# ALGORITHM ANALYSIS

## Algorithm :-

* **Definition** :- An algorithm is a finite set of well-defined instructions.

To know how effective those algorithms are at completing tasks, must access algorithm's time and space complexity.

## Time & Space Complexities :-

Space complexity quantifies how much space or memory it takes to run as a function of the length of the input.

Time complexity measures how long it takes an algorithm to run as a function of the length of the input.

## Problems :-

instance of a problem :- One example of input of a problem is called an instance of the problem.

Eg :- The sorting problem :-

input : 12, 24, 13, 12, 8

output : 8, 12, 12, 13, 24.

This example is instance of the problem.

* We say an algorithm is correct if the algorithmic procedure stops for each problem instance and returns the required output. The algorithm solves the problem.

## Pseudocode :-

* **Definition** : compact and informal high-level description of an algorithm intended for humans rather than machines.

```
Algorithm sum(n)
    sum ← 0
    for i ← 1 to n
        sum += i
    return sum.
```

## Efficiency:-

Different algorithms that solve the same problem can have large performance differences.

Typical Situation - sorting :-

Insertion Sort → $O(n^2)$ i.e. $c_1 n^2$ Quick Sort → $O(n \log_2 n)$ i.e. $c_2 n \log_2 n$

$c_1$ & $c_2$ are implementation dependent constants.

Quick Sort is much faster than Merge Sort when $n$ is large. This holds true even if we use an incompetent implementation of Quick Sort running on a slow computer.

* Example :-

① Insertion Sort on faster computer (computer executes $10^9$ instructions/sec)
$c_1 = 5$. [algorithm implemented by skillful programmer]

② Quick Sort on slower computer (computer executes $10^7$ instructions/sec)
$c_2 = 100$. [algorithm implementation is sloppy].

small instances - $n = 1000$.

① $10^3 \log_2 10^3 = 5 \times 3 \times 10^3 (\log_2 10) \approx 5 \text{ ms}$ ($\log_2 10 \approx 3.33$)

[no. of instructions: $5 \times 10^3 \log_2 10^3 = 3 \times 10^3 \times 5 \times \log_2 10 = 5 \times 10^3$ instructions
time $= \frac{5 \times 10^3}{10^9} = 5 \times 10^{-6}$ sec]

② $(10^3)^2 = 10^6 = 1000 \text{ ms} \approx 1 \text{ s}$.

large instances - $n = 1,000,000$

① $10^6 \log_2 10^6 = 6 \times 10^6 (\log_2 10) = 1.8 \times 10^6 = 1.8 \times 10^3 \text{ ms} = 1.8 \text{ s}$

② $(10^6)^2 = 10^{12} = 10^6 \times 10^6 = 10^6 \text{ s}$.

① small instances :- $n = 1000$.

① Insertion Sort complexity : $c_1 n^2$ Computer = $10^9$/s. $c_1 = 5$.

$\Rightarrow 5 \times (1000)^2 = 5,000,000$ instructions

Time taken $= \frac{\text{instructions}}{\text{instructions/sec}} = \frac{5,000,000}{10^9} = 5 \times 10^{-3} = 5 \text{ ms}$.

② Quick sort complexity = $c_2 n \log_2 n$, $c_2 = 100$, computer = $10^7$/s.

∴ no. of instructions = $100 \times 10^3 \log_2 10^3$

$= 3 \times 10^5 \times (9.3)$

$= 10 \times 10^5$ instructions $= 10^6$ instructions

∴ Time taken = $\frac{\text{instructions}}{\text{instructions/sec}} = \frac{10^6}{10^7} = 10^{-1}$ s

$= 10^2$ ms.

$= 100$ ms

② Large instances $n = 1{,}000{,}000$

fast computer

① Insertion sort = $c_1 n^2 = 5 \times (10^6)^2 = 5 \times 10^{12}$ instructions.

time = $\frac{5 \times 10^{12}}{10^9} = 5 \times 10^3 = 5000$ s (more than 1 hour)

slow computer

② Quick sort = $c_2 n \log_2 n = 100 \times 10^6 \log_2 10^6 = 6 \times 10^8 \times 3.33$

$= 20 \times 10^8$ instructions

time = $\frac{20 \times 10^8}{10^7} = 20 \times 10^1 = 200$s

For small instances (1,000), insertion sort ($c_1 n^2$) is faster than quick sort ($c_2 n \log n$)

($c_1 n^2$) (5ms) ($c_2 n \log n$) (100ms)

For large instances (1,000,000), quick sort ($c_2 n \log n$) is faster than insertion sort ($c_1 n^2$)

($c_2 n \log n$) (200s) ($c_1 n^2$) (5000ms)

even though quick sort is running on slow computer ($10^7$/sec) but insertion sort is running on faster computer ($10^9$/sec)

## How to analyze time complexity : Count your steps

Time complexity estimates the time to run an algorithm. It's calculated by counting elementary operations.

