Time complexity. & Space complexity

* order complexity Analysis

Amount of space or Time taken by an algorithm / code as function of input size.
Not the actual time taken.
i.e it determine the ~~fn~~ relation between fun input

- linear search (largest find)

n ↑ operations ↑

worst case

* Case 1

n ↑ T ↑

T ∝ n

∴ Time is fun of n

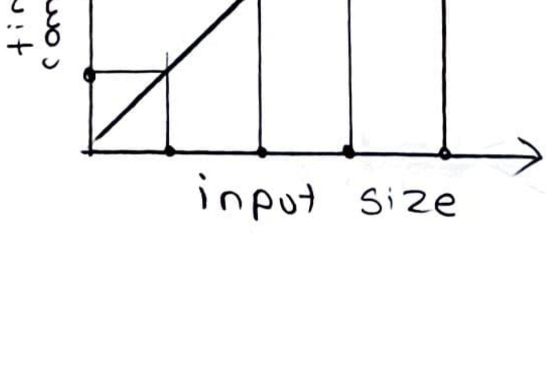
* Case 2

n ↑ T constant

∴ Time is not fun of n

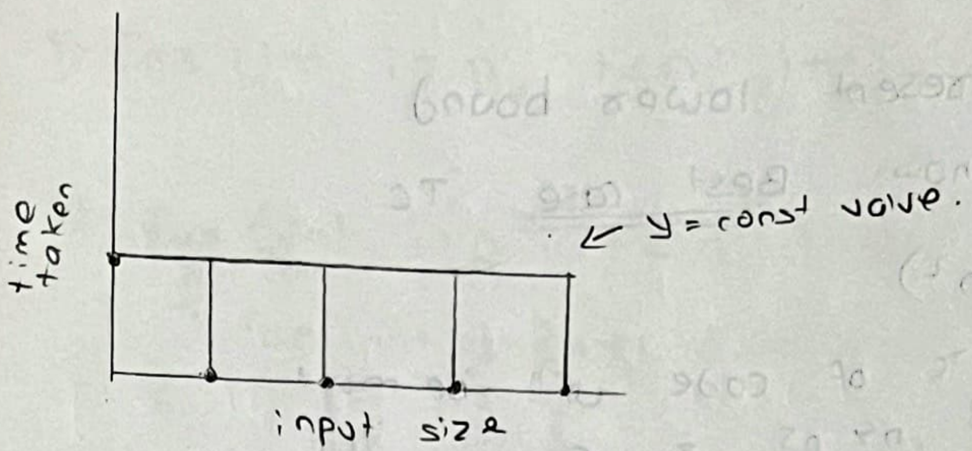
∴ T ∝̸ n

* linear search



→ Time complexity fun
i.e TC = O(n)
$y = a\cancel{n} + b$
i.e it ignore const
$y = n$

* Constant time complexity



$tc = O(1)$
$= O(k)$.

* Big O Notation

→ It gives upper bound ∴ it give worst case tc

→ ie it gives the at max tc
ie program will run in less time but not max than that tc which it specifies

* How to find it

① ignore const
② largest term

ex time $= an^2 + bn + c$
$= \underline{n^2} + n + 1$
i.e $O(n)$

∘ time $= an^3 + b \log n + c$
$= \underline{n^3} + \log n + c$
$= O(n^3)$ — ie consider largest term

∴ time ⇒ $f(n)$

$f(n) = O(g(n))$

$$\lim_{n \to \infty} \frac{|f(n)|}{g(n)} < \infty$$

* Big omega Notation

→ to it represent lower bound

→ ie it g shows Best case TC.

if TC = $\Omega(n^2)$

then TC of code will be more ie n^3, n^4, n^5 & so on

* Big Theta (Θ) Notation

→ it grepresent average TC

when code run in $O(n^2)$ → worst TC

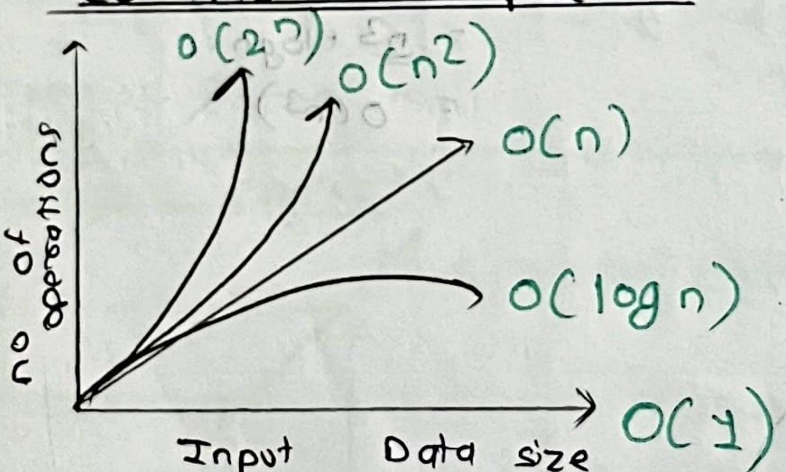
& $\Omega(n^2)$ → Best TC

∴ $\Theta(n^2)$ → average TC

ie LB = UB

∴ $\Theta(n^2)$

* common complexities



* Space complexity

memory space → heap → objects

memory space → stack → functions

space complexity = input space + auxiliary space

* Time complexity Analysis.

