* Metric to measure time :-

→ Complexity analysis just depends on the fact that how much time an algo takes based on the input.

→ No. of instructions executed is a metric to measure time.

→ We assume every single instruction taking C units of time.

→ So if we have total k instructions, we will spend kC unit of time.

→ Therefore, in order to calculate the time, we need to calculate how many instructions were executed with respect to the input.

* Problem 1 :-

```
test.js > ...                          input
1   function foo(n) {
2       let ans = 1;                   → C
3
4       for (let i = 0; i < n; i++) {  → C
5           console.log(i);
6           ans += i;
7       }
8
9       return ans;                    → C
10  }
```

In a loop, variable initialization happens only once. The loop will start from 0 & go upto n-1 which means total of n iterations.

For every i^{th} instruction following things are done :-

1) Condition check (i<n)
2) Line 5 logger
3) Line 6 calculation
4) Incrementing i (i++)

∴ For every i^{th} iteration, we are executing 4 constant instructions.

i = 0 → i = 1 → i = 2 → .... → i = n-1
↓ ↓ ↓ ↓
4 4 4 4

∴ Total iterations = n

In 1 iteration, we execute 4 instructions, so in n iterations, we execute 4×n = 4n instructions.

If 1 instruction takes C unit of time, then 4n instructions takes 4nC units of time.

∴ C + C + C + 4nC = (4nC + 3C)

C: line 2; C: for loop init; C: return; 4nC: for loop

3C → Constant so avoid it.

∴ Time complexity = O(n)

* Problem 2 :-

```
1  function foo(n, m) {
2      let ans = 1; → c
3                  ↷c
4      for (let i = 0; i < n; i++) {
5          ans += 1;
6      }
7                  ↷c
8      for (let j = 0; j < m; j++) {
9          ans += 2;
10     }
11
12     return ans; → c
```

The loop on line 8 will not execute until the previous loop has completed its execution.

For every iteration of the loop on line 4, we perform 3 instructions.

The loop goes from 0 to n, so we have total of n iterations.

∴ Total instructions of loop 1 = 3 * n * c = n

Similarly loop 2 has total of m iterations. For every iterations we perform 3 instructions.

∴ Total instructions of loop 2 = 3 * m * c = m.

∴ Overall total instructions = c + n + m + c

= n + m.

∴ Time complexity = O(n+m)

* Note :-

→ The instructions that does not depend on the input are constants.

Sometimes if $n \leq 10^6$ & $m \leq 10^6$ then O(n) because both are almost same.

Sometimes if $n \leq 10^6$ & $m \leq 10^2$ then $m << n$, so m becomes a lower degree term.

∴ O(n).

* Problem 3 :-

```
test.js > ...
1  function foo(n) {
2      let ans = 9;          → c
3                            ↱ c
4      for (let i = 1; i <= Math.log(n); i++) {
5          console.log(i);
6      }
7
8      return ans;           → c
9  }
```

Total iterations :- $i = 1, i = 2, i = 3, \ldots, i = \log n$

Instructions per iteration :- 3 3 3 3

$\therefore$ Total instructions $= 3 \log n + 3c = \log n$

$\therefore$ Time complexity $= O(\log n)$

* Problem 4 :-

```
test.js > ...
1  function foo(n) {
2      let ans = 0;          → c
3                            ↱ c
4      for (let i = 0; i < n; i++) {      → Outer loop
5          for (let j = 0; j < n; j++) {  → Inner loop
6              ans += 1;
7          }
8      }
9
10     return ans;           → c
11 }
```

1 iteration of outer loop performs $3 n c$ instructions

$j < n$, $ans += 1$, $j++$ — goes upto n — $j = 0$

$\therefore$ Total iterations = $i = 0$, $i = 1$, $i = 2$, ... $i = n-1$

Instructions / iteration = $3nc$, $3nc$, $3nc$, ... $3nc$

$\therefore$ Total instructions $= (n \times 3nc) + 3c = 3n^2c + 3c = 3c(n^2 + 1) = n^2$

$\therefore$ Time complexity $= O(n^2)$

* Problem 5 :-

```
test.js > ...
1   function foo(n) {
2       let ans = 0; → C
3
4       for (let i = 0; i < n; i++) {
5           for (let j = 0; j < i; j++) {
6               ans += 1;
7           }
8       }
9
10      return ans; → C
11  }
```

Total iterations :- $i=0$ $i=1$ $i=2$ $i=3$ $i=n-1$

↓ ↓ ↓ ↓ ↓

Instructions per iteration :- 0 3 6 9 $3\times(n-1)$

Total iterations $= n$

Total instructions $= n \times 3(n-1)$

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

$= 3n^2 - 3n$

$\therefore$ Time complexity $= O(n^2)$

* Problem 6 :-

```
test.js > ...
1   function foo(n) {
2       let ans = 0;
3
4       for (let i = 0; i < n; i++) {
5           for (let j = 0; j < Math.log(n); j++) {
6               ans += 1;
7           }
8       }
9
10      return ans;
11  }
```

