→ What happens for a function call

→ Space Complexity

→ Introduction to Complexity analysis of recursion.

→ Master Theorem

→ What happens when we make a function call??

At the line where you make a function call one operation is considered, and the no. of operations required in the function which is called is added to the overall time complexity

Inside the function fun we have 2 print statements & one function call.

fun → O(1)

inside gun() we have a loop → O(n)

Overall complexity of Time ⇒ O(n)

last

$t = 1000$

always

$O(n^2)$

$n \le 10^?$

per test case

variable

$O(t\,n^2)$

$O(n^2)$

$10^8 \times 10^3$

How to analyse Recursive Algorithms

$f(n) = n * f(n-1)$ → factorial

$f(n) = f(n-1) + f(n-2)$ → fibonacci

code

TC

```
fact(n) {
    if n == 0:
        return 1

    result = n * fact(n-1)
    return result
}
```

Base Case → $O(1)$

Recursive Intuition which is eventually just a function call.

This multiplication operation + returning is the self work.
$O(1)$

For calling function → $O(1)$
+
complexity to create a function → $O(x)$

$O(1) + O(1) + O(1) + O(x) \rightarrow O(x)$

Can I say → for fact(n-1) & fact(n-2) → complexity will be same → $O(x)$

```
fact(n) {
    if n == 0:
        return 1

    result = n * fact(n-1)
    return result
}
```

n = 0

Const

n-2

```
fact(n) {
    if n == 0:
        return 1

    result = n * fact(n-1)
    return result
}
```

n-3

n-1

```
fact(n) {
    if n == 0:
        return 1

    result = n * fact(n-1)
    return result
}
```

n-2

already print

```
fact(n) {
    if n == 0:
        return 1

    result = n * fact(n-1)
    return result
}
```

$f(n) \rightarrow O(x)$

$f(n-1) \rightarrow O(x)$

$f(n-2) \rightarrow O(x)$

$f(1) \rightarrow O(x)$

$f(0) \rightarrow O(x)$

n terms

$x = 1$

TC $O(x) \times n$

$x = ??$

$x = 1$

$\rightarrow$ TC $\rightarrow O(n)$

when you are recursively calling a function

TC → no. of times function is called × time required to execute just one function n=1

```
fun() {
    if n == 0:
        return 1

    for(i = 0 : i < m : i++ )
        print("Hi")
    result = n * fun(n-1)
    return result
}
```

n-2

```
fun() {
    if n == 0:
        return 1

    for(i = 0 : i < m : i++ )
        print("Hi")
    result = n * fun(n-1)
    return result
}
```

n-1

```
fun() {
    if n == 0:
        return 1

    for(i = 0 : i < m : i++ )
        print("Hi")
    result = n * fun(n-1)
    return result
}
```

