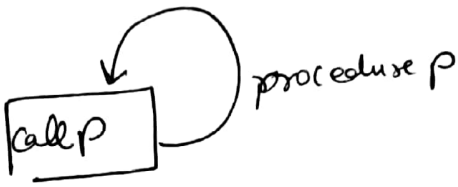
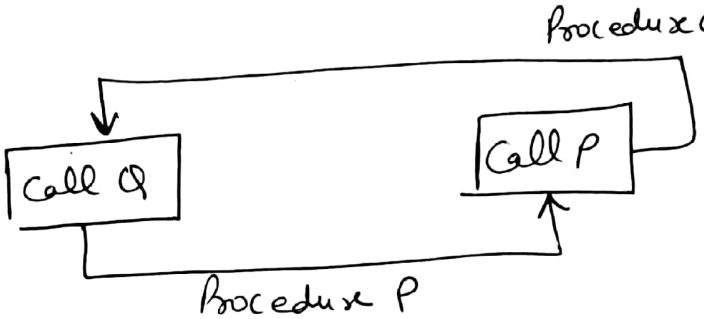
Recursion

Direct Recursion :- If P is a procedure containing a call statement to itself.



Indirect Recursion :- If P is a procedure containing a call statement to a second procedure that may eventually result in a call statement back to the original procedure P.



A recursive procedure must have the following properties :-

1. There must be certain criteria, called base criteria, for which the procedure does not call itself.
2. Each time the procedure does call itself (directly or indirectly), it must be closer to the base criteria.

