Vision 2023

A Course for GATE & PSUs

Computer Science Engineering

Algorithm

CHAPTER 1

Asymptotic Analysis

CHAPTER 1

ALGORITHM

ASYMPTOTIC ANALYSIS

- Combination of a sequence of Finite sets of steps to solve a specific problem is called an algorithm.
- **Properties of algorithm:**
  1. finite time to produce output
  2. should produce correct output
  3. independent of programming language.
  4. every step should perform some tasks.
  5. steps should be unambiguous.
  6. number of input can be zero or more and output should be atleast one.
- Steps means instructions which contains fundamental operators i.e. (+, *, ÷, % , =, etc.)
- **Analysis of algorithm:**

  How to check the available algorithm , which is the best?
  1. **time**: time complexity of the algorithm.

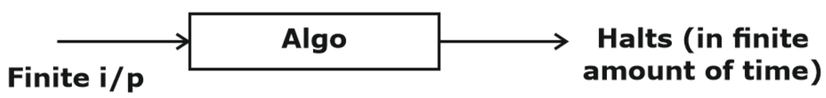
     T (A) = C (A) + R (A)

     C(A): compile time of A, depends on the compiler and software

     R(A): Runtime of A, depends on processor and hardware.
  2. **space:** space complexity of algorithm.
- **Criteria for algorithm-**

  1. Finiteness:

  Algorithms must terminate in a finite amount of time.



Example:

e.g.

i = 1

while (1)

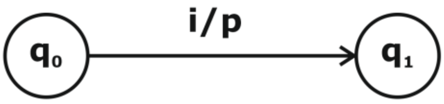
{

i = i + 1 → Since it is executing infinitely. So, this kind of solution. Not allowed in algo.

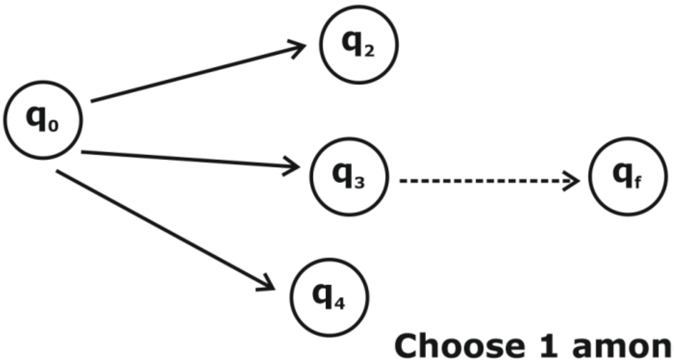
2. Definiteness:

Deterministic Algo:

Each step of the algorithm must have only one unique solution called as deterministic algorithm.



Non Deterministic Algo – Each step of algo consists of a finite no. of solution and algo should choose the correct solution on the 1st attempt.
(Not possible to implement in computer)



Choose 1 among than

<u>Steps to solve any problem:</u>

1. Identifying problem statement:

Example: Arrange 4 Queens Q1 , Q2 , Q3 , Q4 into 4x4 chess board.

2. Identifying constraints :

e.g. No two queens on same ecs & on same column & on same diagonal.

3. Design logic :

Depending on the characteristics of the problem we can choose any one at the following design strategy for design logic.

(i) Divide & Conquer

(ii) Greedy method

(iii) Dynamic programming

(iv) Branch & Bound.

(v) Back tracking etc.....

4. Validation :

Most algorithms are validated by using mathematical indexical.

5. Analysis :

Process of comparing two algo w.r.t time, space, no. of register, network bandwidth etc. is called analysis.

<u>Priory Analysis</u>

Analysis done before executing.

e.g. x = x + 1

→ **Principle :** frequency count of fundamental Ins^n.

Since x = x + 1 being carried out only 1 time

So it's complexity is 0 (1) [order of 1]

→ It provides estimated values.

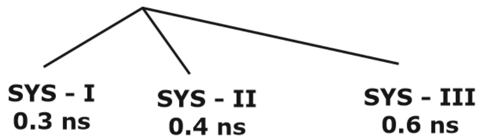
→ It provides uniform values.

→ It is independent of CPU, O/S & system architecture.

<u>Posterior Analysis</u>

→ Analysis done after executing.

e.g. x = x + 1;



→ It provides exact values.

→ It provides non uniform values .

6. Implementation.

7. Testing & Debugger.

Apriori Analysis :

It is a determination of the order of magnitude of a statement.

Example 1:

main(){

int x,y,z; ->order of magnitude of this statement is 1, this statement executes once when the program executes.

x=y+z;

}

Time complexity O(1).

Example-2:

For (i l; i ≤ n; i + +) ------------------→ 2 (n + 1)

{

printf ("ME"); ------------------→ n

}

3n + 2

↓

TC = θ(n)

Time complexity is equal to the inner most statement of loop
∴ directly = θ(n)

Now, time complexity

i --------→ 1

n -------→ 1

∴ SC = constant = θ(1)

Example-3:

For (i = n; i ≥ l; i - -) ------------→ 2(n+1)

Printf (“GRADE UP”) ------------→ n times

∴ T.C = θ (n)

S.C = θ (1)

i --------→ 1

n -------→ 1

constant

Example-4:

For (i = l; i ≤ n; i = i + 5)

printf (“GRADE UP”) ; ---------→ $\left[\frac{n-1}{5}\right]+1$

$$\underbrace{i = 1\ ,\ 1+5\ ,\ 1+2*5\ ,\ 1+3*5\ ,\ \ldots\ldots\ldots\ ,\ 1+K*5 \le n}_{\textbf{K + 1 prints}}$$

$1 + K.5 \le n$

$K.5 \le n - 1$

$K \le \frac{n-1}{5}$

$\boxed{K = \left[\frac{n-1}{5}\right]} \rightarrow TC = \theta(n)$

Example-5

For (i = n; i ≥ l; i= 1 – 5)

print (“GRADEUP”) --------------------→ $\left[\frac{n-1}{5}\right]+1$ - times exactly same as precious loop

T.C = θ (n)

For (i = l; i ≤ n; i = i * 2)

printf (“GRADEUP”); ----------→ [$\log_2$n] + 1 times

$2^{K \le N}$

1' 2'

$$i \longrightarrow \underbrace{1, 2, 2^2, 2^3, \ldots\ldots\ldots\ldots, 2^K}_{\text{Loop runs (K + 1) times}}$$