$i=0$ $i=1$ ... $i=n-1$

↓ ↓ ↓

$\log n$ $\log n$ $\log n$

instruc

Total iterations :- n

Instruction / iteration :- $\log n$

Total instructions $= n \log n$

Time complexity $= O(n \log n)$

* Problem 7 :-

```
test.js > ...
1   function foo(n) {
2       let ans = 0;
3
4       for (let i = 0; i < n; i++) {
5           ans += i;
6       }
7
8       for (let i = 0; i < n; i++) {
9           for (let j = 0; j < n; j++) {
10              ans += i + j;
11          }
12      }
13
14      return ans;
15  }
```

Total instructions = $n + n^2$

Time complexity = $O(n^2)$

* Problem 8 :-

```
test.js > ...
1   function foo(n) {
2       let ans = 0;
3
4       for (let i = 1; i < n; i++) {
5           for (let j = n; j > 1; j--) {
6               ans += i;
7           }
8       }
9
10      return ans;
11  }
```

$i = 1$ $\quad i = 2$ $\quad i = 3$ $\quad \ldots$ $\quad i = n-1 \approx n$

↓ ↓ ↓ ↓

$3n-2$ $\quad 3n-2$ $\quad 3n-2$ $\quad 3n-2$

Total instructions $\approx n^2$

Time complexity = $O(n^2)$

* Problem 9 :-

```
test.js > ...
1   function foo(n) {
2       let ans = 0;
3
4       for (let i = 1; i < n; i *= 2) {
5           ans += i;
6       }
7
8       return ans;
9   }
```

$i = 1$ $\quad i = 2$ $\quad i = 4$ $\quad i = 8$ $\quad i = 16$ $\quad \ldots$

↓ ↓ ↓ ↓ ↓

3 $\quad$ 3 $\quad$ 3 $\quad$ 3 $\quad$ 3

Here, the loop updation is $i *= 2$ which will reach faster to n compared to $i++$.

So, we exactly don't know how many iterations occurred.

Assume total iterations = k.

$\therefore$ Total instructions = $3k$

1st iteration → $i = 1 \longrightarrow 2^0$

2nd iteration → $i = 2 \longrightarrow 2^1$

3^{rd} iteration → $i = 4 \rightarrow 2^2$

4^{th} iteration → $i = 8 \rightarrow 2^3$

k^{th} iteration → $i = 2^{k-1}$

Here, 2^{k-1} will be the last value $< n$ because the next iteration 2^k will be $> n$.

$\therefore 2^{k-1} < n$

Taking $\log_2$ on both sides

$\log_2 2^{k-1} < \log_2 n$

$(k-1)\log_2 2 < \log_2 n \quad (\log_b a^c = c \log_b a)$

$(k-1) < \log_2 n \quad (\log_2 2 = 1)$

$k < \log_2 n + 1$

$\therefore$ The maximum value of k can never exceed $\log_2 n$ because $k < \log_2 n + 1$

$\therefore$ Time complexity = $O(\log n)$

→ We are not using $\log_2 n$ because it can be easily converted to $\log_3 n$ or $\log_4 n$ & so on. So we are making it generic.

* Problem 10 :-

```
test.js > ...
1   function foo(n) {
2       let ans = 0;
3
4       while (n > 0) {
5           ans += n;
6           n /= 2;
7       }
8
9       return ans;
10  }
```

In every iteration, 3 instructions are executed. Since n is updated by $n/2$, we don't know, how many total iterations exists.

Assume total iterations = K.

Total instructions = 3K.

1^{st} iteration → $n = n \rightarrow \frac{n}{1} = \frac{n}{2^0}$

2^{nd} iteration → $n = \frac{n}{2} \rightarrow \frac{n}{2^1}$

3^{rd} iteration $\rightarrow n = \frac{n}{4} \rightarrow \frac{n}{2^2}$

4^{th} iteration $\rightarrow n = \frac{n}{8} \rightarrow \frac{n}{2^3}$

k^{th} iteration $\rightarrow n = \frac{n}{2^{k-1}}$

The last value of n should be 1 because after another $n/2$, n becomes ≈ 0.

$$\therefore \frac{n}{2^{k-1}} \approx 1$$

$$n \approx 2^{k-1}$$

Taking $\log_2$ on both sides

$$\log_2 n \approx \log_2 2^{k-1}$$

$$\log_2 n \approx (k-1)\log_2 2$$

$$\log_2 n \approx k-1$$

$$k \approx \log_2 n + 1$$

$\therefore$ Time complexity $= O(\log n)$

* Problem 11 :-

```
test.js > ...
1   function foo(n) {
2     let ans = 0;
3
4     for (let i = n; i > 0; i /= 2) {
5       for (let j = 0; j < i; j++) {
6         console.log(i, j);
7       }
8     }
9
10    return ans;
11  }
```

$i = n \rightarrow$ j will go n iterations.

