Recursion - 1

What is Recursion?

(Bookish lang.) → when a function calls itself (directly or indirectly).

Desi lang. → when solution of a bigger problem is dependent upon similar type of smaller problem.

When to apply recursion → जब bigger problem का solution depend करता small वाली similar problem पर।

eg → 2^5 ← Bigger problem

$\Rightarrow \boxed{2^5 = 2 \times 2^4}$

↓

2×2^4 ← chotti problem

$f(n) = 2^n$

$f(n-1) = 2^{n-1}$

$f(n) = 2 \times 2^{n-1}$

$\boxed{f(n) = 2 \times f(n-1)}$

⊙ **Factorial →**

Bigger Problem = 5!

↓ ← smaller problem (similar type)

$5 \times 4!$

B.P (5!) = 5 × (4!) C.P

⊙ **Counting (Reverse) →**

n = 5

o/p → 5, 4, 3, 2, 1

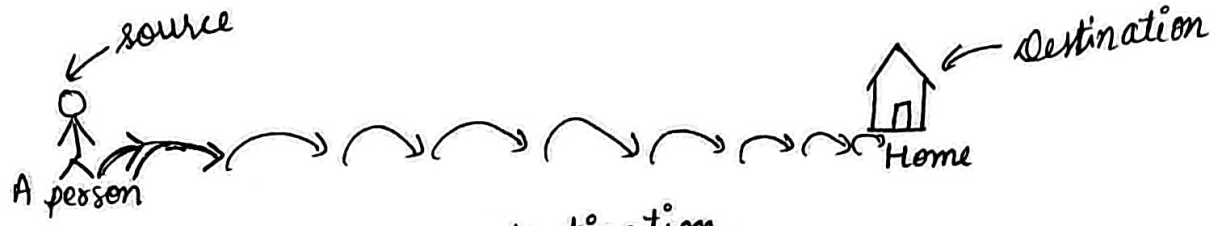
f(n) → print counting from n to 1

f(5) → print counting from 5 to 1

f(5) → print 5 + f(4)

```
void solve(){
    ---
    solve();
}
```

Problem Statement → if bigger problem of similar type depending upon similar prob ⇒ Use recursion



We have to reach to destination.
We will start by taking one step and repeat this step.
And we will stop wherever we reach destination.

eg → Fibonacci Series →

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ----

0+1=1 , 1+2=3 , 8+13=21

n^{th} term = $(n-1)^{th}$ term + $(n-2)^{th}$ term

bigger problem → $f(n) = f(n-1) + f(n-2)$ ← similar type of smaller problems.

Ques i/p ⇒ n
o/p ⇒ 2^n

eg → n = 3
o/p → $2^3 = 8$

n = 6
o/p → $2^6 = 64$

Bigger problem → 2 to the power of n → 2^n

$2^n = 2 \times 2 \times 2 \times 2 - - - - \times 2$ (n times)

$2 \times 2 = 2^2$
$2^2 \times 2^3 = 2^5$
$2^a \times 2^b = 2^{a+b}$

बड़ी problem → $2^n = 2 \times 2^{n-1}$ ← छोटी problem ($n-1$ times)

$f(n) = 2 \times f(n-1)$

⇒ factorial → i/p → n
o/p → n!

eg ⇒ n = 3
o/p = 6
n = 5 → o/p ⇒ 120

Bigger problem is to find n!

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

$n! = n \times (n-1)!$

$n! = n \times (n-1)!$ → smaller prob.
↑ Bigger prob.

$$f(n) = n \times f(n-1)$$

⇒ Counting (Reverse) → i/p = n
o/p ⇒ reverse counting from n to 1.

f(n) → reverse counting from n to 1

$$f(n) \Rightarrow \text{print } n + f(n-1)$$

f(5) ⇒ print 5 ,then f(4)

⊙ Codes →

factorial →

```
int factorial(int n){

    int ans = n * factorial(n-1);
    return ans;
}
```

← we will end up getting segmentation fault.

