RECURSION

What is Recursion?

- A function is said to be RECURSIVE if it calls itself.

Example:

```
        void function_1(int x)
{   int y;
    ……….
        if ( condition )
                function_1(y);
}
```

Function calls it self repeatedly until some specified condition is satisfied.

Two cases

- A recursive definition has two parts:
  - One or more recursive cases where the function calls itself
  - One or more base cases that returns a result without a recursive call
- There must be at least one base case
- Every recursive case must make progress towards a base case

Recursive and Base case

```
int foo(int x)
{  int y,n;
   if (condition)
       y = foo( x );
    else
      return n; …
}
```

Recursive Case

Base Case

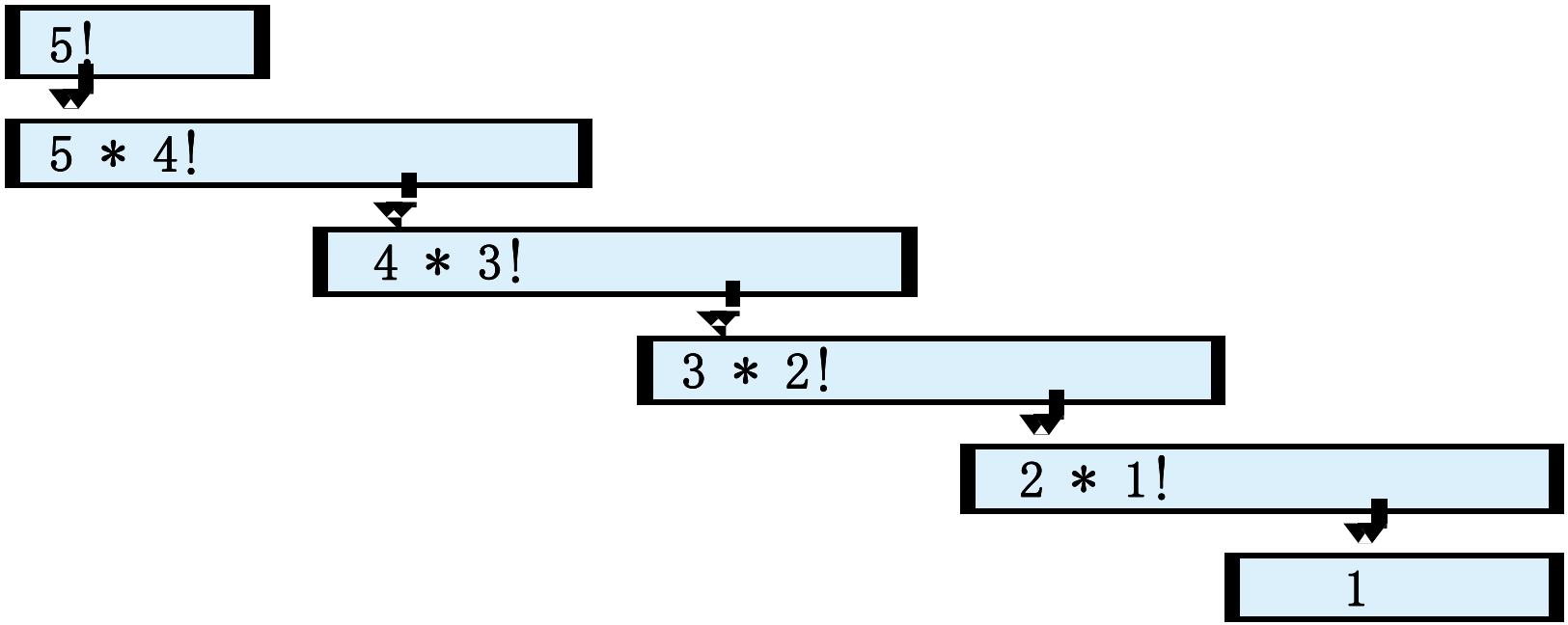
Example 1

Factorial of a number n! is defined mathematically as :

