ALGORITHM

- program is called **recursive** when an entity calls itself.
- A program is called **iterative** when there is a loop (or repetition).

Time Complexity

- The time complexity of the method may vary depending on whether the algorithm is implemented using recursion or iteration.

- **Recursion:** The time complexity of recursion can be found by finding the value of the nth recursive call in terms of the previous calls. Thus, finding the destination case in terms of the base case, and solving in terms of the base case gives us an idea of the time complexity of recursive equations. Please see Solving Recurrences for more details.

- **Iteration:** The time complexity of iteration can be found by finding the number of cycles being repeated inside the loop.

Recursion: Recursion involves calling the same function again, and hence, has a very small length of code. However, as we saw in the analysis, the time complexity of recursion can get to be exponential when there are a considerable number of recursive calls. Hence, usage of recursion is advantageous in shorter code, but higher time complexity.

Iteration: Iteration is the repetition of a block of code. This involves a larger size of code, but the time complexity is generally lesser than it is for recursion.

TIME COMPLEXITY ANALYSIS

- Time complexity is a function that describes the amount of time required to run an algorithm, as input size of the algorithm increases.
- In the above definition, “time” doesn’t mean wall clock time, it basically means how the number of operations performed by the algorithm grows as the input size increases. Operations are the steps performed by an algorithm for a given input.

asymptotic notation

Asymptotic Notations are mathematical tools used to represent the complexities of an algorithm.

There are three forms of asymptotic notations

Big O (Big-Oh)

Big Ω (Big-Omega)

Big Θ (Big-Theta)