Eg:- Compute maximum element in the array a.

```
Algorithm max(a):
    max ← a[0]
    for i = 1 to len(a) - 1
        if a[i] > max
            max ← a[i]
    return max.
```

→ choose this as an elementary operation since
① This captures running time well as comparison dominates all other operations in this algorithm
② Time to perform comparison is constant: it doesn't depend on the size of a.

We want to reason about execution time in a way that depends only on the algorithm and its input.
This can be achieved by choosing an elementary operation, which the algorithm performs repeatedly, and define the Time Complexity $T(n)$ as the number of such operations the algorithm performs given an array of length $n$.

$$T(n) = n - 1 \quad [\because i \to 1 \text{ to } len(a) - 1].$$

In general, an elementary operation must have two properties:

1. There can't be any other operations that are performed more frequently as the size of the input grows.
2. The time to execute an elementary operation must be constant. It mustn't increase as the size of the input grows. This is known as unit cost.

## Asymptotic Notations :-

Asymptotic Notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

Example:- Bubble Sort

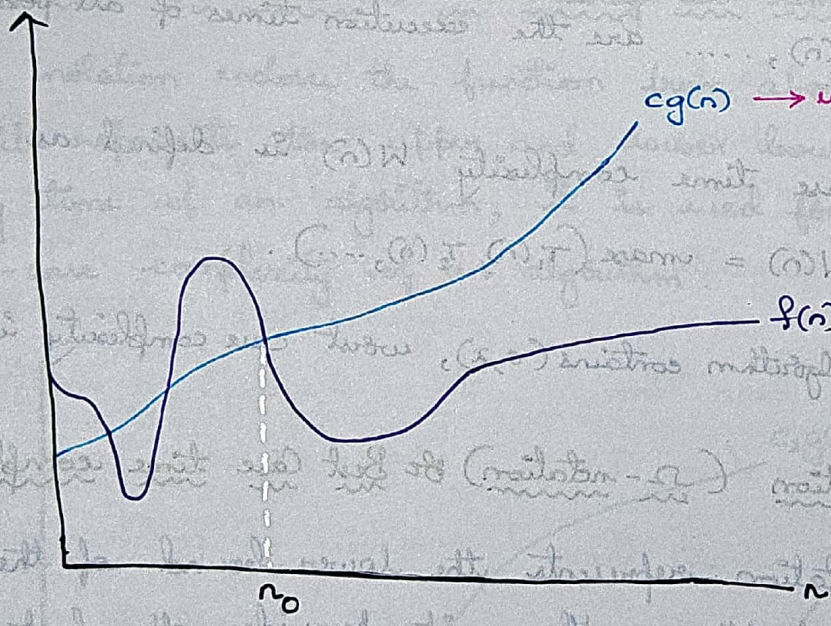
① input array already sorted — linear (Best case)
② input array sorted in reverse order — Quadratic (Worst case)
③ neither ① & ② — (Average case).

These are denoted using asymptotic notations.

* Big-O notation.
* Omega notation
* Theta notation.

① Big-O notation & Worst case time complexity:-

Big-O represents upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm




$cg(n)$ → upper bound for $n \geq n_0$ and a constant c so $cg(n)$ is the worst case and can be written as $O(g(n))$ for a time complexity $f(n)$.

Eg:- $f(n) = x^2 + 2x + 3$
$g(n) = cx^2$
$O(n^2)$ for n i/p.

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

$$O(g(n)) = \{ f(n) : \exists \text{ positive constants } c \text{ and } n_0 \Rightarrow 0 \leq f(n) \leq cg(n) \ \forall \ n \geq n_0 \}$$

$f(n)$ belongs to the set $O(g(n))$ if there exists a positive constant c such that it lies between 0 and $cg(n)$, for sufficiently large n.

For any n, running time doesn't cross $O(g(n))$

Example:- Tell whether array a contains x.