∴ (K + 1) times printf execude

$2^K \le n$

$\log_2 2^K \le \log_2 n$ **(Taking log)**

$K \log_2 2^2 \le \log_2 n$

$\boxed{K = [\log_2 n]}$

∴ loop runs [$\log_2$n] + 1 times

T.C. = θ ($\log_2$ n)

Example-6

6) For (i = n; i ≥ l; i = i/2)

printf (“ME”); -----------------→ [$\log_2$ n] + 1 times

same as above

$\frac{n}{2K}$ ³ 1

$$i \longrightarrow \underbrace{n, \quad \frac{n}{2}, \quad \frac{n}{2^2}, \quad \cdots\cdots\cdots\cdots \quad \frac{n}{2^K}}_{\text{(K+1) times}}$$

$\frac{n}{2^K} \ge 1$

$\Rightarrow n \ge 2^k$

$K = [\log_2 n]$

$\therefore = \theta(\log_2 n)$

Example-7

For (i = l; i ≤ n; i = i * 5)

printf (“ME”) ; --------------------→ [$\log_5$ n] + 1 ← (case 5)

$$i \longrightarrow \underbrace{1,\ 5,\ 5^2,\ \ldots\ldots\ldots\ldots\ 5^K}_{\textbf{K+1 times}}$$

$5^K \le n$

$\log_5 (5^K) \le \log_5 n$

$K \log_5 5 \le \log_5 n$

$K = [\log_5 n]$

Example 8:

For (i =2; i ≤ n; i = i^2)

printf (“ME”) ;

$$2^{2^K} \le n$$

$$i \longrightarrow \underbrace{2^{1^0},\ 2^{2^1},\ 2^{2^2},\ 2^{2^3},\ 2^{2^4},\ \text{-------------}\ 2^{2^K}}_{\textbf{upon run (K+1) times}}$$

$2^{2^K} \le n$

$\Rightarrow \log_2 2^{2^K} \le \log_2 n$

$\Rightarrow 2^K \log_2 2 \le \log_2 n$

$\Rightarrow 2^K \le \log_2 n$

Again take log

$\Rightarrow K \log_2 2 \le \log_2 (\log_2 (n))$

$\Rightarrow \boxed{K = [\log_2 (\log_2 n)]}$

$\therefore TC = \theta(\log_2 \log_2 n)$

Example 9:

For $\left(i = n;\ \ i \ge 2;\ \ i = \sqrt{i}\right)$

printf (“GRADE UP”) --------------→ [$\log_2 \log_2$ n] + 1 times

same as above

$$i \longrightarrow \quad \underbrace{n^{\frac{1}{2^0}},\ n^{\frac{1}{2^1}},\ n^{\frac{1}{2^2}},\ \text{-------}\ n^{\frac{1}{2^t}}}_{\textbf{(K+1) times}}$$

$n^{\frac{1}{2^K}} \geq 2$

$\Rightarrow$ Apply log

$\Rightarrow \frac{1}{2^K}\log_2 n \geq \log_2 2$

$\Rightarrow \frac{1}{2^K}\log_2 n \geq 1$

$\Rightarrow \frac{1}{2^K}\log_2 n \geq 1$

$\Rightarrow \log_2 n \geq 2^K$

Again take log

$\Rightarrow K = \left[\log_2 \log_2 n\right]$

$T.C = \theta\left(\log_2 \log_2 n\right)$

if we have $\quad i = i^3$

$\downarrow$

$\therefore [\log_3 \log_3 n] \leftarrow$ (case 3)

Example 10:

```
main() {
int x,y,z,i,j,k,n;
for(i=1 to n) { ---------------------> order of magnitude =n
for(j=1 to i){ -----------------------> 1,2,3,...n
for(k=1 to 135){ ------------------> order of magnitude =135
x=y+z;
}
}
}
}
```

i=1	i=2	i=3	. . .	i=n
j=1 to 1	j=1 to 2	j=1,2,3	. . .	j=1,2,3,...n
k=135	135, 135	135,135,135	. . .	135,135,135...135

T= 1*135+2*135+3*135+....+n*135

=135(1+2+3+....+n)

=135*n*(n+1)/2

=O(N^2)

Shortcut

(1)	$\text{for}(i = 1; i \le n; i = i + c)$ $\boxed{\text{or}}$ $\text{for}(i = n; i \ge 1; i = i - c)$	$\text{T.C} = \theta(n)$
(2)	$\text{for}(i = 1; i \le n; i = i * c)$ $\boxed{\text{or}}$ $\text{for}(i = n; i \ge 1; i = i / c)$	$\text{T.C} = \theta(\log_c n)$
(3)	$\text{for}(i = c; i \le n; i = i^c)$ $\boxed{\text{or}}$ $\text{for}(i = n; i \ge c; i = i\sqrt{i})$	$\text{T.C} = \theta(\log_c \log_c n)$

Example 11:

1) For (i =2 ; i ≤ 2^n; i = i^2)

printf ("GRADE UP");

1) for (i = 2 ; i ≤ 2^n ; i = i^2)

printf ("GRADE UP");

$i \rightarrow 2^1, 2^{2^1}, 2^{2^2}, \dots\dots\dots 2^{2^K} \le 2^n$

(Shortcut)

Compare,

for (i = 2 ; i ≤ 2^n, i = i^2)

$\log_2\log_2 n$

↑

log2 log2 2^n

↓

(log n)

$2^{2^K} \le 2^n$

$\Rightarrow 2^K \log 2^2 \le \log 2^2 \Rightarrow 2^K \le n$

TC = θ (log n)

$\Rightarrow \boxed{K = \log_2 n}$

2) for (i = n/2 ; i ≤ n ; i = i * 2)

printf ("GRADE UP");

$i \rightarrow \frac{N}{2}, \quad n, \quad \boxed{2n}$..........