Calls goes for n = 5, 4, 3, 2, 1, 0, -1, -2, -3, - - - -

⊙ Component of Recursive Code → 3 components

- Mandatory
  - → Base condition ← tells the function when to stop? (Rukna kab hai?)
  - → Recursive Relation ← Relation
- Optional → Processing

now,

```
int factorial(int n){
    //base case.
    if(n==1)
        return 1;
    return n* factorial(n-1);
}
```

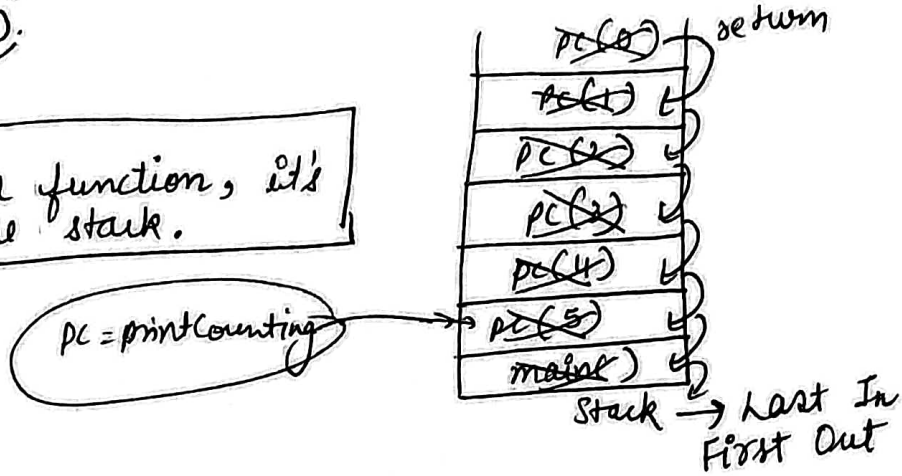
← Recursion call will be made only till 1 and 1 will be returned.

```
int factorial(int n){
    //base case.
    if(n==1)
        return 1;
    int chotiProblemKaAns = factorial(n-1);
    int badiProblemKaAns = n* chotiProblemKaAns;
}
```

```
void printCounting(int n){
    if(n==0)          ← base case
        return;
    // Processing
    cout << n;
    // Recursive Relation
    printCounting(n-1);
}
```

we have to always use return keyword in base case.

whenever we call a function, it's entry is stored in the stack.



let n = 5

5==0 → F

cout<< 5.