Algorithm contains (a, x):

```
for i = 0 to len(a) - 1
    if (x == a[i])  ---> elementary operation.
        return true
return false.
```

For this algorithm, the number of comparisons (operations) not only on the size of input n but also on x and values in a.

Case ①: if x is not present → n comparisons.

Case ②: if x is present at a[0] → 1 comparison.

Hence, we often choose to study worst-case time complexity.

* $T_1(n), T_2(n), \ldots$ are the execution times of all possible input sizes n.
* Worst-case time complexity $W(n)$ is defined as

$$W(n) = \max(T_1(n), T_2(n), \ldots).$$

For given algorithm contains(a,x), worst case complexity is $O(n)$.

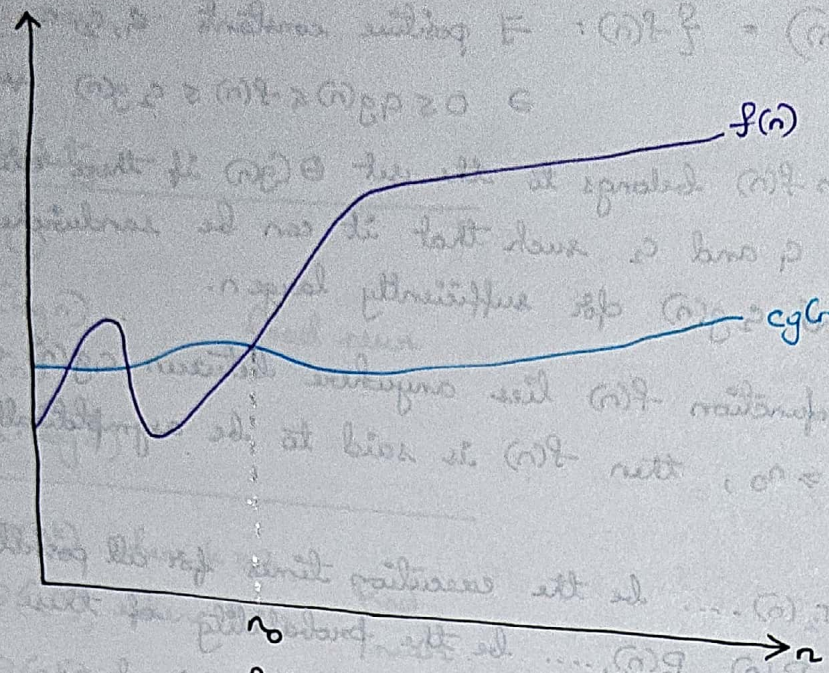
② Omega Notation ($\Omega$-notation) or Best Case time complexity :-

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

$$\Omega(g(n)) = \{f(n) : \exists \text{ positive constants } c \text{ and } n_0 \Rightarrow 0 \le cg(n) \le f(n) \ \forall \ n \ge n_0\}$$

Function $f(n)$ belongs to the set $\Omega(g(n))$ if there exists a positive constant c such that it lies above $cg(n)$ for sufficiently large n.

