Recursion

```
③ int main()
{
    Cout << "Before fun2 \n";
    fun2();
    Cout << "After fun2 \n";
    return 0;
}
```

```
② Void fun2()
{
    Cout << "Before
    fun1();
    Cout << "After
}
```

```
① Void fun1()
{
    Cout << "fun1 \n";
}
```

Output

Bef fun 2
Bef fun 1
fun 1
Aft fun 1
Aft fun 2

// function Calls

A fn calls itself directly (or) indirectly.

```
Void f1()
{
   f1();
}
```

a) direct

```
Void f1()
{
   f2()
}
Void f2()
{
   f1()
}
```

b) indirect.

```
Void f1()
{  cout << "GFG" << endl;
   f1();
}
int main()
{
   f1();
   return 0;
}
```

// it keeps on calling

No termination condition.

: Segmentation fault.

```
void f1(int n)
{
    if (n==0)          ] base case
        return;
    cout << "GFG" << endl;
    f1(n-1);
}
```

```
int main()
{
    f1(2);
    return 0;
}
```

output

GFG
f1(2-1);

GFG
f1(1-1);

return;

Structure

```
.... f1(....)
{
    base cases
    ....
    Recursive Call (i.e Call to fn())
    with atleast one change in parameter
    ....
}
```

Applications of Recursion

Many algorithm techniques are based on Recursion.

→ Dynamic Programming

→ Back tracking

→ Divide & Conquer.
- quick
- merge
- Binary Search.

iteration ↑
↳ recursion
program Solve.

- Many problems inherently recursive
  - Tower of Hanoi.
  - DFS based traversals (DFS of graph)
    - Inorder
    - Post order
    - Pre order.

traversals of tree.

(1)

```
Void fun( int n)
{
    if (n==0)
    return ;
    Cout << n << Endl ;
    fun(n-1) ;
    Cout << n << Endl ;
}

int main( )
{
    fun(3) ;
    return 0;
}
```

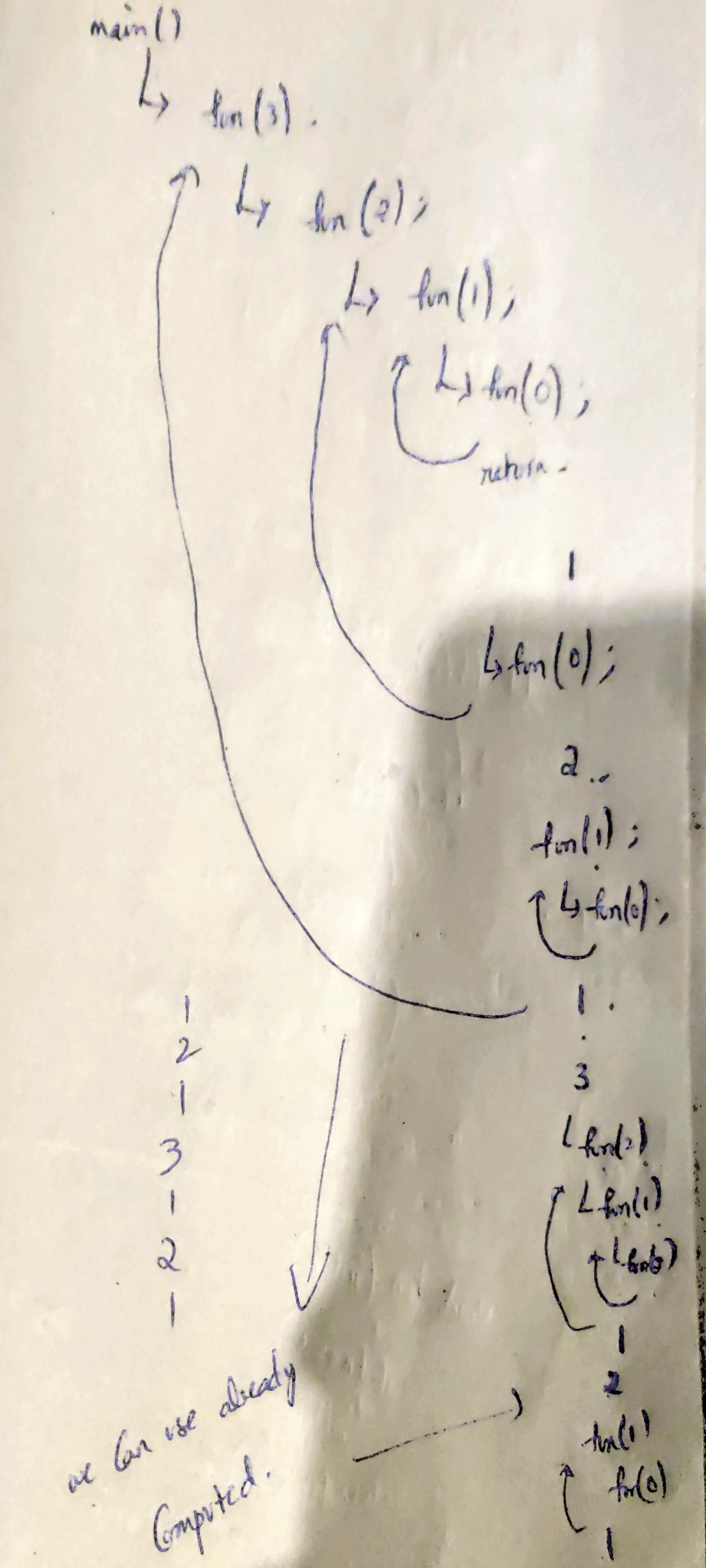
3
f(3-1),
2
f(2-1),
1
f(1-1) ;
ret
1
2
3