funcall for n-1=4

o/p → 5 4 3 2 1

n = 4

4==0 → F

cout<< 4

func call for n-1=3

n = 3

3==0 → F

cout<<3

fun call for 2

n = 2

2==0 → F

cout<< 2

fun call for 1

n=1

1==0 → F

cout<<1

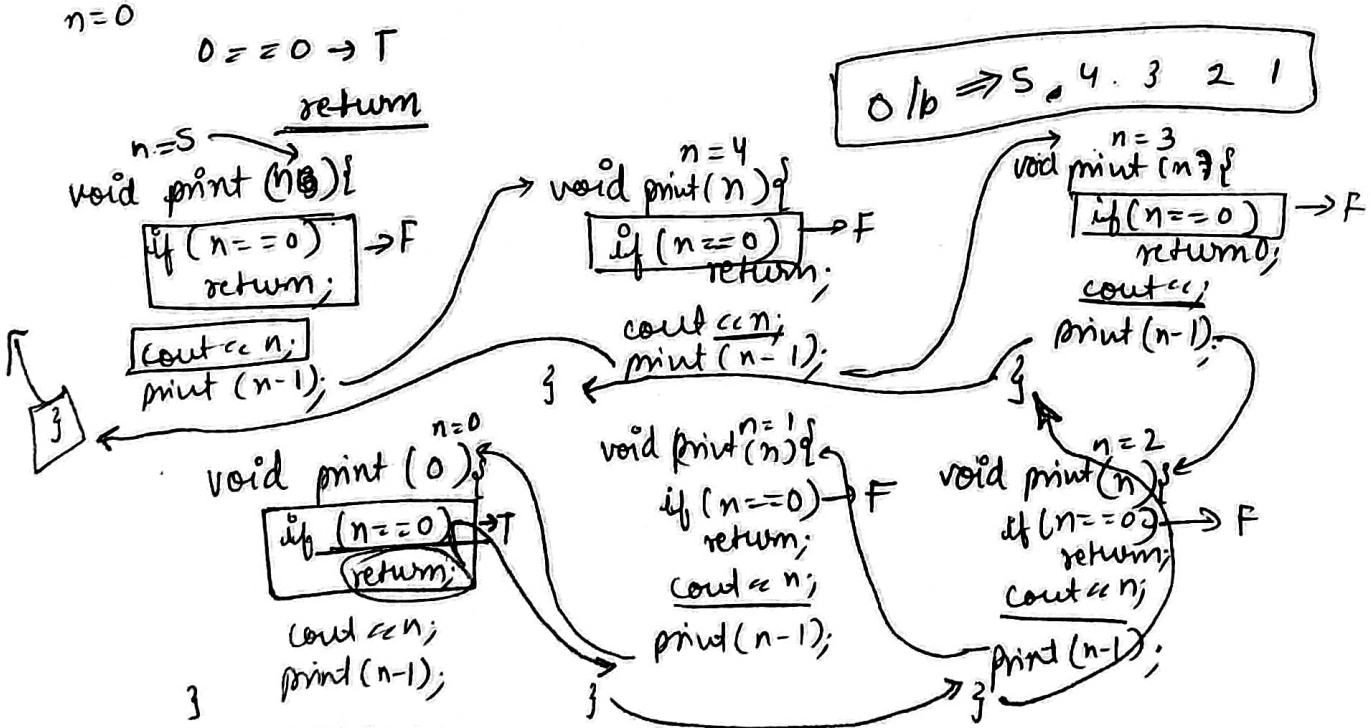
fun call for 0

n=0

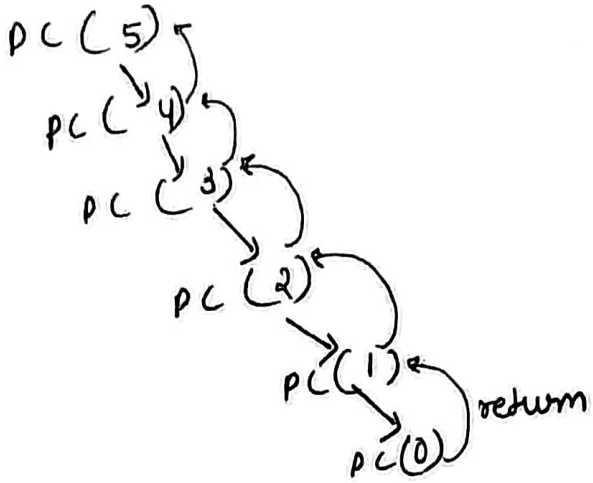
0==0 → T

return

o/p ⇒ 5, 4, 3, 2, 1

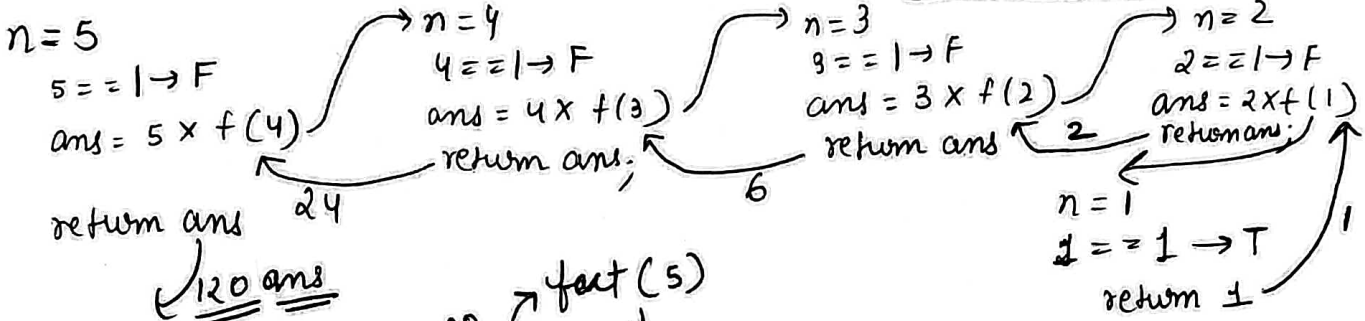


Recur Tree → n=5



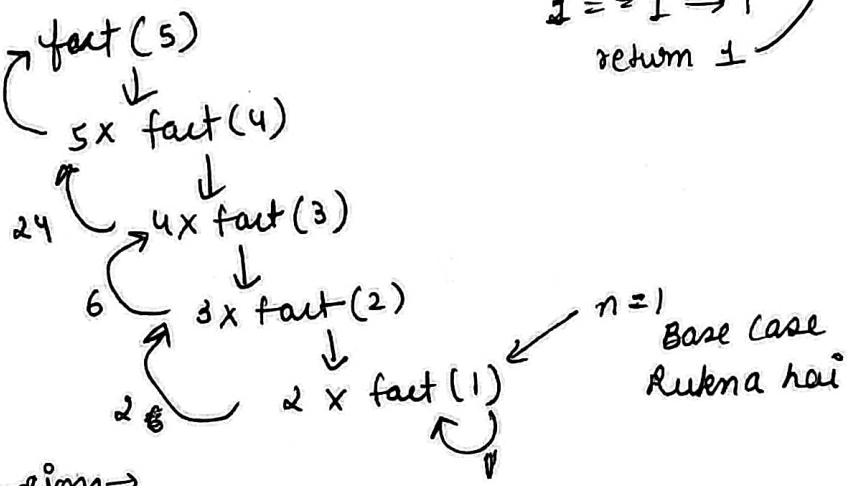
① Factorial →

```
int factorial(int n){
    if(n==1) return 1;
    int ans = n* factorial(n-1);
    return ans;
}
```



120 ans

Recurs Tree → n=5



② Head And Tail Recursion →

```
