Recursion → Time Complexity [HFN]

→ it will help to understand recursive task/algo's time complexity, also divide & conquer approach will be using this property.

simple recursive code with recursion tree breakdown.

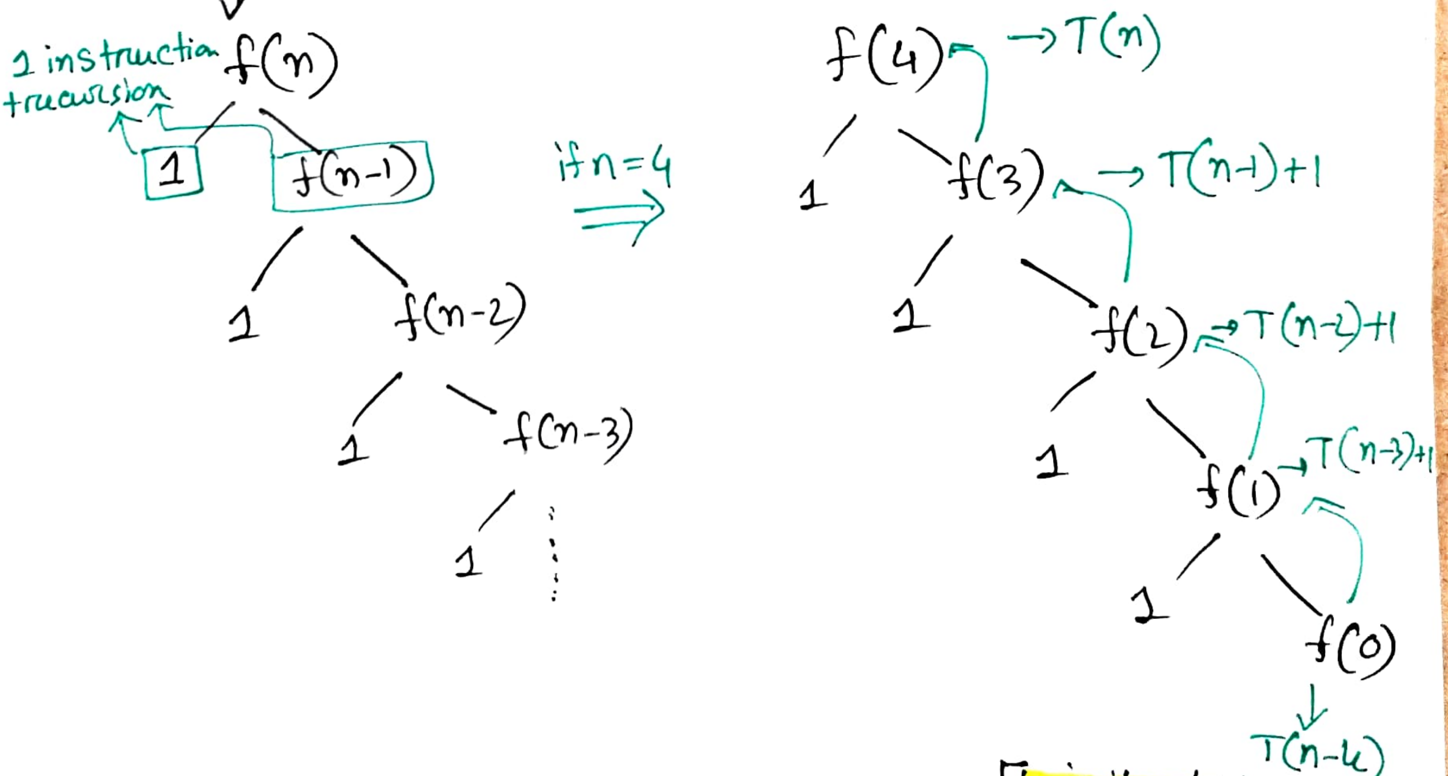
```
def f(x):
    if x == 0:
        return x
    print("1")
    f(x-1)
```