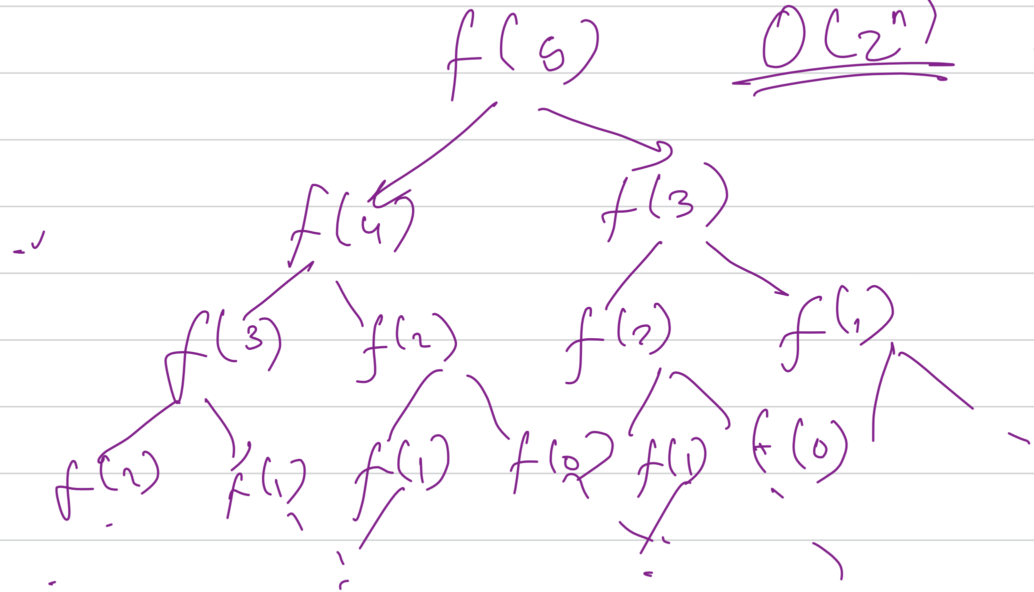
n

```
fun() {
    if n == 0:
        return 1

    for(i = 0 : i < m : i++ )
        print("Hi")
    result = n * fun(n-1)
    return result
}
```

```
fib(n) {
    if n == 0 or n == 1:
        return n

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

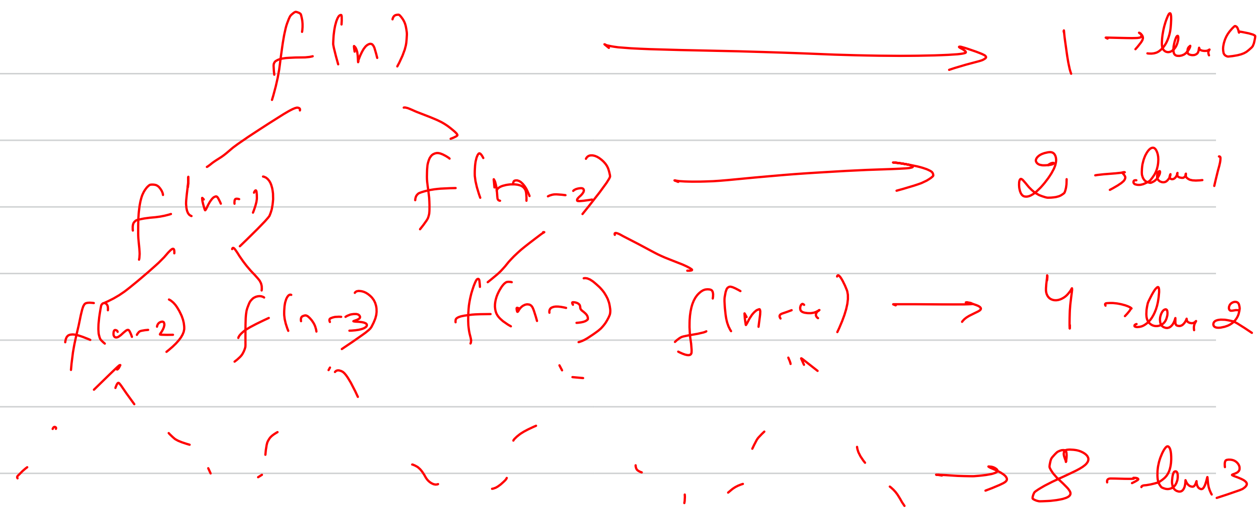
$$\boxed{\text{no of times fib is called} \times \text{TC for one fib call}}$$

$$O(2^n) \times O(1)$$



$$TC \rightarrow O(2^n)$$

$\frac{2^{n+2}}{2^n}$



$2^n + 2^{n-1} + 2^{n-2} - - - - - - - 8 + 4 + 2 + 1$

$\to \overbrace{1 + 2 + 4 + 8 - - - - - - 2^{n-1} + 2^n}^{n+1 \text{ terms}}$ GP

$\to 1 \times \frac{(2^{n+1} - 1)}{2 - 1} \approx O(2^n)$

$2^n \to$ lev n

~~Discrete maths~~

$$f(n) = 3 * f(n-1) \quad \text{if } n > 0$$

$$= 1 \quad \text{otherwise}$$

$$\Downarrow$$

$$f(n) = f(n-1) + f(n-1) + f(n-1) \quad O(3^n)$$

$$f(n) = 3f(n-1) \qquad f(n-1) = 3f(n-2)$$

$$= 3 \times (3f(n-2)) \qquad f(n-2) = 3f(n-3)$$

$$= 3 \times (3 \times 3f(n-3)) = 3^3 f(n-3)$$

$$= 3^3 \times 3f(n-4) = 3^4 f(n-4)$$

$$= 3^4 \times 3f(n-5) = 3^5 f(n-5)$$

$$\vdots$$

$$= 3^n f(n-n) = 3^n f(0) \qquad f(0) = 1$$

$$f(n) = 3^n$$

$$f(n) = \underbrace{a}_{\text{no of subproblems}} \times f(\underbrace{n-b}_{\text{size of smaller subproblem}})$$