For any n, minimum time is $\Omega(g(n))$

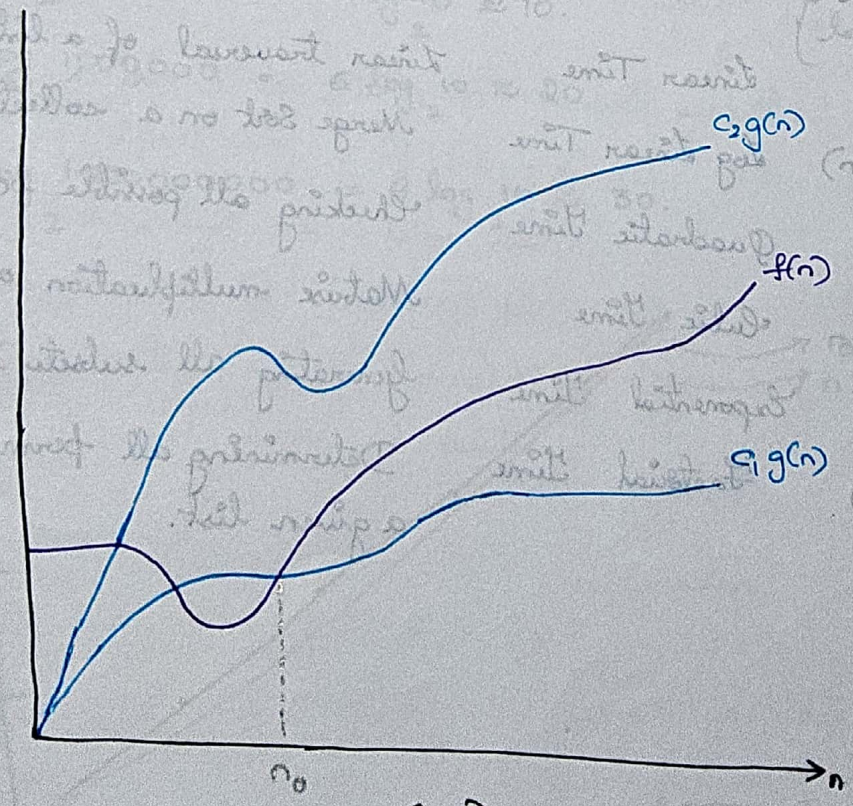



$f(n) = \Omega(g(n))$

$T_1(n), T_2(n), \ldots$ $\Omega \Rightarrow \min(T_1(n), T_2(n), \ldots)$

③ Theta Notation ($\Theta$-notation) or Average case time complexity :-

Theta notation encloses the function from above and below. Since it represents the upper and lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.




Theta bounds the function within constant factor.

$f(n) = \Theta(g(n))$

$\Theta(g(n)) = \{ f(n) : \exists$ positive constants $c_1, c_2, n_0$

$\Rightarrow 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall \; n \geq n_0 \}$

Function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants $c_1$ and $c_2$ such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$ for sufficiently large $n$.

If a function $f(n)$ lies anywhere between $c_1 g(n)$ and $c_2 g(n)$ for all $n \geq n_0$, then $f(n)$ is said to be asymptotically tight bound.

$T_1(n), T_2(n) \ldots$ be the execution times for all possible input sizes $n$ and let $P_1(n), P_2(n), \ldots$ be the probability of these inputs, then $\Theta$ (average case time complexity) defined as

$$P_1(n) T_1(n) + P_2(n) T_2(n) + \ldots$$

Big-O Cheat Sheet :-

| Big O | Name | Example |
|---|---|---|
| $O(1)$ | Constant Time | Checking if stack is empty. |
| $O(\log n)$ | Logarithmic Time | Finding an item in an Balanced Search Tree. |
| $O(n)$ | Linear Time | Linear traversal of a list. |
| $O(n \log n)$ | Log Linear Time | Merge Sort on a collection of items |
| $O(n^2)$ | Quadratic Time | Checking all possible pairs in an array. |
| $O(n^3)$ | Cubic Time | Matrix multiplication of $n \times n$ matrices. |
| $O(2^n)$ | Exponential Time | Generating all subsets of a given set. |
| $O(n!)$ | Factorial Time | Determining all permutations of a given list. |

Key takeaways :-

Complexity

$\Theta(1)$
$\Theta(\log n)$
$\Theta(n)$
$\Theta(n \log n)$

good news.

$\Theta(n^k)$, where $k \geq 2$
$\Theta(k^n)$, where $k \geq 2$
$\Theta(n!)$

Bad news.