base case: `if x == 0: return x` — consider it as a 1 instruction

`print("1")` — 1 instruction

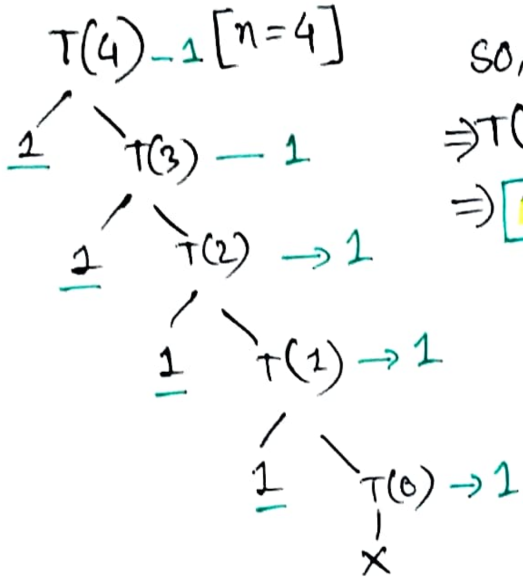
`f(x-1)` — recursive call

Assume, for input n size, which will be fed into the f(x) when calling, time function is T(n);



[k is the last recursive call where base case hits]

1// then we can just calculate the instruction for each recursive call to get the time.



so, for $n=4$, total 5 instructions [1 T(n) call first, then 4 printing instruction]

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

$\Rightarrow \boxed{T(n) = O(N)}$

This method of creating a recursion tree and observing the behavior. So this is the first way to solve a recursive time complexity problem : $\boxed{\text{① Analysing Recursion Tree.}}$ — ①

Keep in mind, this may seem easy for a simple recursive task such as this, but with different task where instructions per method call is bigger, then we have to count them as well, example: $c * T(n) \rightarrow$ ~~c*~~ $c * T(n/2)$
$\quad\quad\quad\quad\quad\quad\quad\quad \hookrightarrow c * T(n/2)$

2// Substitute method

same code,

```
f(x):                 ——— T(n)
    if x = =0:
        return        ] 1
    print("1")        — 1
    f(x-1)            ——— T(n-1)
```

we can establish recursive relation from here:

$$T(n) = \begin{cases} 1 & ; \text{ if } n=0 \\ T(n-1); & \text{ if } n>0 \end{cases}$$

now, for $n>0$, we need to solve the recursive process to get the time function;

$$T(n) = \boxed{T(n-1)} + 1$$

if $T(n) = T(n-1)+1$
then, $T(n-1) = T(n-2)+1$.
so substituting $T(n-1)$ there

$$\Rightarrow T(n) = [T(n-2)+1]+1$$

$$\Rightarrow T(n) = \boxed{T(n-2)}+2$$

same process

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

assume, the recursion will end at k^{th} time. then;

↳ ending means $T(n)$'s $n=0$ since that's the base case.

$$\Rightarrow T(n) = T(n-k) + k \quad (*)$$

so, for this to be the last call, $(n-k)$ must be 0.

$$n-k = 0 \Rightarrow n = k. \Rightarrow k = n$$

now substituting the k's value:

$$T(n) = T(n-n)+n$$
$$= \boxed{T(0)} + n$$
$$= 1+n = n$$

$T(n) = 1$ if $n=0$

$$\Rightarrow \boxed{T(n) = O(N)}$$

This was **substitute method**. It will always work for any recursive problem.

