

12/10/2023
Thursday

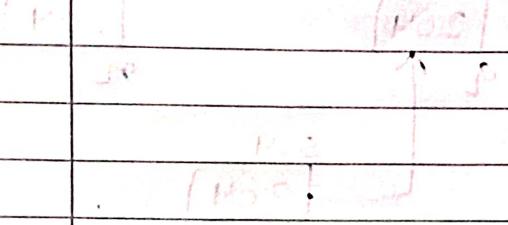
* How recursion works?

Let,

```
main() ->
{ > func(n);
```

func(n);

}



```
func(n) ->
```

```
{ > if (n==0) return; // Base case
```

// Processing

int a, b, c;

func(n-1);

}

PO1 : 1

PO2 : 2

PO3 : 3

PO4 : 4

PO5 : 5

PO6 : 6

PO7 : 7

PO8 : 8

PO9 : 9

PO10 : 10

PO11 : 11

PO12 : 12

PO13 : 13

PO14 : 14

PO15 : 15

PO16 : 16

PO17 : 17

PO18 : 18

PO19 : 19

PO20 : 20

PO21 : 21

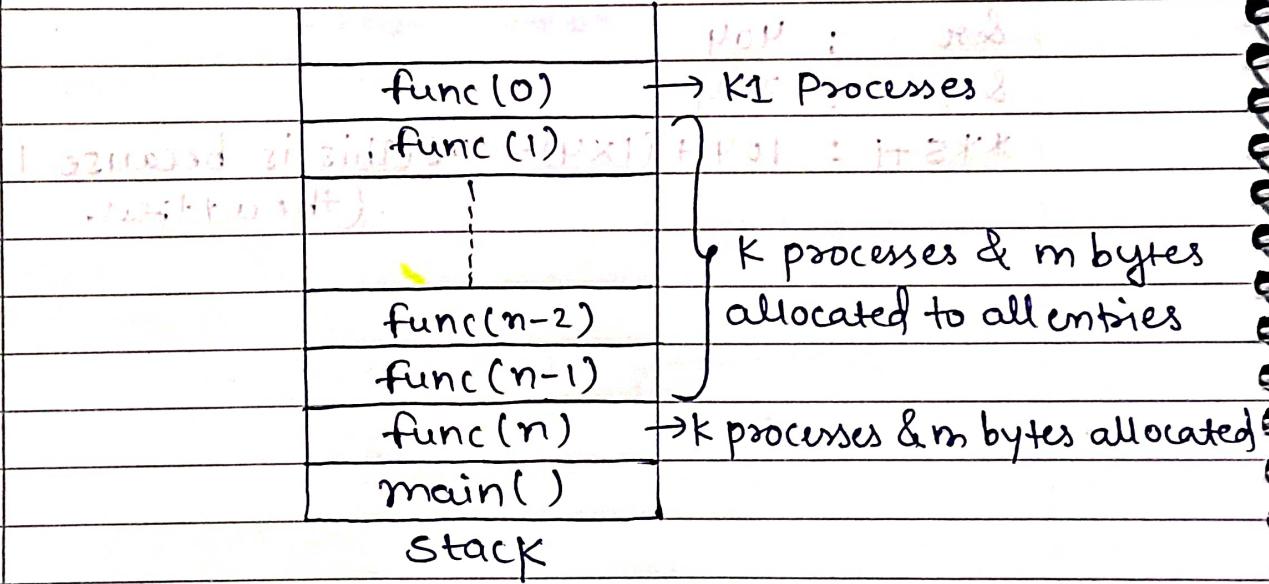
PO22 : 22

PO23 : 23

PO24 : 24

PO25 : 25

Stack → Stack works on last In: first Out principle (LIFO).



main() calls func(n). func(n) further calls func(n-1) & func(n-1) calls to func(n-2) till the base case condition gets true. Entries get created in stack when they are called. When system finds the base case, now, stack unwinding occurs and entries get removed one by one.