Factorial Function

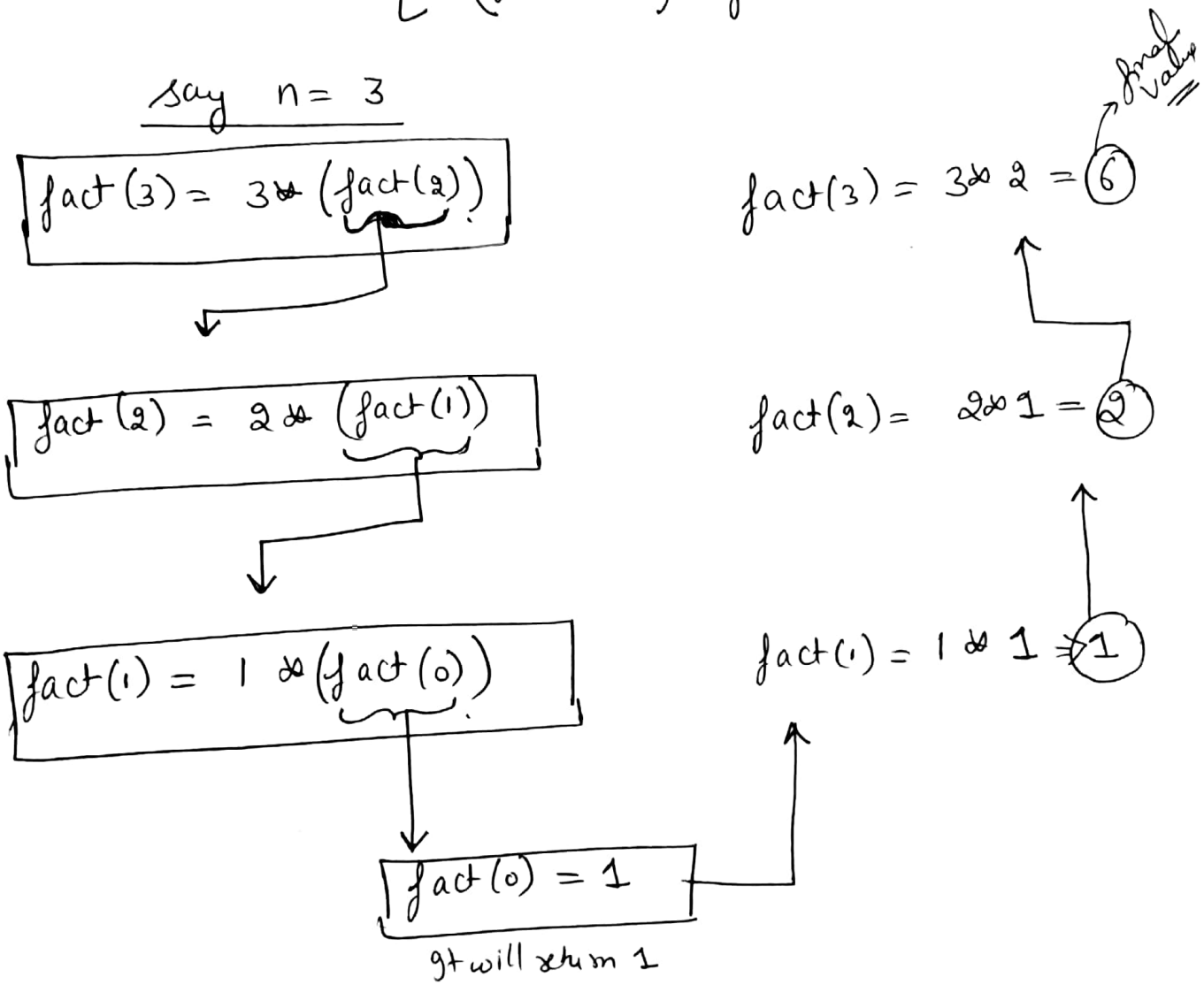
if $n=0$ then $n! = 1$

if $n>0$ then $n! = n \cdot (n-1)!$

Factorial of a number is the product of the integral values from 1 to the number.

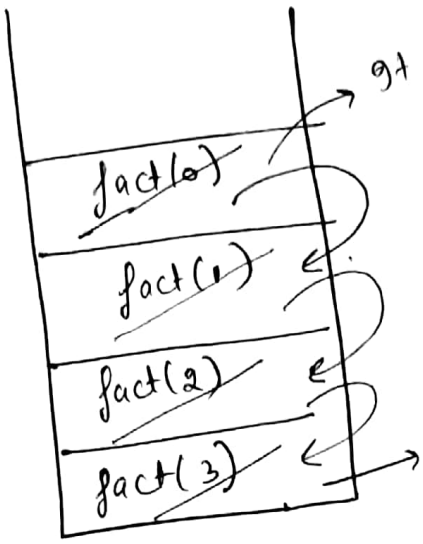
$$fact(n) = \begin{cases} 1 & \text{if } n=0 \\ n * (fact(n-1)) & \text{if } n>0 \end{cases}$$

Say $n = 3$



This I had shown the working of factorial function.

Now Consider the same in form of stack



It will pop out & return the value to the calling function.

Then this function will return the value to the function who called it from main().

Consider the function factorial

```
fact ( int n )
{
(1)   if (n == 0)        ] → (d)
(2)   return 1 ;

(3)   else → (C1)
(4)   return ( n * fact (n-1) ) ;
                 n → (C2)
                 fact(n-1) → T(n-1)
}
```

Say Instruction (1) & (2) takes d time.

Instruction (3) takes C_1 time

Upto Instruction (4) it takes $d + C_1 + C_2 + T(n-1)$

$$T(n) = \begin{cases} d & \text{if } n = 0 \\ \underbrace{d + c_1 + c_2}_{\Downarrow} + T(n-1) & \text{if } n > 0 \end{cases}$$

say we treat all these constant as c

so we have $c + T(n-1)$

Using iterative method, we can solve the complexity for recursive ~~function~~ relation.

$$\begin{aligned} T(n) &= c + T(n-1) \\ &= c + (c + T(n-2)) \\ &= 2c + T(n-2) \\ &= 2c + (c + T(n-3)) \\ &= 3c + T(n-3) \\ &\equiv \end{aligned}$$

After k steps

$$kc + \underbrace{T(n-k)}_{\Downarrow} \quad \text{---------- (1)}$$

when it becomes $T(1)$

$$n - k = 1$$
$$k = n - 1$$

Put this value in eqn (1)

$$T(n) = (n-1)c + T(1)$$

$$T(n) = (n-1)c + T(1)$$

$$= nc - c + T(1)$$

$$= nc - c + (d) \rightarrow \text{we had already defined this time}$$

$$= nc - c + d$$

$\Downarrow$ constant value

& acc. to Big-O we have to ignore the constant

$$= nc$$ again eliminating constant

$$= O(n)$$

Fibonacci Series :-

0, 1, 1, 2, 3, 5, - - - -

If $n=0$ or $n=1$, then $f_n = n$

or If $n>1$, then $f_n = f_{n-2} + f_{n-1}$] Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, - - -

usually denoted by $(f_0, f_1, f_2, - - -)$

That is $f_0 = 0$ and $f_1 = 1$ and each succeeding term is the sum of the two preceding terms.