void solve(){
    // B.C
    // Processing
    // R.R
}
```

Tail Recursion
↓
Recursive relation at the last.

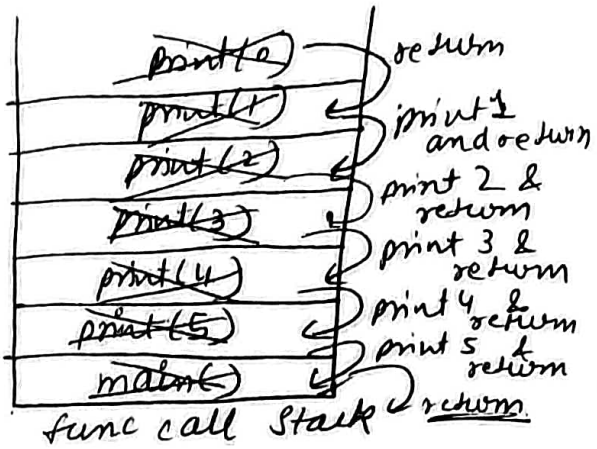
```
void solve()
    // B.C
    // R.R
    // Processing
}
```

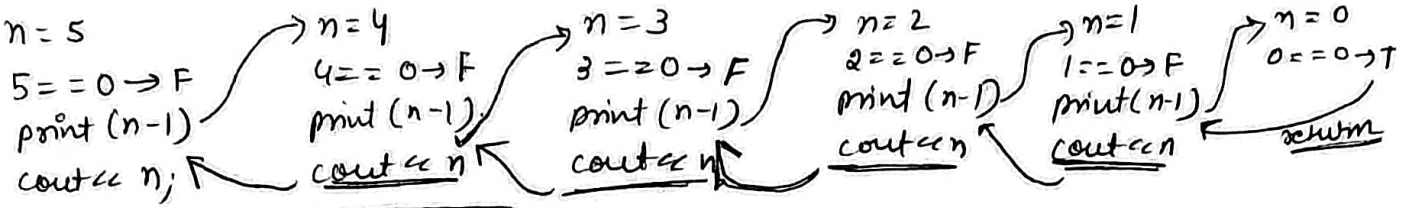
Head Recursion
↓
Recursive relation before processing.

① Counting 1 to n :→

```
void print (int n){
    if (n == 0) return;
    print (n-1);
    cout << n;
}
```

i/p → 5





o/p → 1 2 3 4 5

② Fibonacci series →

This rule can't be applicable on the first two numbers in the series.

n^{th} term = $(n-1)^{th}$ term + $(n-2)^{th}$ term

0, 1, 1, 2, 3, 5, 8, ——— .