So, OS makes the recursion possible by maintaining the stack.

- * Time & Space Complexity, calculation for recursive Solutions →

- (1) Print Array →

```
void printArray (int a[], int n){  
    if (n==0) return; } K - processings  
    cout << *a << " ";  
    printArray (a+1, n-1); } Function gets  
} called again  

$$(24 + 24)(0 - (n-1))T$$
  

$$(24)0 = (n-1)T$$
  
till base case found.
```

$$\text{So, } f(n) = k + f(n-1)$$

- Time Complexity
- formula Based Method →

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

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

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

$$\vdots \quad \vdots \quad \vdots$$

$$T(1) = k + T(0)$$

$$T(0) = k_1$$

Adding both sides →

$$T(n) + T(n-1) + T(n-2) + \dots + T(1) + T(0) = k + T(n-1) + k + T(n-2) + k + T(n-3) + \dots + T(0) + k + k_1$$

$$[T(n) = nk + k_1] \rightarrow O(nk + k_1) \rightarrow O(n)$$

Recursive Tree Method (visual method)

$f(n) \rightarrow K$ time

$f(n-1) \rightarrow K$ time

$f(n-2) \rightarrow K$

$f(1) \rightarrow K$ time

$f(0) \rightarrow K_1$ time

$$T(n) = O(nK + K_1)$$

$$T(n) = O(n)$$

Space Complexity

func(0)	$\rightarrow m_1 T$
func(1)	$\rightarrow (l-1)T$
func(2)	$\rightarrow (l-2)T$
func(n-2)	$\rightarrow m$ bytes allocated per memory entry
func(n-1)	
func(n)	
main()	

So, $O(n^*m + m_1) \rightarrow O(n)$ as m_1 and m are constant.

(2) Factorial → $n! = n \times (n-1) \times (n-2) \times \dots \times 1$

int fact(int n)

{ }

if ($n == 1$)

~~return 1;~~

1. (a) 3

return n * fact()

卷之三

Based Method →

Formula Based Method \rightarrow $(a^p)^q = a^{pq}$

$$\text{So, } f(n) = \boxed{n *} f(n-1)$$

let it takes

k time

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

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

$$T(n-2) = K + I(n-3)$$

~~the results~~

and all the world

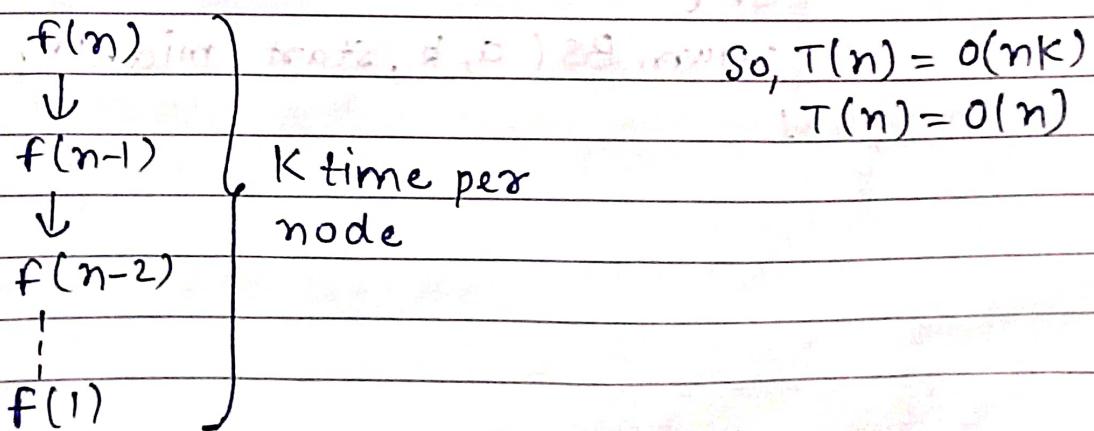
$$V = 2 \text{ km/s} + \text{factor} \approx \text{binst}$$