If next two terms of the sequence are

$8+13 = 21$ and $13+21 = 34$

Procedure for finding the n^{th} term F_N of the Fibonacci sequence follows:-

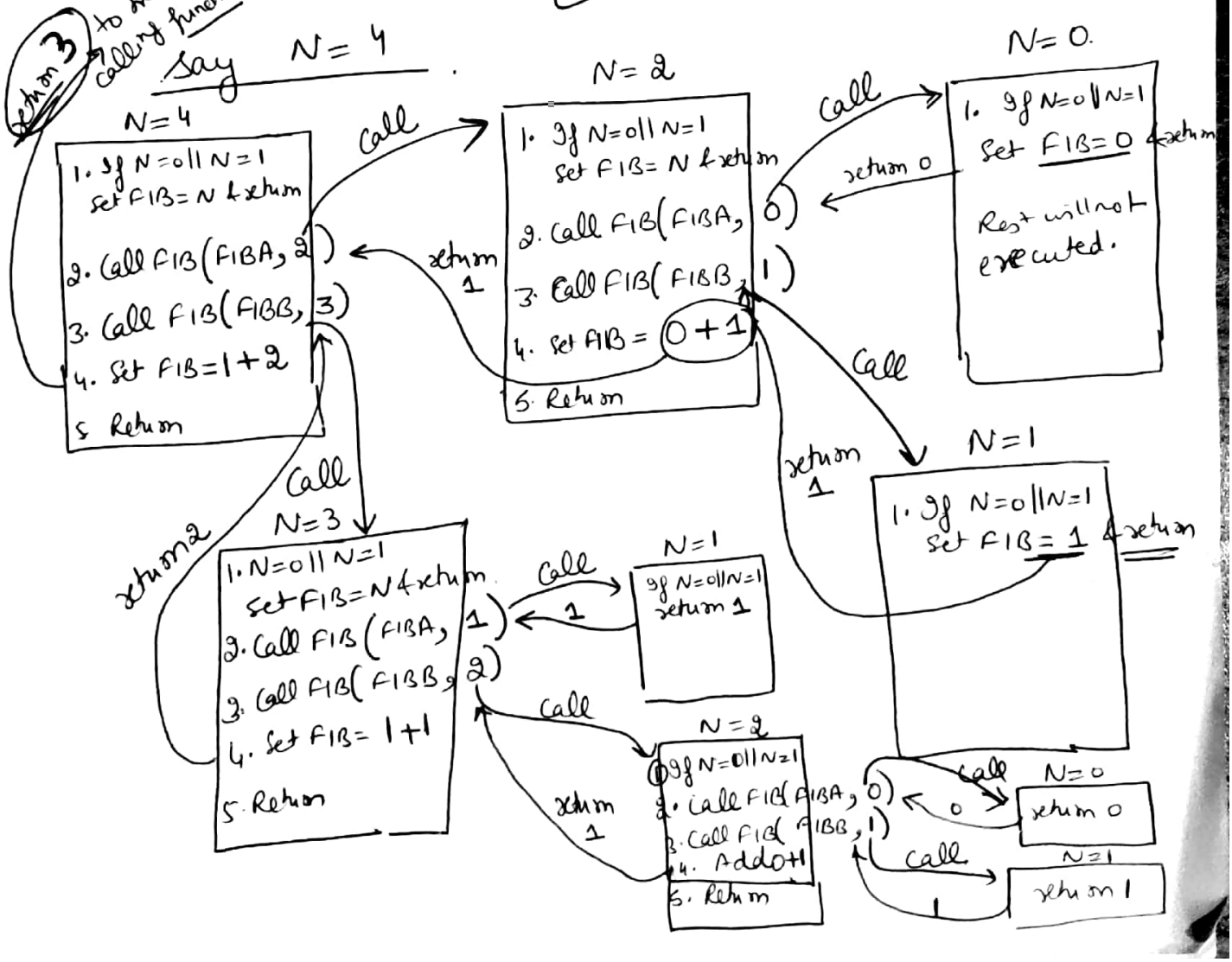
This procedure Calculates F_N & return the value in the first parameter FIB.

FIBONACCI (FIB , N)

1. If N=0 or N=1 then:
   Set FIB = N & Return
2. Call FIBONACCI (FIB A , N-2)
3. Call FIBONACCI (FIB B , N-1)
4. Set FIB = FIB A + FIB B
5. Return.