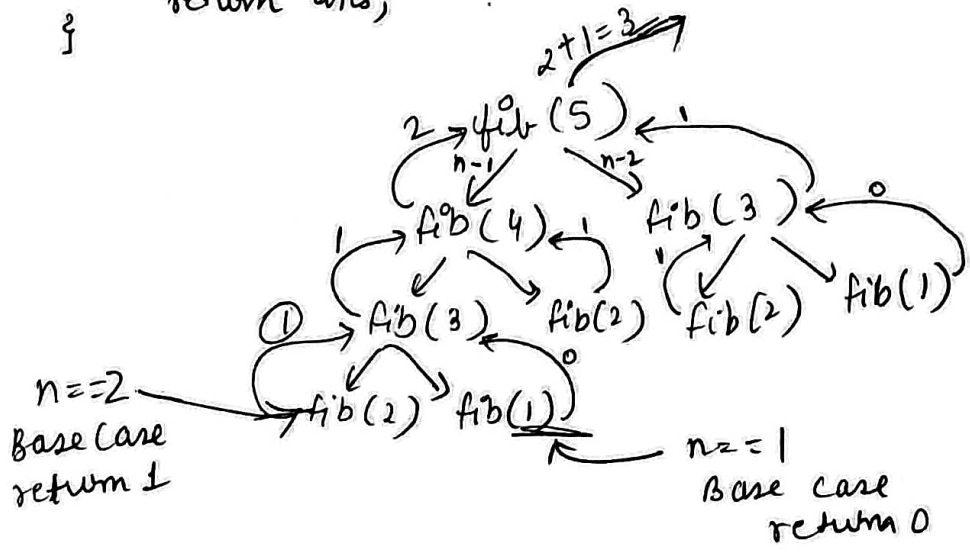
↓ Base cases

Har ek no. pichle 2 no. ka sum hota hai.

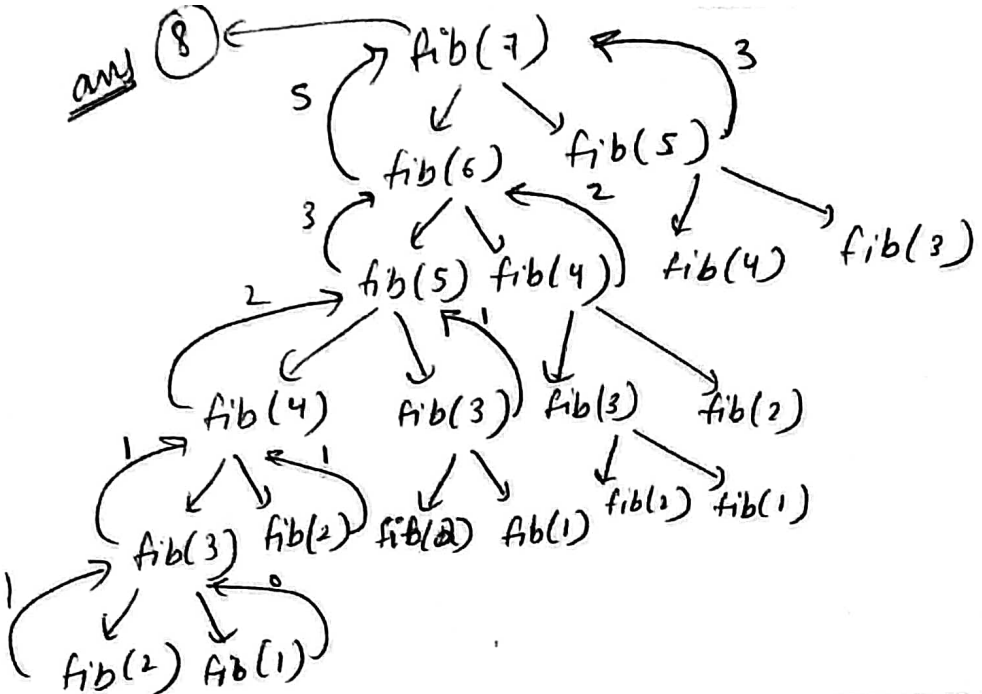
```
int fib( int n){
    if (n == 1)
        return 0;
    if (n == 2)
        return 1;
```

} Base Cases (Here considering 0 is the first term and 1 is the second term

```
// RR → f(n) = f(n-1) + f(n-2)
int ans = fib(n-1) + fib(n-2);
return ans;
}
```



n=7 ans (8)



Magical line → 1 case solve karlo baki recursion sambhal lega

for eg → 5!

$5! = 5 \times 4!$ → Recursion will solve this.

↓
1 case solve
(case will solve)

$fib(n) = fib(n-1) + fib(n-2)$

↖ we only did this addition

this is done by recursion.

Questions →

① Factorial
② Counting (Forward and Backward)
③ Power of 2
④ Fibonacci
⑤ Climb Stairs (Leetcode Ques.)
⑥ Print all digit of a number.
⑦ Max element in a array.
⑧ Min element " "
⑨ Find key in " "
⑩ Find Index of key " "
⑪ Find the no. of times key is existing in the array.