Recursion (class 1)

10-03-2023

Date.....

A recursion means, a function calls itself, directly or indirectly.

↳ The problem is solved by repeatedly breaking into smaller problems, which is similar in nature to the original prob-lem. The smaller problems are solved & their solutions are applied to get the final soluti-on of the original problem.

Example :-

```
main(){
    rec();
}
void rec(){
    rec();
}
```

//recursive call

Here, the function rec() is calling itself inside its own function body, so rec() is a recursive function. when main() calls rec(), the code of rec() will be executed & since there is a call to rec() inside rec(), again rec() will be executed.

It seems that this process will go on infinitely but in practice, a term -inating condition is written inside the recursive function which ends this recursion.
The terminating condition is also knowns as **exit condition** or the **Base condition**.

↳ this is the case when functi -on will stop calling itself & will finally start returni -ng.

The two main steps in writing a recursive function are :-

i) Identification of the Base case & its solution, i.e., the case where solution can be achieved without recursion. There may be more than one base case.

ii) Identification of the general case or the recursive case, i.e., the case in which recursive call will be made.

Identifying the base call is very important because without it the function will keep on calling itself resulting in infinite recursion.

Date.....

Ques. Print counting from n to 1 using recursion.

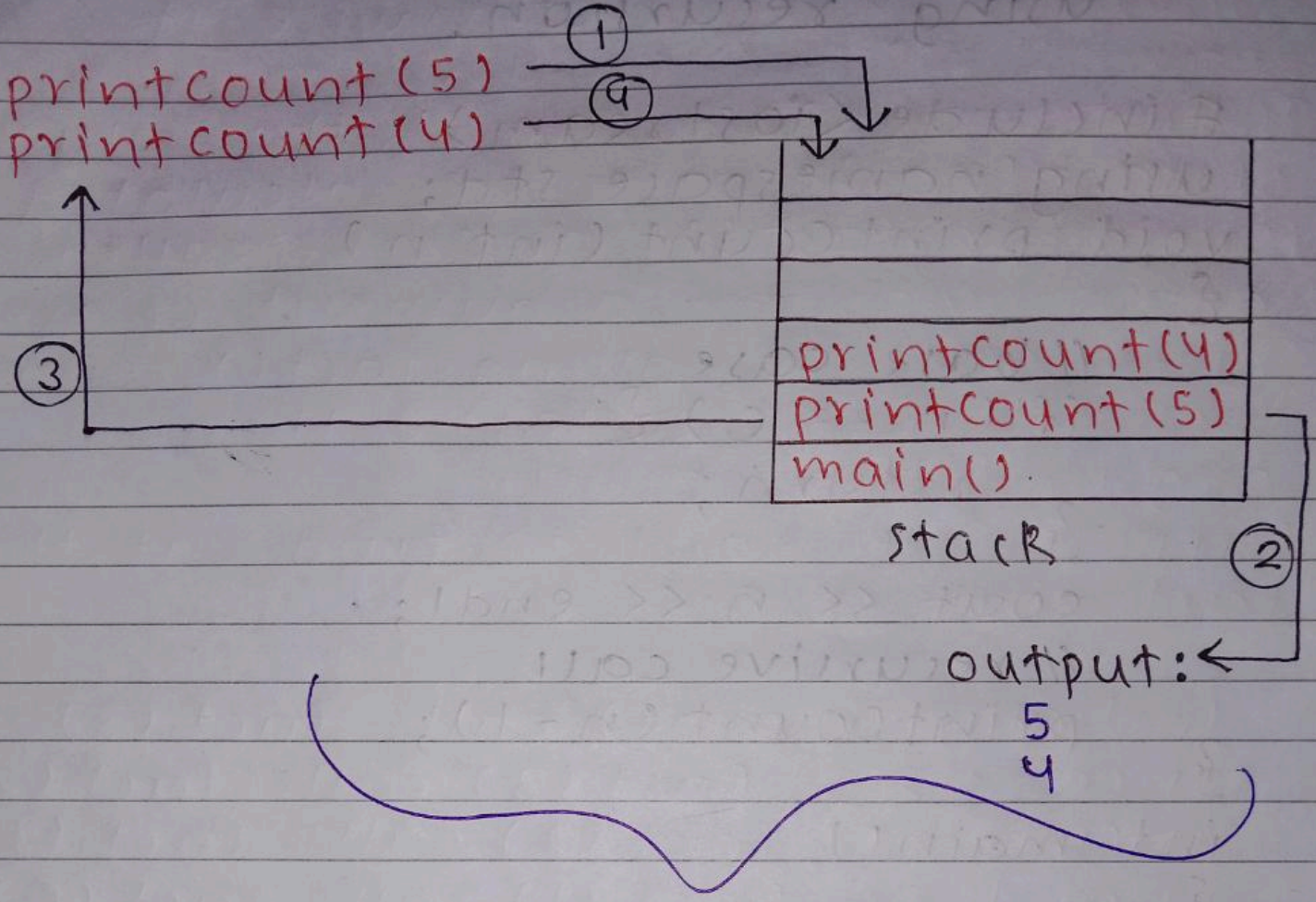
```
#include<iostream>
using namespace std;
void printCount(int n)
{
    // base case
    if(n == 0){
        return;
    }
    cout << n << endl;
    // recursive call
    printCount(n-1);
}
int main()
{
    int n;
    cout<<"enter num: ";
    cin >>n;
    printCount(n);
    return 0;
}
```

