Recursion

↳ function calling itself.

→ A problem is broken down into smaller problems to solve the bigger problem.

↓ subproblem.

Ex :- sum(n) = 1 + 2 + 3 + 4 . . . n

↓

sum(n) = sum(n-1) + n

↓ subproblem.

How to write a recursive code?

1) Assumption :- Assume your function works for a smaller input

2) Main logic :- Solving bigger problem with smaller problem.

3) Base case :- Answer of smallest i/p we know.

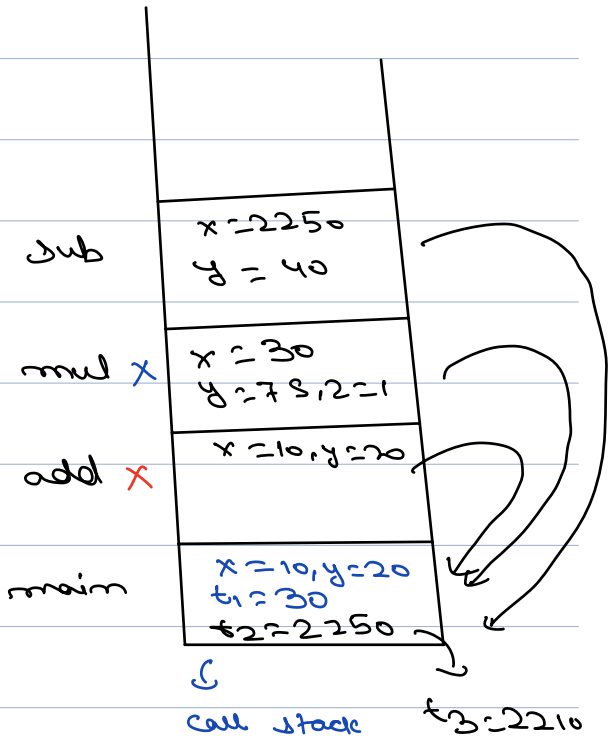
function call Tracing

```
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() {
    x = 10;
    y = 20;
    t1 = add(x,y);
    t2 = mul(t1,t5,1)
    t3 = sub(t2,40);
    print(t3)
}
```

Ques Given n, find its factorial.