$$n! = \begin{cases} 1 & n = 0 \\ n * (n - 1)! & \text{otherwise} \end{cases}$$

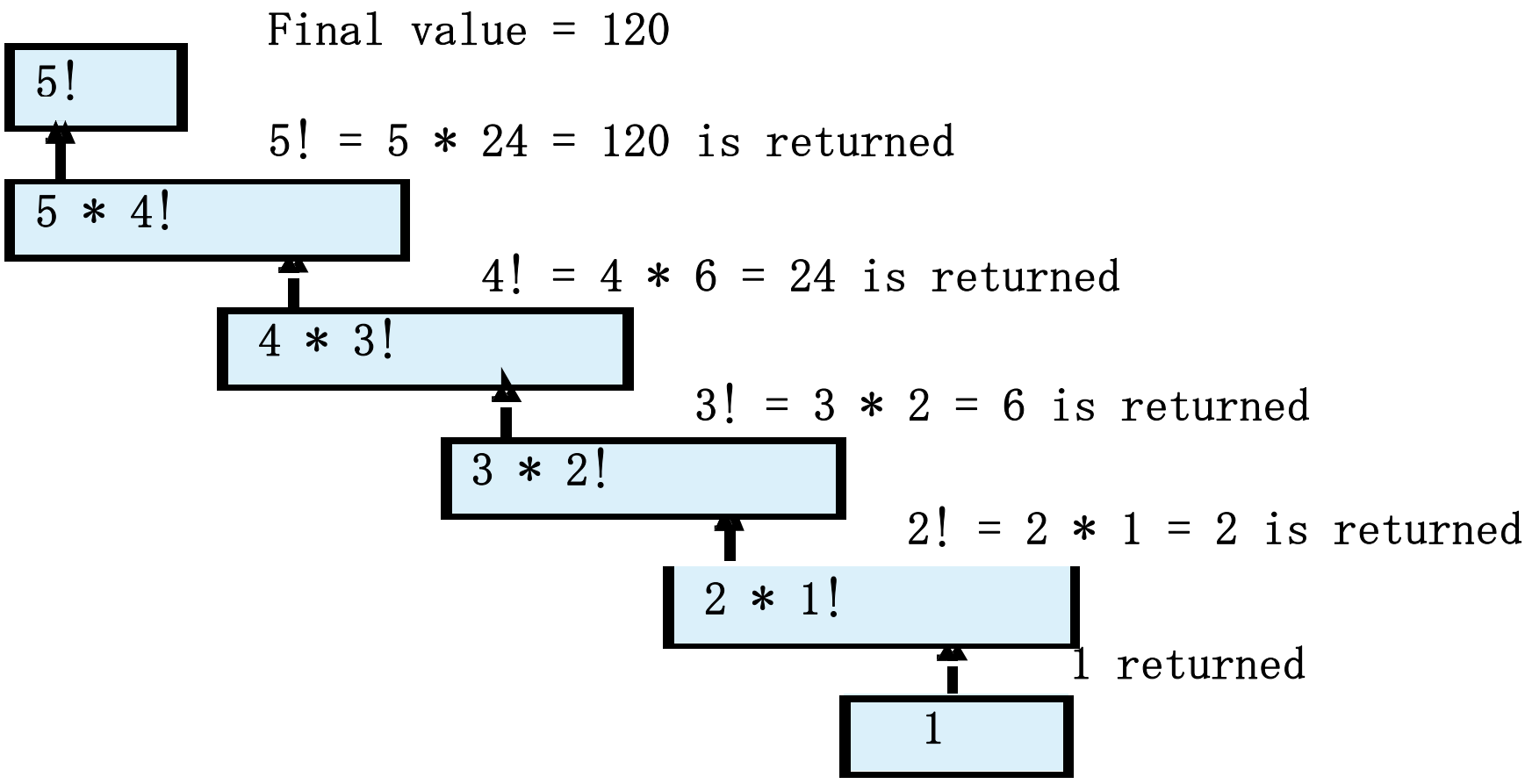
For calculating n! first (n-1)! should be calculated and for that (n-2)! should be computed and so on …

- Factorial of 5 (5 !) ->



Sequence of recursive calls.

Values returned from each recursive call.