×

2 times ← constant

T.C = θ (1)

3) for (i = 1 ; i ≤ 2^n ; i = i * 2)

printf ("GRADE UP")

$i \rightarrow 1, \quad 2, \quad 4, \quad 2^3 \ldots\ldots\ldots\ldots 2^K \quad \leq \quad 2^n$

2^2

$2^K \leq 2^n$

$\boxed{K = n}$

TC = θ (n)

Nested Loops –

(1) Independent Nested Loop :

the inner loop variable is independent of the outer loop.

E.g.-

for (i = 1; i ≤ n ; i + +)

for (j = 1 ; j ≤ n^2 ; j * 2)

↑

j value does not depend on i

E.g.-

```
          ┌ for (i = 1; i ≤ n; i + +)
          │ ┌ for (j = 1; j ≤ n^2 ; j *2)
          │ │ ┌ for (k = n; k ≥ 2; k = √K)
n times   │ │ │         printf ("ME")
          │ │ └
          │ │ *log2 log2 n
          │ └
          │ * 2 log2n
          └
```

Overall time complexity is no. of times each loop runs.

$\Rightarrow n * 2 \log_2 n * \log_2 \log_2 n$

$\Rightarrow n \log_2 n * \log_2 \log_2 n$

TC = θ (n log n. log log n)

2) for ($i = 1$; $i \leq n^2$; $i = i * 2$) $\longrightarrow$ $\theta(\log_2 n^2) \to 2 \log n$

for ($j = 1$; $i \leq n^2$; $j++$) $\longrightarrow$ $\theta(n^2) \to n^2$

for ($K = n^2$; $K \geq 1$; $K = K/2$) $\longrightarrow$ $\theta(\log_2 n^2) \to 2 \log n$

printf ("M.E") ;

Now,

TC = $2 \log n * n^2 * 2 \log n$

TC = $\theta(n^2 . \log^2 n)$

Time complexity of loop

Loop only depend on i.

$\Rightarrow$ i does not depend on j

$\therefore$ for ($i = n$; $i \geq 1$; $i = i/2$)

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

(2) Depending Loop

Inner loop variable depends on the outer loop.

x = 0

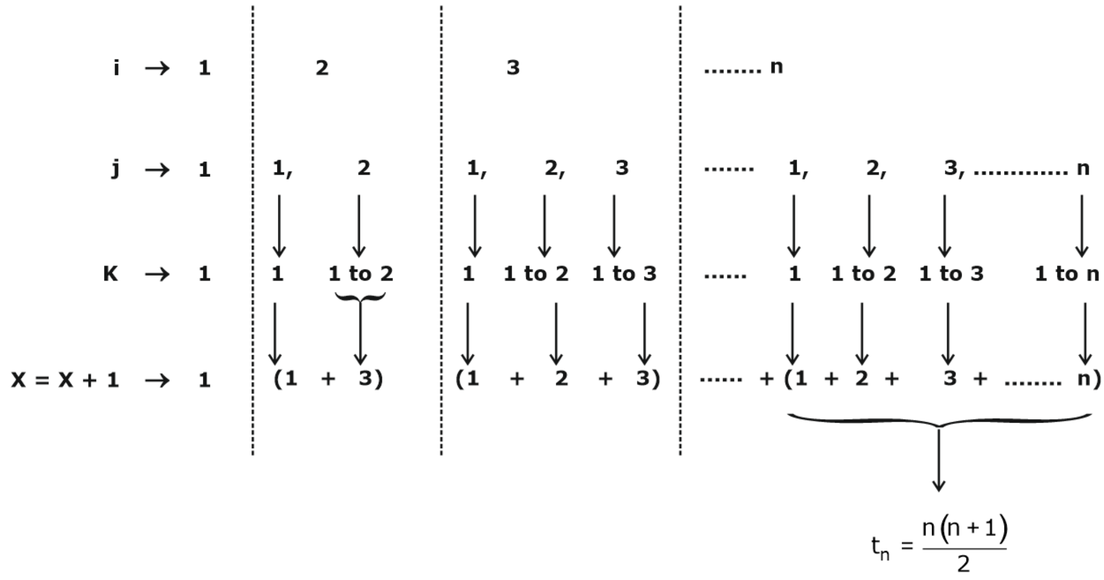
E.g.-

```
for (i = 1 ; i ≤ n ; i + +)
{
        for (j = 1 ; j ≤ i ; j + +)
                for ( K = 1 ; K ≤ j ; K + + )
                        X = X + 1
{
```

a) What is the frequency count of the loop?

b) If n = 10, what is the final value of n?

Expansion of loop



Above is in arithmetic progression.

$S_n = \Sigma tn$

n the term

$$= \frac{\Sigma n(n+1)}{2} \Rightarrow \frac{1}{2}\left[\Sigma n^2 + \Sigma n\right]$$

$$\Rightarrow \frac{1}{2}\left[\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}\right]$$

$$\Rightarrow \frac{1}{2} \cdot \frac{n(n+1)}{2}\left[\frac{2n+1}{3} + 1\right]$$

$$\Rightarrow 1.\frac{n(n+1)(n+2)}{6} \Rightarrow \theta\left(n^3\right)$$

()

if n = 10

$$\Rightarrow \frac{10(10+1)(10+2)}{6}$$

$$\Rightarrow \frac{10 \times 11 \times 12}{6} \Rightarrow 220$$

Asymptotic analysis:

Why performance analysis?

There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc. Why worry about performance?

The answer to this is simple, we can have all the above things only if we have performance. So performance is like currency through which we can buy all the above things. Another reason for studying performance is – speed is fun!

To summarize, performance == scale. Imagine a text editor that can load 1000 pages, but can spell check 1 page per minute OR an image editor that takes 1 hour to rotate your image 90 degrees left OR … you get it. If a software feature can not cope with the scale of tasks users need to perform – it is as good as dead.

Given two algorithms for a task, how do we find out which one is better?

One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.

1) It might be possible that for some inputs, the first algorithm performs better than the second. And for some inputs the second performs better.
2) It might also be possible that for some inputs, the first algorithm perform better on one machine and the second works better on other machine for some other inputs.

* **Asymptotic Notation :**

To compare two algorithms' rate of growth with respect to time & space we need asymptotic notation.

