1) DP
2) Trees
3) Sorting
4) Backtracking.
5) linked list
6) graph

Recursion ⇒ Function calling itself.

⇒ Solve a large problem using solution of smaller instances of the same problem.
↳ smaller input

Sum (n) ⇒ 1 + 2 + 3 + 4 ..... n-1 + n

Sum (n) ⇒ Sum (n-1) + n

How to Code Recursion :

1) Manifestation / ~~Assumption~~ / Leap of faith.

Sum (n) ⇒ Given n return the sum of 1st n natural numbers.

2) Main logic : (How are you using smaller instances to solve the bigger problem).

Sum (n) = Sum (n-1) + n

3) Base Condition.

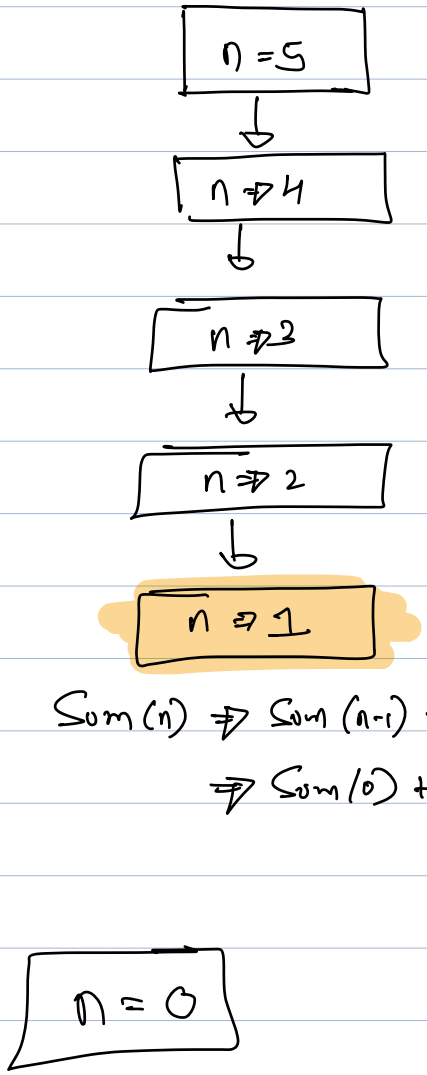
```
if (n == 1)
    return 1;

int sum (int n) {

    if (n <= 0)
        return 0

    return sum(n-1) + n;

}
```



Sum(n) ⇒ Sum(n-1) + 1

⇒ Sum(0) + 1

Q Factorial of n number.

$f(n) \Rightarrow 1 \times 2 \times 3 \times 4 \cdots (n-1) \times n$

$f(n) \Rightarrow f(n-1) \times n$

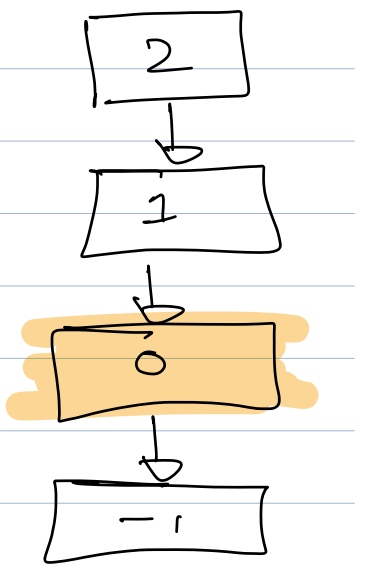
1) Manifestation

: $f(n) \Rightarrow$ returns $n!$.

2) Main logic

$f(n) = f(n-1) \times n$

$0! \Rightarrow 1$



3) Base Condition.

```
if (n == 0)
    return 1;
```

```
# int f(int n) {
    if (n == 0)
        return 1;
    return f(n-1) * n
}
```