1) For (int i = 0; i<n; i++) :- $O(n)$

2) for (int i = 0 ; i < n; i++)
{ for (int j = i + 1; j <n; j++) { }}

i = 0   n-1 +
i = 1   n-2 +
i = 2   n-3 +
i = n-1   0

$\therefore O(n^2)$

3) for (int i = 0; i<n; i++)
{ for (int j = 0; j < i; j++) { }

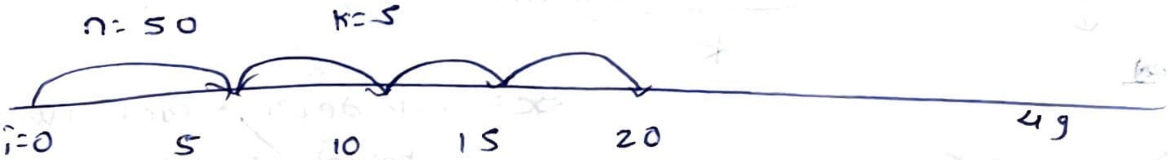
$O(n^2)$

i = 0   ~~j = 0~~ 1   0 times
i = 1   j = 0 to 0   -- k
i = 2   j = 0 to 1   2k   ie AP
i = 3   j = 0 to 2   3k
4k
5k
i = n-1   j = 0 to n-1   (n-1)k

4) for (int i = 0; i <n; i = i + k)
{ for (int j = i+1; j <= k; j++) { }}

$O(n)$

n = 50   k = 5



outer loop = $\frac{n}{k}$ inner loop = k

$O(\frac{n}{k} \times k) = O(n)$

* Binary search

```
While (start <= end)
{
    ⋮      const
}
```

→ ① ⇒ n = n

② ⇒ $\frac{n}{2^1} = \frac{n}{2}$

③ ⇒ $\frac{n}{2^2} = \frac{n}{4}$

④ = $\frac{n}{2^3} = \frac{n}{8}$

⋮

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

∴ $\frac{n}{2^k} = 1$

⋮ log n

* Recursive Algorithm

* Time complexity

- total work done = (no of calls * work in each call)

- Recurrence eqn

* Space complexity

Space complexity = (max depth * memory in each call)

1) Factorial

p.s. int fact(int n)

return n * fact(n-1);

WD = no of calls * work in each call
↓ n, k

WD = n * k

TC = O(n)

n = 4

f(0)	1
f(1)	1×1 = 1
f(2)	1×2 = 2
f(3)	3×2 = 6
f(4)	6×4 = 24

SC = max depth * each memory

height of call stack

(n) * K

SC = O(n)

2) Sum of n NO $\quad f(n) = n + f(n-1)$

```
int Sum(int n)
{ return n + Sum(n-1);
```