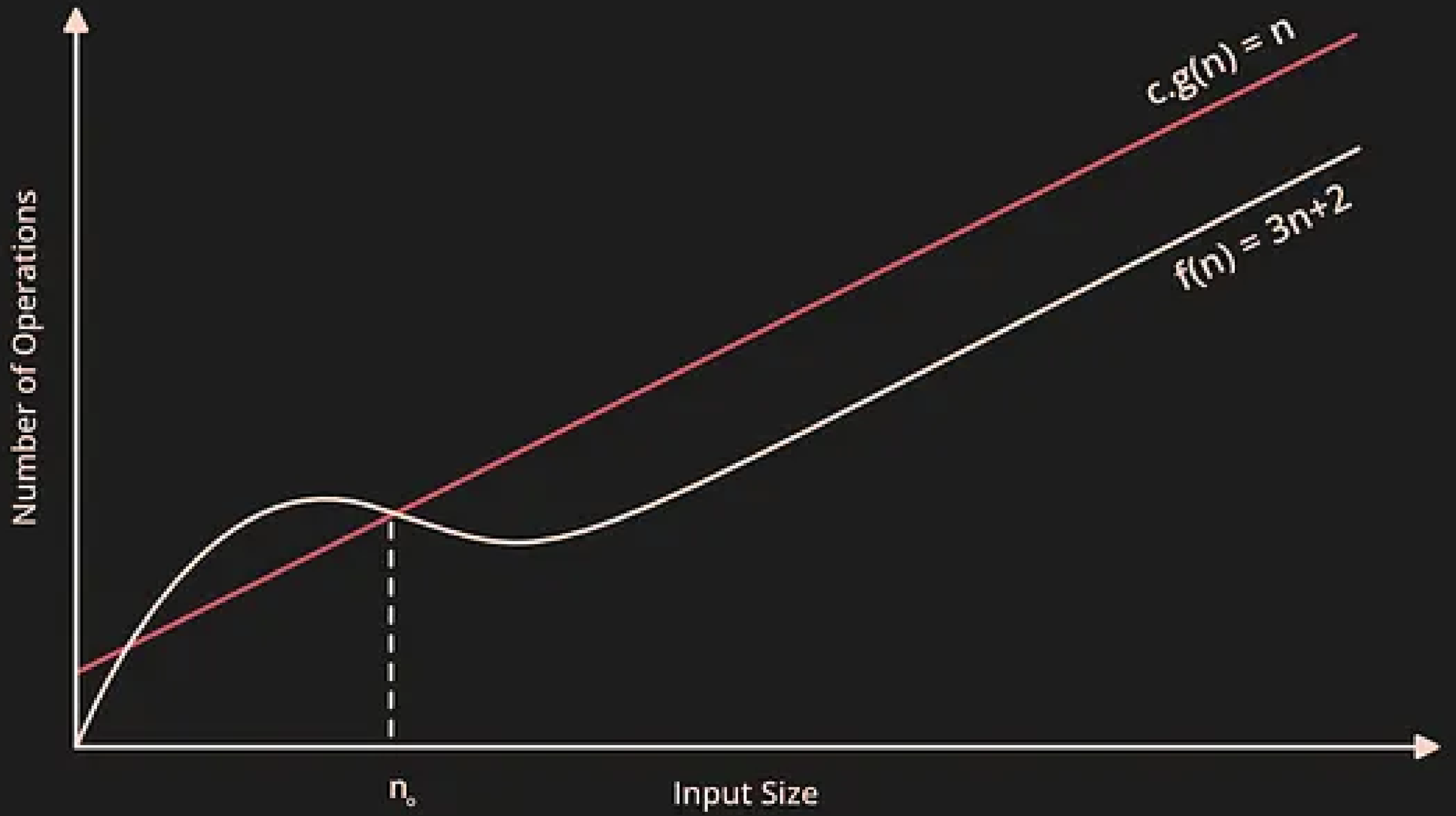
Big-Oh Notation

Big Oh is an Asymptotic Notation for the **worst-case scenario**, which is also the ceiling of growth for a given function. It provides us with what is called an asymptotic upper bound for the growth, of an algorithm.

The Mathematical Definition of Big-Oh

f(n) ∈ O(g(n)) if and only if there exist some positive constant c and some non-negative integer n_0 such that, $f(n) \leq c\, g(n)$ for all $n \geq n_0$, $n_0 \geq 1$ and $c>0$.

The above definition says in the worst case, let the function f(n) be your algorithm's runtime, and g(n) be an arbitrary time complexity you are trying to relate to your algorithm. T**hen O(g(n)) says that the function f(n) never grows faster than g(n) that is f(n)<=g(n)and g(n) is the maximum number of steps that the algorithm can attain.**


Number of Operations
c.g(n) = n
f(n) = 3n+2
n₀
Input Size

- In the above graph, the c.g(n)is a function that gives the maximum runtime(upper bound) and f(n) is the algorithm’s runtime.

Example 1: Find the upper bound for $f(n) = 3n + 2$

To show that function $3n + 2$ is $O(n)$,
For it to happen, it has to satisfy the condition $f(n) \leq c.g(n)$.

$=> 3n + 2 \leq cn$

Increase the value of every term to the highest power.

$=> 3n + 2n \leq cn$

$=> 5n \leq cn$

Divide by n we get the values of
$c = 5$ and $n_0 = 1$
Now we can say $=> 3n + 2 \leq 5(n)$
$O(n)$ is the upper bound, with $c = 5$ and $n_0 = 1$.

Example 2: Find the upper bound for $f(n) = 1000n^2 + 1000n$

Example 2: Find the upper bound for $f(n) = 1000n^2 + 1000n$

To show that function $1000n^2 + 1000n$ is $O(n^2)$,
For it to happen, it has to satisfy the condition $f(n) \leq c.g(n)$.

$$=> 1000n^2 + 1000n \leq cn^2$$

Increase the value of every term to the highest power.

$$=> 1000n^2 + 1000n^2 \leq cn^2$$

$$=> 2000n^2 \leq cn^2$$

Divide by n^2 we get the values of
$c = 2000$ and $n_0 = 1$
Now we can say $=> 1000n^2 + 1000n \leq 2000(n)$
$O(n^2)$ is the upper bound, with $c = 2000$ and $n_0 = 1$.