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

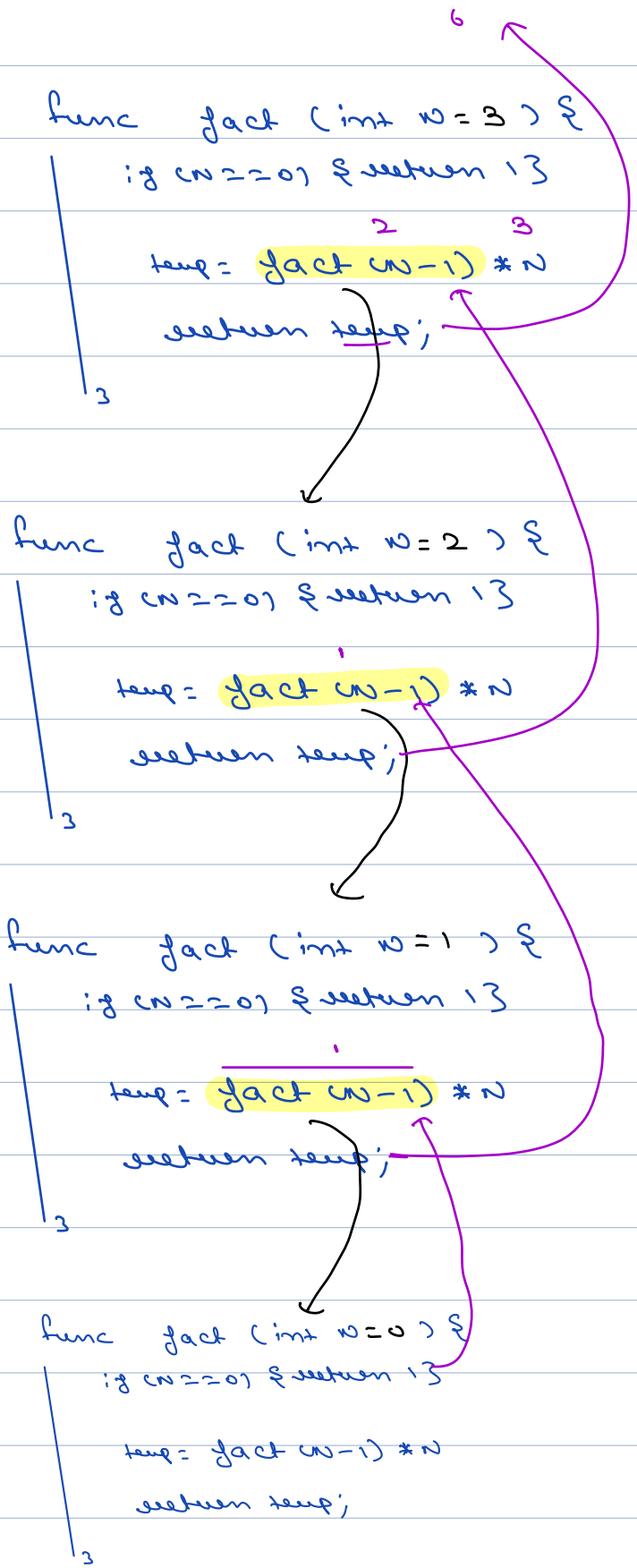
factorial(n) = factorial(n-1) * n

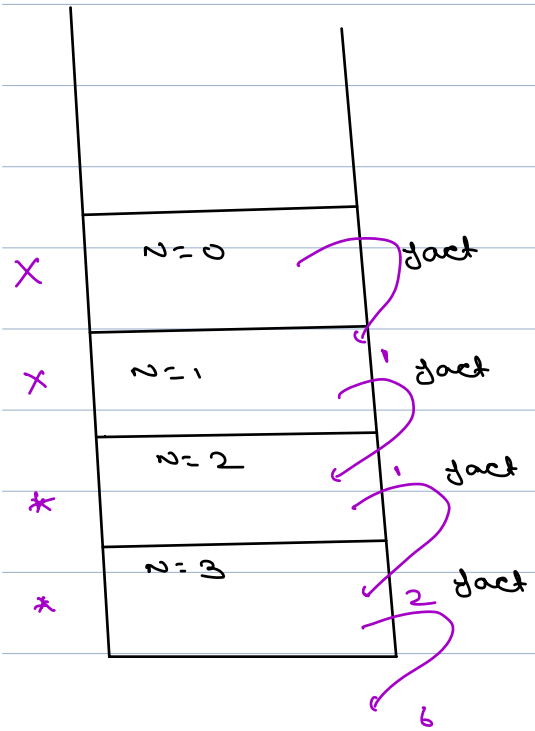
fact(5) = fact(4) * 5

$0! = 1$.

```
func fact (int n) {
    if (n==0) {return 1;}
    temp = fact (n-1) * n
    return temp;
}
```

fact(3)


6
func fact (int n = 3) {
if (n == 0) {return 1}
temp = fact (n-1) * n
return temp;
3
2
3
func fact (int n = 2) {
if (n == 0) {return 1}
temp = fact (n-1) * n
return temp;
3
1
2
func fact (int n = 1) {
if (n == 0) {return 1}
temp = fact (n-1) * n
return temp;
3
1
func fact (int n = 0) {
if (n == 0) {return 1}
temp = fact (n-1) * n
return temp;
3




```
func fact (int n) {
1  if (n==0) {return 1}
2  temp = fact (n-1) * n
3  return temp;
}3
```

fact(3) = 6.