$f(n) = n * f(n-1)$ $\rightarrow$ Self work

depends on $f(n-1)$ only

$f(n) = 5 x f(n-1)$ $\rightarrow$ Smart work

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

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

$f(n) \longrightarrow$

$n = k \rightarrow$ "the ir is true"

$k+1$

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

$f(n-1) = (n-1) \times f(n-2)$

$f(n)$ {

if ____

for (i = 0; i < n; i++)

$f(n-1)$

}

$= n \times ((n-1) \times f(n-2))$

$f(n-2) = (n-2) f(n-3)$

$= n \times (n-1) \times n-2 \times f(n-2)$

$= n f(n-4)$

$n \times n-1 \times n-2 \cdots$

$n!$

$f(n) \Rightarrow O(n^n)$

$n! < n^n$

f( )
{
  if ______
  return n*f(n-1)
}

$\longrightarrow$ f(n) $\longleftarrow$ f(n-1)

n*f(n-1)

$$f(n) = f(n-1) - O(1) \quad \text{if } n > 0$$

$$= 1 \quad \text{otherwise}$$

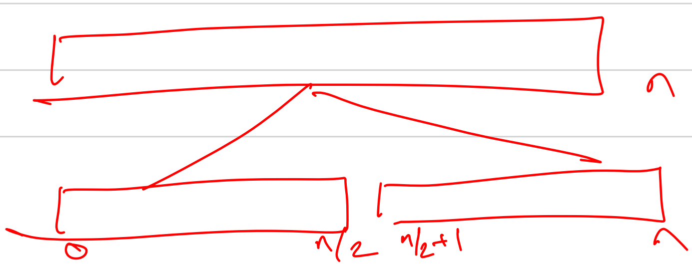
$$T(n) = aT\left(\frac{n}{b}\right) + O(n^d \times \log^p n)$$

recurrence of divide n conquer

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n^1 \times \log^0 n)$$

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

Merge Sort



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

Sat → 4-6

$O(2^n)$