$i = \frac{n}{2} \rightarrow$ j will go $\frac{n}{2}$ iterations

$i = \frac{n}{4} \rightarrow$ j will go $\frac{n}{4}$ iterations.

:

$i = \frac{n}{2^k} \rightarrow$ j will go $\frac{n}{2^k}$ iterations.

$\therefore$ Total iterations $= n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \cdots + \frac{n}{2^k}$

$$= n\left(\frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^k}\right)$$

From previous couple of examples, we observed how k is calculated. Similarly, we can deduce $k = \log n$

$$= n\left(\frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{\log n}}\right)$$

This is a geometric progression.

Sum of GP $= \frac{a(1-r^n)}{1-r}$ where

a = first term, r = ratio & n = total terms

$a = 1 \qquad r = \frac{1}{2} \qquad n = \log_2 n + 1$

$$\text{Sum} = \frac{1 \times \left(1 - (1/2)^{\log_2 n + 1}\right)}{1 - 1/2}$$

$$= \frac{\left(1 - (1/2)^{\log_2 n + 1}\right)}{1/2}$$

$$= 2\left(1 - \frac{1}{2^{\log_2 n + 1}}\right)$$

$$= 2 - \frac{2}{2^{\log_2 n + 1}}$$

$$= 2 - \frac{2}{2^{\log n} \times 2}$$

$$= 2 - \frac{1}{2^{\log_2 n}}$$

$$\simeq 2 - \frac{1}{n} \qquad \left(2^{\log_2 n} = n \log_2 2 = n \right)$$

If n is very large, $\frac{1}{n}$ is negligible. (Because we only consider higher values of n)

$\simeq 2 \rightarrow$ constants.

$$\therefore \text{Total iterations} = n \left(\frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots + \frac{1}{2^{\log n}} \right)$$

$$= n \times c$$

$$= n$$

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

* Problem 12 :-

```
test.js > ...
1  function foo(n) {
2    for (let j = 1; j <= n; j++) {
3      for (let i = 0; i < n; i = i + j) {
4        console.log(i, j);
5      }
6    }
7  }
```

$j = 1$ $\quad i \rightarrow (0 \text{ to } n-1)$ with inc 1 $\rightarrow$ n iterations in total

$j = 2$ $\quad i \rightarrow (0 \text{ to } n-1)$ with inc 2 $\rightarrow$ $n/2$ iterations in total

$j = 3$ $\quad i \rightarrow (0 \text{ to } n-1)$ with inc 3 $\rightarrow$ $n/3$ iterations in total

$\vdots$

$j = n$ $\quad i \rightarrow (0 \text{ to } n-1)$ with inc n $\rightarrow$ 1 iteration in total

$$\therefore \text{Total iterations} = n + \frac{n}{2} + \frac{n}{3} + \frac{n}{4} + \cdots + 1$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}\right)$$

$\log n$ (Harmonic progression)

Total iterations $= n \log n$

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

Harmonic progression $= \frac{1}{a}, \frac{1}{a+d}, \frac{1}{a+2d}, \ldots, \frac{1}{a+(n-1)d}$

$$S_n = \frac{1}{d} \log_e \left(\frac{2a + (2n-1)d}{2a-d}\right)$$

$$= \frac{1}{1} \log_e \left(\frac{2 + 2n - 1}{2 - 1}\right)$$

$$= \log_e \left(\frac{2n+1}{1}\right)$$

$\tilde{\sim} \log n$ (where n is very large)

* Problem 13 :-

```
test.js > ...
1  function foo(n) {
2    let ans = 0;
3    for (let i = 2; i <= n; i *= i) {
4      ans++;
5    }
6  }
```

$i = 2 \to 4 \to 16 \to \ldots$

$= 2^1 \to 2^2 \to 2^4 \to 2^8 \to \ldots \to 2^k$

1 2 3 4 T

2^k some final value of $i <= n$

$\therefore 2^k \leq n$

Taking log on both sides

$k \leq \log n$

But here, k doesn't denotes no. of iterations.

itr	Power
1st	$1 \to 2^0$
2nd	$2 \to 2^1$
3rd	2^2
4th	2^3
⋮	
T^{th}	2^{T-1}

$k = 2^{T-1}$

Taking log both sides

$\log k = \log 2^{T-1}$

$\log k = T-1 \log_2 2$

$\log k = T-1$

$T = \log k + 1$

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

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