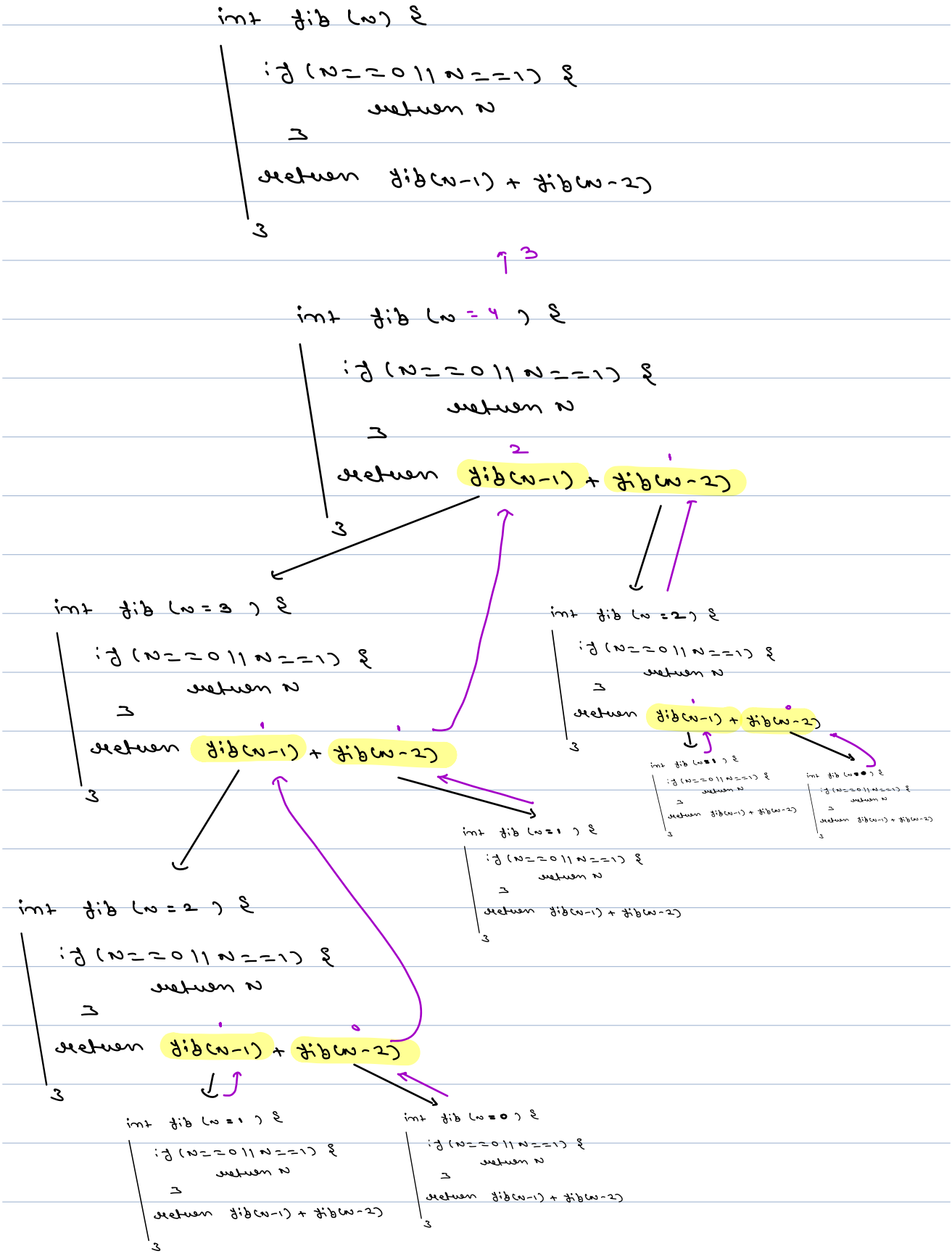
Ques 2 Fibonacci series

n =	0	1	2	3	4	5	6	7	8
	0	1	1	2	3	5	8	13	21

I/p → 6
O/p → 8

I/p → 8
O/p → 21

fib(n) = fib(n-1) + fib(n-2)


int fib (n) {
if (n == 0 || n == 1) {
return n
}
return fib(n-1) + fib(n-2)
}
int fib (n = 4) {
if (n == 0 || n == 1) {
return n
}
return fib(n-1) + fib(n-2)
2
1
int fib (n = 3) {
if (n == 0 || n == 1) {
return n
}
return fib(n-1) + fib(n-2)
int fib (n = 2) {
if (n == 0 || n == 1) {
return n
}
return fib(n-1) + fib(n-2)
int fib (n = 1) {
if (n == 0 || n == 1) {
return n
}
return fib(n-1) + fib(n-2)
int fib (n = 2) {
if (n == 0 || n == 1) {
return n
}
return fib(n-1) + fib(n-2)
int fib (n = 1) {
if (n == 0 || n == 1) {
return n
}
return fib(n-1) + fib(n-2)
int fib (n = 0) {
if (n == 0 || n == 1) {
return n
}
return fib(n-1) + fib(n-2)

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

Qu: Given $a, n \rightarrow$ calculate a^n

e.g., $a = 2, n = 3 \longrightarrow 8$.