Big-O Analysis of Algorithms

We can express algorithmic complexity with the big-O notation. For a problem of size N:

- A constant-time function is is "order 1" : O(1)
- A linear-time function is "order N" : O(N)
- A quadratic-time function/method is "order N squared" : O(N 2)

Definition: g and f be functions from the set of natural numbers to itself. The function f is said to be O(g) (read big-oh of g), if there is a constant c and a natural n 0 such that f (n) ≤ cg(n) for all n > n0 .

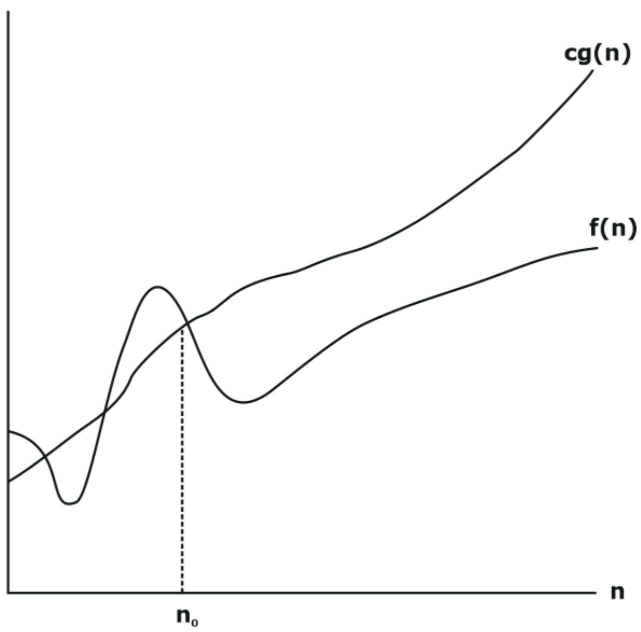
Note: O(g) is a set!

Abuse of notation: f = O(g) does not mean f ∈ O(g).

The Big-O Asymptotic Notation gives us the Upper Bound Idea,

f(n) = O(g(n)) if there exists a positive integer n_0 and a positive constant c, such that f(n)≤ c.g(n) ∀ $n \geq n_0$

Diagram for Big oh notation:



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

Note :

Even Though $n^2 > n^3 > n^4$ are upper bound's to the

$t(n) = n^2 + n + 1$; we have to take the least **upper base** only.

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

Shortcut :

If $t(n)\ a_0 + a_1 n + a_2 n^2 + \ldots\ldots + a_m n^m\ (a_m \neq O)$

then $t(n) = O(n^m)$

The steps for Big-O runtime analysis is as follows:

1. find what the input is and what n represents.
2. get the maximum number of operations that the algorithm performs in terms of n.
3. remove all the highest order terms.
4. eliminate all the constant factors.

Example 1:

$f(n) = n^2 \log n$; $g(n) = n (\log n)^{10}$, which of the following is true?

A. $f(n) = (g(n)), g(n) \neq 0 (f(n))$

B. $f(n) \neq 0(g(n)), g(n) = 0(f(n))$

C. $f(n) = 0(g(n)), g(n) = 0 (f(n))$

D. $f(n) \neq 0(g(n)), g(n) \neq 0 f(n)$

Ans. B

Example 2:

$\rightarrow f(n) = n^2 \log n \qquad g(n) = n (\log n)^{10}$

$= n \log \times n \qquad = n \log n (\log n)^9$

Since $n > (\log n)^n$

$f(n) > g(n)$

$\Rightarrow \quad g(n) < f(n)$

Low = 0 (high) $\quad \Rightarrow \quad g(n) = 0(f(n))$

Big – Omega (Ω) :

$f(n)$ is $\Omega\ (g(n))$ iff $\exists$ some $C > 0$ and $K \geq 0$ such that $t(n) \geq C.\ g(n)$; $\forall\ n \geq K$.

Ex:

If $f(n) = n^2 + n + 1$, then $f(n) = \Omega\ (\ )$

$\rightarrow \quad n^2 \geq n^2$

$n^2 + n \geq n^2$

$n^2 + n + 1 \geq 1\ n^2$; $\forall\ n \geq 0$

$n^2 + n + 1 = \Omega\ (n^2)$

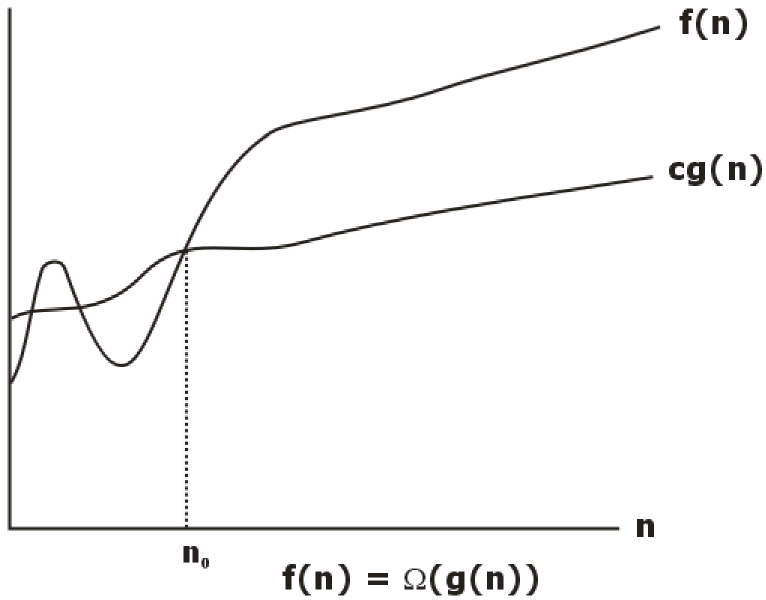


Fig. Omega notation

→ $n^2 \geq n$

$n^2 + n \geq n$

$n^2 + n + 1 \geq 1.\ n$; $\forall n \geq 0$

$n^2 + n + 1 = \Omega\ (n)$

Always take higher value from the lower frequency.

Note :

Even though n^2, n are lower bonus to f(n) you have to take the greatest lower bound only.