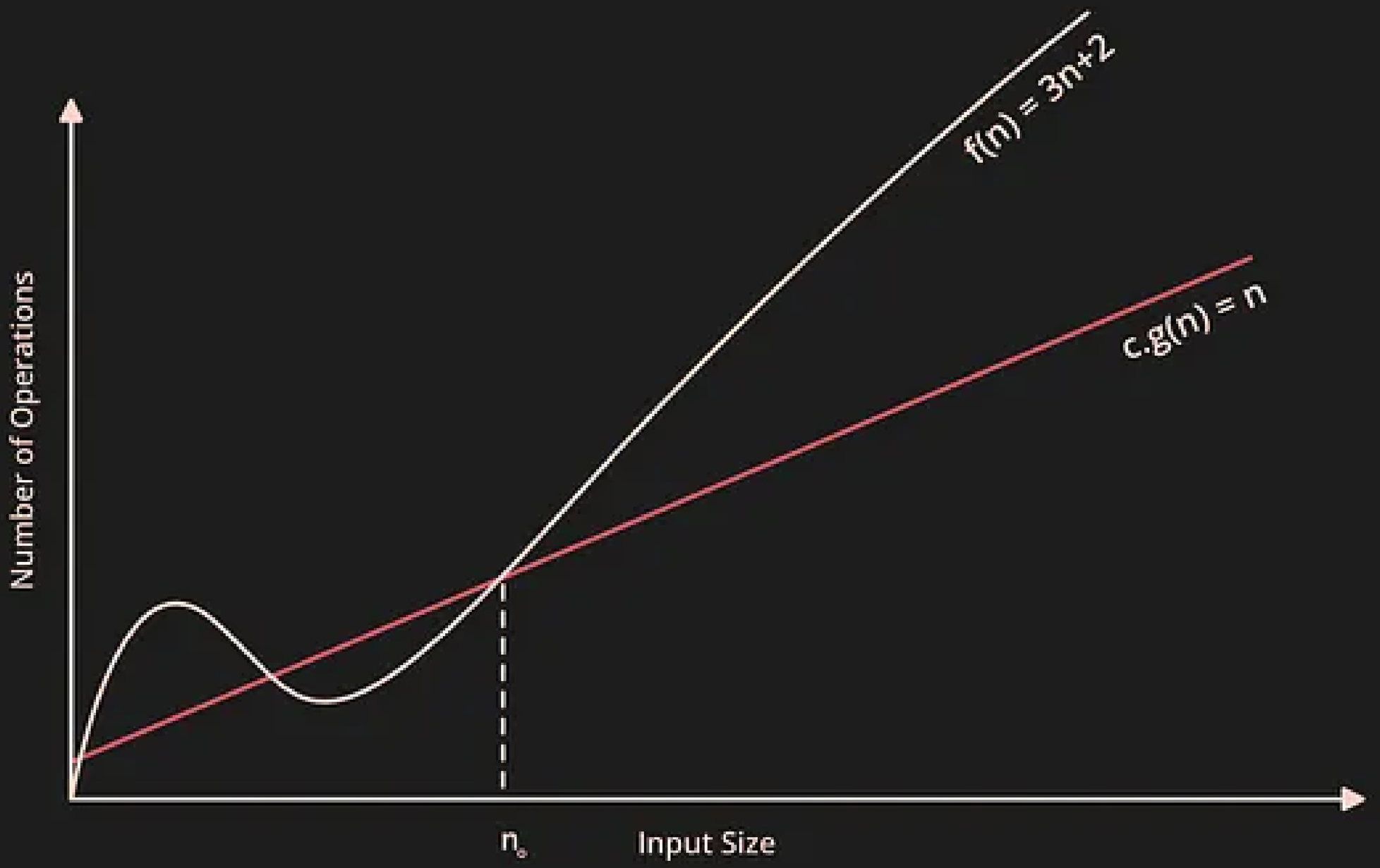
Big-Omega Notation

Big Omega is an Asymptotic Notation for the best-case scenario. Which is the floor of growth for a given function. It provides us with what is called an asymptotic lower bound for the growth rate of an algorithm.

The Mathematical Definition of Big-Omega

$f(n) \in \Omega(g(n))$ if and only if there exist some positive constant c and some non-negative integer n_0 such that, $f(n) \geq c\ g(n)$ for all $n \geq n_0$, $n_0 \geq 1$ and $c>0$.

