Introduction → function calling itself.

A problem is broken into smaller problems, (subproblems)
and soln to bigger problem is generated
using subproblems.

Example

Sum of first N natural numbers.

$sum(N) = 1 + 2 + 3 + \cdots (N-1) + N.$

$sum(N) = \underbrace{sum(N-1)}_{\text{subproblem}} + N.$

How do we write a recursive code:-

1) Assumption :- Decide what your function does, and assume its working for smaller i/p.

2) Main logic :- Solving bigger problem using smaller problem

3) Base Case:- Smallest input for which you know the soln.

function call tracing

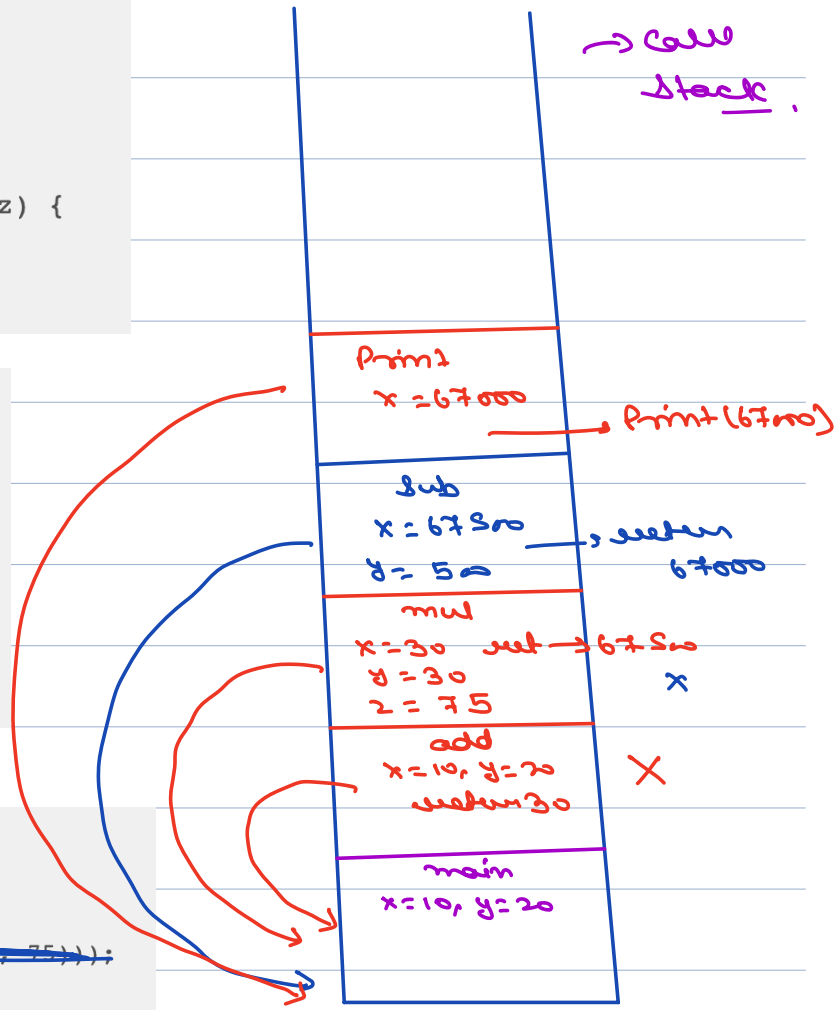
```
int add(int x, int y) {
    return x + y;
}

int mul(int x, int y, int z) {
    return x * y * z;
}
```

```
int sub(int x, int y) {
    return x - y;
}

void print(int x) {
    cout << x << endl;
}
```

```
int main() {
    int x = 10;
    int y = 20;
    ~~print(sub(mul(add(x, y), 30, 75)));~~
    return 0;
}
```



$t_1 =$ add(x, y);

$t_2 =$ mult(t_1, 30, 75);

$t_3 =$ sub(t_2, 500)

Print(t_3);

Ques) finding factorial of n.