Iterative version of factorial program

```
int n, m;
int fact=1;
printf(“Enter the number:”);
scanf(“ %d”,&n);
for( m = n ; m>=1; m- -)
    fact= fact x m;
printf(“The factorial is %d”, fact);
```

Recursive version of factorial program

```
int factorial(int n)
{
    int result;

    if (n == 0||n==1))
            return 1;
    else     result = n * factorial(n - 1);
    return result;
}
```

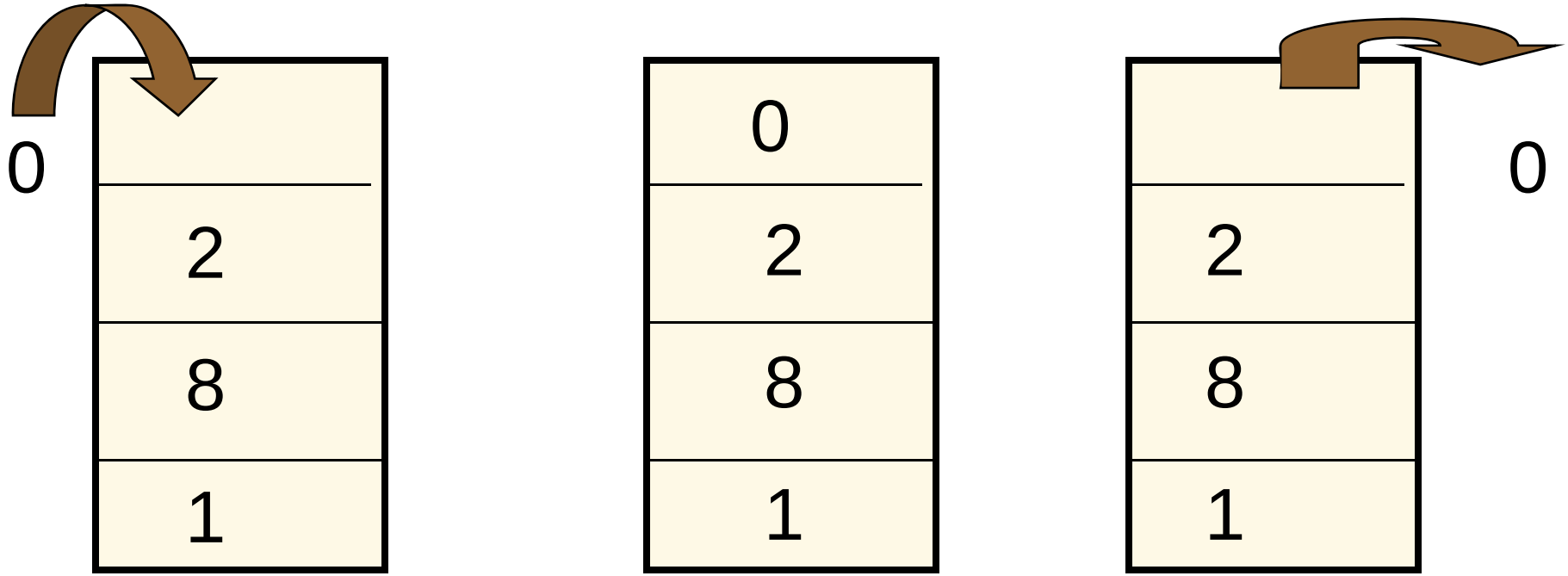
LOGIC

- When a recursive function is executed, the recursive calls are not executed immediately. Rather, they are placed on a stack until the condition that terminates the recursion is encountered.

- The function calls are then executed in reverse order, as they are popped off the stack.

