317

03

10/ /2023

Recursion

When a function calls itself directly or undirectly is known as recursion. This is the bookish language.

function calling itself

```
void solve() {

    solve()
}
```

When solution of bigger problem is depending on smaller & same type problem, then here recursion will be applied. This is the way by which we can tell whether recursion will be applied or not.

Ex → Solve 2^5 by recursion

2^5 is bigger problem

$2^5 = 2 \times 2^4$

↳ small problem

↳ Bigger problem

$f(n) = 2^n$

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

$2^n = 2 \times 2^{n-1}$

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

function calling itself

Ex→ Solve 5! by recursion.

5! = 5 X 4!
↳ Bigger problem ↳ Smaller problem
Hence recursion can be applied

Ex→ Reverse counting
n = 5 ⇒ 5 4 3 2 1

f(5) ⇒ print from 5 to 1

f(5) = print(5) + f(4)
↳ Bigger problem
ignore (just for understanding)
print from 4 to 1
↳ Smaller problem

Whenever we will be given a problem statement & if we are able to find a formulae or relation in which bigger problem into smaller problem, then we can apply recursion.

Ex→ Example to understand recursion

Source —— 10 steps ——→ Destination
1 step

Hence we can divide 10 steps into one one steps each & hence recursion can be applied here.

Ex→ Fibonacci Series

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

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

n^{th} term $\Rightarrow f(n)$
$(n-1)^{th}$ term $\Rightarrow f(n-1)$
$(n-2)^{th}$ term $\Rightarrow f(n-2)$

$f(n) = f(n-1) + f(n-2)$] Relation

function calling itself

Hence recursion is applied for fibonacci series also.

Understanding recursion by code

```
int factorial (int n) {
        // Base case

        // Recursive relation
}
```

Our recursive code will have 3 components namely base condition, recursive relation & processing.

1) Base case will tell the recursion to stop. It is a mandatory component
2) Recursive relation is also mandatory component.
3) Processing is like sometimes we have to print the values but it is not a mandatory component.

Code of factorial using recursion

```
int factorial (int n) {
```

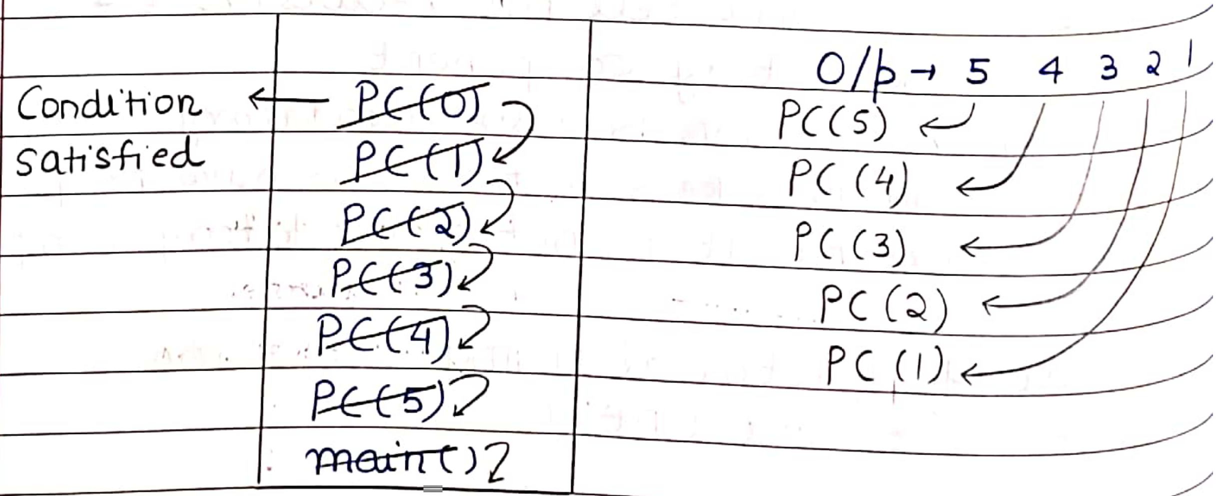
```
    //Base case
    if (n == 0 || n == 1)
            return 1;
    int ans = n * factorial (n-1); //
    return ans;              Recursive relation
}
```

<u>Note</u> → In most of cases, return keyword is used in the base case.

Recursive call stack

```
void printCounting (int n){
    //Base Case
    if (n == 0)
        return ;
    // Processing
    cout << n << "  ";
    // Recursive Relation
    printCounting (n-1);
}
```

Stack is basically like plates in the marriage.
Stack follows LIFO i.e last in first out.



