* Recursion: Bookish: when a func call itself directly or indirectly called recursion.

Def: Bigger Problm sol depend on small and same type of a problem.

Ex: intsolve() {

solve()

}

Problem Statement

(1) i/p = n
o/p = print 2^n Ex: n=4
$2^4 = 16$

Bigger problem: 2^n
Smaller problem: 2^{n-1}

As $2^n = 2 \times 2^{n-1}$
↓ Bigger Problem. ↓ smaller problem
as B.P depend on S.P

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

It is clear that $f(n-1)$ i.e S.P depend on $f(n)$ B.P

(2) i/p → n
o/p → n!
Ex: n=5, n=3
5! = 120 3! = 6

Big Problem: n!

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

$n! = n * (n-1)!$
↓ B.P ↓ S.P
as B.P depend on S.P

As. $f(n) = n * f(n-1)$

It is clear that $f(n-1)$ i.e S.P depend on $f(n)$ B.P

③ counting in reverse order Ex: n=5 5 4 3 2 1 output

I/P → n
O/P → numbers counting from n to 1.

Bigger problem : numbers counting from n → 1

f(n) → print(n) + f(n-1) → S.P
ignore
print from (n-1) to 1
↓
B.P

B.P depends on S.P.

* Recursive code:

① Base case → when code need to be stopped (condition)

② Recursive relation: Ex: f(n) = n * f(n-1)] B.P depends on S.P or same type Problem
Recursive call

③ Processing → optional
Calculation, updation etc.

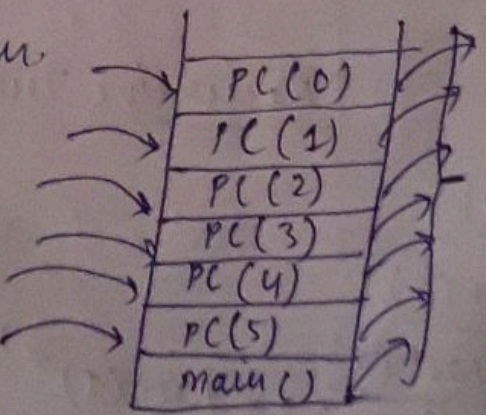
* we need to write return along with base case.

* Recursive Call stack:

```
void printCount(int n){
    //base case
    if(n==0){
        return;
    }
    //processing
    cout<<n;
    //Recursive relation
    printCount(n-1);
}
```

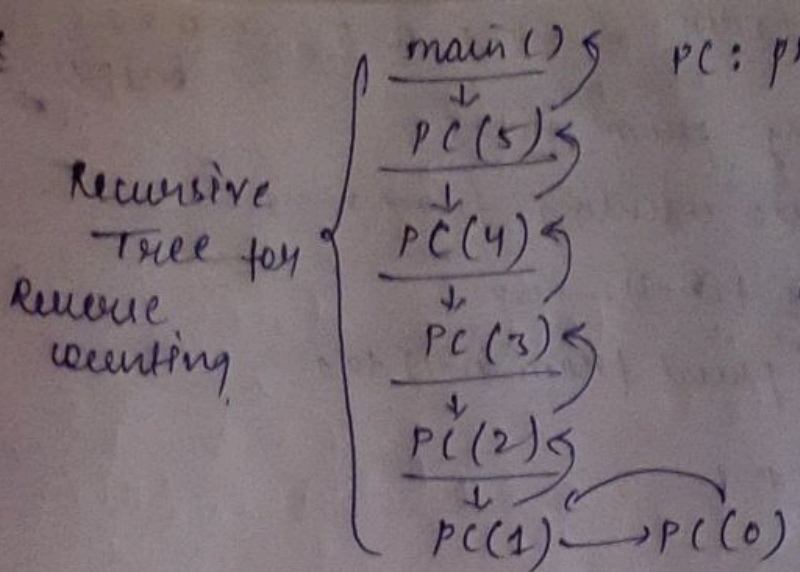
```
main(){
    cin>>n;
    printCount(n);
}
```