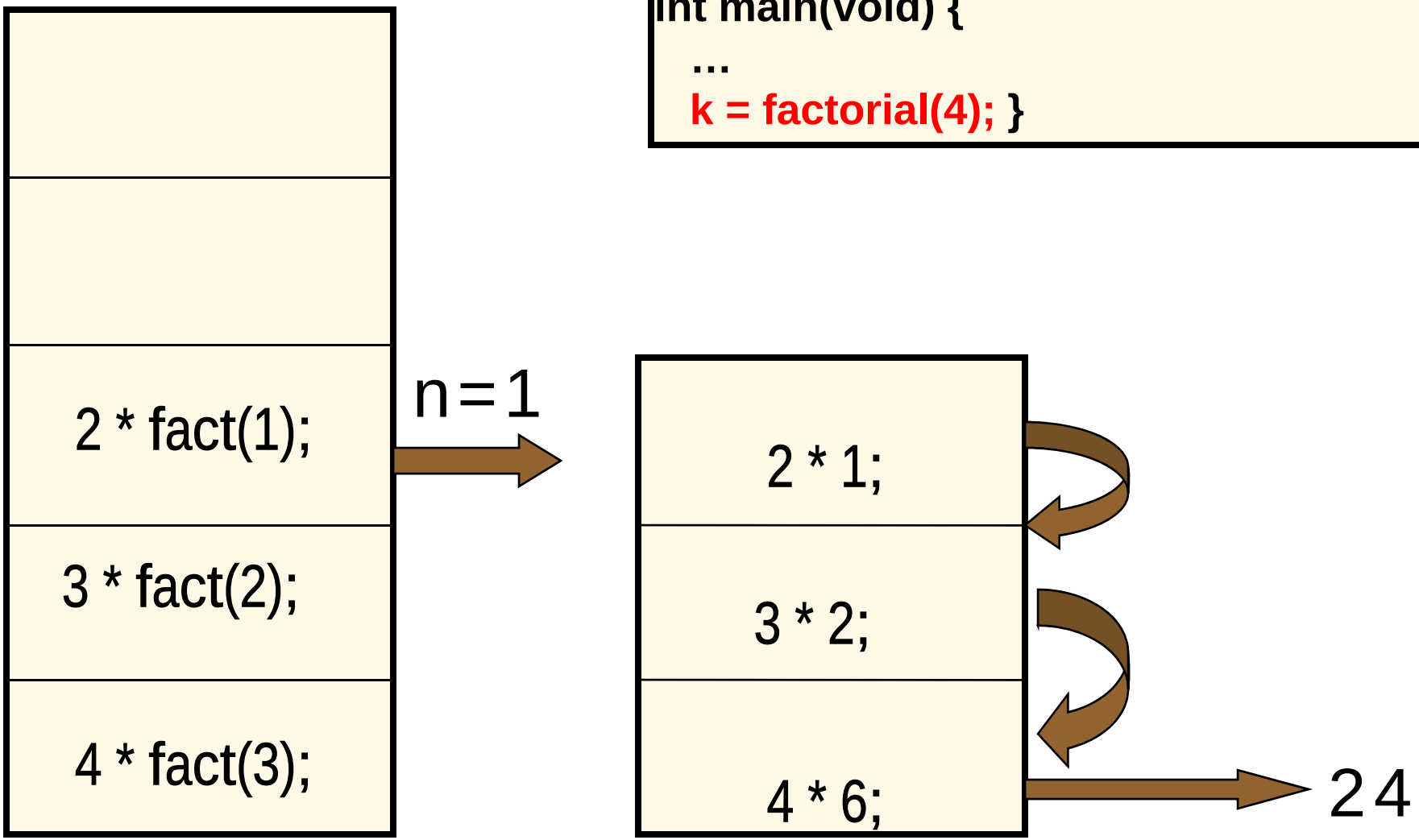
STACK

- A STACK is a LAST-IN, FIRST-OUT data structure in which successive data items are pushed down upon the preceding data items. The data items are removed (popped off) from the stack in REVERSE order.