[$T(n) = T(n-1) + n$] related problem. [sol^n 1]

take this code as an example:

```
def f(x):                      ——— T(n)
    if x == 0:
        return "The end"       ] — 1
    c = 0                      ——— 1
    for i in range (x):        ——— n+1
        c += x                 ——— n
    print(c)                   ——— 1
    f(x-1)                     ——— T(n-1)
```

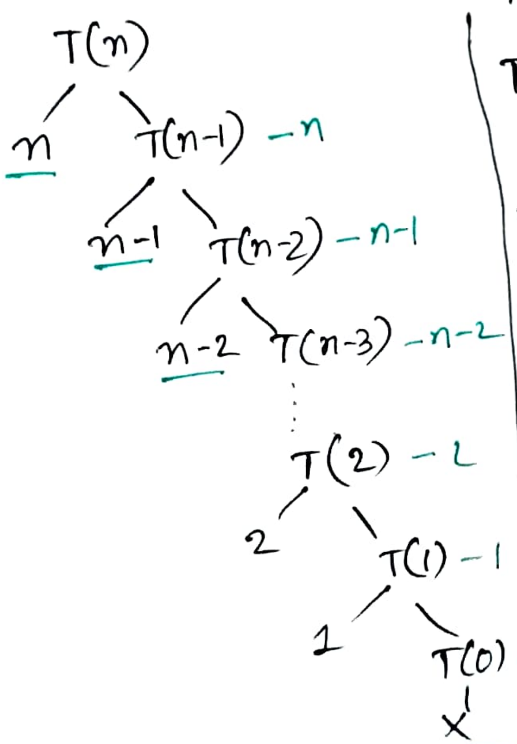
[in python, it runs $0 \to x-1$; checking x and breaking; so, $x+1$]

$$\text{so, } T(n) = 1 + n + 1 + n + 1 + T(n-1)$$
$$= \underline{2n+3} + T(n-1)$$
$$= T(n-1) + n$$

asymptotically, this is a linear time, replacing with n

$$\Rightarrow T(n) = \begin{cases} 1 & ; n = 0 \\ T(n-1) + n & ; n > 0 \end{cases}$$

now using $\boxed{\text{sol}^n \ 1}$ [rec. tree]



$T(n)$ = sum of the instructions in this tree

$$\Rightarrow T(n) = n + (n-1) + (n-2) + \dots + 2 + 1$$
$$\Rightarrow T(n) = 1 + 2 + 3 + \dots + (n)$$
$$\Rightarrow T(n) = \frac{n(n+1)}{2}$$
$$\Rightarrow \boxed{T(n) = O(N^2)}$$

[assuming you know how this is calculated in big-oh notation]

Sol^N 2 : ~~back~~ substitute method

$$T(n) = \begin{cases} 1 & ; n = 0 \\ T(n-1) + n & ; n > 0 \end{cases}$$

$T(n) = T(n-1) + n$ — step 1

$\Rightarrow T(n) = [T(n-2) + (n-1)] + n$ — step 2

$\Rightarrow T(n) = [T(n-3) + (n-2)] + (n-1) + n$ — step 3

$\vdots$

k^{th} step:

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

in k^{th} step, $n-k = 0$; $k = n$

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

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

$= 1 + 1 + 2 + 3 + \dots (n-1) + n$

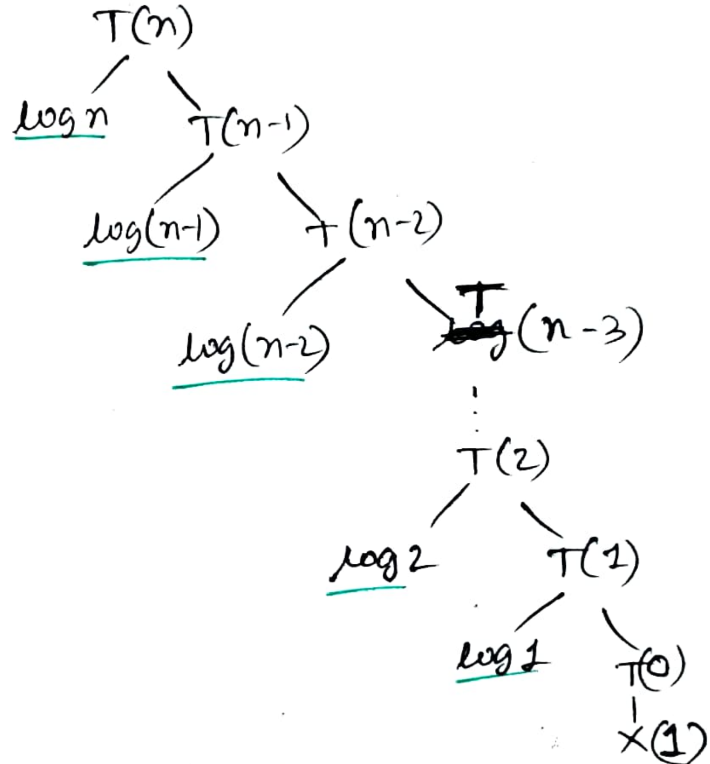
$= \dfrac{n(n+1)}{2}$

$\Rightarrow \boxed{T(n) = O(N^2)}$

log based problems:

```
def f(x):            — T(n)
    if x == 0:
        return       ] 1
    i = 1, c = 0     — 1
    while(i<x):
        c = i        ] log(n)
        i = i*2
    f(x-1)           — T(n-1)
```

Sol^n 1



Summing the instructions:

$\log n + \log(n-1) + \log(n-2) + \dots \log 2 + \log 1 + 1$

$\Rightarrow \log(n \times (n-1) \times (n-2) \times \dots 2 \times 1) + 1$

$\Rightarrow T(n) = \log(n!) + 1$

* mathematical explanation:

$\log(n!) = \log 1 + \log 2 + \dots + \log(n-1) + \log(n)$; as you can see, you cannot find the solution through summation since it's not a ~~[illegible]~~ functional series where a formula will give you the answer. Thankfully we don't need the exact answer!

so, consider $n! \Rightarrow 1 \times 2 \times 3 \times \dots \times (n-1) \times n$. if we try to find the upper bound of this function, there are "n" terms where the maximum value is n. so, rewriting:

$n! = n(n-1)(n-2)(n-3) \dots 2 \cdot 1$.

assume, the upperbound for all this term is n, throwing

...the constant terms,

So, $O(n!) \Rightarrow n \times n \times n \times \ldots n$ [n number of ns are multiplied]

$\Rightarrow O(n!) = O(n^n)$

$\Rightarrow \boxed{n! = O(N^N)}$

So, $T(n) = \log(n!)$

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

$= O(n \log n)$ $\quad \left[\log_{a} b^{b} = b \log_{a} a\right]$ (a = base)

Soln 2: Substitute

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + \log n; & n > 0 \end{cases}$$

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

$\Rightarrow T(n) = [T(n-2) + \log(n-1)] + \log n$

$\Rightarrow T(n) = [T(n-k)] + \log n + \log_{n-1} + \ldots \log_2 + \log_1$

in kth step, $n-k=0$; $n=k \Rightarrow k=n$

$\Rightarrow T(n) = T(n-n) + \log(n!)$

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

$= 1 + n \log n$

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

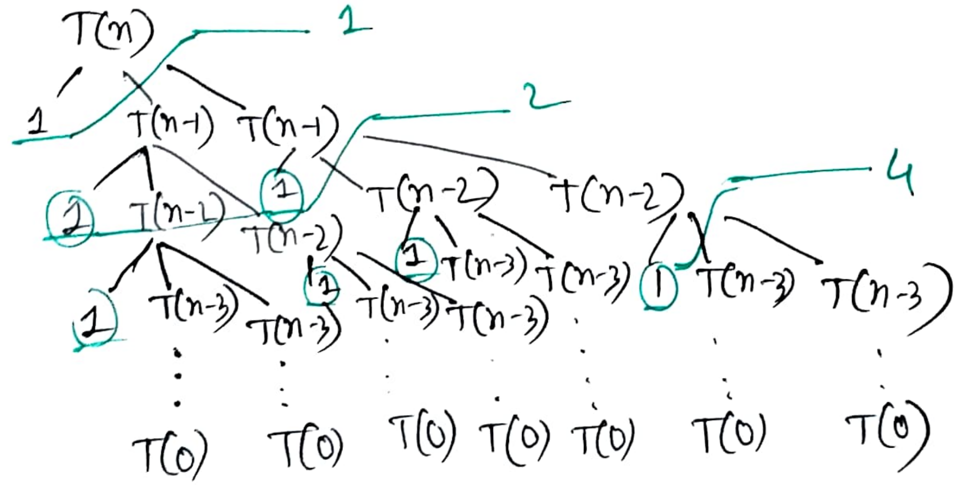
recursion branching/multiple rec.