```
fun(n)
    if (n ≤ 1) return 0;
    x = 2 * fun(n-1)
    y = f(n-1)
    return x + y
}
```

```
fun(n) {
    if (n ≤ 1) return 0;
    return 3 * f(n-1)
}
```

$O(n)$

$$f(n) = 2 * f(n-1) + f(n-1)$$

$$= 2\left(2f(n-2) + f(n-2)\right) + 2f(n-2) + f(n-1)$$

$$= \cancel{4}\, 6f(n-2) + 3f(n-2)$$

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

$+$

$$f(n) = 2f(n-1) - 1$$

$$= 2\left(2f(n-2) - 1\right) - 1$$

$f(n) \; O(g(n))$

$$= 2^2 f(n-2) - 2 - 1$$

$$= 2^2 \times \left(2 \times f(n-3) - 1\right) - 2 - 1$$

$$\Rightarrow 2^3 f(n-3) - 2^2 - 2 - 1$$

$$\vdots$$

$$\Rightarrow 2^n f(n-n) - 2^{n-1} - 2^{n-2} - 2^{n-3} - \cdots - 2 - 1$$

$$f(0) = 1$$

$$2^n(1) - 2^{n-1} - 2^{n-2} - 2^{n-3} - \cdots - 2 - 1$$

$$\Rightarrow 2^n - (\underbrace{2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2 + 1})$$

$$\Rightarrow 2^n - (\underline{1 \times 2^n} - 1)$$

$$\Rightarrow \cancel{2^n} - \cancel{2^n} + 1$$

$$\Rightarrow +1$$

$$f(n) = 3f(n-1) \longrightarrow 3^n$$

$$f(n) \to 2f(n-1) - 1 \longrightarrow +1$$

$$f(n) = 2f(n-1) - 1$$

→ Divide N Conquer based recurrences & there Time Complexity

→ Space Complexity

→ Merge Sort, Quick Sort, Binary Search etc asymptotic analysis

$$T(n) = a \times T\left(\frac{n}{b}\right) + O\left(n^d \log^p n\right)$$

no of operations reqd to complete a problem of size n

$a \geq 1$

$b > 1$

$d \geq 0$

$p \rightarrow$ real no.

Recurrence of Merge Sort $\longrightarrow$ $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

$a = 2,\ b = 2,\ d = 1,\ p = 0$

Recurrence of Binary Search $\longrightarrow$ $T(n) = T(n/2) + O(1)$

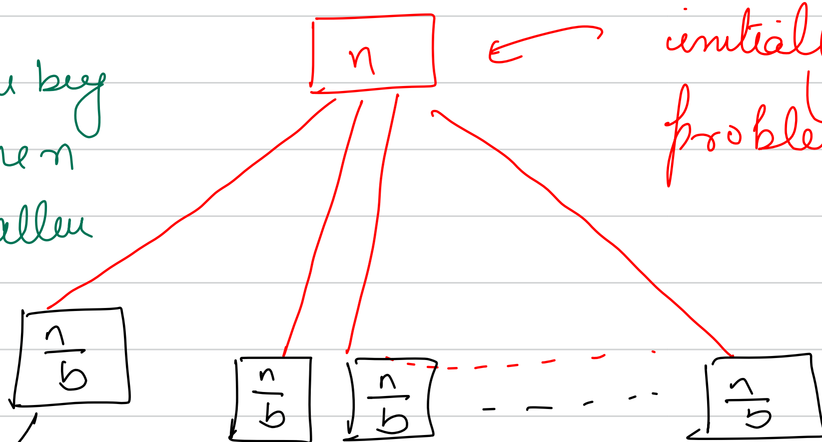
$a = 1,\ b = 2,\ d = 0,\ p = 0$

$$T(n) = a \times T\left(\frac{n}{b}\right) + O(n^d \log^p n)$$

meaning?

you divide the big problem of size n into 'a' smaller subproblems all of size $\frac{n}{b}$

initially you've a problem of size n



Total 'a' subproblem

size of smaller subproblem

This denotes the amount of extra work done in terms of no. of operation on a problem of size n.

Note → In terms of asymptotic notation, base of log doesn't **matter**.

$$O(\log_2 n) \xrightarrow{\text{log property}} O\Big(\log_3 n \times \underbrace{\frac{1}{\log_2 3}}_{\text{const}}\Big)$$

Qn why recurrences like $f(n) = \boxed{a}\, \underline{f(n-b)} + c f(n-d)$