$5! = 1 \times 2 \times 3 \times 4 \times 5 \Rightarrow 120.$

$fact(n) = \underbrace{fact(n-1)}_{(1 \times 2 \times \cdots (n-1))} * n$

```
int factorial (int n) {
    if (n==0) { return 1 }
    return factorial(n-1) * n
}
```

6

print (fact (3))

```
int factorial ( int n = 3 ) {
    if (n==0) { return 1 }
    return factorial (n-1) * n
```

2 3

3

```
int factorial ( int n = 2 ) {
    if (n==0) { return 1 }
    return factorial (n-1) * n
```

1 2

3

```
int factorial ( int n ==1 ) {
    if (n==0) { return 1 }
    return factorial (n-1) * n
```

1 1

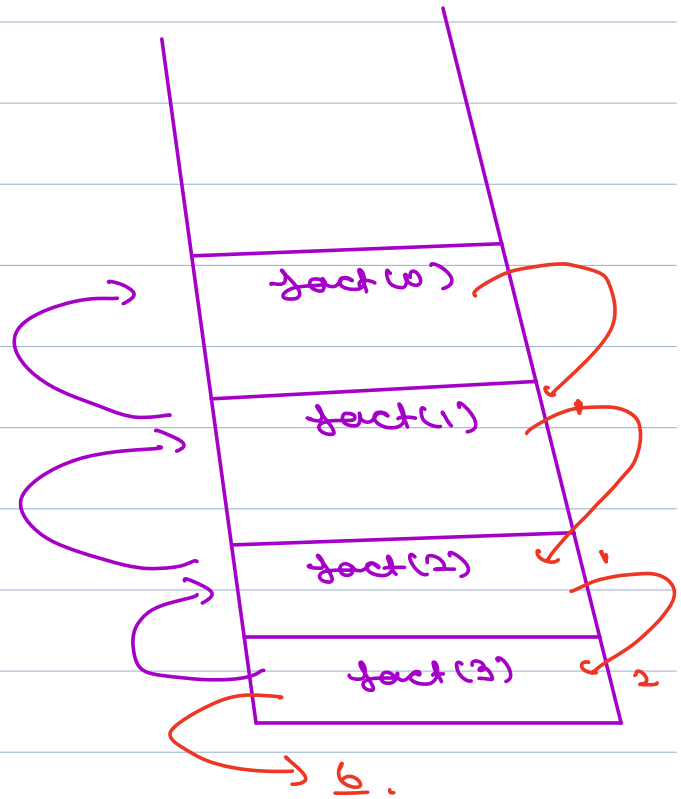
3

```
int factorial ( int n ==0 ) {
    if (n==0) { return 1 }
    return factorial (n-1) * n
```

3

fact (3)

```
int factorial ( int n) {
    if (n==0) { return 1 }
    return factorial (n-1) * n
}
```



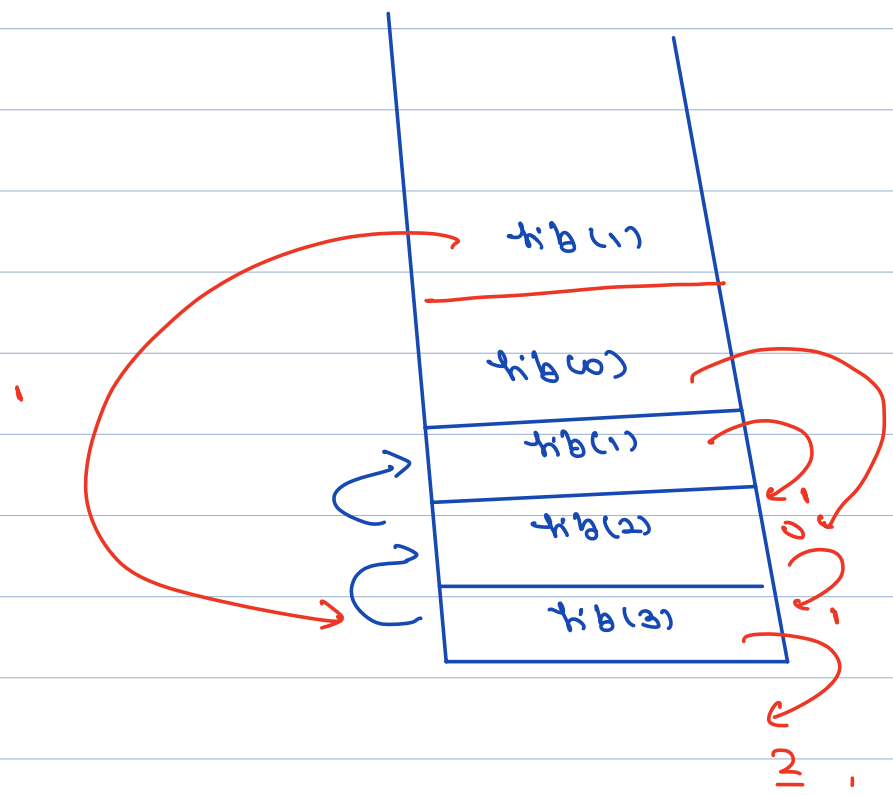
Ques) n^{th} number in fibonnaci Series :-

n	0	1	2	3	4	5	6	7
fib	0	1	1	2	3	5	8	13

```
int fib(n) {
    if(n==0 || n==1) { return n}

    return fib(n-1) + fib(n-2)
}
```

fib(3)



fib(4)