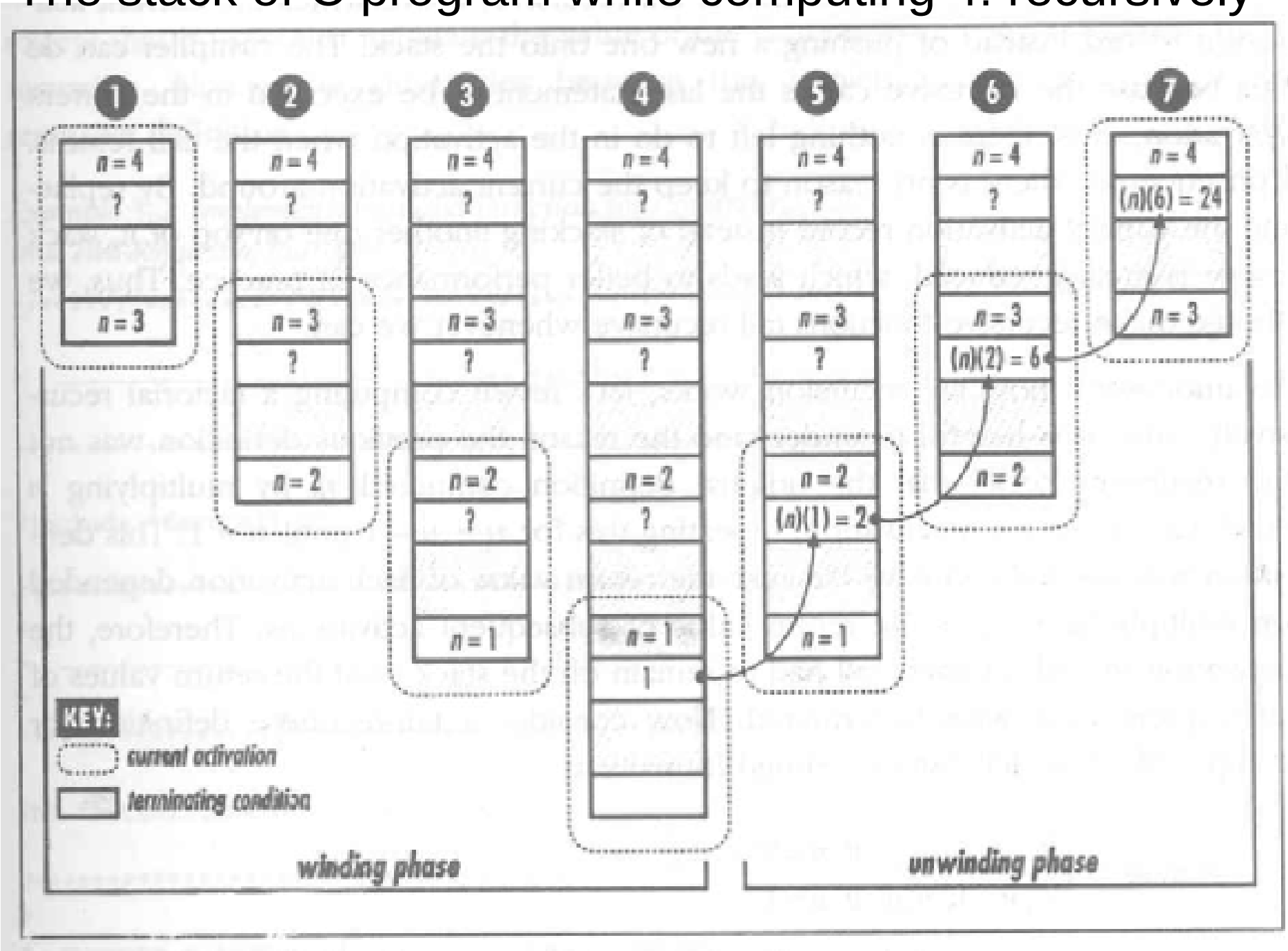


TRACE

```
int factorial(int n){
    if (n <= 1)
        return  1;
    else
        return ( n *   factorial(n - 1)); }
int main(void) {
   ...
   k = factorial(4); }
```



The stack of C program while computing 4! recursively



Example 2

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - **Each term is the sum of the previous two terms.**
- Recursive definition:
  - `F(0) = 0;`
  - `F(1) = 1;`
  - `F( n ) = F( n - 1 ) + F( n – 2 );`

if n = 6 , 6 th term

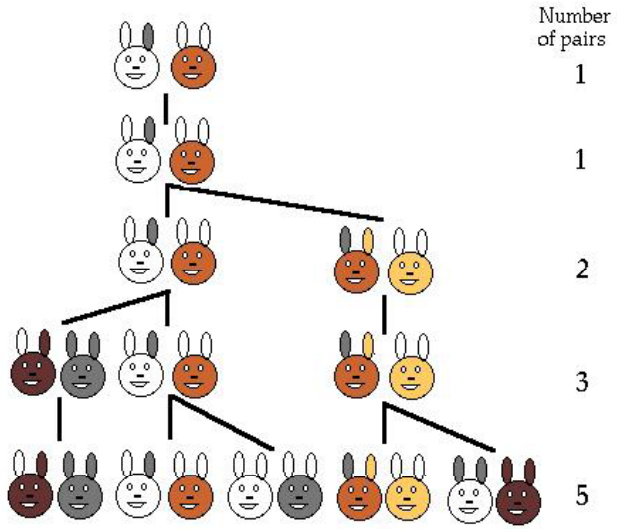
fib(6) = fib (5) + fib (4)

How many pairs of rabbits can be produced from a single pair in a year's time?