code:

```
f(x):           — T(n)
if x==0:        ] 1
  return        ]
print(x)        — 1
f(x-1)          — T(n-1)
f(x-1)          — T(n-1)
p
```

$$T(n) = \begin{cases} 1 & ; n=0 \\ 2(T(n-1))+1 & ; n>0 \end{cases}$$

Sol^N 1



the instructions will go like,

1
2
4
8
16
⋮

for k^{th} step ⇒ summing all

$1 + 2^1 + 2^2 + 2^3 + 2^4 + \dots 2^k$

$\Rightarrow 2^{k+1} - 1$

$$\left[a + ax + ax^2 + ax^3 + \dots ax^n = \frac{a(x^{n+1}-1)}{x-1} \right]$$

since, $n-k=0$; $k=n$

$$T(n) = 2^{n+1} - 1$$

$$\Rightarrow T(n) = O(2^n)$$

Sol^N 2 sub.

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

$= 2[2T(n-2)+1]+1 = 2^2T(n-2) + 2+1$

$(k^{th}) \Rightarrow 2^k \cdot T(n-k) + 2^{k-1} + 2^{k-2} + \dots + 2^3 + 2^2 + 2^1 + 1$

$k = n \Rightarrow 2^n \cdot T(n-n) + 2^{n-1} + 2^{n-2} + \dots + 2^2 + 2^1 + 1$

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

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

$= 2^{n+1} - 1$

$\Rightarrow T(n) = O(2^n)$

Observations:

$T(n) = \underset{n}{T(n-1)} \times \underset{1}{+1} \Rightarrow O(n)$

$T(n) = \underset{n}{T(n-1)} \times \underset{n}{+n} \Rightarrow O(n^2)$

$T(n) = \underset{n}{T(n-1)} \times + \log n \Rightarrow O(n \log n)$

$T(n) = \underset{n}{T(n-1)} \times + n^2 \Rightarrow O(n^3)$

$T(n) = \underset{n}{T(n-2)} \times + n \Rightarrow O(n)$

if the recursive process is only branching 1 function per call + it's size is $\boxed{\text{decreasing}}$ in a linear fashion, we can interchange it with n and multiply with the instruction in each call.

$T(n) = 2T(n-1) + 1 \Rightarrow O(2^n)$

$T(n) = 3T(n-1) + 1 \Rightarrow O(3^n)$

$T(n) = 3T(n-1) + n \Rightarrow O(n \cdot 3^n)$

same scenario but multiple branching, then branching^(decrease linear) x instr

All of the tasks shown before, are the recursion of decreasing function: $T(n) = T(n-1)$ in this pattern; now see some dividing function:

```
f(x) :                  ------ T(n)
   if x == 0:
       return "done"    ] 1
   print (x)            ------ 1
   f(x/2)               ------ T(n/2)
```

$$T(n) = \begin{cases} 1, & n = 0 \\ T(n/2) + 1, & n > 0 \end{cases}$$

$T(n)$ —— 1
$1 \quad T(n/2)$ —— 1
$1 \quad T(n/4)$ —— 1
$1 \quad T(n/2^3)$ —— 1
$\vdots$
$T(0) \quad [T(n/2^k)]$

at k^{th} step; $T(n/2^k) \Rightarrow n/2^k = 1$ [base case]