Shortcut :

If $f(n) = a_0 + a_1n + a_2n^2 + ... + a_mn^m$ ($a_m \neq 0$)

then $f(n) = \Omega\ (n^m)$

Little ω asymptotic notation

Definition : Let f(n) and g(n) are the two functions that maps + integers to + real numbers. We say that f(n) is ω(g(n)) (or f(n) ∊ ω(g(n))) if for any real constant c > 0, there exists an integer constant n0 ≥ 1 such that f(n) > c * g(n) ≥ 0 for every integer n ≥ n0.

f has a higher growth rate than g so difference between Ω and ω lies in between the definitions. In the case of Big Omega f(n)=Ω(g(n)) and the bound is 0<=cg(n)<=f(n), but in case of little omega, it is true for 0<=c*g(n)<f(n).

The relationship between Big Omega (Ω) and Little Omega (ω) is similar to that of Big-O and Little o except that now we are looking at the lower bounds. Little Omega (ω) is a rough estimate of the order of the growth whereas Big Omega (Ω) may represent the exact order of growth. We use ω notation to denote a lower bound that is not asymptotically tight.

And, f(n) ∊ ω(g(n)) if and only if g(n) ∊ o((f(n)).

In mathematical relation,

if $f(n) \in \omega(g(n))$ then,

lim f(n)/g(n) = ∞

n→∞

Example-1

$F(n) = n$

$G(n) = n^2$

$F(n) = \Omega(G(n))$

$n >= c.n^2 \ \forall n, n >= n_0$

Example-2

$f(n) = n-10$

$g(n) = n+10$

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

$n-10 >= c.n+10, \forall n, n >= n_0$

Note :

Even though n^2, n are lower bonus to f(n) you have to take the greatest lower bound only.

Shortcut :

If $f(n) = a_0 + a_1n + a_2n^2 + ... + a_mn^m$ ($a_m \neq 0$)

then $f(n) = \Omega(n^m)$

Theta (θ) :

f(n) is θ (g(n)) iff f(n) is 0 (g(n)) **and** f(n) is Ω (g(n)).

$f(n) = \theta(g(n)) \Leftrightarrow \exists\ C_1, C_2 > 0$ and $K_1, K_2 \geq 0$ and

$K_1 > 0$ such that

$C_1g(n) \leq +(n) \leq C_2.\ g(n)\ ; \forall\ n \geq K_1$

Example-1

If $f(n) = n^2 + n + 1$ then $f(n) = \theta(\ )$

→ $n^2 + n + 1 = 0(n^2)\ ; \forall\ n \geq 1$ and for $C_2 = 3$

&

$n^2 + n + 1 = \Omega(n^2)\ ; \forall n \geq 0$ and for $C_1 = 1$

$1\ n^2 \leq n^2 + n + 1 \leq 3\ .\ n^2\ ; \forall\ n \geq 1$

↑ ↑ ↑

C_1 C_2 K_1

$n^2 + n + 1 = \theta(n^2)$

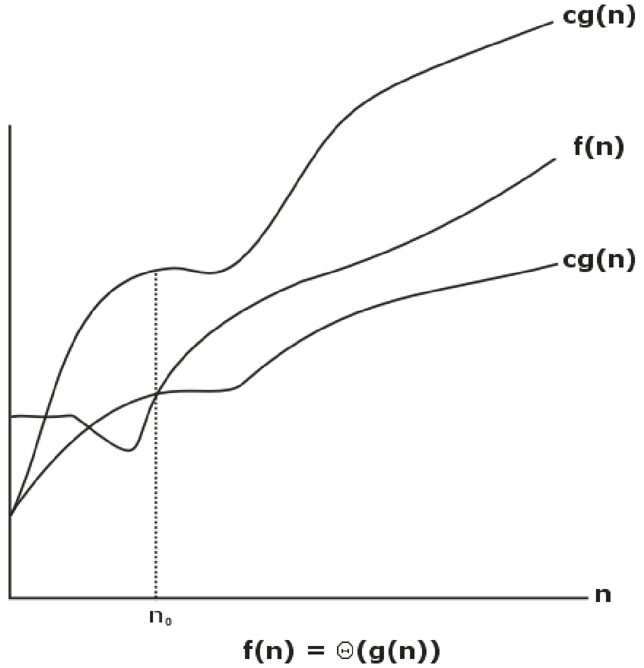


Fig. theta notation

Example-2

$F(n)=n-10$

$G(n)=n+10$

$F(n)<= c_1G(n)$

$n-10<= c_1(n+10)\ \forall n$, $n>=n_0$

$f(n)>= c_2g(n)$

$n-10 >=c_2(n+10)\ \forall n$, $n>=n_0$

$n-10 =\theta(n+10)$ for $c_1 =1$, $c_2 =1/2$, $n_0 =30$

Properties of Asymptotic:

1. **Reflexivity:**

   If f(n) is given then, **$f(n) = O(f(n))$**

   Example:

   If $f(n) = n^3 \Rightarrow O(n^3)$

   Similarly,

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

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

2. **Symmetry:**

   $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$

   Example:

   If $f(n) = n^2$ and $g(n) = n^2$ then $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$

3. **Transitivity:**

   $f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$

   Example:

   If $f(n) = n$, $g(n) = n^2$ and $h(n) = n^3$

   $\Rightarrow$ n is $O(n^2)$ and n^2 is $O(n^3)$ then n is $O(n^3)$

4. **Transpose Symmetry:**

   $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

   Example:

   If $f(n) = n$ and $g(n) = n^2$ then n is $O(n^2)$ and n^2 is $\Omega(n)$

5. **Since these properties hold for asymptotic notations, analogies can be drawn between functions f(n) and g(n) and two real numbers a and b.**

   - $g(n) = O(f(n))$ is similar to $a \le b$
   - $g(n) = \Omega(f(n))$ is similar to $a \ge b$
   - $g(n) = \Theta(f(n))$ is similar to $a = b$
   - $g(n) = o(f(n))$ is similar to $a < b$
   - $g(n) = \omega(f(n))$ is similar to $a > b$

6. **$\max(f(n), g(n)) = \Theta(f(n) + g(n))$**

7. **$O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$**

Difference Between Big oh, Big Omega and Big Theta :

