12/10/2023
Thursday

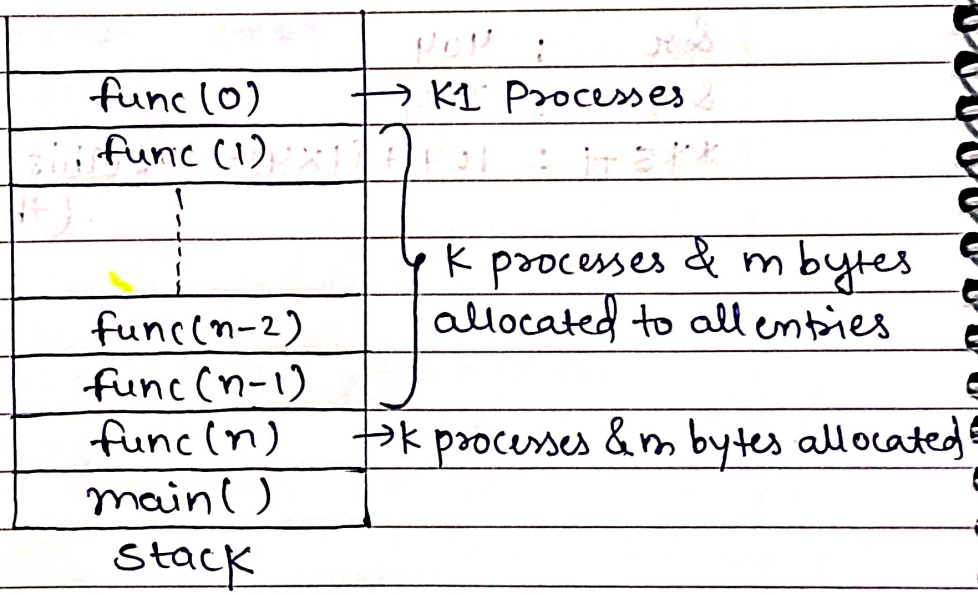
* **How recursion works?**

let,

```
main()
{
  func(n);
}
```

```
func(n)
{
  if(n==0) return; //Base case
  //Processing
    int a, b, c;
    func(n-1);
}
```

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 gets 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 maintaing the stack.

* Time & Space Complexity calculation for recursive solutions →

(1) Print Array →

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

K = processings

→ Function gets called again till base case found.

So, $f(n) = K + f(n-1)$

Time Complexity

- Formula Based Method →

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

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

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

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

$T(0) = K_1$

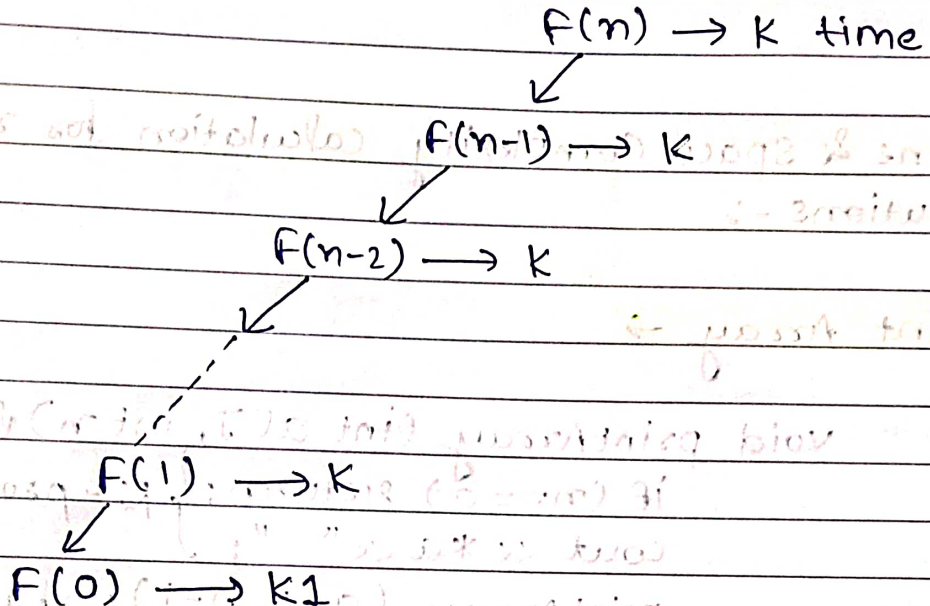
Adding both sides →

$T(n) + \cancel{T(n-1)} + \cancel{T(n-2)} + \dots + \cancel{T(1)} + \cancel{T(0)} = K + \cancel{T(n-1)} + K + \cancel{T(n-2)} + K + \cancel{T(n-3)} + \dots + \cancel{T(0)} + K + K_1$