$$l(n) = O(nk) = O(n \log n)$$

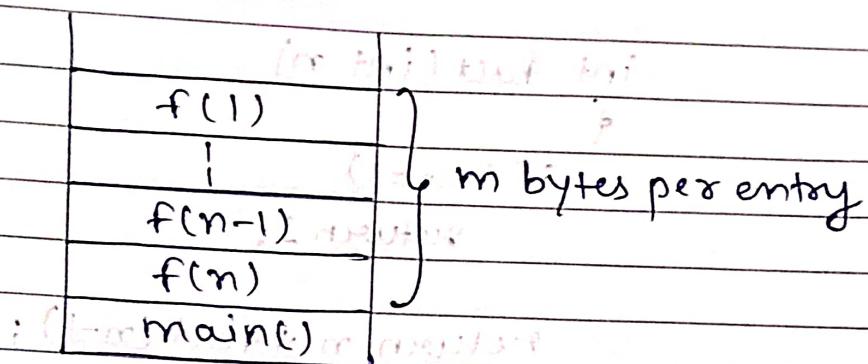
正月三日，同人游南湖，天晴，水碧，游人如织。

$I(n) = O(n)$ \leftarrow $I(n) \leq cn$ for some constant c

Recursive Tree Method →



Space complexity of factorial \rightarrow Recursion Tree



So, $O(n * m) \rightarrow O(n)$ as m is constant.

(3)

Binary Search \rightarrow Recursion Tree

```
int BS(int a[], int k, int start, int end)  
{  
    if (start > end) return -1;  
    int mid = start + (end - start) / 2;
```

```
    if (a[mid] == k) return mid;  
    else if (k > a[mid]) {  
        return BS(a, k, mid + 1, end);  
    } else {
```

```
        return BS(a, k, start, mid - 1);  
    }  
}
```

Time Complexity →

($\Theta(\log n)$) using

$$F(n) = K + F(n/2)$$

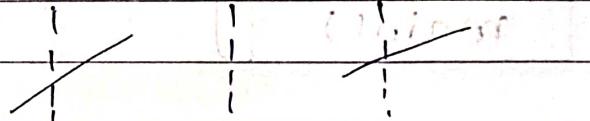
formula Based Method →

base case is

$$\text{start from } T(n) = K + F(n/2)$$

$$T(n/2) = K + F(n/4)$$

$$T(n/4) = K + F(n/8)$$



$$T(2) = K + T(1)$$

$$T(1) = K$$

$$T(n) = \alpha K$$

$$n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow \dots \rightarrow n/2^a$$

$$\text{as we start } \frac{n}{2^a} \text{ is called } 2^{-a} \text{ times}$$

$$2^{(a-1)} \text{ diff } + 2^a = n \text{ diff counts}$$

$$a = \log n$$

$$T(n) = O(K \log n)$$

$$T(n) = O(\log n)$$

Recursive Tree Method → f(n)