S.NO.	BIG OH	BIG OMEGA	BIG THETA
1.	It is like <= rate of growth of an algorithm is less than or equal to a specific value	It is like >= rate of growth is greater than or equal to a specified value	It is like == meaning the rate of growth is equal to a specified value
2.	The upper bound of algorithm is represented by Big O notation. Only the above function is bounded by Big O. asymptotic upper bond is it given by Big O notation.	The algorithm lower bound is represented by Omega notation. The asymptotic lower bond is given by Omega notation	The bonding of function from above and below is represented by theta notation. The exact asymptotic behavior is done by this theta notation.
3.	Big oh (O) – Worst case	Big Omega (Ω) – Best case	Big Theta (Θ) – Average case
4.	Big-O is a measure of the longest amount of time it could possibly take for the algorithm to complete.	Big- Ω takes a small amount of time as compared to Big-O it could possibly take for the algorithm to complete.	Big- Θ is take very short amount of time as compare to Big-O and Big-? it could possibly take for the algorithm to complete.
5.	Mathematically – Big Oh is 0 <=f(n) <= c g(n) for all n>=n0	Mathematically – Big Omega is O<= C g(n) <= f(n) for all n>=n 0	Mathematically – Big Theta is O<=C 2 g(n)<=f(n)<=C 1 g(n) for n>=n 0

Recurrence Relation

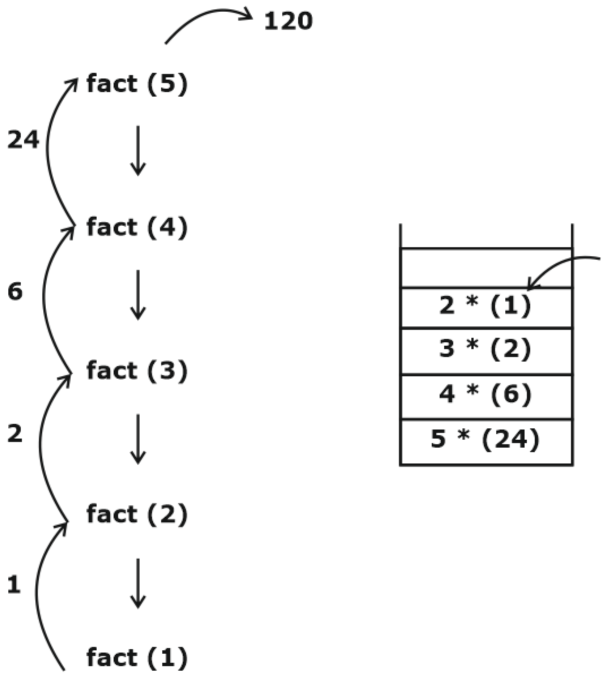
A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

A. Substitution Method
B. Recurrence Tree Method
C. Master's theorem

Recursive Algorithm :

```
int fact (int n)
{
      if (n = = 0 || n = = 1)
      return 1 ;                // Base condition
      else
      return n * fact (n – 1) ;
```

Here if we i/p n = 5 then,



Notes :

1. Time complexity of recursive algorithm = No. of function call
   ∴ Time complexity of fact (n) = 0(n)
2. Space complexity of recursive algorithm = Depth of recursive tree
   or
   = No. of activation record.
3. Space complexity of fact (n) = 0 (n – 1)
   = 0 (n)

Ex : Find time complexity of recursive tan^{n}. of Fibonacci sequence.

```
int fib (in + n)
{
      ⎧ it (n == 0)
B.C.  ⎨ return 0;
      ⎩ if (n == 1)
return 1
else
I.C. {return tib (n – 1) + fib (n – 2);
}
```

A. 0 (n^2)
B. 0 (2^n)
C. 0 (n)
D. 0 (n log n)

Ans. B

Here we take n = 5

Note :

For small values of n fib (n) = 0 (n^2) & for large values of n fib (n) = (2^n)

Since our analysis is only for large values of n. So time complexity of $\boxed{\text{fib}(n) = 0(2^n)}$

Note :

No. of tun^n. calls on i/p size n is Fibonacci sequence $= \underline{2.\text{fib}(n+1)-1}$

e.g. : n = 5, tun^n. call = 15

= 16 − 1

= 2 × 8 − 1

= 2f (6) − 1

Note :

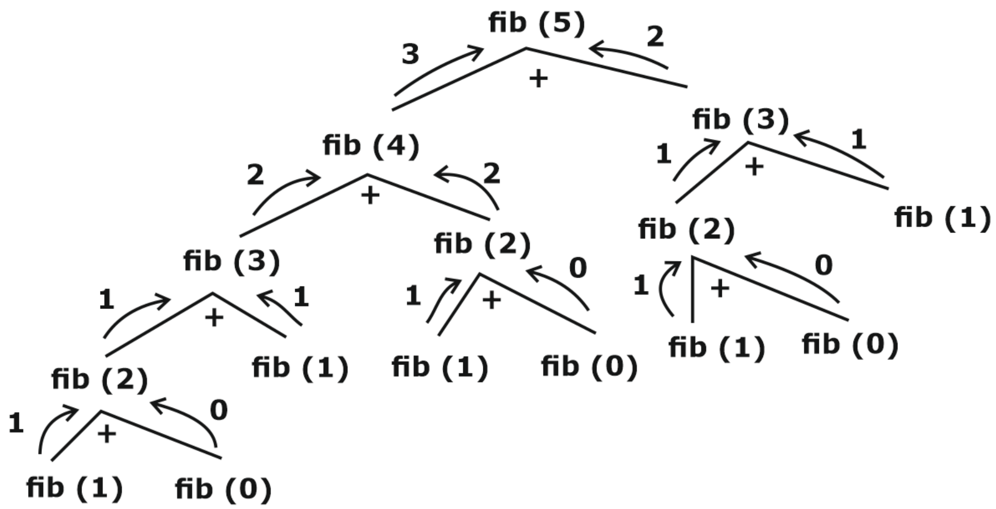
No. of addition perform on input size n in fib (n) = fib (n + 1) − 1

e.g.

n = 5 ,addition = 7

= 8 − 1

= fib (6) − 1



n	0	1	2	3	4	5	6	7
fib (n)	0	1	1	2	3	5	8	13

Function call = 15 (Total no. of nodes)

Total addition = 7

A. Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

Solve the equation by Substitution Method.