Function Call Stack



output:
5 4 3 2 1

*



PC : printcounting() func

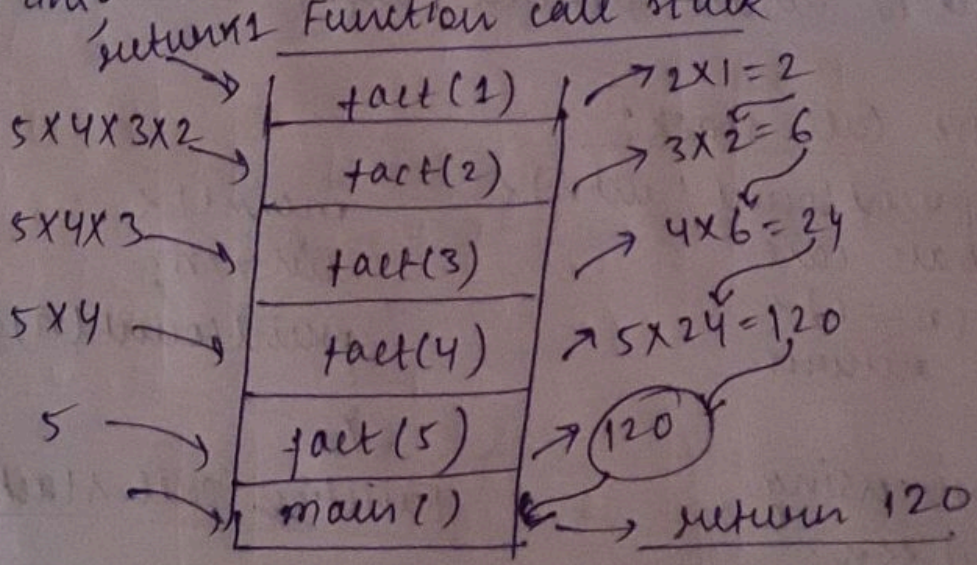
output

Reverse counting
5 4 3 2 1

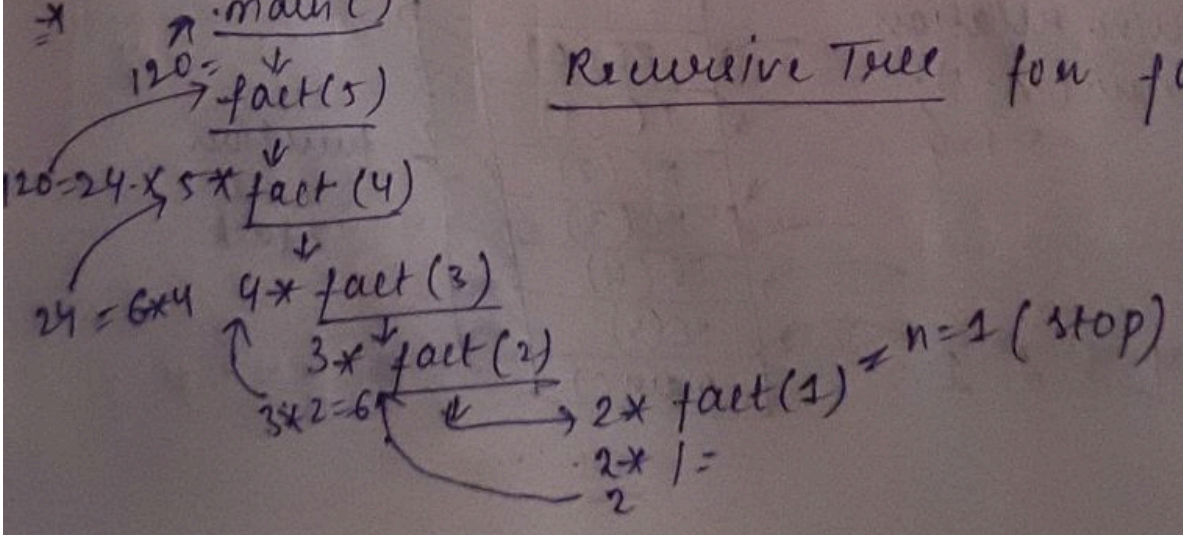
* Factorial

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

```
int main(){
int n;
cin>>n;
int ans = factorial(n);
cout<<
```