2

```
int   fib(n = 3) {
      if (n==0 || n==1) { return n}

      return  fib(n-1) + fib(n-2)
```

3

```
int   fib(n = 2) {
      if (n==0 || n==1) { return n}

      return  fib(n-1) + fib(n-2)
```

3

```
int   fib(n = 1) {
      if (n==0 || n==1) { return n}

      return  fib(n-1) + fib(n-2)
```

3

```
int   fib(n = 1) {
      if (n==0 || n==1) { return n}

      return  fib(n-1) + fib(n-2)
```

3

```
int   fib(n = 0) {
      if (n==0 || n==1) { return n}

      return  fib(n-1) + fib(n-2)
```

3

Break 8:15Am - 8:25Am

Ques) a, n, a^n using recursion

$a = 2, \; n = 3, \qquad 2^3 \Rightarrow 8.$

$$a^n \Rightarrow \underbrace{a * a * a \cdots a}_{n}.$$

← recursively →

$a > 0.$

```
function pow (a, n) {
    if (n==0) {return 1}
    return pow (a, n-1) * a;
}
```

$a^n \rightarrow a^{n-1} * a$

$2^{10} \rightarrow 2^9 * 2. \rightarrow 2^5 * 2^5$

$3^{18} \rightarrow 3^{17} * 3 \rightarrow 3^9 * 3^9.$

$4^{10} \rightarrow 4^9 * 4$

$\rightarrow 4^5 * 4^5$

$4^{11} \rightarrow 4^5 * 4^5 * 4$.

```
function pow ( a, n) {
        if (n==0) {return 1}
    if (n%2 == 0) {
        return pow (a, n/2) * pow (a, n/2);
    }
    else {
        return pow(a, n/2) * pow (a, n/2) * a;
    }
}
```

fast Power or fast Exponentiation.

```
function pow ( a, n) {
    if (n==0) {return 1}
        p = pow (a, n/2)
    if (n%2 == 0) {
        return p*p;
    }
    else {
        return p*p*a;
    }
}
```

pow (2, 9) 512.

```
function pow ( a=2, n=9 ) {
    if (n==0) {return 1}
        p = pow (a, n/2)   16
    if (n%2 == 0) {
        return p*p;
    }
    else {
        return p*p*a;
    }
}
```

```
function pow ( a=2, n=4 ) {
    if (n==0) {return 1}
        p = pow (a, n/2)   4
    if (n%2 == 0) {
        return p*p;
    }
    else {
        return p*p*a;
    }
}
```

```
function pow ( a=2, n=2 ) {
    if (n==0) {return 1}
        p = pow (a, n/2)   2
    if (n%2 == 0) {
        return p*p;
    }
    else {
        return p*p*a;
    }
}
```

```
function pow ( a=2, n=1 ) {
    if (n==0) {return 1}
        p = pow (a, n/2)
    if (n%2 == 0) {
        return p*p;
    }
    else {
        return p*p*a;
    }
}
```

```
function pow ( a=2, n=0 ) {
    if (n==0) {return 1}
        p = pow (a, n/2)
    if (n%2 == 0) {
        return p*p;
    }
    else {
        return p*p*a;
    }
}
```

Time Complexity of Recursive functions :-

$T(n)$

```
int factorial(int N) {
    // base case
    if (N == 0) {
        return 1;
    }
    // recursive case
    return N * factorial(N-1);
}
```

$T(n-1)$.

Recurrence Relation

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

$T(0) = 1$

$\Rightarrow T(n) = T(n-1) + 1$

$T(n-1) = T(n-2) + 1$, $T(n-2) = T(n-3) + 1$

$\Rightarrow T(n) = T(n-2) + 2$

$\Rightarrow T(n) = T(n-3) + 3$

$\Rightarrow T(n) = T(n-4) + 4$

After k iterations,

$\Rightarrow T(n) = T(n-k) + k$.

if , $n-k = 0$,

$k = n$.