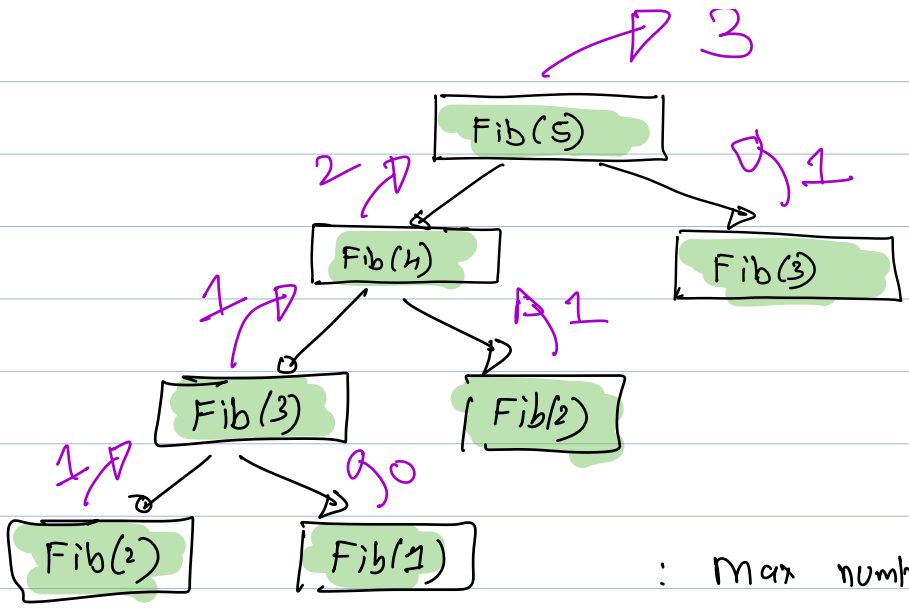
Fibonacci

1	2	3	4	5	6	7
0	1	1	2	3	5	8

$$f(n) \Rightarrow f(n-1) + f(n-2)$$

1) $f(n) \Rightarrow$ return n^{th} fibo number.
2) $f(n) \Rightarrow f(n-1) + f(n-2)$
3) if (n == 1) return 0; if (n == 2) return 1

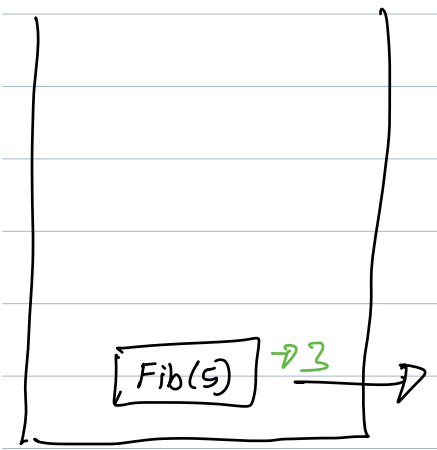
```
int f(n) {
    if (n == 1)          - 1
        return 0;
    if (n == 2)          - 2
        return 1;
    return f(n-1) + f(n-2);   - 3
}
```



Fib(n)

: max number of function in call stack = height of the recursion tree.

For this problem ⇒ height ⇒ n



fib(4) + fib(3)

⇒ Space Complexity ⇒ number of function in call stack + memory taken by the function.

SC: O(n)

Time Complexity.

1) Method of Substitution

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

$T(n-1) \Rightarrow 1 + T(n-2)$

$T(n) \Rightarrow 1 + 1 + T(n-2)$

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

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

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

$T(1) = 1$

$n-k = 1$

$k = n-1$

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

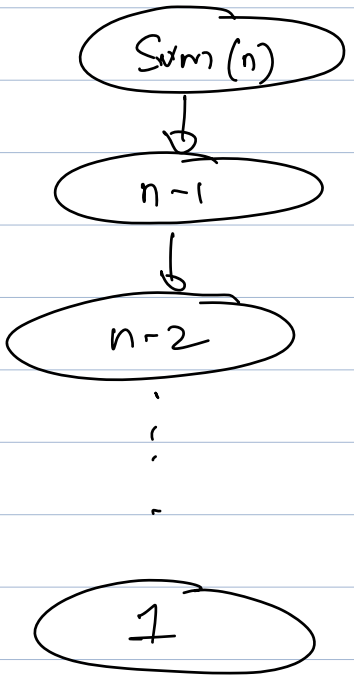
$n-1 + 1$

$\boxed{T(n) \Rightarrow n}$

Tc: $O(n)$

2) Time taken ⇒ (no of function calls) × (time taken by each function)

if (time taken is same for every func call)



no of function calls ⇒ n

Time taken ⇒ n × 1

TC : O(n)