$n \rightarrow \frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \frac{n}{8} \; \cdots\cdots$

$n \rightarrow n-b \rightarrow n-2b \rightarrow \underline{n-3b}$

$n-d$

$n-2d$

$n-3d$

$\vdots$

for simplicity in deriving the solution

Let's assume $p = 0$

$$\overbrace{n \rightarrow \frac{n}{b} \rightarrow \frac{n}{b^2} \rightarrow \frac{n}{b^3} - - - - - \frac{n}{b^k}}^{k \text{ terms}}$$

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

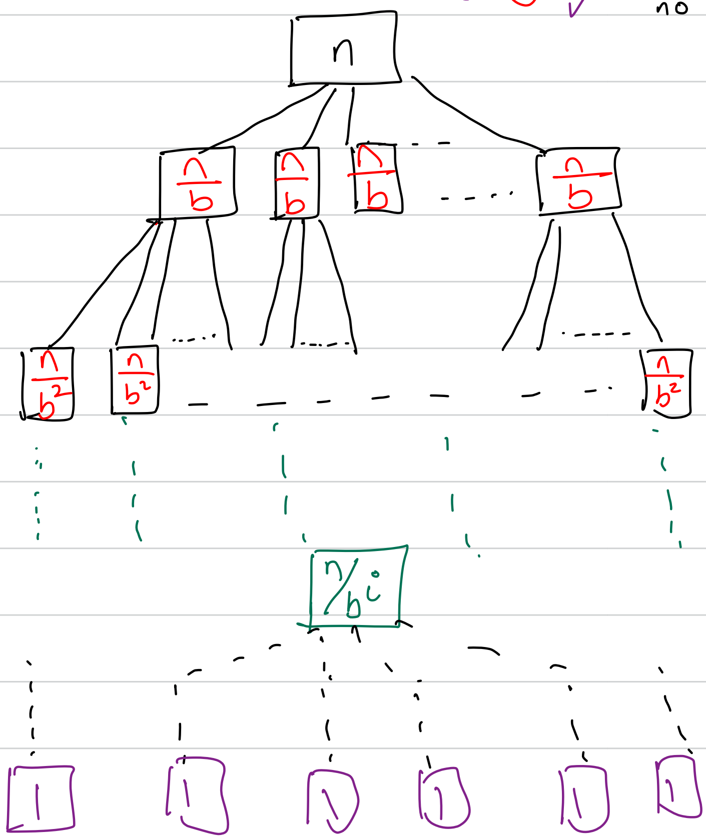
$$n = b^k$$

$$k = \log_b n$$

PTO

Let's try to solve the eqn

$$T(n) = a \times T\left(\frac{n}{b}\right) + O(n^d \log^p n)$$



level	no of problem	Time Complex
0	1	$O(n^d)$
1	a	$a \times O\left(\frac{n}{b}\right)^d = O(n^d) \times \frac{a}{b^d}$
2	a^2	$O(n^d)\left(\frac{a}{b^d}\right)^2$
i	a^i	$O(n^d)\left(\frac{a}{b^d}\right)^i$
last level $\log_b n$	$a^{\log_b n}$	

Time Complexity = $\sum_{i=0}^{\log_b n} O(n^d)\left(\frac{a}{b^d}\right)^i \rightarrow$ GP

$r = \frac{a}{b^d}$

$r \neq 1$

$a + ar + ar^2 + a^3 \text{ - - - - - - - - } ar^{n-1}$

Sum of GP $\rightarrow$ $a \times \frac{1 - r^n}{1 - r}$

Sum = $O(a)$ if $r < 1$

Sum = $O(ar^{n-1})$ if $r > 1$

Sum $\rightarrow$ $O(n \times a)$ if $r = 1$ $\rightarrow$ won't come from GP

Sum = $O(a)$ if $r < 1$

Sum = $O(ar^{n-1})$ if $r > 1$

Sum $\rightarrow$ $\underline{O(n \times a)}$ if $r = 1$

$\boxed{\text{Case 1}}$ $\quad r < 1 \rightarrow \frac{a}{b^d} < 1 \rightarrow a < b^d \rightarrow \log_b a < d$

$TC \rightarrow O(n^d \log^p n)$

$\rightarrow$ if $p < 0 \rightarrow T(n) = O\left(n^d \times \frac{1}{\log^p n}\right) \approx O(n^d)$

if $p \geq 0 \rightarrow T(n) = O(n^d \log^p n)$

Case II

$r > 1 \longrightarrow \frac{a}{b^d} > 1 \longrightarrow a > b^d \longrightarrow \log_b a > d$

$$TC \to O(n^d) \times \left(\frac{a}{b^d}\right)^{\log_b n} \Rightarrow O(n^d) \times \frac{a^{\log_b n}}{b^{d \times \log_b n}}$$

$b^{d \times \log_b n} \Rightarrow n^d$

$$\Rightarrow O(\cancel{n^d}) \times \frac{a^{\log_b n}}{\cancel{n^d}}$$

$$TC \Rightarrow O(a^{\log_b n}) \Longrightarrow O(n^{\log_b a})$$

(By log property)

Case III

$r = 1$

$\frac{a}{b^d} = 1 \longrightarrow d = \log_b a$

$r = 1$

$TC = O(n^d)(1 + \log_b n)$

$\approx O(n^d)(\log_b n)$ (avoid +1)

$\approx O(n^d \log n)$

$O(n^d \log^b n \times \log n) \approx O(n^d \log^{b+1} n)$

The previous 3 cases are called as