$T(n) = T(0) + n$

$T(n) = 1 + n$

T.C $\rightarrow O(n)$.

Time Complexity of Power function :-

$\rightarrow T(n)$

```
Function pow(int a, int n){
    if(n == 0) return 1;

    if(n % 2 == 0) {
        return pow(a,n/2) * pow(a,n/2);
    }
    else {
        return pow(a,n/2) * pow(a,n/2) * a;
    }
}
```

$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + 1$

$T(n) = 2T(\frac{n}{2}) + 1$

$T(n) = 2T(\frac{n}{2}) + 1 \quad \text{- - - -} (1)$

$T(\frac{n}{2}) = 2T(\frac{n}{4}) + 1 \quad (\text{Put } n = \frac{n}{2})$

$T(n) = 2(2T(\frac{n}{4}) + 1) + 1$

$\Rightarrow T(n) = 4T(\frac{n}{4}) + 2 + 1$

$\Rightarrow T(n) = 4T(\frac{n}{4}) + 3$

$T(\frac{n}{4}) = 2T(\frac{n}{8}) + 1 \quad (\text{from eqn } 1)$

$T(n) = 4(2T(\frac{n}{8}) + 1) + 3$

$T(n) = 8T(\frac{n}{8}) + 7$

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

$$T(n) = 4T\left(\frac{n}{4}\right) + 3$$

$$T(n) = 8T\left(\frac{n}{8}\right) + 7$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + 2^k - 1$$

$T(0)$ or $T(1) = 1$.

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$\Rightarrow k = \log_2 N$$

$$T(n) = 2^{\log_2 N} T(1) + 2^{\log_2 N} - 1$$

$$T(n) = N * (1) + N - 1$$

$$T(n) = 2N - 1$$

T.C $\Rightarrow O(N)$

fast power

```
Function pow(int a, int n){
    if(n == 0) return 1;

    long p = pow(a, n/2);

    if(n % 2 == 0) {
        return p * p;
    }
    else {
        return p * p * a;
    }
}
```

$T(n) = T(\frac{n}{2}) + 1$

$T(\frac{n}{2}) = T(\frac{n}{4}) + 1$

$T(n) = T(\frac{n}{4}) + 2$

$T(\frac{n}{4}) = T(\frac{n}{8}) + 1$

$T(n) = T(\frac{n}{8}) + 3$

$T(n) = T(\frac{n}{2^k}) + k$

$T(1) = 1$

$\frac{n}{2^k} = 1$

$n = 2^k$

$k = \log_2 N$

$T(n) = \underbrace{T(1)}_{(1)} + \log_2 N$

$T(n) = 1 + \log_2 N$

$T.C \rightarrow O(\log_2 N)$

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

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

Solve very complex.

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

$T(n) = 2T(n-1) + 1 \rightarrow O(2^n)$

T.C. will be done with one more method.

T.C → Time of a single function call * no. of function calls.

2^5

$\downarrow$

2^2

$\downarrow$

2^1

$\downarrow$

2^0

Space Complexity of recursive function

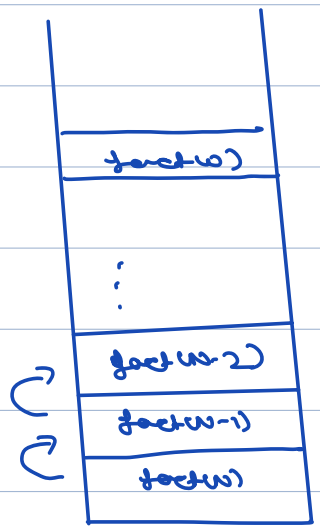
↳ Max stack space used.

1 (n)

```
int factorial(int N) {
    // base case
    if (N == 0) {
        return 1;
    }
    // recursive case
    return N * factorial(N-1);
}
```

T(n-1).

S.C → O(n).