The above definition says in the **best case,** let the function f(n)be your algorithm's runtime and g(n) be an arbitrary time complexity you are trying to relate to your algorithm. Then Ω(g(n)) says that the function g(n)never grows more than f(n)i.e. f(n)>=g(n), g(n)indicates the minimum number of steps that the algorithm will attain.


f(n) = 3n+2
c.g(n) = n
Number of Operations
n₀
Input Size

In the above graph, the c.g(n)is a function that gives the minimum runtime(lower bound) and f(n)is the algorithm’s runtime.

Example 1: Find the lower bound for $f(n) = 3n + 2$

To show that function $3n + 2$ is $\Omega(n)$
It has to satisfy the condition $f(n) \geq c.g(n)$.

$=> cn \leq 3n + 2$

Rearrange the terms, we get.

$=> cn - 3n \leq 2$

Take n as common, we get.

$=> n(c - 3) \leq 2$

On rearranging, we get.

$=> n \leq \frac{2}{(c-3)}$

Choose the value of c, where n should not be negative or undefined.
If we assume $c = 4$, then $n_0 = 2$
Now we can say $=> 3n + 2 \leq 4(n)$
$\Omega(n)$ is the lower bound of $3n + 2$, with $c = 4$ and $n_0 = 2$.

Big-Theta Notation

Big Theta is an Asymptotic Notation for the average case, which gives the average growth for a given function. Theta Notation is always between the lower bound and the upper bound. It provides us with what is called an asymptotic average bound for the growth rate of an algorithm. If the upper bound and lower bound of the function give the same result, then the Θ notation will also have the same rate of growth.

The Mathematical Definition of Big-Theta

$f(n) \in \Theta(g(n))$ if and only if there exist some positive constant c_1 and c_2 some non-negative integer n_0 such that, $c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$, $n_0 \geq 1$ and $c>0$.

Calculating the Time Complexity

Example 1: Loops

```
// executes n times
for (i = 1; i <= n; i++)
{
      a = a + i; // constant time, c
}
```

The running time of a loop is the multiple of, statements inside the loop by the number of iterations.

Total Time = constant c × n = c n = O(n)

Example 2: Nested Loops

```
// outer loop executed n times
for (int i = 1; i <= n; i++)
{
    // inner loop executed n times
    for (j = 1; j <= n; j++)
    {
        a = a + 1 // constant time
    }
}
```

The total running time of this algorithm is the product of the sizes of all the loops.

Total Time = $c \times n \times n = cn^2 = O(n^2)$

Example 3: Consecutive Statements

```
a = a + 1; // constant time
// executed n times
for (int i = 1; i < n; i++)
{
    b = b + 1; // constant time
}
// outer loop executed n times
for (int i = 1; i <= n; i++)
{
    // inner loop executed n times
    for (int j = 1; j <= n; j++)
    {
        c = c + 1; // constant time
    }
}
```

The total running time of this algorithm is the sum of all the time complexities of each statement of the algorithm.

Total Time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$

Example 4: Logarithmic Complexity

```
for (i = 1; i <= n;)
{
    i = i * 2;
}
```

we observe the value of i is doubling on every iteration i=1,2,4,8.. , and the original problem is cut by a fraction (1/2)on each iteration, the complexity of this kind is of log time.

Total Time = O(logn)

Complexity analysis for recursion

Recurrence relation

- Recurrence relation is used to analyze the time complexity of recursive algorithms in terms of input size.
- A recurrence relation is a mathematical equation in which any term is defined by its previous terms.
- Recurrence relation is used to analyze the time complexity of recursive algorithms in terms of input size.

Example — Reverse an array (Decrease by a constant)

solving the subproblem of reversing (n-2) size array.

```
public void reverse(int[] a, int l, int r){
  if(l >= r)
    return;

  swap(a[l], a[r]);
  revserse(a, l-1, r+1); // inut size decreased by 2
}
```

Time complexity => Time complexity of solving an (n-2) size problem +

Time complexity of swapping operation

recurrence relation => T(n) = T(n-2) + c, where T(1) = c

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

Usually, the running time T(n)
is expressed as a conditional statement having the following two conditions

Base Case: The base case is the running time of an algorithm when the value of n is small such that the problem can be solved trivially.

Recursive case: The recursive running time is given by a recurrence relation for a large value of n

.

Using base and recursive case, the running time of a recursive function would look like the following.