- Assumptions:
  - Each pair of rabbits produces a new pair of offspring every month;
  - each new pair becomes fertile at the age of one month;
  - none of the rabbits dies in that year.

- Example:
  - After 1 month there will be 2 pairs of rabbits;
  - after 2 months, there will be 3 pairs;
  - after 3 months, there will be 5 pairs (since the following month the original pair and the pair born during the first month will both produce a new pair and there will be 5 in all).

Population Growth in Nature



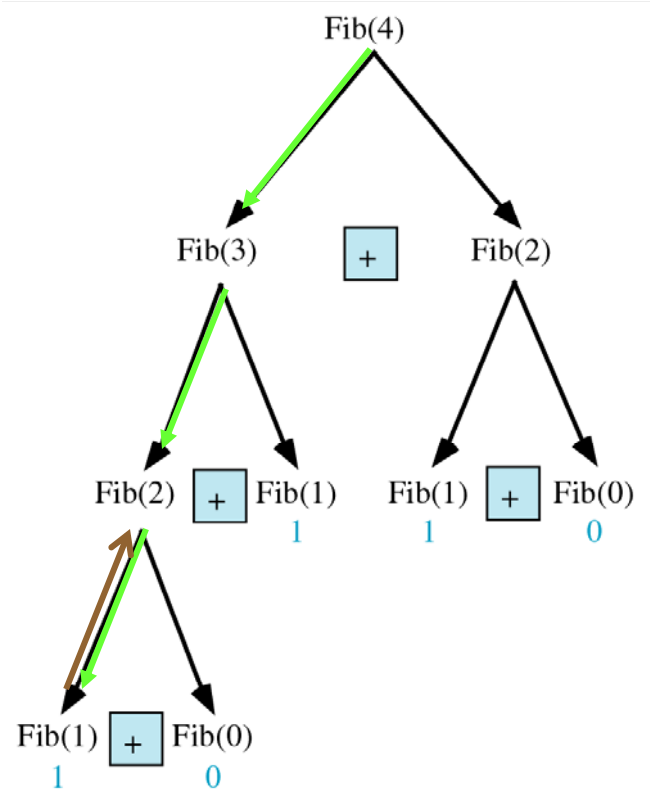
- Leonardo Pisano (Leonardo Fibonacci = Leonardo, son of Bonaccio) proposed the sequence in 1202 in *The Book of the Abacus*.
- Fibonacci numbers are believed to model nature to a certain extent, such as Kepler's observation of leaves and flowers in 1611.

Trace a Fibonacci Number

- Assume the input number is 4, that is, num=4:

```
fib(4):
    4 == 0 ? No;   4 == 1?  No.
    fib(4) = fib(3) + fib(2)
    fib(3):
       3 == 0 ? No; 3 == 1? No.
       fib(3) = fib(2) + fib(1)
       fib(2):
          2 == 0? No; 2==1? No.
          fib(2) = fib(1)+fib(0)
          fib(1):
             1== 0 ? No; 1 == 1? Yes.
              fib(1) = 1;
              return fib(1);
```

```
int fib(int num)
{
    if (num == 0) return 0;
    if (num == 1) return 1;
    return
      (fib(num-1)+fib(num-2));
}
```