Suppose n = 4, first function is called for 4, 4 ≠ 0, print 4, the function is called for 3, 3 ≠ 0, print 3, then function is called for 2, 2 ≠ 0, print 2, then function is called for 1, 1 ≠ 0, then print 1, & function is called for 0. 0 = 0. From Here we return back from function.

Teacher's Sign.....

Date.....

* call stack for previous program:-



This process will continues until n=0, then unwinding phase will start & all the function calls will start returning the values.

we must ensure that each recursive call takes us closer to the base case, i.e., the size of the problem should be diminished at each recursive call.

The recursive call should be made in such a way that finally we arrive at the base case.

If we don't do so, we will have infinite recursion.

calculate 4! using recursion

$4! = 4 \times 3!$

$\downarrow$

$3! = 3 \times 2!$

$\downarrow$

$2! = 2 \times 1!$

$\downarrow$

$1! = 1 \times 0!$

$\downarrow$

$0! = 1$

$\downarrow$

$1! = 1 \times 1$, i.e., 1

$\downarrow$

$2! = 2 \times 1$, i.e., 2

$\downarrow$

$3! = 3 \times 2$, i.e., 6

$\downarrow$

$4! = 4 \times 6$, i.e., 24. This means, that factorial of 4 is 24.

Spiral

Date.....

Step1, This defines 4! in terms of 3!, so we must postpone evalutating 4! until we evaluate 3!.

Step2, Here 3! is defined in terms of 2!, so we must postpone evaluating 3! until we evalua-te 2!.

Step3, This defines 2! in terms of 1!.

Step 4, This defines 1! in terms of 0!.

Step 5, This step can explicitly evaluate 0!, since 0 is the base value of the recursion.

Step 6 to 9, we backtrack, using 0! to find 1!, using 1! to find 2!, using 2! to find 3! and finally using 3! to find 4!.

Note:- Jab bhi kisi bigger problem ka solution depend krega choti & same type ki problem pr. To recursion use hoga.

(4!) = 4 x (3!) → choti problem

↓

Badi Problem

Date.....

Ques Program to find the factorial using recursion.

```
#include<bits/stdc++.h>
using namespace std;
int fact(int n)
{
   if(n==0)
    {
      return 1;
    }
    int ans = n * fact(n-1);
    return ans;
}
int main()
{
    int n;
    cout<<"Enter num: ";
    cin>>n;
    int ans = fact(n);
    cout<<"Factorial :"<<ans;
    return 0;
}
```

Date.....

call stack

Recursive functions use something called "call stack".
when a function is called by the prog -ram, that function goes on top of the call stack.
The call stack is used for several rela -ted purposes, but the main reason for having one is to keep track of the point to which each active subroutine should return control when it finish executing.

* what happens to the method call stack while a recursive method is executing?
  whenever a base condition is hit in recursion, we stop making recursive calls & then method calls in stack keeps executing & gets popped out of the memory stack one by one.

A call stack is a stack Data structure that stores information about the active functions. The call stack is also known as Execution stack, control stack, function stack or Runtime stack.

Date.....