$T(n)=\{O(1)$ if $n\leq 2$

otherwise $T(n-1)+O(1)$

```
int fact(int n) {
    if (n == 0) {
        return 1;
    }
    return n * fact(n - 1);
}
```

The base condition happens when the value of n is 0.

We can find the value of 0! in constant time O(1) trivially i.e. for base condition

T(0)=O(1)

Next, we look for the recursive call. There is only one recursive call with input size n−1 and after the recursive function returns we do a simple multiplication that takes O(1) time i.e. the total running time is $T(n)=T(n-1)+O(1)$

Combining this with the base condition, we get

$$T(n) = \begin{cases} O(1) & \text{if } n = 0 \\ T(n-1) + O(1) & \text{otherwise} \end{cases}$$

```
int fib(int n) {
    if (n <= 1)
     {
        return n;
     }

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

Time complexity

Combining with the base case, we get

$$T(n)=\{O(1) \text{ if } n\leq 1$$
$$T(n-1)+T(n-2)+O(1) \text{ otherwise}$$

Methods to solve recurrence relations

- Recursion Tree Method
- Master Theorem
- Iterative method / substitution method

1. **substitution method**

Master’s theorem solves recurrence relations of the form-

$$T(n) = a\,T\left(\frac{n}{b}\right) + \theta\left(n^{k} \log^{p} n\right)$$

Master's Theorem

Here, a >= 1, b > 1, k >= 0 and p is a real number.

masters method

[redacted] If the recurrence is of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^k log^p n)$, where $a \geq 1, b$ $1, k \geq 0$ and p is a real number, then:

1) If $a > b^k$, then $T(n) = \Theta\left(n^{log_b^a}\right)$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$
   b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$
   c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

PRACTICE PROBLEMS BASED ON MASTER THEOREM-

Problem-01: Solve the following recurrence relation using Master's theorem- T(n) = 3T(n/2) + n^2

Solution- We compare the given recurrence relation with T(n) = aT(n/b) + θ (n^klogp^n).

a = 3 b = 2 k = 2 p = 0 Now, a = 3 and b^k = 2^2 = 4.

Clearly, a < b^k. So, we follow case-03.
Since p = 0, so we have- T(n) = θ (n^klogpn) = T(n) = θ (n^2log0n)

Thus, T(n) = θ (n2)

Problem-02: solve the following recurrence relation using Master's theorem- T(n) = 2T(n/2) + nlogn

We compare the given recurrence relation with T(n) = aT(n/b) + θ (n^klogp^n).

Clearly, a = b^k. So, we follow case-02.

Since p = 1, so we have- T(n) = θ (nlogb^a.log^p+1n)

T(n) = θ (nlog2^2.log^1+1n) **Thus,**

T(n) = θ (nlog^2n)

Problem-03: Solve the following recurrence relation using Master's theorem $T(n) = 8T(n/4) – n^2\log n$

Solution-

The given recurrence relation does not correspond to the general form of Master's theorem.
So, it can not be solved using Master's theorem.

Steps to Solve Recurrence Relations Using Recursion Tree Method-

Step-01: Draw a recursion tree based on the given recurrence relation.

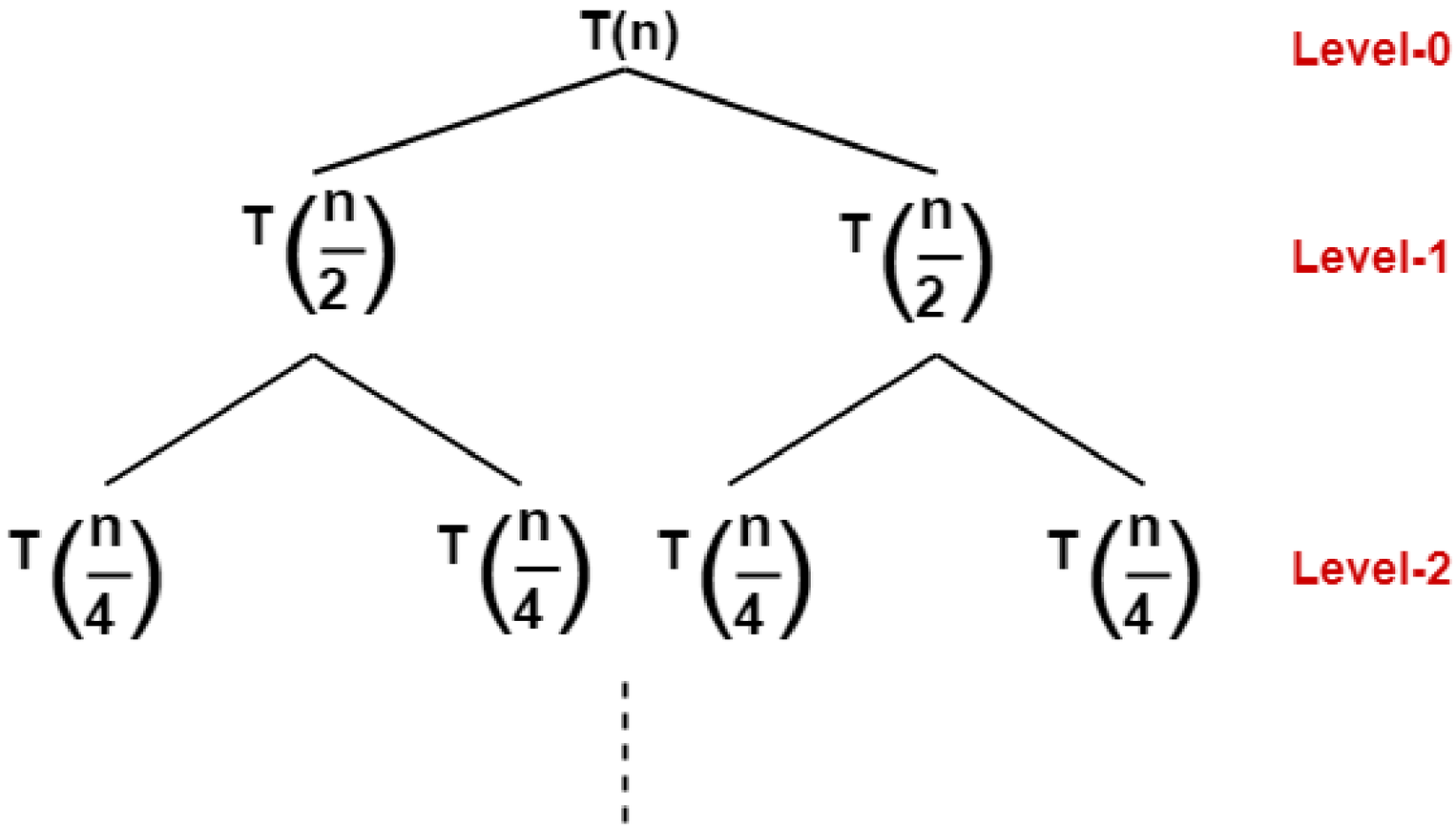
Step-02: Determine-

Cost of each level