$$T(n) = O(\alpha K)$$

$$T(n) = O(K \log n)$$

$$T(n) = O(\log n)$$

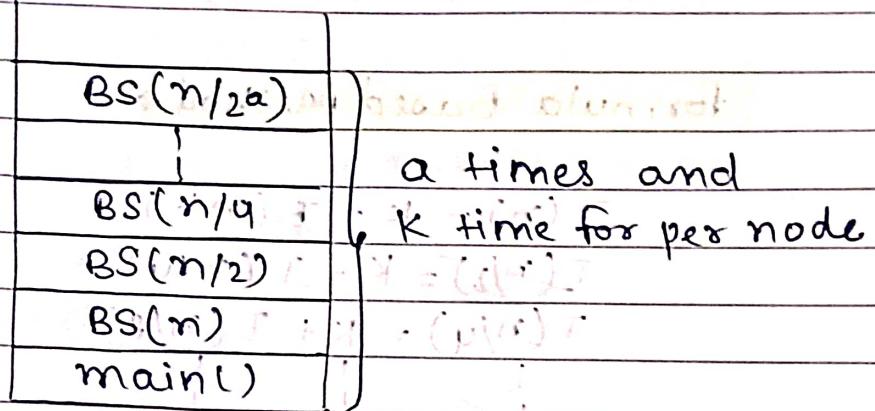
K = K time for per node

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

$$n = 2^a$$

a times α , $\alpha = \log n$

Space Complexity →



So, $O(a) \rightarrow O(\log n)$

(4) Fibonacci Series →

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

$\text{fib}(n) \rightarrow 2^0$ times

$\text{fib}(n-1) \rightarrow 2^1$ times

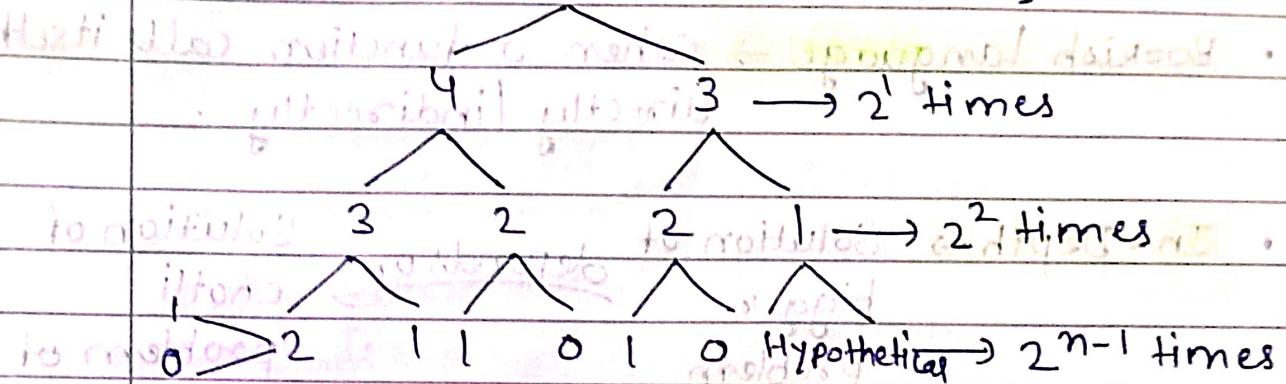
$\text{fib}(n-2) \rightarrow 2^2$ times

$\text{fib}(n-3) \rightarrow 2^3$ times

$\text{fib}(1) \rightarrow 2^{n-1}$ times

let for $\text{fib}(5)$,

$\text{fib}(5) \rightarrow 2^0$ times



Time complexity \rightarrow

$$T(n) \leq 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{n-1}$$

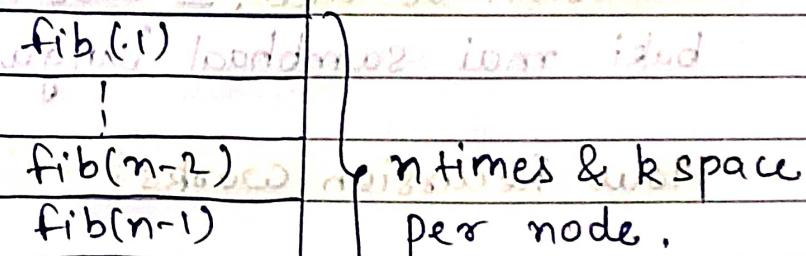
$$T(n) \leq 1 + 2 + 4 + 8 + \dots + 2^{n-1}$$

So upper bound

$$T(n) = O(2^n) \rightarrow \text{Upper bound}$$

Upper bound means max to max kitna time lgega.

Space complexity \rightarrow



$$\text{So, } O(nk) \rightarrow O(n)$$