Example- 1

```
Long power (long x, long n)
{        if (n==0) return 1;
 if (n==1) return x;
if (( n % 2) == 0)
         return power (x*x, n/2);
else
return power (x*x, n/2) * x;
}
```

$T(0) = c_1$

$T(1) = c_2$

$T(n) = T(n/2) + c_3$

(Assume n is a power of 2)

$T(n) = T(n/2) + c_3$ ☐ $T(n/2) = T(n/4) + c_3$

$=T(n/4) + c_3 + c_3$

$= T(n/4)2c_3$ ☐ $T(n/4) = T(n/8) + c_3$

$=T(n/8) + c_3 + 2c_3$

$= T(n/8)+2c_3$ ☐ $T(n/8) = T(n/16) + c_3$

$= T(n/16) + c_3 + 3c_3$

$= T(n/32) + c_3 + 4c_3$

$= T(n/32) + 5c_3$

=…..

$= T(n/2k) + kc_3$

$T(0) = c_1$

$T(1) = c_2$

$T(n) = T(n / 2) + c_3$

$T(n) = T(n/2^k) + kc_3$

We want to get rid of $T(n/2^k)$. We get to a relation we can solve directly when we reach T(1)

$\lg n = k$

$T(n) = T(n/2^{\lg n}) + \lg n c_3$

$= T(1) + c_3 \lg n$

$= c_2 + c_3 \lg n$

$= \Theta(\lg n)$

Example 2- For the given program find the recurrence relation.

```
int mid=0;
int S[]={4,6,8,10,14,18,20};
binsearch(int low, int high, int S[], int x)
{      if low ≤ high
{   mid = (low + high) / 2 ;
                if x = S[mid]
                      return mid;
         }
           else if x < S[mid]
               return binsearch(n, low, mid-1, S, x);
           else
               return binsearch(n, mid+1, high, S, x)
        else
           return 0
  }
```

For binsearch(n), how many times is binsearch called in the worst case?

T(0) = 1

T(1) = 2

T(2) = T(1) + 1 = 3

T(4) = T(2) + 1 = 4

T(8) = T(4) + 1 = 4 + 1 = 5

So the recurrence relation can be written as-

o T(n) = T(n/2) + c ------- c is constant here.

Solving the recurrence relation is similar as in example 1.

Example 3- Consider the following code

```
fun(n)
{ if(n>1)
printf("%d", n);       ☐ this statement will take constant time
      return (fun(n-1));   ☐ recursive function, every time n value
   decremented by 1
}
```

i). Find the recurrence relation.

ii). Compute the time complexity.

Solution:

i) Recurrence Relation

T (n) = T (n-1) +1 and T (1) = θ (1).

ii) For time complexity-

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

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

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

$= T(n-4) + 4$

$= T(n-k) + k$

Where $k = n-1$

$T(n-k) = T(1) = \theta(1)$

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

$= 1+n-1=n=$

$T(n)=\theta(n)$.

Example 4- Consider the Recurrence

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

$T(n) = 2T(n-1)$ **if** $n>1$

Solution:

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

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

$= 4[2T(n-3)] = 2^3T(n-3)$

$= 8[2T(n-4)] = 2^4T(n-4)$

Repeat the procedure for i times

$T(n) = 2^i T(n-i)$

Put $n-i=1$ or $i= n-1$

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

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

$= 2^{n-1} = O(2^n)$

B. Recurrence Tree Method: In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels.

1. Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.
2. In general, we consider the second term in recurrence as root.
3. It is useful when the divide & Conquer algorithm is used.
4. It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represent the cost of a single subproblem.
5. We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.

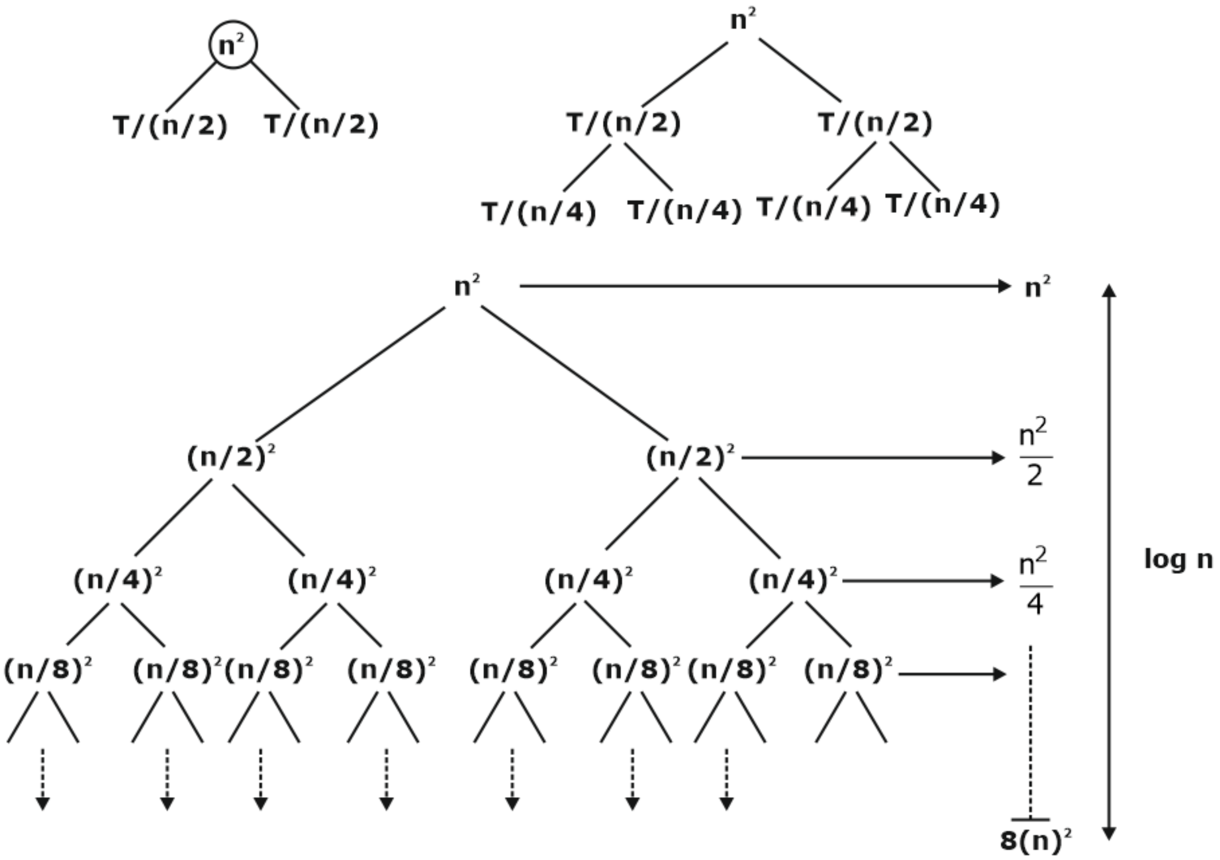
6. A Recursion Tree is best used to generate a good guess, which can be verified by the Substitution Method.