$\Rightarrow n/2^k = 1$

$\Rightarrow n = 2^k$

$\Rightarrow k = \log n$

so, k amount of instruction (1) is running, $k = \log n$

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

Sol^N 2

$$T(n) = T(n/2)+1$$
$$= [T(n/4)+1]+1$$
$$= T(n/2^2)+2$$
$$= [T(n/2^3)+1]+2$$
$$= T(n/2^3)+3$$

kth $= T(n/2^k)+k$

$T(n/2^k)$ in kth time, it should be 1 [base case]

$n/2^k = 1 \Rightarrow k = \log n.$

so, $T(n) = T(n/2^{\log_2 n}) + \log n$

$$= \underbrace{1}_{\text{base case}} + \log n$$

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

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + n ; & n > 1 \end{cases}$$

Sol^N 1

$T(n)$

n \ $T(n/2)$ —— n

$\frac{n}{2}$ \ $T(n/2^2)$ —— $n/2$

$\frac{n}{2^2}$ \ $T(n/2^3)$ —— $n/2^2$

$\vdots$

$n/2^{k-1}$ \ $T(n/2^k)$ —— $n/2^{k-1}$

summing $\Rightarrow$ $n + n/2 + n/2^2 + \dots n/2^{(k-1)} + \underline{n/2^k}$

↳ approximating since big oh notation

$$\Rightarrow T(n) = n\left[\underline{1 + 1/2 + 1/2^2 + \dots 1/2^k}\right]$$

$$= n\,(1 + .5 + .25 + \dots)$$

$$= n \times O(1)$$

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

↳ this is a forever going function's finite version which will approximately be around 1.9999 ; it will never reach more than 2, making it linear regardless [big Oh]

Soln 2

$$T(n) = T(n/2) + n \quad \text{— st 1}$$

$$\Rightarrow [T(n/4) + n/2] + n \quad \text{— st 2}$$

$$= [T(n/2^2) + n/2] + n$$

$$\Rightarrow T(n/2^3) + n/2^2 + n/2 + n \quad \text{— st 3}$$

$$k^{th} \Rightarrow T(n/2^k) + n/2^{k-1} + n/2^{k-2} + \dots n/2^1 + n/2^0$$

$$\Rightarrow T(n) = T(n/2^k) + n\left[1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \dots \frac{1}{2^k}\right]$$