Total number of levels in the recursion tree
Number of nodes in the last level
Cost of the last level

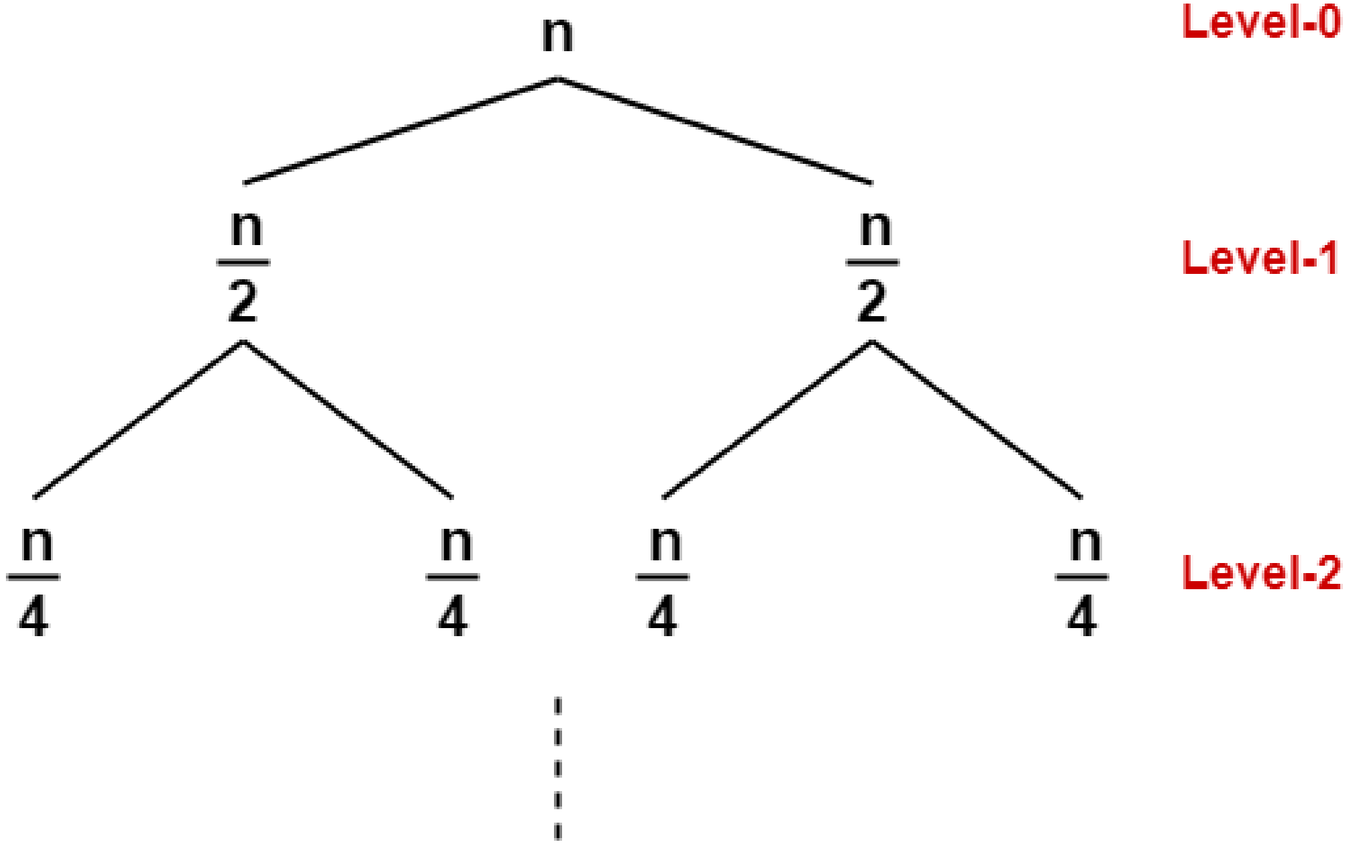
Step-03: Add cost of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation.

Solve the following recurrence relation using recursion tree method-

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



cost of each level



step 2

Determine cost of each level-

Cost of level-0 = n

Cost of level-1 = n/2 + n/2 = n

Cost of level-2 = n/4 + n/4 + n/4 + n/4 = n and so on.

step 3: Determine total number of levels in the recursion tree-

Size of sub-problem at level-0 = n/2^0
Size of sub-problem at level-1 = n/2^1
Size of sub-problem at level-2 = n/2^2
Continuing in similar manner, we have-
Size of sub-problem at level-i = n/2^i

Suppose at level-x (last level), size of sub-problem becomes 1. Then-

n / 2^x = 1 = 2^x = n

Taking log on both sides, we get- x = log(base2)n

Step-04:

Determine number of nodes in the last level-

Level-0 has 2^0 nodes i.e. 1 node
Level-1 has 2^1 nodes i.e. 2 nodes
Level-2 has 2^2 nodes i.e. 4 nodes

Continuing in similar manner, we have-

Level-log2n has 2^log(base2)n nodes i.e. n nodes

Step-05: Determine cost of last level-

Cost of last level = n x T(1) = θ(n)

Step-06: Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = \underbrace{\{\ n + n + n + \ldots\ldots\ \}}_{\text{For } \log_2 n \text{ levels}} + \theta\ (n)$$

n x log(base2)n + θ (n)

= nlog(base2)n + θ (n)

= θ (nlog(base2)n)