(2)

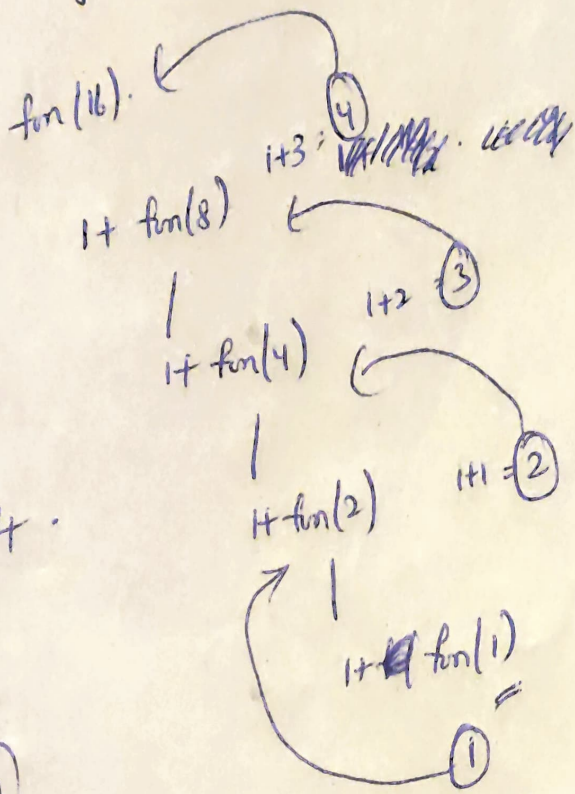
```
Void fun( int n)
{
    if (n==0)
    return
    fun(n-1) ;
    Cout << n << Endl ;
    fun(n-1) ;
}

int main( )
{
    fun(3)
    return 0;
}
```

3


main()
fun(3).
fun(2);
fun(1);
fun(0);
return.
1
fun(0);
2..
fun(1);
fun(0);
1.
3
fun(2)
fun(1)
fun(0)
1
2
fun(1)
fun(0)
1
1
2
1
3
1
2
1
we can use already
computed.

```
int fun(int n)
  if (n==1)
    return 0;
  Else
    return 1+ fun(n/2)
}
```



= 4.

$(\log_2 n)$

```
Void fun (int n)
  if (n==0)
    return ;
  fun(n/2);
  print(n%2);
```

n=7 ?

fun(1)
↳ fun(7/2 = 3.5 ≃ 3)

↳ fun(1)

↳ fun(0)

1 % 2 . 1

3 % 2 . 1

7 % 2 . 1

111

it prints binary equivalent

// n = 5

// 5 4 3 2 1

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

int main()
{
    int n>4;
    f(n);
    return 0;
}
```

base case

n > 1

5 1

↳ f(4)

4 9

↳ f(3)

3

↳ f(2)

2

↳ f(1)

1

↳ f(0)

1 to n

n = 4

1 2 3 4

(n-1)

Cout << n;

fn(n-1);

base case

if n == 1

return 0;

We need to call recursively upto n-1 &

print n.

```
void print(int n)
{   if(n==0)
        return;
    Print(n-1);
    Cout << n ;
}
int main()
{   int n=4
    Print(n);
    return 0;
}
```

Tail Recursion

- When parent fn has nothing more to do when child fn has finished executing

Ex:

```
void fn(int n)
{
    if (n==0)
        return;
    cout << n;          // prints n, Calls recursive fn.
    fn(n-1);
}
```

At last at f(1) Executes & program terminates.

but for 1 to n Recursive program.

```
if (n==0)
    return;
fn(n-1);
cout << n;
```

Calls upto last no & later prints.

So after fn, it need to print & return so recursive call Completes.

...So on -

So tail Recursion Executes Slight faster.