TODO

1) Fibonacci

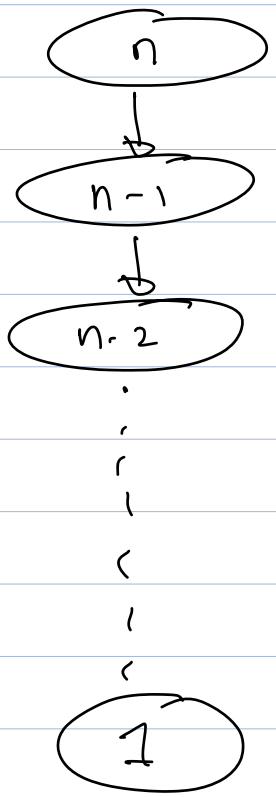
$T(n) \Rightarrow 1 + T(n-1) + T(n-2)$

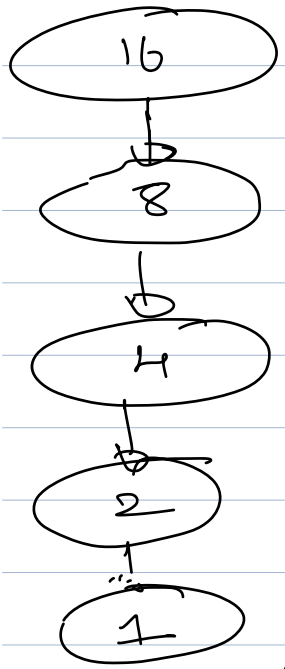
$T(n) \Rightarrow 1 + 2T(n-1)$

pow (a, n) mod m

$a^n \Rightarrow a \times a^{n-1}$

$a^n \Rightarrow a^{n/2} \times a^{n/2}$



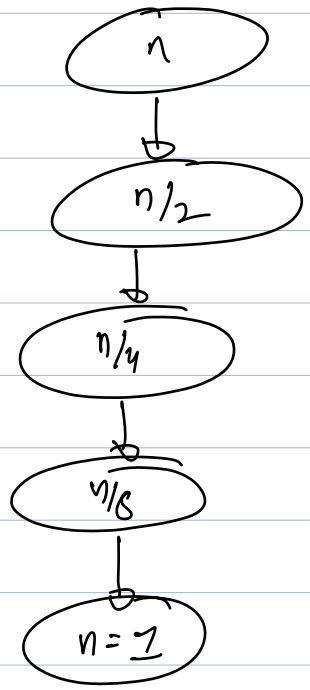


if n => odd

$a^5 \Rightarrow a^2 a^2 \times a$

$a^n \Rightarrow a^{n/2} \times a^{n/2} \times a$

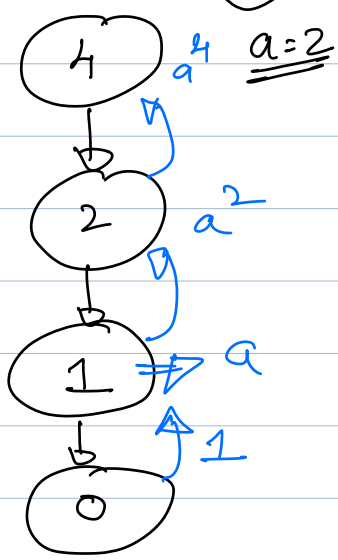
#



TC: (log n)

SC: (log n)

2^4



```
int power (int a, int n, int m) {

    if (n==0)                                   m => 10^9+7
        return 1

    if (n%2 ==0) {
return ((long)power (a,n/2,m) x power (a,n/2,m)) %m
    } else
        return (((long)power (a,n/2,m) x power (a,n/2,m))%m
    }                  xa) %m .

}
```

$$\Rightarrow ((a \times b)\%m \times c\%m)\%m$$

$$T(n) \Rightarrow 1 + 2T(n/2)$$

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

$$T(n) \Rightarrow 3 + 4T(n/4)$$

$$T(n) \Rightarrow 7 + 8T(n/8)$$

$$a^{\log_a n} \Rightarrow n$$

$$a^{\log_a n} \Rightarrow n$$

$$T(n) \Rightarrow 2^k - 1 + 2^k T\left(n/2^k\right)$$

$$T(n) \Rightarrow 2^{\log n} - 1 + 2^{\log n} T(1)$$

$$n - 1 + n$$

$$\geq n$$

$$T(n) \Rightarrow n$$

$$T(1) \Rightarrow 1$$

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

$$n \Rightarrow 2^k$$

$$k \Rightarrow \log n$$

FAST Power. → $a^n \% m$

```
int fast-power (int a, int n, int m) {
    if (n==0)
        return 1;
    int temp => fast-power (a, n/2, m);
    temp => ((long) temp x temp) % m
    if (n % 2 == 0) {
        return temp;
    } else {
        return ((long) temp x a) % m
    }
}
```