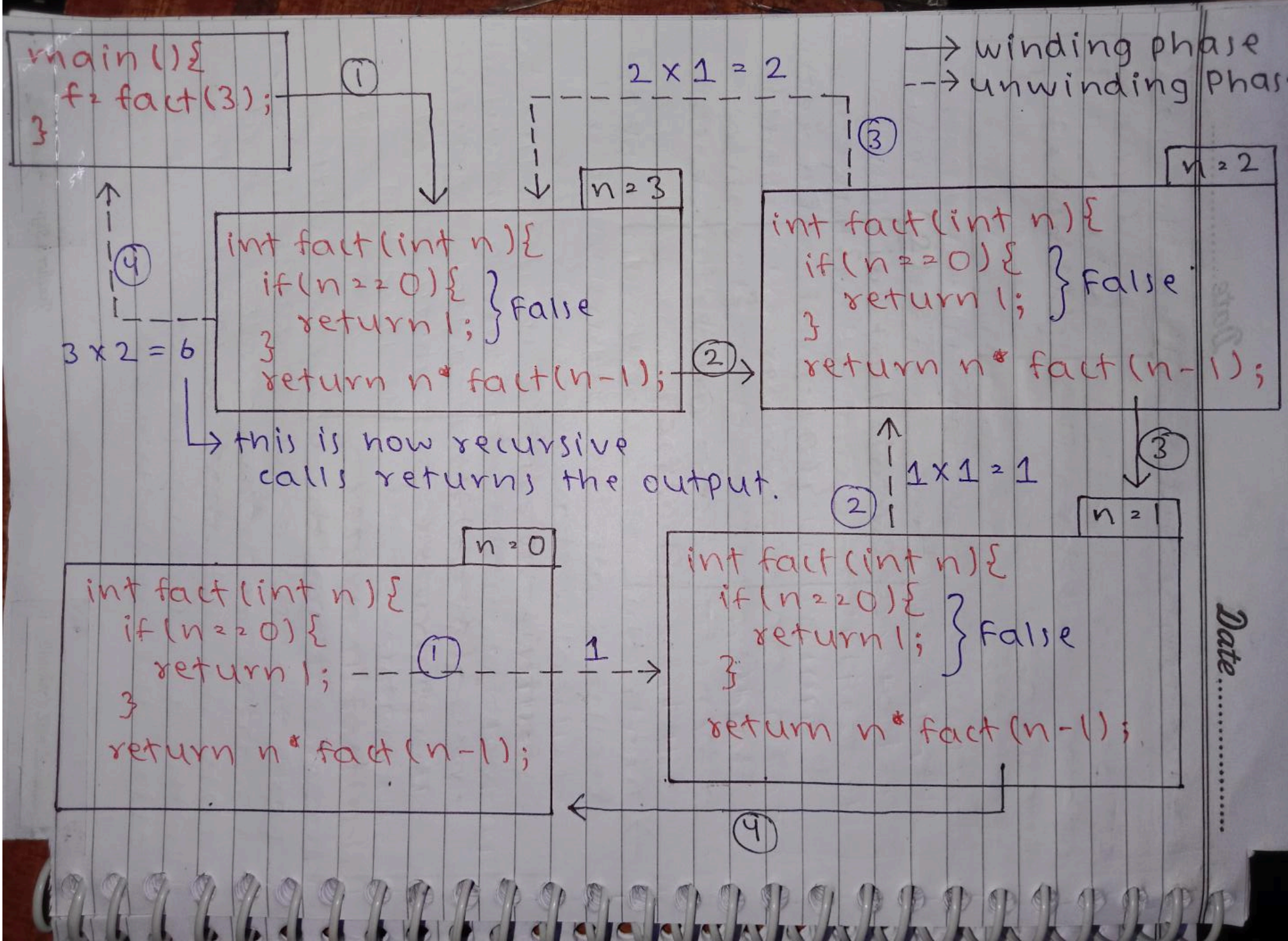
winding and unwinding Phase :-

All recursive function works in 2 phases - winding Phase & unwindi -ng Phase.

→ winding Phase begins when the recursive function is called for the first time, and each recursiv -e call continues the winding Phase. In this phase, the function keeps on calling itself & no retu -rn statement are executed in this phase. This phase terminates when the Base condition become true in a call.

→ After this the unwinding phase begins & all the recursive functi -on calls start returning in reverse order till the first insta -nce of function returns. In unwi -nding phase, the control returns through each instance of the function.