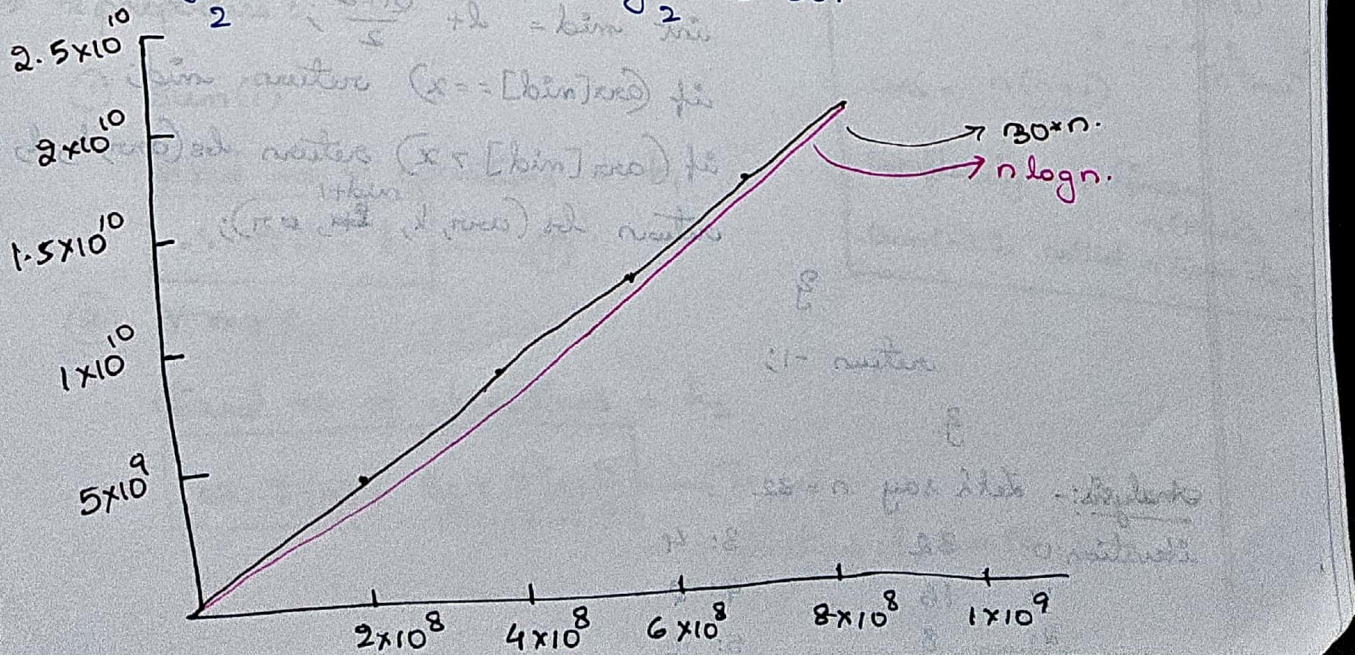
$O(n \log n)$ :-

An algorithm with worst case time complexity $W(n) \in O(n \log n)$ scales very well, since logarithms grow very slowly.

$\log_2 1000 = 3 \log_2 10 \approx 10.$ $\left[\log_2 10 \approx 3.3\right]$

$\log_2 1{,}000{,}000 = 6 \log_2 10 \approx 20$

$\log_2 1000000000 = 9 \log_2 10 \approx 30.$



$\log_2 n$ :-

From previous example:

$\log_2 10 \rightarrow 3.33$

$\log_2 10^2 \rightarrow 6.33$

$\log_2 10^3 \rightarrow 10$

$\log 10^6 \rightarrow 20$

$\log 10^9 \rightarrow 30.$

As input n increases exponentially ($10^3, 10^6, 10^9$) the time increases linearly (10, 20, 30)

So if 10 elements take 1 second, then
100 elements take 2 seconds,
1000 elements take 3 seconds ...

Divide and Conquer algorithms take logarithmic time complexity.

Example:-
```
int BinarySearch (int arr[], int k, int l, int r) {
    if (r >= l) {
        int mid = l + (r-l)/2;
        if (arr[mid] == x) return mid;
        if (arr[mid] > x) return bs(arr, k, l, mid-1);
        return bs(arr, k, mid+1, r);
    }
    return -1;
}
```

Analysis:- Let's say n = 32

| | | | |
|---|---|---|---|
| iteration 0: | 32 | 3: | 4 |
| 1: | 16 | 4: | 2 |
| 2: | 8 | 5: | 1 |

6 iterations for $n = 32$ (1st excluded).

Every iteration n reduces by 2.

After 5 iterations n reduces by $\frac{n}{2^5} = 1$.

For after k iterations $\Rightarrow \frac{n}{2^k} = 1$.

$$n = 2^k$$

$$\boxed{k = \log_2 n}$$

## Time complexity of recursive functions [Master Theorem] :-

The time complexity of a recursive function can be computed by formulating and solving a recurrence relation.

* Recurrence relation :-

```
int algorithm sum(int n) {
    if n==1 return 1;
    return n + sum(n-1);
}
```

Let's prove that this algorithm has linear time complexity.

→ 2 properties :

① sum(1)

fixed no. of operations = $k_1$

$$\boxed{\therefore T(1) = k_1} \quad \text{—— ①}$$

② $\forall n > 1$,

fixed no. of operations = $k_2$

$$\boxed{\therefore T(n) = k_2 + T(n-1)} \quad \text{—— ②}$$

$T(n-1)$ → no. of operations of sum(n-1).