Trace a Fibonacci Number

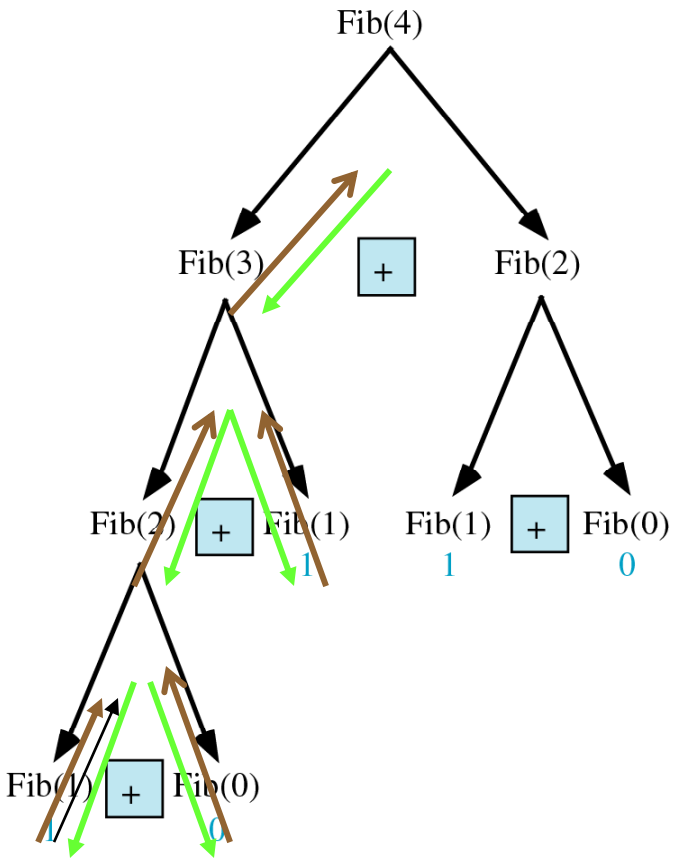
```
fib(0):
0 == 0 ?  Yes.
fib(0) = 0;
return fib(0);
```

```
fib(2) = 1 + 0 = 1;
return fib(2);
```

```
fib(3) = 1 + fib(1)
```

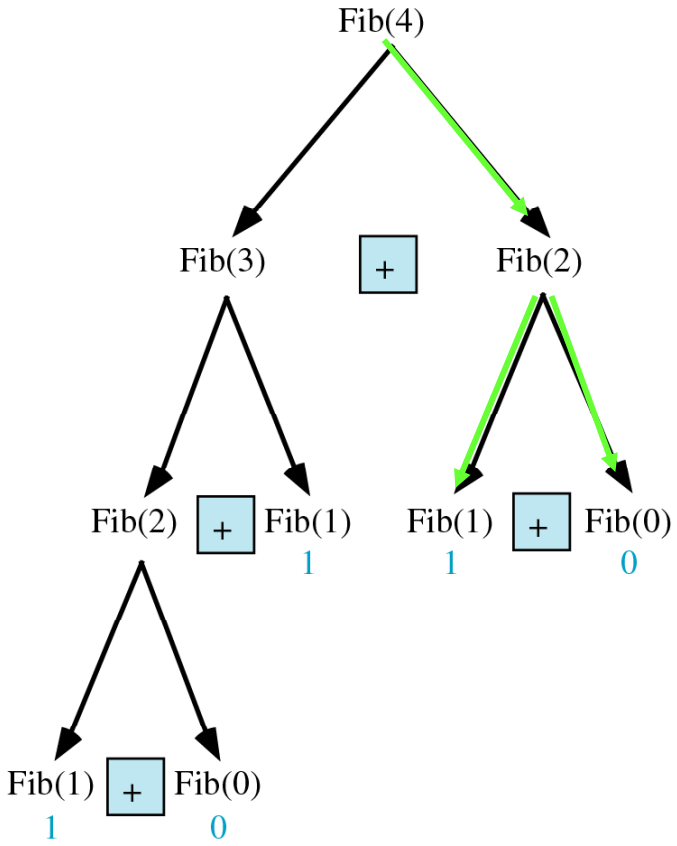
```
  fib(1):
 1 == 0 ? No; 1 == 1? Yes
 fib(1) = 1;
return fib(1);
```

```
fib(3) = 1 + 1 = 2;
 return fib(3)
```



Trace a Fibonacci Number

```
fib(2):
  2 == 0 ? No; 2 == 1?     No.
  fib(2) = fib(1) + fib(0)
  fib(1):
     1== 0 ? No; 1 == 1?  Yes.
      fib(1) = 1;
      return fib(1);
  fib(0):
     0 == 0 ?   Yes.
     fib(0) = 0;
     return fib(0);
   fib(2) = 1 + 0 = 1;
   return fib(2);
fib(4) = fib(3) + fib(2)
       = 2 + 1 = 3;
return fib(4);
```



```
#include <stdio.h>
long fibonacci( long n ); /* function prototype */
/* function main begins program execution */
int main ( )
{
        long result , number ;
        printf( "Enter an integer: " );
        scanf( "%ld", &number );
        result = fibonacci( number );
 printf( "Fibonacci( %ld ) = %ld\n", number, result );
      return 0;
 }
```

```
/* Recursive definition of function fibonacci */
long fibonacci( long n )
 {
    /* base case */
    if ( n == 0 || n == 1 )
          {
              return n;
          } /* end if */
    else
              { /* recursive step */
                  return fibonacci( n - 1 ) + fibonacci( n - 2 );
              } /* end else */

 } /* end function fibonacci */
```

How do I write a recursive function?

