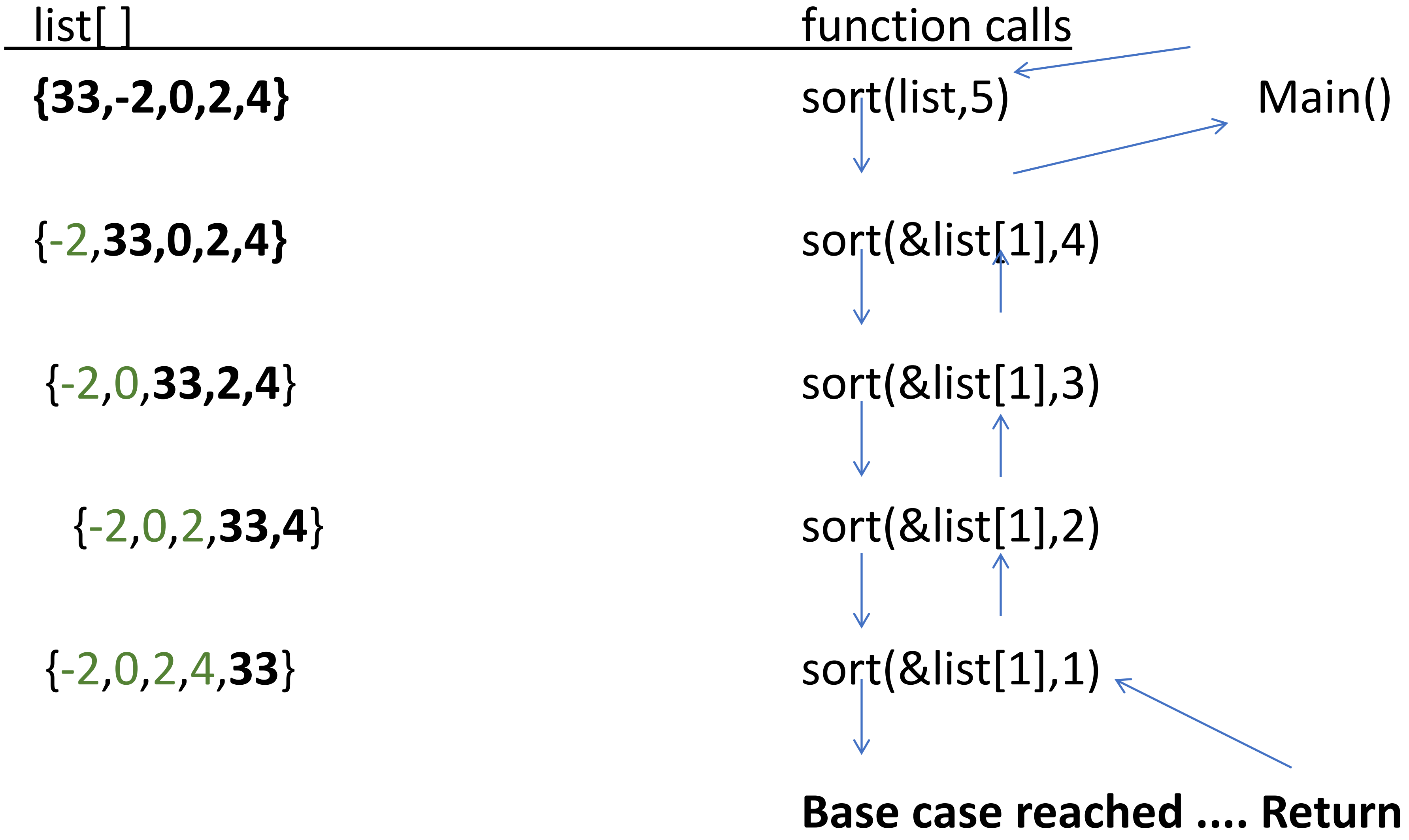
Recursive Call:

1. Find the smallest element in the list and place it in the 0th position
2. Sort the unsorted array from 1.. n-1

```
        sortR(&list[1], n-1)
```

For eg: list [ ] = {33,-2,0,2,4}   n=5

Sorting a list



list[ ]	function calls
{33,-2,0,2,4}	sort(list,5)
{-2,**33,0,2,4**}	sort(&list[1],4)
{-2,0,**33,2,4**}	sort(&list[1],3)
{-2,0,2,**33,4**}	sort(&list[1],2)
{-2,0,2,4,**33**}	sort(&list[1],1)

Main()

Base case reached .... Return

Sorting a list

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

```
int sortR(int list[], int ln){
int i, tmp, min;
if (ln == 1)
   return 0;
/* 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;
/* recursive call */
return sortR(&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 recursive 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

Recursion *vs* Iteration

RECURSION	ITERATION
Uses more storage space requirement	Less storage space requirement
Overhead during runtime	Less Overhead during runtime
Runs slower	Runs faster
A better choice, a more elegant solution for recursive problems	Less elegant solution for recursive problems

Recursion – Do’s

- You must include a **termination** condition or **Base** Condition in recursive function; Otherwise your recursive function will run “forever” or **infinite**.
- Each successive call to the recursive function must be nearer to the base condition.

Summary

- Definition
- Recursive functions
- Problems Solving Using Recursion
- Pros and Cons
- Recursive Vs Iterative Function