$\boxed{T(n) = nK + K1} \rightarrow O(nK + K1) \rightarrow O(n)$

__/__/____

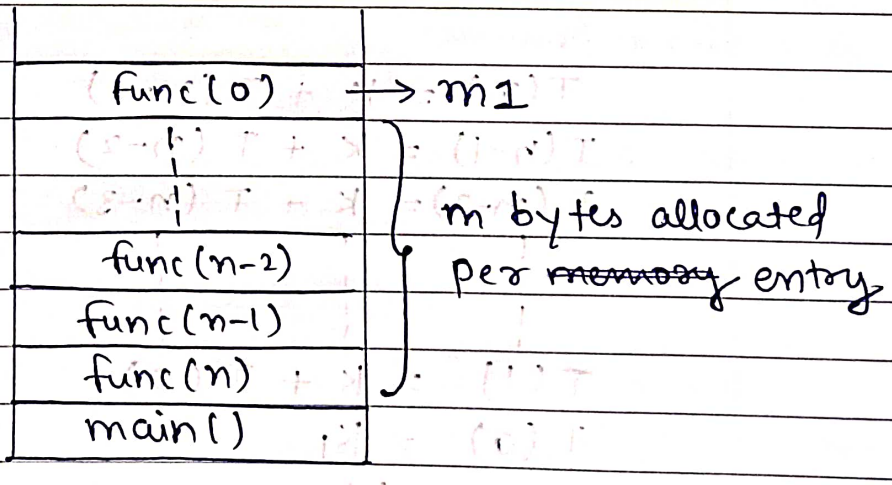
Recursive Tree Method (Visual method)



$T(n) = O(nK + K1)$

$T(n) = O(n)$

Space Complexity



So, $O(n*m + m1) \rightarrow O(n)$ as m and m1 are constant.

(2) **Factorial →**

```
int fact(int n)
{
    if (n==1)
        return 1;

    return n * fact(n-1);
}
```