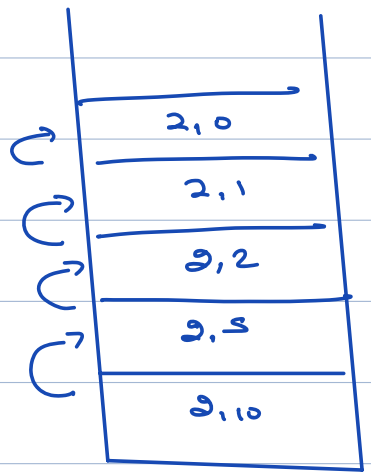


2, 10

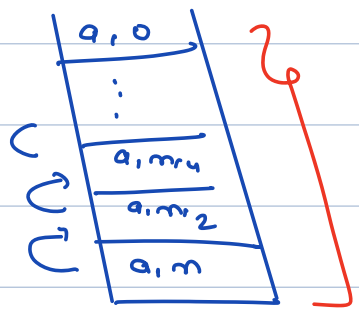
```
Function pow(int a, int n){
    if(n == 0) return 1;

    long p = pow(a, n/2);

    if(n % 2 == 0) {
        return p * p;
    }
    else {
        return p * p * a;
    }
}
```

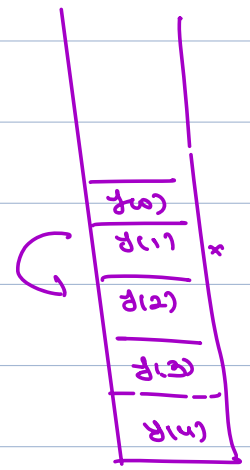


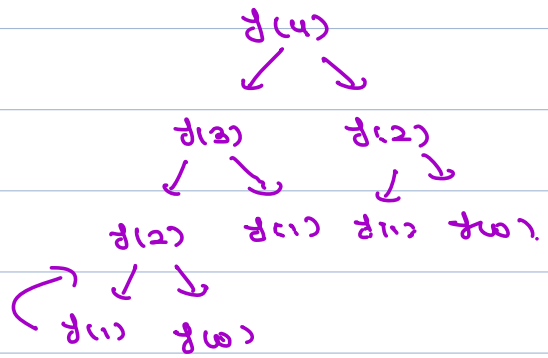
S.C → O(log n).

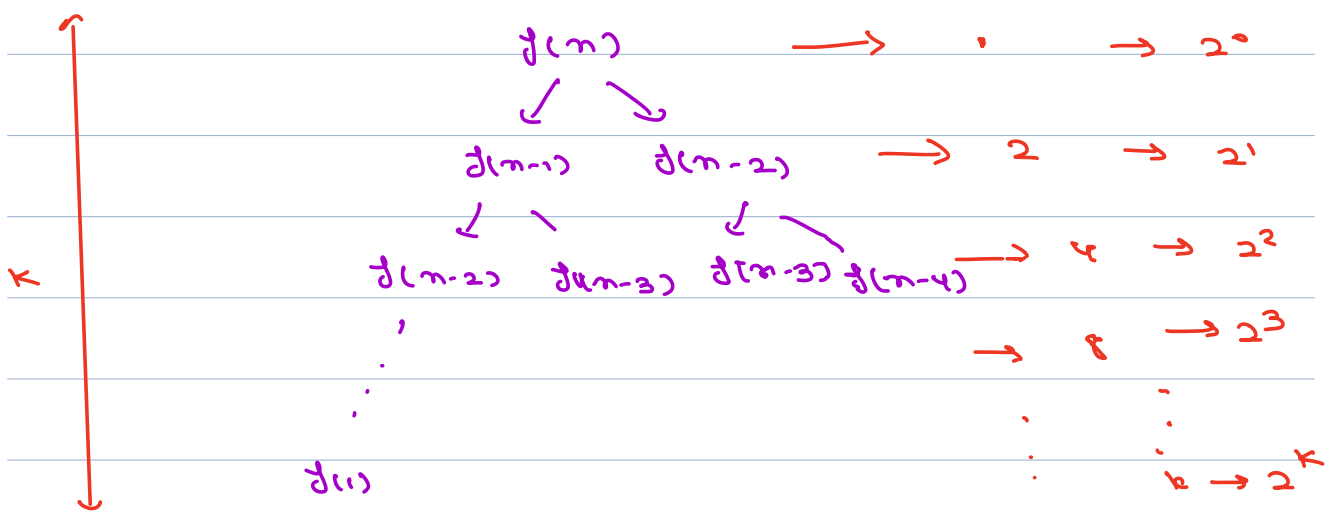


```
Function fibonacci(int n){
    if(n == 0 || n == 1) return n;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

S.C → O(n).







$T.C \rightarrow \quad 2^0 + 2^1 + 2^2 + 2^3 + \dots 2^{k}$

$T.C \rightarrow \quad 2^0 + 2^1 + 2^2 + \dots 2^n$

$T.C \rightarrow \quad 2^{n+1} \rightarrow T.C \rightarrow O(\underline{2^n})$.