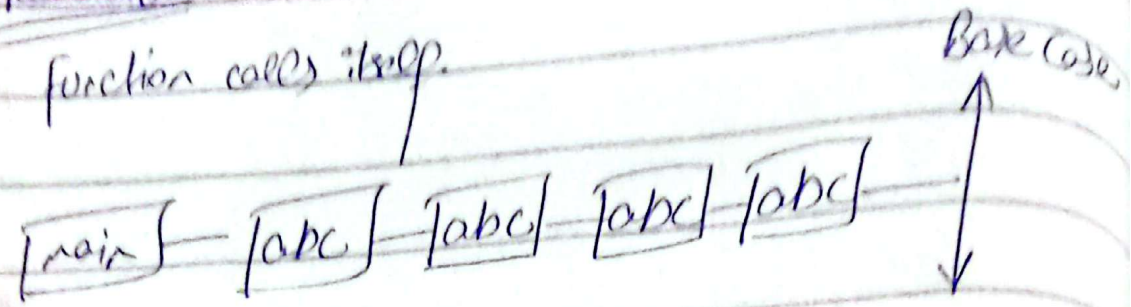
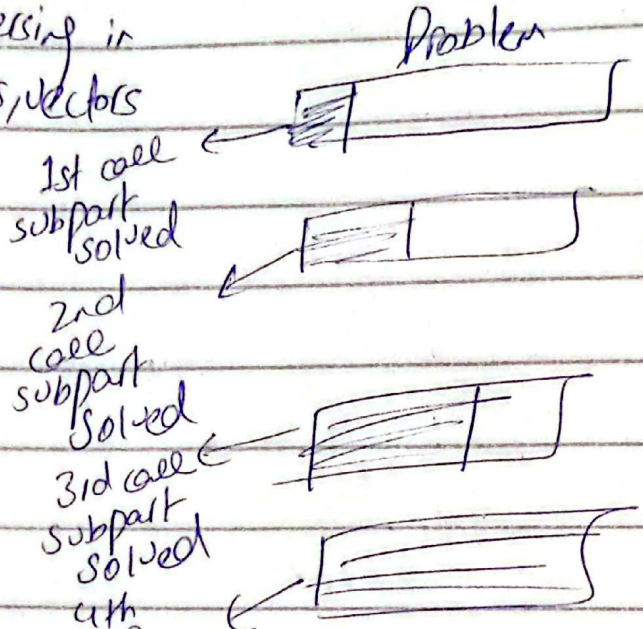
Recursion

function calls itself.



All works which we can do with the loops, we can also do them with the recursion.

- traversing in arrays, vectors
- Graphs, Trees etc.



n to 1

4 3 2 1

3 2 1

2 1

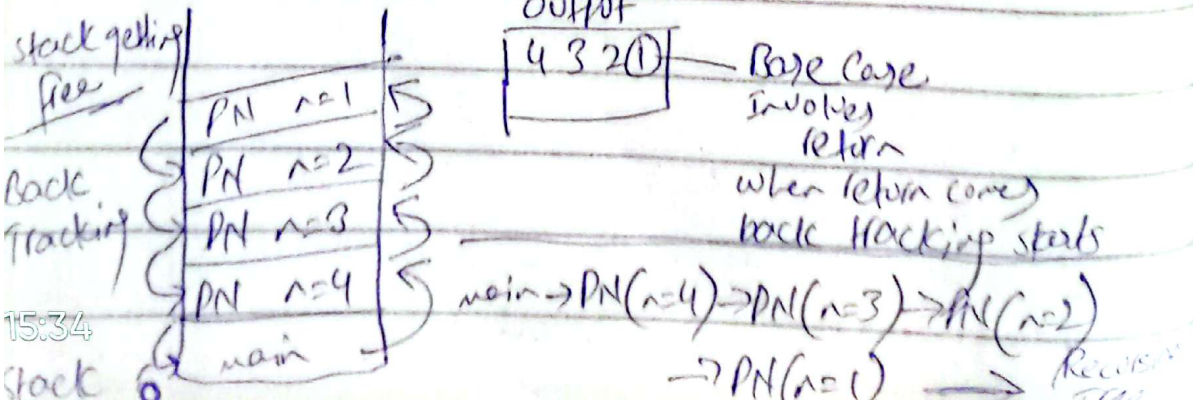
1

Base case:

```
if(n==1){
cout<<n<<1;
return;
```

```
void PrintNum(int n){
cout<<n;
PrintNum(n-1);
}
```

Call stack



Recurrence Relation

$f(n) = print\ "n" + f(n-1)$

$T(C) = O(1) + T(n-1)$

(Example)

N Factorial

n! like $4! = 4 \times 3 \times 2 \times 1 = 4 \times 3!$

$3! = 3 \times 2!$ and $2! = 2 \times 1!$

$f(n=4) = 4 * f(n=3) \rightarrow 3 * f(n=2) \rightarrow 2 * f(n=1) \rightarrow 1 * (f(n=0))$

↓

Base case return 1

```
int factorial(n){
    if(n==0){ ——→ Base case
        return 1;
    return n * factorial(n-1)
}
```

Recursion (Time complexity)

① Using Recurrence Relation

② TC = total No of Rec calls * work done in each call (More Practical)

Using ①:-

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

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

$f(n-2) = 1C + f(n-3)$

$f(n-3) = 1C + f(n-4)$

$f(1) = 1C + f(0)$

$f(n) = C * n + C_2$

$f(n) = O(n)$

$T.C = O(n)$

method 2

Step Recursion Tree :-

n=4 f(4) → f(3) → f(2) → f(1) → f(0)

(n → (n-1)) (n → 1+n)

O(n * 1)

O(n)

Space Complexity

Although, there is no extra memory allocated structure is used, but stack frames are being used for each call

1C →	n=0
1C →	n=1
1C →	n=2
1C →	n=3
1C →	n=4

n

SC = Depth of Rec. Tree (Height of call stack) * Memory in each call ①

S.C = n * 1C

S.C = n

* Sum of N nums

S(4) → S(3) → S(2) → S(1)

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

f(4) = 4 + f(4-1)

```
int Sum(int n){
  if(n==1){
    return 1
  }
  return n + Sum(n-1)
}
```

T.C = total Calls * W.D in each call

= n * O(1)

T.C = O(n)

S.C = Depth of Rec. Tree * Memory in each call

= n * O(1)

S.C = O(n)