In the function call stack, 1st entry will be of main function always. As the base case matches the entry of function in stack is removed & control will go to the previously called function.

Note → When the base case is matched & as the return keyword is encountered, the control will go to the next line of the call of previously called function.

Recursion by Tree

main ()
↓
PC(5) → 5
↓
PC(4) → 4
↓
PC(3) → 3
↓
PC(2) → 2
↓
PC(1) → 1
↓
PC (0)

Recursive call stack of factorial function

~~factorial(1)~~	1 returned
~~factorial(2)~~	2 returned
~~factorial(3)~~	2×3 = 6 returned
~~factorial(4)~~	4×6 = 24 returned
~~factorial(5)~~	5×24 = 120 returned to main
~~main()~~	n × factorial (4) = 5 × 24

Function will be popped out once it is finished (returned the value).

Note → Whenever a function is called, its entry is pushed into the call stack so as to track it.

Tree

fact(5) ← 120
↓
5 × fact(4) ← 24
↓
4 × fact(3) ← 6
↓
3 × fact(2) ← 2
↓
2 × fact(1)
1

Head and Tail recursion

```
void solve () {
    //Base case
    // Processing
    //Recursive Relation
}
```

Tail Recursion

```
Void solve () {
    //Base case
    //    Recursive Relation
    // Processing
}
```

Head Recursion

In tail recursion, processing is done before calling of function whereas in head recursion processing is done after the recursive relation.

Ex→ Write code for printing counting from 1 to n.

```
void count (int n) {
        //Base Case
        if (n = = 0)
              return;
        // Recursive Relation
        count (n-1);
        // Processing
        cout << n;
}
```

The above code is of the head recursion. This will print counting from 1 to n as whenever we encounter return in base case, we move to the next line of the calling of function.

Understanding recursion of fibonacci series
We have already derived the relation i.e
$f(n) = f(n-1) + f(n-2)$
But the question comes to our mind that what will be the base case.

2nd term

0 1 1 2 3 5 8 ---

1st term

$n == 2 \Rightarrow$ return 1;
$n == 1 \Rightarrow$ return 0;

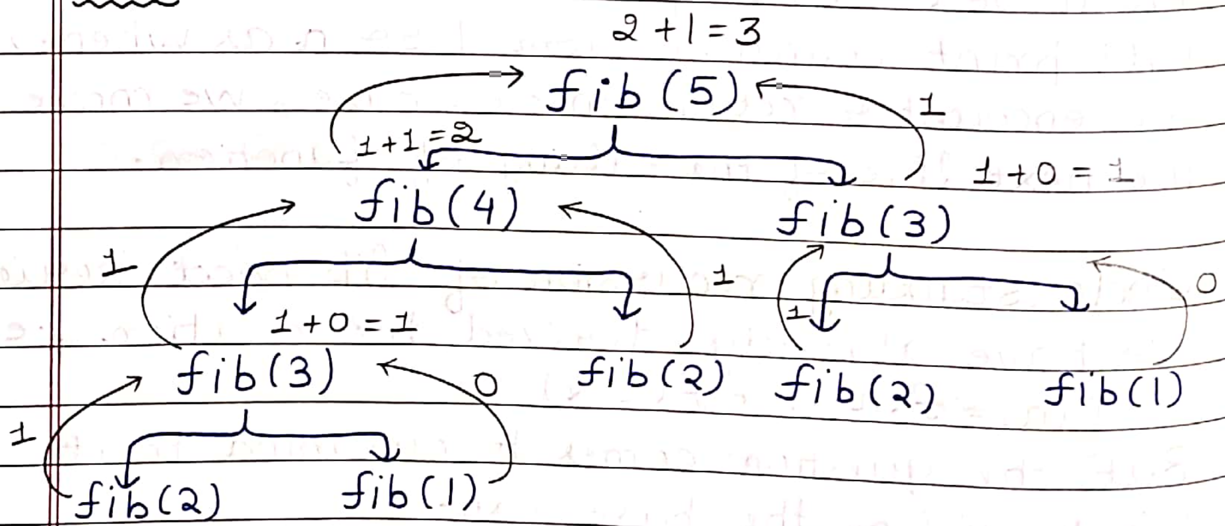
Base case.

We have selected this as the base case as behind them there are no 2 numbers. Hence we handled this case explicitly.