*



Recursive Tree for factorial

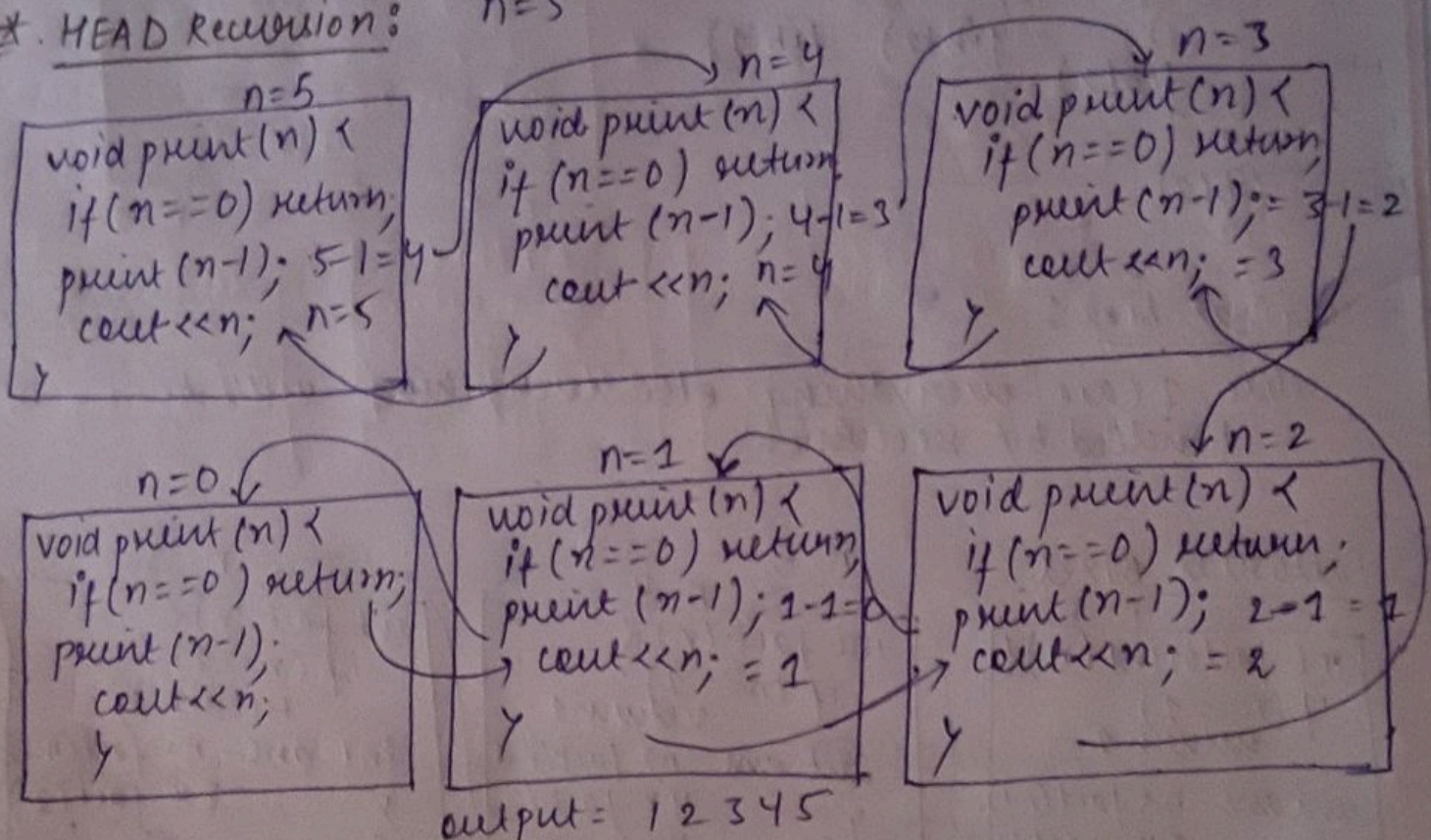
```
void solve(){
  // Base case
  // Processing
  // Recursive relation
}
```

TAIL Recursion

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

HEAD Recursion

HEAD Recursion:



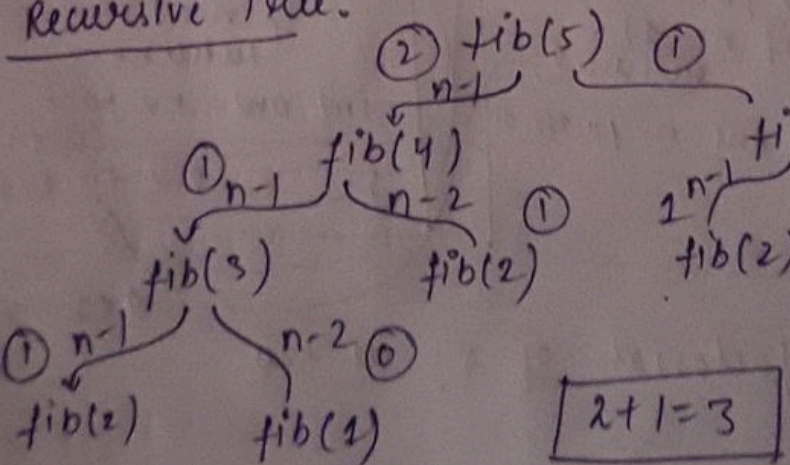
output = 1 2 3 4 5

Fibonnaci Series:

Recursive Tree:

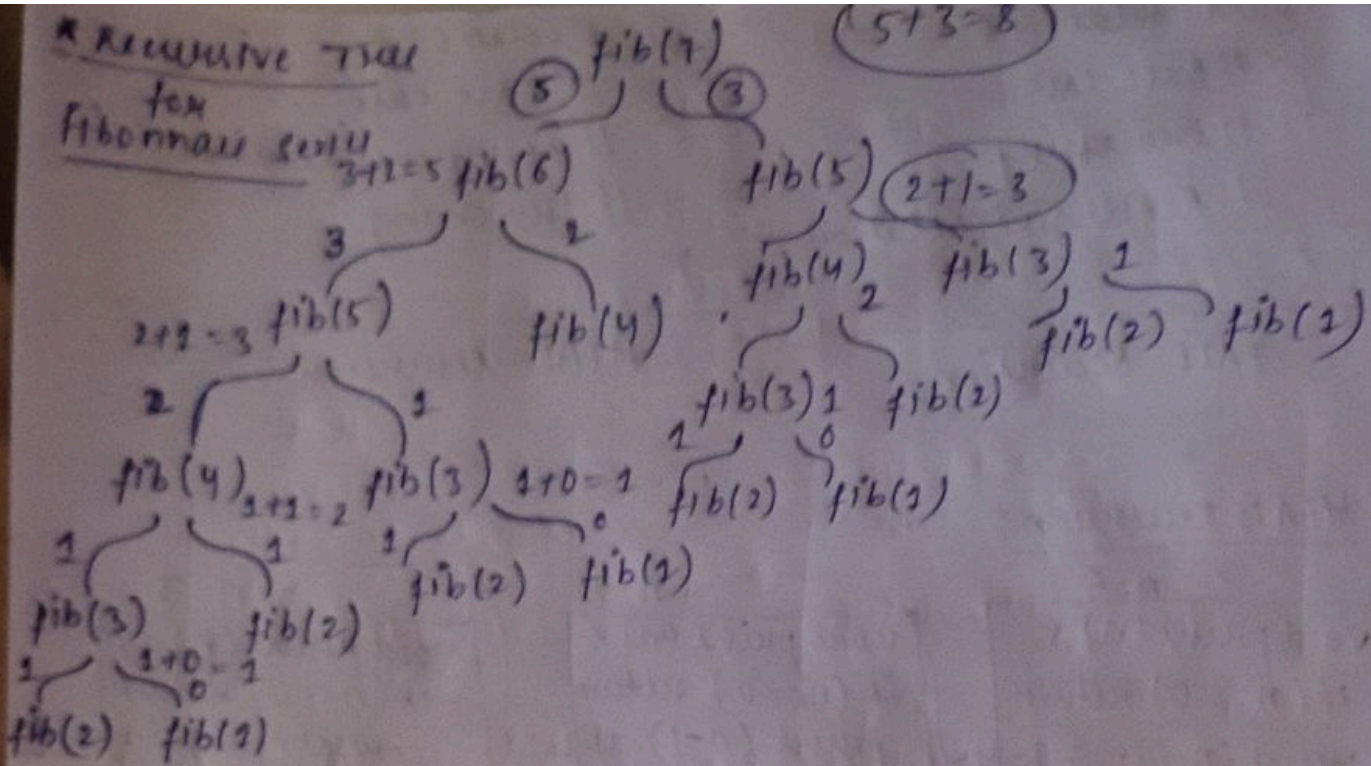
R.R for fibonacci:

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



Every end node is B.C

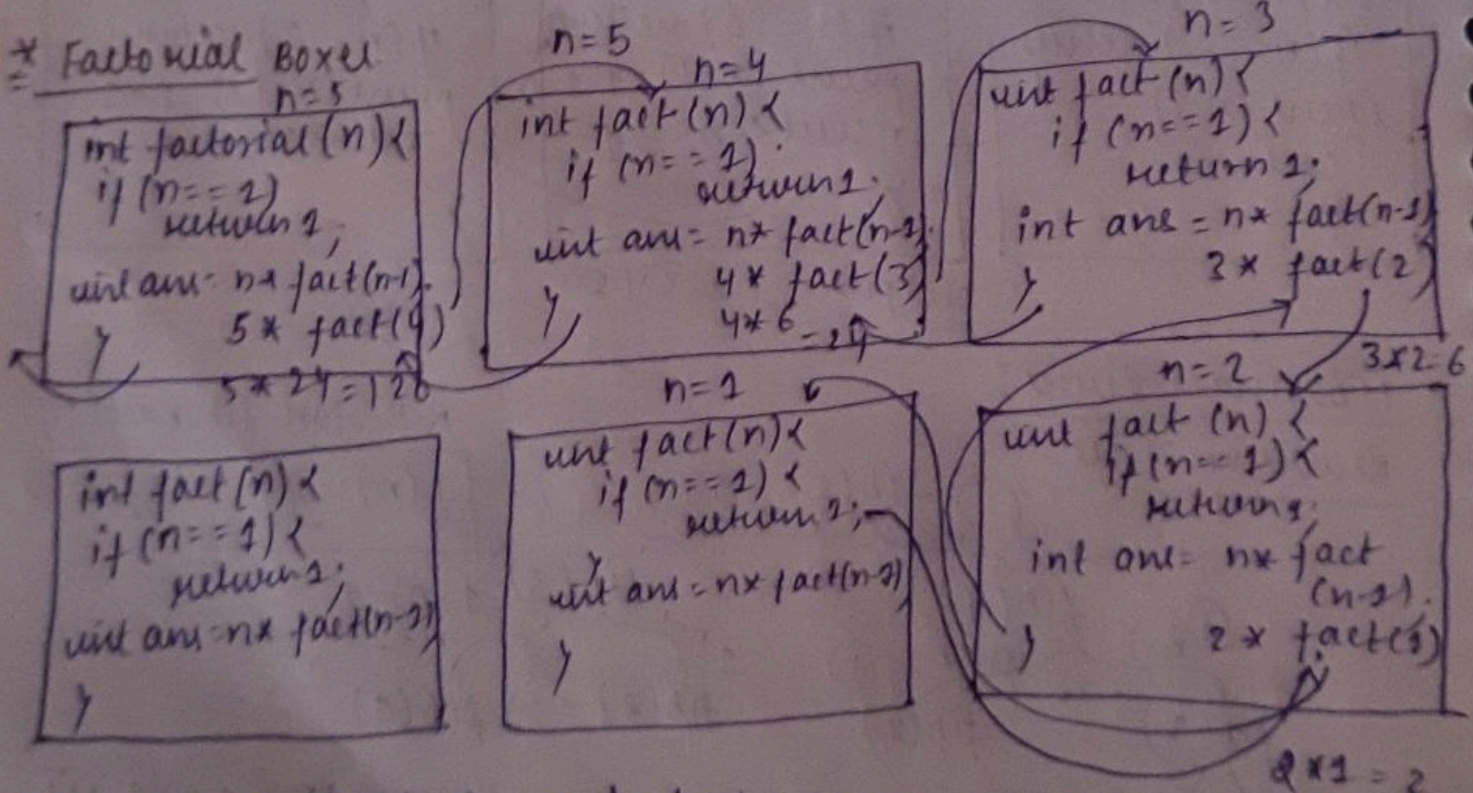
* Recursive Trace for Fibonnaci series



* Magical Line:

Solve 1 case everything else everything will be handled by recursion.

* Factorial Boxes



The output i.e factorial of 5! is 120.

* Examples of Recursion:

① 

You can reach to the Destination in 5 steps

B·P is you can reach in 5 steps.

S·P is you can reach in 1+4 steps.

5 steps	=	1 + 4 steps
↓		↓
B·P		S·P

* Fibonnaci series:

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

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

$f(n) = \underbrace{f(n-1)}_{S\cdot P} + \underbrace{f(n-2)}_{S\cdot P}$

$\underset{\downarrow}{f(n)}$
B·P

* Print Counting in Reverse Order TAIL Recursion: [Boxes]

n=5
```
void print(n){
if(n==0) return;
cout<<n; 5
print(n-1);
}
```
5-1=4

n=4
```
void print(n)
if(n==0) return;
cout<<n; 4
print(n-1);
}
```
4-1=3

n=3
```
void print(n)
if(n==0) return;
cout<<n 3
print(n-1);
}
```
3-1=2

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

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

n=2
```
void print(n){
if(n==0) return;
cout<<n; 2
print(n-1);
}
```
2-1=1

output: 5 4 3 2 1

Fibonacci Boxes:

n=5

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

n=5

2+1=3

n=4

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

2 + 1

n=3

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

n=3

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

1+1

n=2

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

n=2

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

n=1

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