$2^5 = 2^4 * 2$

$pow(a, n) = pow(a, n-1) * a$

```
func Power (a, n) {
    if (n == 0) { return 1 }
    return Power(a, n-1) * a
}
```

Power (2, 3) ⤴ 8

↓ ⤴ 4

Power (2, 2)

↓ ⤴ 2

Power (2, 1)

↓ ⤴ 1

Power (2, 0)

$2^4 \longrightarrow 2^3 * 2$

$2^{10} \longrightarrow 2^9 * 2$

$2^4 = 2^2 * 2^2$

$2^{10} = 2^5 * 2^5$

$2^9 = 2^4 * 2^4 * 2$

n == even

$pow(a, n) = pow(a, n/2) * pow(a, n/2)$

n == odd

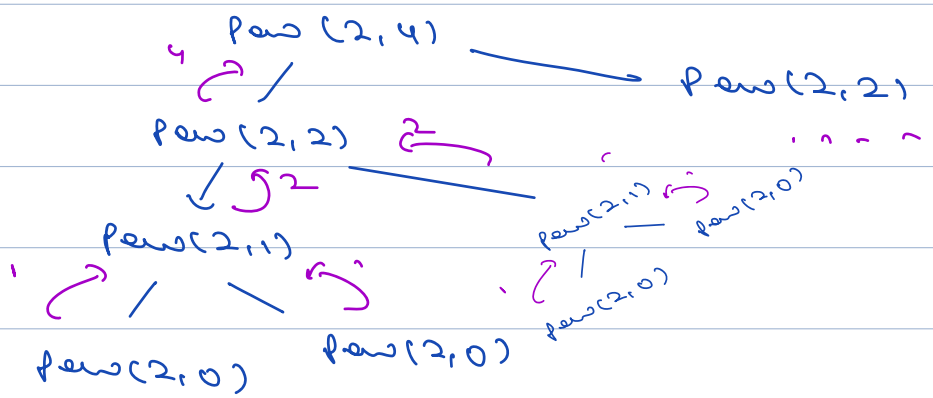
$pow(a, n) = pow(a, n/2) * pow(a, n/2) * a$

$2^7 \longrightarrow 2^3 * 2^3 * 2$

```
func 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;
    }
}
```

```
func   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 Exponentiation

```
func    pow2 ( a, n)
        if (n==0) { return 1 }
        temp = pow(a, n/2);
        if (n%2==0) {
            return temp*temp;
        }
        else {
            return temp*temp*a;
        }
    }
```

→ 1024

pow(2,10)
→ 32

pow(2,5)
→ 4

pow(2,2)
→ 2

pow(2,1)
→ 1

pow(2,0)

Time Complexity

$T(0) = 1$

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

`factorial(int N)` → $T(n)$

`factorial(N-1)` → $T(n-1)$

→ recurrence relation

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

$$T(n) = T(n-1) + 1 \quad \smile$$

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

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

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

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

Generic

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

if $n-k = 0$,

$n = k$

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

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

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

T.C → $O(n)$.

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

$T(0) = 1 \simeq T(1)$

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

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

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

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

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

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

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

$T(\frac{n}{8}) = 2T(\frac{n}{16}) + 1$

$\Rightarrow T(n) = 16T(\frac{n}{16}) + 15$

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

$\frac{n}{2^k} = 1 \Rightarrow n = 2^k \Rightarrow k = \log_2 n$

$$T(n) = 2^{\log_2 n} * T\left(\frac{n}{2^{\log_2 n}}\right) + 2^{\log_2 n} - 1$$

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

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

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

$$T(n) = 2n - 1 \Rightarrow T.C \rightarrow O(n)$$

→ fast Exponentiation