TC = ~~?~~ no of call * work in each call

$= n * K$

$\underline{O(n)}$

SC = height * depth × memory in each IV

$= n * K$

$SC = \underline{O(n)}$

3) Fibonacci

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

∴ Recurrence eqⁿ ⇒ $\boxed{f(n) = f(n-1) + f(n-2)}$

$T(n) = T(n-1) + T(n-2) + K$ -- by master theorem

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

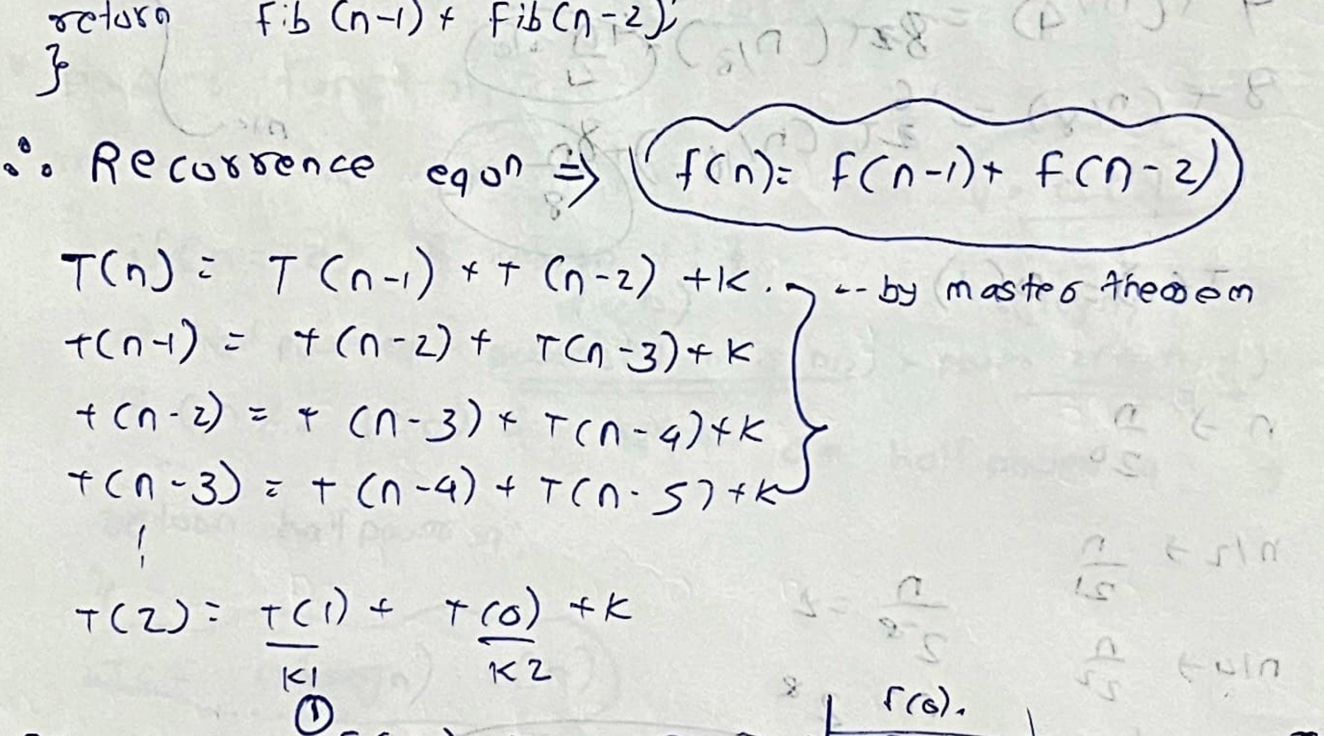
$T(n-2) = T(n-3) + T(n-4) + K$

$T(n-3) = T(n-4) + T(n-5) + K$

$\vdots$

$T(2) = \underset{K1}{T(1)} + \underset{K2}{T(0)} + K$

ex $\underline{n=4}$



$SC = n * K$

$\boxed{SC = O(n)}$

$\boxed{TC = O(2^n)}$

4) Merge Sort

```
mergesort (int arr[], int si, int ei)
{ if (si>=ei) { return; }
  int mid = si + (ei - si)/2 ;
  mergesort (arr, si, mid);
  mergesort (arr, mid+1, ei);
  merge (arr, si, mid, si);  -> O(n)
}
```

$T(n) = T(n/2) + T(n/2) + nk \rightarrow nk$

$T(n) = 2T(n/2) + nk$ $\rightarrow nk$

$2T(n/2) = 4T(n/4) + 2 \cdot \frac{n}{2} k$ $\rightarrow nk$

$4T(n/4) = 8T(n/8) + 4\frac{n}{4} k$

$8T(n/8) = 16T(n/16) + 8\frac{n}{8} k$ $\quad nk$

$\left.\right\} x n.$

$T(1) = O(1)$

$\therefore n \rightarrow \frac{n}{2^0}$

$n/2 \rightarrow \frac{n}{2^1}$

$n/4 \rightarrow \frac{n}{2^2}$ $\quad \frac{n}{2^x} = 1$

$n = 2^x$

$\cancel{\log_2 2^x} = \boxed{\log_2 n = x}$

$\therefore TC = O(\log n)$

jb bhi n -- half hogo s, 1 hoga

so $\log n$

$\therefore TC = O(1) + \log n\,(nk)$

$\therefore TC = O(n \log n)$

$SC = O(n)$ in merge

S. power function

```
int power (int a, int n)
{ if (n==0)
    { return 1; }
  return a * power(a, n-1); }
```

$f(a,n)$ a^n
↓
$a * f(a, n-1)$ a^{n-1}
↓
$a * f(a, n-2)$ a^{n-2}
↓
$a * f(a, n-3)$ a^{n-3}
;
(1) $f(a,0)$ a^0

WD = no of call * time in each cell

= n * k

$\boxed{TC = O(n)}$

SC: no of call * memory space in each call

= n * O(1)

$\boxed{SC = O(n)}$

* power function-2

```
int power2( int a, int n)
{
  if(n==0) { return 1; }
  int half power sq = power2(-2 int a, n/2) * power 2(a, n/2);
  if (n%2 != 0) { return a* half power sq; }
  return half power sq;
}
```

O(log n)   O(log n)

i.e O(n)

$\boxed{TC = \cancel{O(\log n)}\ O(n)}$

* PW Fn3

```
int power3 (int a, int n)
{ int half power = power3(a, n/2);
  int halfpowersq = ~~power~~ half power * ~~po~~ half power;
  if (n%2!=0) { return a * half powersq; }
  return half powersq;
}
```

O(log n)

* Master Theorem for dividing function

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

$$f(n) = O(n^k \log^p n)$$

Case 1 :- if $\log_b a > k$ then $O(n^{\log_b a})$

case 2 :- if $\log_b a = k$

if $p > -1$ -- $O(n^k \log^{p+1} n)$

if $p = -1$ -- $O(n^k \log \log n)$

if $p < -1$ $O(n^k)$.

Case 3 :- if $\log_b a < k$ -- if $p \geq 0$ - $O(n^k \log^p n)$

-- if $p < 0$ -- $O(n^k)$

* Focus on

① $\log_b a$

② k