So the 4th Term is
0, 1, 1, 2, 3

Say N = 4



Recurrence Relation for Fibonacci series

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

$$T(N) = \begin{cases} c & \text{if } N=1 \\ T(N-1)+T(N-2)+c' & \end{cases}$$

$$T(N) = 2T(N-1) + c \quad \text{in general}$$

Apply iterative Method

$$T(N) = 2T(N-1)+c$$
$$= 2(2T(N-2)+c)+c$$
$$= 2^2T(N-2)+2c+c$$
$$= 2^2T(N-2)+c(2+1)$$
$$= 2^2T(N-2)+c(1+2)$$

Again putting

$$= 2^2(2T(N-3)+c)+c(1+2)$$
$$= 2^3T(N-3)+2^2c+c(1+2)$$
$$= 2^3T(N-3)+c(1+2+2^2)$$

=

=

$= 2^3 T(N-3) + C(1+2+2^2)$

After i^{th} step

$2^i T(N-i) + C(\underbrace{1+2+2^2+\cdots 2^i}_{\text{Geometric expression // G.P.}})$

$\Downarrow$ Geometric expression // G.P.

$\Downarrow$ $C(2^{i+1}-1)$

$\Downarrow$ when it becomes $T(0)$

$N-i = 0$

$N = i$

$= 2^N T(N-N) + C(2^{N+1}-1)$

$= 2^N T(0) + C(2^{N+1}-1)$

$= 2^N \cdot 0 + C(2^{N+1}-1)$

$= O(2^N)$ // As constant will be eliminated

To avoid duplicate calls

Previous → p c → current

| 0 | 1 | 1 | 2 | 3 | 5 | 8 |

say this $n = 6$

fib (0, 1, 6);

0 → previous, 1 → current, 6 → n

```
fib( int p , int c , int n)
{
    if ( n == 1 )
        return p;
    else
        return ( fib ( c , p+c , n-1 ));
}
```

∴ For the next time our C will become p and p+C will become our new C & n will be decreased.

So The recurrence relation for this

$$T(N) = \begin{cases} C & \text{if } N = 1 \\ C + T(N-1) & \text{if } N > 1 \end{cases}$$

$\Phi \approx O(N)$

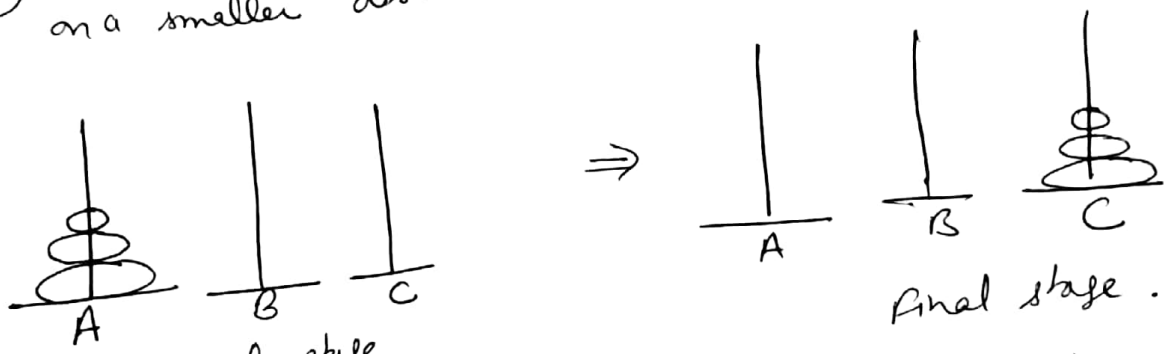
∴ in this code, no duplication of function call will occur.

Tower of HANOI :-

Suppose three pegs, labelled A, B and C are given and suppose on peg A there are placed a finite number n of disks with decreasing size. The objective of the game is to move the disks from peg A to peg C using peg B as an auxiliary.

The rules of the game are :-

1. Only one disk may be moved at a time. Specifically, only the top disk on any peg may be moved to any other peg.
2. At no time can a larger disk be placed on a smaller disk.



Intial stage

Final stage.

We had moved the disk in the following order

(1) A → C (2) A → B (3) C → B

(4) A → C (5) B → A (6) B → C (7) A → C

We can use the technique of recursion to develop a general solution.

(1) Move the top (n-1) disk from Peg A to peg B [By using peg C as an auxiliary]

(2) Move the top disk peg A to peg C : A → C
i.e only one disk

(3) Move the top (n-1) disk from peg B to peg C [By using peg A as an auxiliary]