$n + (n-1)$

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

$+ (n-2) + (n-3)$

$1$

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

$\text{ans} = \frac{n(n+1)}{2}$

complexity → linear.

n (times).

sum(n) is called n times itself

We need not specify $k_1$ and $k_2$ values, we can make them 1.

$$T(1) = 1$$

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

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

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

$$= 2+1+T(n-3) \quad = 3+T(n-3)$$

$$\cdots\cdots$$

$$= k+T(n-k)$$

$$\cdots\cdots$$

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

* Binary Search.

Recurrence relation: $T(1) = 1$

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

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

$$= 1+1+T(n/4) \quad = 2+T(n/4)$$

$$= 1+1+1+T(n/8) \quad = 3+T(n/8)$$

$$\cdots\cdots$$

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

$$\cdots\cdots$$

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

$$= \underline{\log n} = \Theta(\log n).$$

$n/2^k = 1$

$2^k = n$

$k = \log_2 n$

## Master theorem :-

The master theorem is a recipe that gives asymptotic estimates for a class of recurrence relations that often show up when analyzing recursive algorithms.

$$T(n) = aT(n/b) + f(n)$$

where $n$ = size of input

$a$ = number of subproblems in the recursion.

$n/b$ = size of each subproblem. All subproblems are assumed to have the same size.

$f(n)$ = cost of work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions.

Here, $a \geq 1$ and $b > 1$ are constants, and

$f(n)$ is an asymptotically positive function.

An asymptotically positive function means that for a sufficiently large value of $n$, we have $f(n) > 0$.

The master theorem is used in calculating the time complexity of recurrence relations (divide & conquer) algorithms in a simple and quick way.

$$T(n) = aT(n/b) + f(n)$$

where $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b a - \varepsilon})$, then $T(n) = \Theta(n^{\log_b a})$
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} * \log n)$.
3. If $f(n) = \Omega(n^{\log_b a + \varepsilon})$, then $T(n) = \Theta(f(n))$

$\varepsilon > 0$ is a constant.

Each of the above conditions can be interpreted as:

(1). If the cost of solving the sub-problems at each level increases by a certain factor, the value of $f(n)$ will become polynomially smaller than $n^{\log_b^a}$.

Thus the time complexity is oppressed by the cost of the last level i.e., $n^{\log_b^a}$.

(2) If the cost of solving the sub-problem at each level is nearly equal, then the value of $f(n)$ will be $n^{\log_b^a}$.

Thus, the time complexity will be $f(n)$ times the total number of levels i.e., $n^{\log_b^a} * \log n$.

(3) If the cost of solving the subproblems at each level decreases by a certain factor, the value of $f(n)$ will become polynomially larger than $n^{\log_b^a}$.

Thus, the time complexity is oppressed by the cost of $f(n)$.

* Solved Example of Master Theorem:-

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

$a = 3 \quad b = 2$

$\log_b^a \pm \varepsilon = 2.$

$\log_2^3 \pm \varepsilon = 2.$

$1.58 \pm \varepsilon = 2.$

$+\varepsilon = 2 - 1.58$

$\Rightarrow 1.58 < 2.$

$\therefore f(n) = O(n^{\log_b^a + \varepsilon}).$ → Case 3

$\therefore O(f(n)) = O(n^2)$

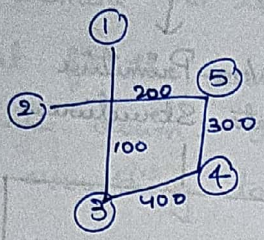
* Limitations:- cannot use master theorem if:

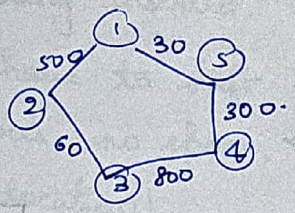
(1) $T(n)$ is not monotone i.e., $T(n) = \sin n$ (2) $f(n)$ is not polynomial (eg: $2^n$)

(3) $a$ is not constant eg, $a = 2n$ (4) $a < 1$.

## Example of n! algorithm :-

* The traveling salesperson :- Person wants to travel all cities with minimum distance.




There are 120 such connections to all cities and can be written 5!

5 cities → 120 ways to travel all cities.
6 cities → 720
7 cities → 5040
8 cities → 40320.
- - -
15 → 1,307,674,368,000
- - - -

Time complexity = O(n!)

This problem needs to be optimised.