In some algorithms, we need to perform some work while returni -ng from recursive calls, in that case we put that particular code in the unwinding phase, i.e., just after the recursive call.


main(){
f= fact(3);
}
①
n=3
int fact(int n){
if(n==0){
return 1;
}
return n* fact(n-1);
}
False
②
n=2
int fact(int n){
if(n==0){
return 1;
}
return n* fact(n-1);
}
False
③
n=1
int fact(int n){
if(n==0){
return 1;
}
return n* fact(n-1);
}
False
n=0
int fact(int n){
if(n==0){
return 1;
}
return n* fact(n-1);
}
1
④
2 x 1 = 2
3 x 2 = 6
1 x 1 = 1
→ winding phase
--→ unwinding phase
this is how recursive calls returns the output.
Date.....

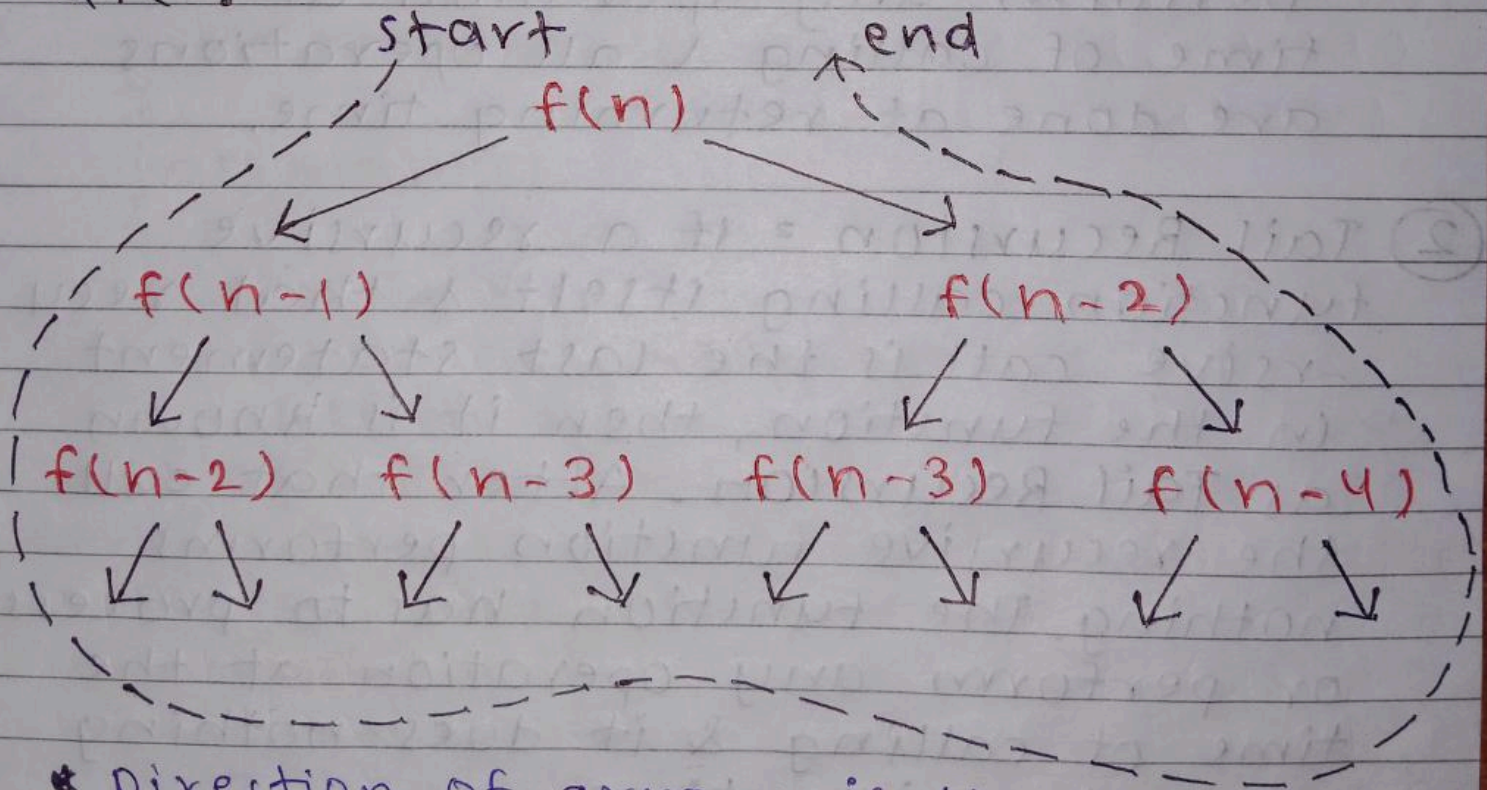
Date.....