$$\Rightarrow T(n) = \underbrace{T(1)}_{\text{base case}} + n \times (1+1)$$

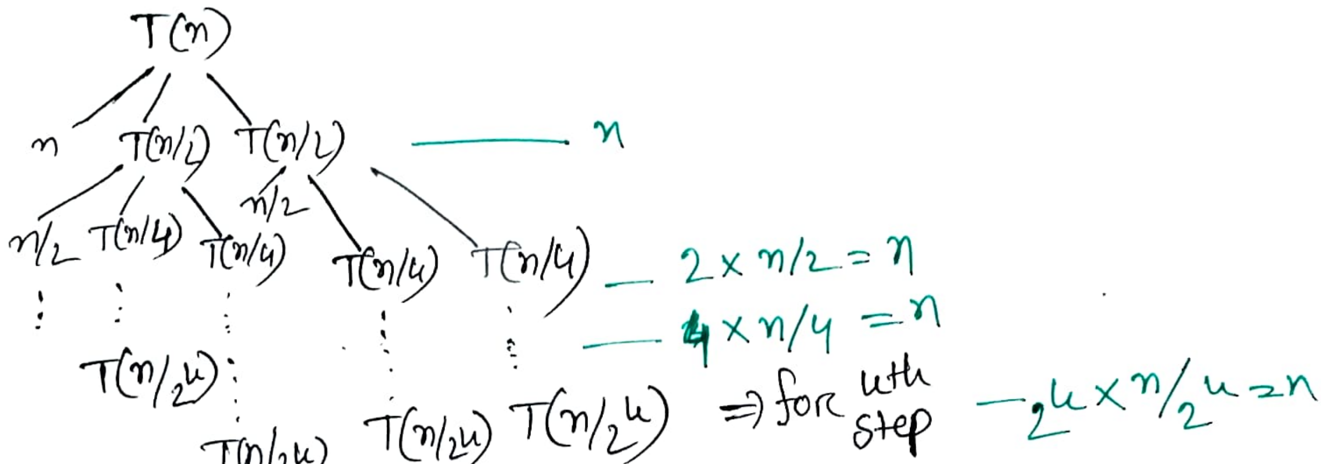
$$\Rightarrow T(n) = O(n)$$

---

another case (will be useful in sorting)

$$T(n) = \begin{cases} 1 & ; n = 1 \\ 2T(n/2) + n & ; n > 1 \end{cases}$$

(P.T.O.)



base case, $n/2^k = 1$ [Sol^N 1]

$\Rightarrow k = \log n$.

Summing all steps, total k steps, each step n operation;

$T(n) = k \times n$

$\Rightarrow T(n) = n \times \log n$

$= O(n \log n)$

Sol^N 2

$T(n) = 2T(n/2) + n$ —— 1

$= 2^2 T(n/2^2) + 2 * n/2 + n$ —— (11)

$= 2^2 T(n/2^2) + n + n$

$= 2^3 T(n/2^3) + 3n$

kth $\Rightarrow 2^k T(n/2^k) + k \cdot n$; $n/2^k = 1$; $n = 2^k$

$k = \log n$

$T(n) = 2^k T(1) + n.\log n$

$= n \times 1 + n.\log n \Rightarrow \boxed{T(n) = O(n.\log n)}$

Another way to solve generalized recursive Algorithm: $\boxed{\text{Masters Theorem.}}$

⇒ it's also known as master method. we discussed two types of recursive cases, "decreasing recursion & dividing recursion". The generalised formula is different for both of them.

for $\boxed{\text{decreasing recursive}}$ function:

format: $T(n) = a\,T(n-b) + f(n)$; $a > 0$ & $b > 0$ & $f(n) = O(n^d)$

must satisfy these three

(i) if, $a = 1 \Rightarrow T(n) = O(n \times f(n))$

(ii) if, $a > 1 \Rightarrow T(n) = O(a^{n/b} \cdot n^d) = O(\boxed{n^d} \cdot a^{n/b})$

f(n)

upper bound of f(n); you can write f(n) instead if you want

(iii) if $a < 1 \Rightarrow T(n) = O(n^d) = O(f(n))$

for dividing recursive function:

since it divides it's input; the instructions/ f(n) cannot be represented with the previous formula

⇒ format for dividing rec. master theorem

$$T(n) = a.T(n/b) + f(n) \; ; \quad a \geq 1, \; b > 1$$

we will need: $f(n) = O(n^d)$

This is a more generalized formula. the conditions for this theorem are:

$$\boxed{T(n) = \begin{cases} O(n^d . \log n) & ; \text{ if } a = b^d \\ O(n^d) & ; \text{ if } a < b^d \\ O(n^{\log_b a}) & ; \text{ if } a > b^d . \end{cases}}$$

divide recursion

to repeat; for decreasing recursion:

$$T(n) = a.\ T(n-b) + f(n) \; ; \quad a > 0; \; b > 0; \quad f(n) = O(n^d)$$

$$\boxed{T(n) = \begin{cases} O(n^d) & ; \text{ if } a < 1 \\ O(n \times n^d) & ; \text{ if } a = 1 \\ O(n^d \times a^{n/b}) & ; \text{ if } a > 1 \end{cases}}$$