Code

```
int fib (int n){
      //Base case
      if(n = = 1)  // 1st term
            return 0;
      if (n = = 2)  //2nd term
            return 1;
      //Recursive Relation
      int ans = fib(n-1) + fib(n-2);
      return ans;
}.
```

Tree



This is how our algorithm works. First n-1 call will go & then n-2 call will go.

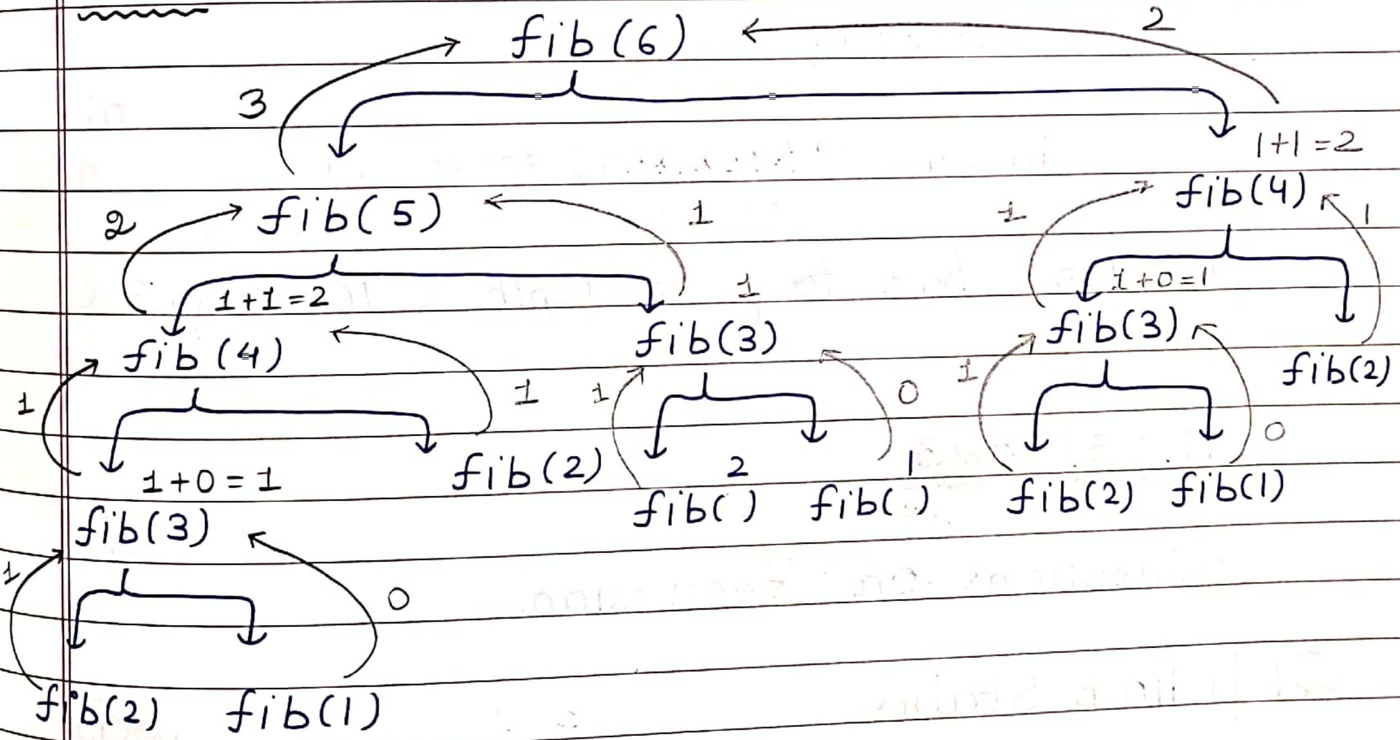
$$\left.\begin{array}{l} fib(2) = 1 \\ fib(1) = 0 \end{array}\right\} \text{Base Case}$$

Note → In our entire coding journey, we don't have to see whether head or tail recursion will be applied. It will just be dependent on the problem statement.

Also note that the call which is encountered first gets executed.

$n = 6$

3 + 2 = 5 Ans



The above is the tree for $n = 6$ & we can say that 6th term is 5.

Also note that the nodes which are at the end are the calls in which base case condition is satisfied.

Note → Recursion call stack and tree are just for the understanding purpose.

Very important line

MAGICAL LINE

Just solve 1 case and rest recursion will handle.

Suppose that we need to find 5! & lets assume that 4! is known as recursion has given the answer. So simply multiply 5 & 4! and hence we get the answer.

$$5 \times 4!$$

Recursion knows the answer (Assume it true)

We don't have to go in depth of the magical line.