Recursion Tree

Recursion Tree is useful for visualizing what happens when a recurrence function called.
It diagrams the tree of recursive calls & the amount of work done at each call.
For example :- fibonacci function looks like :-

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

* Recursion Tree for above function are :-

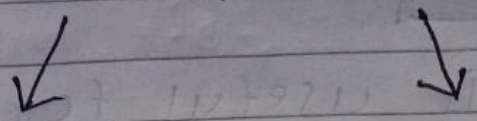


* Direction of arrow is the order of execution of function call
* They execute in Depth First manner, top to bottom, left to right.

Teacher's Sign .....

Date.....

Recursion



Head Recursion.

Tail Recursion.

(1) Head Recursion = If a recursive function calling itself & that recursive call is the first state-ment in the function, then it is known as Head Recursion. There is no statement, no operation before the call. The function does not have to process or perform any operation at the time of calling & all operations are done at returning time.

(2) Tail Recursion = If a recursive function calling itself & that recu-rsive call is the last statement in the function, then it is known as Tail Recursion. After that call the recursive function performs nothing. The function has to process or perform any operation at the time of calling & it does nothing at returning time.

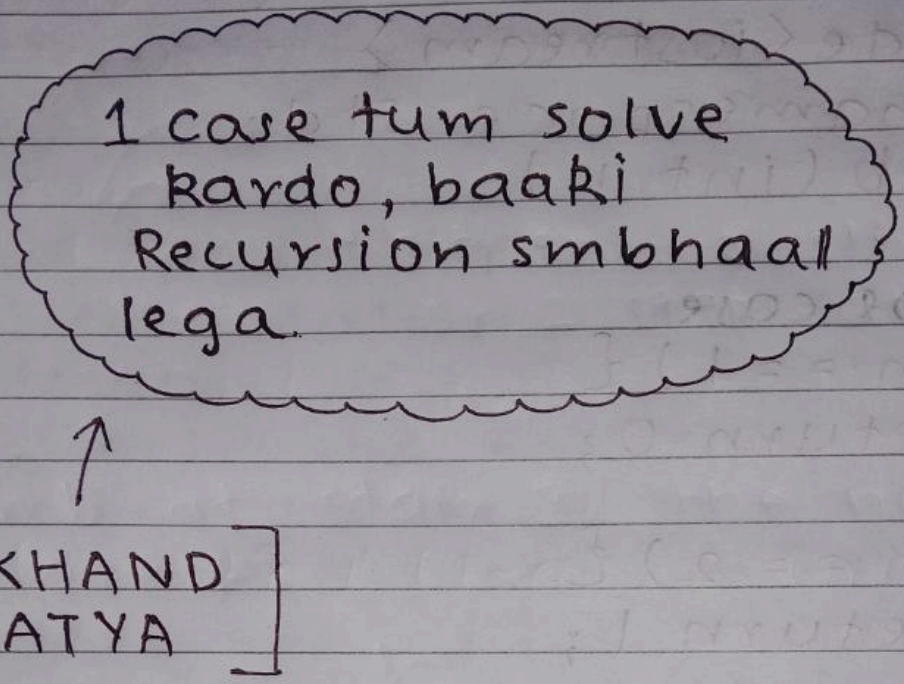
Date.....

Ques Print n^{th} fibonacci term using recursion.

```
#include<iostream>
using namespace std;
int fib(int n)
{
    // base case
    if(n==1){
        return 0;
    }
    if(n==2){
        return 1;
    }
    int ans = fib(n-1) + fib(n-2);
    return ans;
}
int main()
{
    int n;
    cout << "enter term:";
    cin >> n;
    int ans = fib(n);
    cout << n << "term of fibonacci
        series is " << ans;
    return 0;
}
```

Date.....

There is a magical line in concept of recursion, i.e.,



[AKHAND SATYA]