MASTER THEOREM

$$\rightarrow T(n) = \begin{cases} O(n^d) & d > \log_b a \\ O(n^d \log n) & d = \log_b a \\ O(n^{\log_b a}) & d < \log_b a \end{cases}$$

$$T(n) = a\,T\left(\frac{n}{b}\right) + O(n^d)$$

$$a \geq 1, \quad b > 1, \quad \underline{\underline{d \geq 0}}$$

Q → $T(n) = 3T\left(\frac{n}{2}\right) + O(n^2)$

$d = 2$ $\quad \log_b a \to \log_2 3$

$d > \log_b a$

$TC \to O(n^2)$

$$T(n) = \begin{cases} O(n^d) & d > \log_b a \\ O(n^d \log n) & d = \log_b a \\ O(n^{\log_b a}) & d < \log_b a \end{cases}$$

$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$

$a \geq 1, \; b > 1, \; d \geq 0$

Q_5 $T(n) = T\left(\frac{n}{2}\right) + O(n^2)$

$\rightarrow O(n^2)$

Q_6 $T(n) = 2T\left(\frac{n}{2}\right) + O(n\log n)$

$a = 2$ $b = 2$ $\log_b a \Rightarrow 1$

$d = 1$ $p = 1$ $d = \log_b a$

$n \times \log n$ $\log^p n$

$\Rightarrow O(n\log^2 n)$

$T(n) = \begin{cases} O(n^d) \\ O(n^d \log n) \\ O(n^{\log_b a}) \end{cases}$ $d > \log_b a$ / $d = \log_b a$ / $d < \log_b a$

$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$

$a > 0$, $b > 1$, $d \geq 0$

Q) $T(n) = 2^n T\left(\frac{n}{2}\right) + O(n^n)$ $\longrightarrow$ MT not applicable

Q) $T(n) = 2T(\sqrt{n}) + O(\log n)$

$\rightarrow$ $n = 2^m$ assume

$\log n = m$

$T(n) = T(2^m) = 2T\left(\sqrt{2^m}\right) + m$

$= 2T\left(2^{m/2}\right) + m$

assume $T(2^m) = S(m)$

$T(2^{m/2}) = S\left(\frac{m}{2}\right)$

$\Rightarrow \underbrace{S(m) = 2 \times S\left(\frac{m}{2}\right) + m}$ $\longrightarrow O(m \log m) \longrightarrow O(\log n \times \log(\log n))$

$$T(n) = \begin{cases} O(n^d) & d > \log_b a \\ O(n^d \log n) & d = \log_b a \\ O(n^{\log_b a}) & d < \log_b a \end{cases}$$

$T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$

$a > 0$, $b > 1$, $d \geq 0$

Q~ $T(n) = T(\sqrt{n}) + O(1)$

$\rightarrow O(\log\log n)$

$S(m) = S\left(\frac{m}{2}\right) + 1$

Space Complexity → Maximum space your algorithm took during the course of execution

→ Recursive logic → call stack
→ lists / arrays
→ datastructure → maps / Dict, ---

can contribute to space complexity

variables → X donot contribute to space complexity

garbage collector

In java → garbage collector → mark and sweep algo

manual access to free the memory

→ do import module from internal space → Yes my can know

https://pastebin.com/NUA9aX6W