Formula Based Method →

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

↓

let it takes K time

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

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

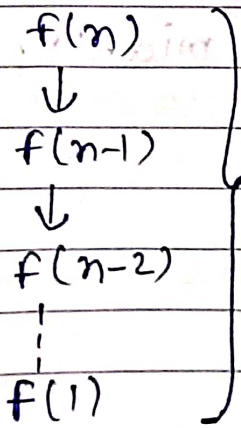
$T(n-2) = K + T(n-3)$

$T(1) = K$

$T(n) = O(nK)$

$T(n) = O(n)$

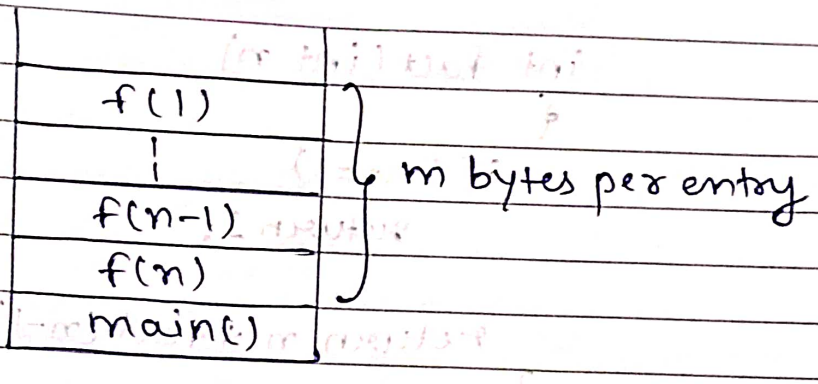
Recursive Tree Method →



So, $T(n) = O(nK)$

$T(n) = O(n)$

Space complexity of factorial →



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

(3) **Binary Search →**

```
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 →

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

formula Based Method →

$$T(n) = K + T(n/2)$$
$$T(n/2) = K + T(n/4)$$
$$T(n/4) = K + T(n/8)$$

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

$$T(n) = aK$$

$$n \longrightarrow n/2 \longrightarrow n/4 \longrightarrow n/8 \longrightarrow \cdots \longrightarrow n/2^a$$

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

$$2^a = n$$

$$a = \log n$$

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

Recursive Tree Method →

$f(n)$
↓
$f(n/2^1)$
↓
$f(n/2^2)$
↓
⋮
$f(n/2^a)$

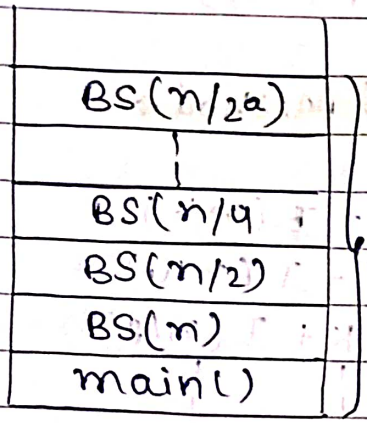
a times

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

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

K = K time for per node

Space Complexity →

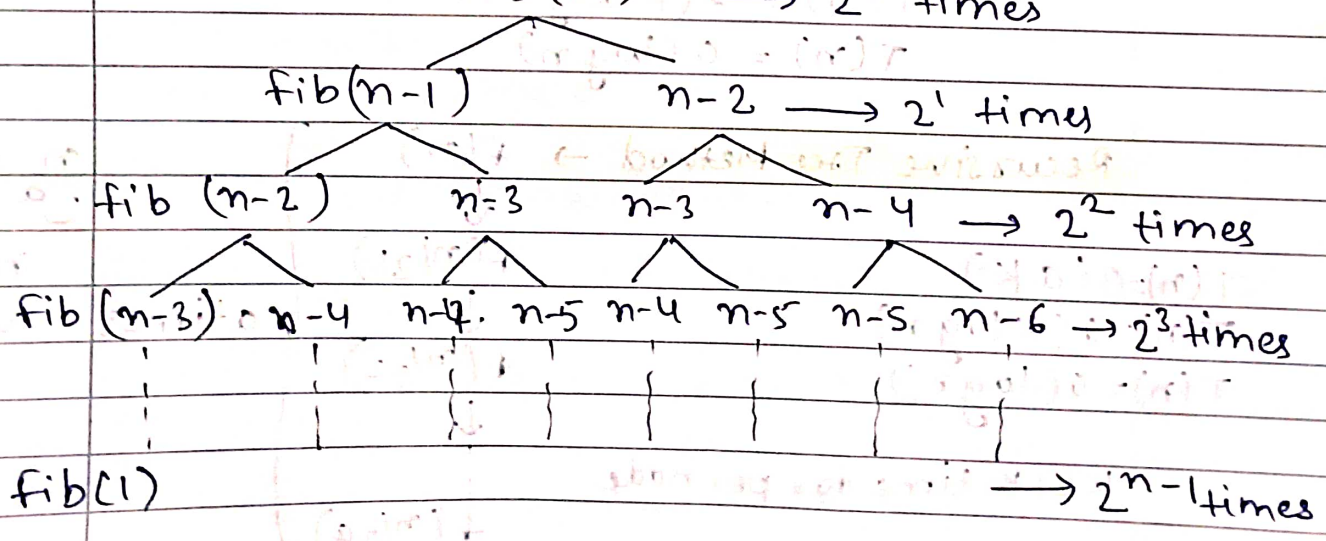


So, $O(a) \rightarrow O(\log n)$

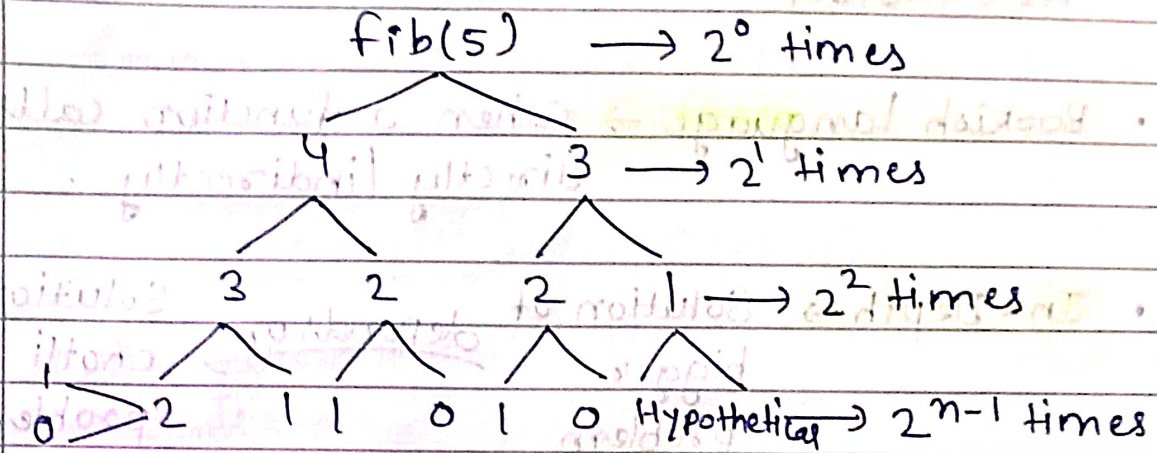
(4) Fibonacci Series →

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

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



let for fib(5),



Time complexity →

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

$T(n) \leq 2^n - 1$

$\boxed{T(n) = O(2^n)}$ → Upper bound

Upper bound means max to max kitna time lgega.

Space complexity →

fib(1)
!
fib(n-2)
fib(n-1)
fib(n)
main()

} n times & k space per node.

So, $O(nk) \rightarrow O(n)$