- Determine the size factor
- Determine the base case(s)

  (the one for which you know the answer)
- Determine the general case(s)

  (the one where the problem is expressed as a smaller version of itself)
- Verify the algorithm

  (use the "Three-Question-Method")

Three-Question Verification Method

The Base-Case Question:

Is there a non-recursive way out of the function, and does the routine work correctly for this "base" case?

The Smaller-Caller Question:

Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?

The General-Case Question:

Assuming that the recursive call(s) work correctly, does the whole function work correctly?

Handling infinite loop

If we use iteration, we must be careful not to create an infinite loop by accident:

```
for(int incr=1; incr!=10;incr+=2)
   ...
```

Oops!

```
int result = 1;
while(result >0){
   ...
   result++;
}
```

Oops!

Similarly, if we use recursion we must be careful not to create an infinite chain of function calls:

```
int fac(int numb){
   return numb * fac(numb-1);
}
```

Oops! No termination condition

Or:

```
int fac(int numb){
   if (numb<=1)
      return 1;
   else
      return numb * fac(numb+1);
}
```

Oops!

Recursion General Form

- How to write recursively?

```
int recur_fn(parameters){
  if(stopping condition)
     return stopping value;
  // other stopping conditions if needed
  return function of recur_fn(revised parameters)

}
```

Problem Solving Using Recursion

Let us consider a simple problem of printing a message for n times. You can break the problem into two subproblems: one is to print the message one time and the other is to print the message for n-1 times. The second problem is the same as the original problem with a smaller size. The base case for the problem is n==0. You can solve this problem using recursion as follows:

```
void nPrintln(char message[], int times)
{
  if (times >= 1) {
    printf("%s" message);
    nPrintln(message, times - 1);
  } // The base case is n == 0
}
```

Example 3: exponential function

- How to write **exp(int numb, int power)** recursively?

```
int exp(int numb, int power){
     if(power ==0)
          return 1;
     return numb * exp(numb, power -1);
}
```

Recursion vs Iteration

- Recursion can simplify the solution of a problem, often resulting in shorter, more easily understood source code

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls

- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized

- Both can have infinite loops

- Balance
  - Choice between performance (iteration) and good software engineering (recursion)

Performance

Recursive functions are not efficient over the loops,

IN THE POINT OF VIEW OF MEMORY ALLOCATI0N

Each time when you call the function, memory space is reserved for all the local variables in the function.

IN THE POINT OF VIEW OF EXECUTION TIME

Recursive is slower than loops. Because there is an overhead of calling the function repeatedly, apart from the operations performed inside the function.

Deciding whether to use a recursive solution

- When the depth of recursive calls is relatively "shallow"
- The recursive version does about the same amount of work as the nonrecursive version
- The recursive version is shorter and simpler than the nonrecursive solution

Practice Problems

Example 5

Write a recursive function for the Ackermann's Function

$$A(x,y) \equiv \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1,1) & \text{if } y = 0 \\ A(x-1,A(x,y-1)) & \text{otherwise.} \end{cases}$$

Example 6: number of zero

- Write a recursive function that counts the number of zero digits in an integer
- **`zeros(10200)`** returns 3.

```
int zeros(int numb){
   if(numb==0)              // 1 digit (zero/non-zero):
      return 1;             // bottom out.
   else if(numb < 10 && numb > -10)
      return 0;
   else                     // > 1 digits: recursion
      return zeros(numb/10) + zeros(numb%10);
}
```

```
zeros(10200)
zeros(1020)                                  + zeros(0)
zeros(102)                       + zeros(0) + zeros(0)
zeros(10)           + zeros(2) + zeros(0) + zeros(0)
zeros(1) + zeros(0) + zeros(2) + zeros(0) + zeros(0)
```

- Write a recursive function to determine how many factors m are part of n. For example, if n=48 and m=4, then the result is 2 (48=4*4*3)