```
func  pow2 ( a, n) → T(n)
    if (n==0) { return 1 }
    temp = pow(a, n/2); → T(n/2)
    if (n%2==0) {
        return temp * temp;
    }
    else {
        return temp * temp * a;
    }
```

$T(1) = 1$.

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

$T(n/2) = T(n/4) + 1$

$$\Rightarrow T(n) = T(n/4) + 2$$

$T(n/4) = T(n/8) + 1$

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

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

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

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

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

$T(1) = 1$.

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

$n = 2^k$

$k = \log_2 n$

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

$T(n)$

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

$T(n-1)$

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

$T(n) = 2T(n-1) + 1 \quad \dashrightarrow$ H.W.

$O(2^n)$.

Another way of calculating T.C.

T.C → Time taken by one function call * no. of function calls.

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

fact(6) → fact(5) → fact(4) → fact(3) → fact(2) → fact(1) → fact(0)

fact(n) → fact(n-1) → fact(n-2) → ... → fact(0)

T.C → 1 * n ⇒ O(n).

→ Fast Exponentiation

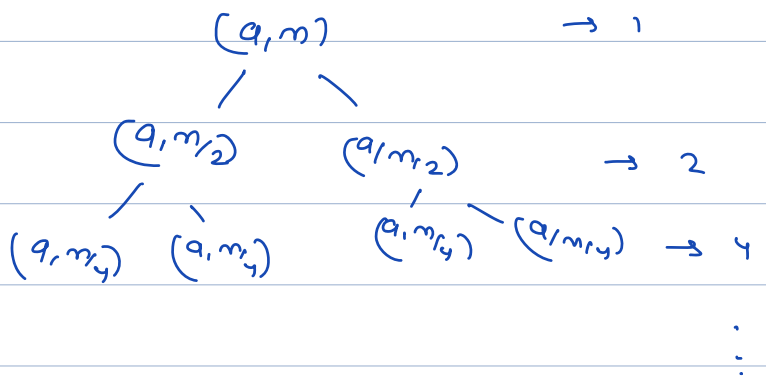
```
func  pow2 ( a, n) → T(n)
if (n==0) { return 1 }
temp = pow(a, n/2); → T(n/2)
if (n%2==0) {
    return temp * temp;
}
else {
    return temp * temp * a;
}
```

pow(2,5) → pow(2,2) → pow(2,1) → pow(2,0)

pow(a,n) → pow(a,n/2) → pow(a,n/4) → ... → pow(a,0)

log n

T.C → O(log n)

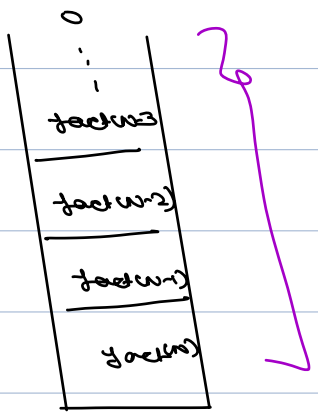


S.C of Recursive codes.

↓

Max^m fun^cs in stack at any time.

for factorial



S.C → O(n).

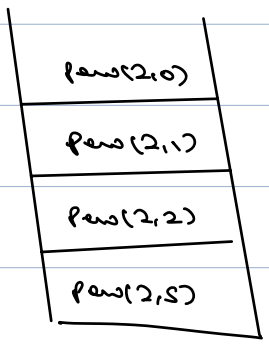
→ fast Exponentiation

```
func  pow2 ( a, n)  → T(n)
{
    if (n==0) { return 1 }
    temp = pow(a, n/2);  → T(n/2)
    if (n%2==0) {
        return temp*temp;
    }
    else {
        return temp*temp*a;
    }
}
```

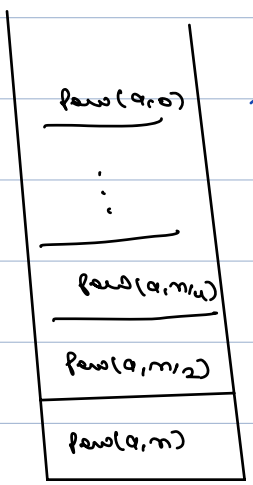
pow(2,5)
↓
pow(2,2)
↓
pow(2,1)
↓
pow(2,0)

pow(a,n)
↓
pow(a,n/2)
↓
pow(a,n/4)
⋮
pow(a,0)

log n



S.C → O(log n)



log n

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