Example-1

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

We have to obtain the asymptotic bound using recursion tree method.

Solution: The Recursion tree for the above recurrence is



$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots \log n \text{ times.}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

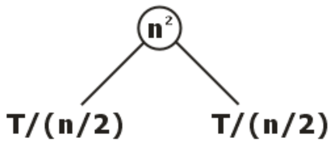
T(n) = $\theta(n^2)$

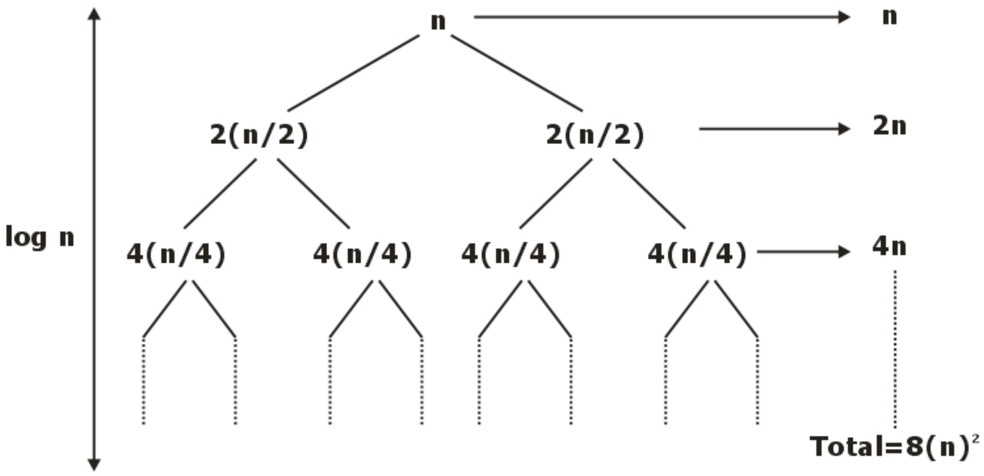
Example 2: Consider the following recurrence

T (n) = 4T (n/2) + n

Obtain the asymptotic bound using recursion tree method.

Solution: The recursion trees for the above recurrence





We have n + 2n + 4n + …… $\log_2 n$ times

= n (1 + 2 + 4 + …… $\log_2$ n times)

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

T (n) = θ (n^2)

C. Master Method

The Master Method is used for solving the following types of recurrence

T (n) = a T($\frac{n}{b}$) + f (n) with a≥1 and b≥1 be constant & f(n) be a function and $\frac{n}{b}$ can be interpreted as

Let T (n) is defined on non-negative integers by the recurrence.

T (n) = a T($\frac{n}{b}$)+ f (n)

In the function to the analysis of a recursive algorithm, the constants and function take on the following significance:

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- f (n) is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the subproblems.
- It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function.

$$T(n) = \{\Theta(n^{\ a}) \quad f(n) = O(n^{\ a-\varepsilon})\ \Theta(n^{a})\ f(n) = \Theta(n^{a})\ \Theta(f(n))\ f(n) = \Omega(n^{a+\varepsilon}) \quad AND \quad af\left(\frac{n}{b}\right) < cf(n) for\ large\ n \quad \}\varepsilon > 0\ c < 1$$

Case1: If f (n) = O (n $\log_b$ a — $\mathcal{E}$) for some constant ε >0, then it follows that:

T (n) = Θ (n $^{\log_b a}$)

Example-1

T (n) = 8 T (n/2) + 1000n^2 apply master theorem on it.

Solution:

Compare T (n) = 8 T (n/2) + 1000n^2 with

T (n) = a T(n/b) + f(n) with a>= 1 and b>1

a = 8, b=2, f (n) = 1000 n^2, $\log_b a = \log_2 8 = 3$

Put all the values in: f (n) = O(n $^{\log_b a - \varepsilon}$)

1000 n^2 = O ($n^{3-\varepsilon}$)

If we choose ε=1, we get: 1000 n^2 = O (n^{3-1}) = O (n^2)

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

T (n) = Θ (n $^{\log b\ a}$)

Therefore: T (n) = Θ (n^3)

Case 2: If it is true, for some constant k ≥ 0 that:

F (n) = Θ ($n^{\log_b a} \log^k$ n) then it follows that : T (n) Θ (n $^{\log_b a} \log^{k+1}$ n)

Example-2

T(n) = 2 T(n/2) + 10n, solve the recurrence by using the master method.

As compare the given problem with T (n) = a T(n/b) + f (n) with a ≥ 1 and b > 1 a = 2, b = 2, k = 0, f (n) = 10n, $\log_b a$

Put all the values in f (n) = Θ (n $^{\log_b a} \log^k$ n), we will get

10 n = Θ (n^1) = Θ (n) which is true.

Therefore,

T (n) = Θ (n $^{\log_b a} \log^{k+1}$ n)

= Θ (n log n)

Case 3: If it is true f(n) = Ω (n $\log_b$ a + $\mathcal{E}$) for some constant ε >0 and it also true that:

af ($\frac{n}{b}$) ≤ c f (n) for some constant c<1 for large value of n ,

then : T (n) = Θ((f (n))

Example-3

Solve the recurrence relation: $T\left(\frac{n}{2}\right) + n^2$

Compare the given problem with T (n) = a T (n/b) + f(n) with a ≥ 1 and b > 1 a = 2, b = 2, $f(n) = n^2$, $\log_b a = \log_2 2 = 1$

Put all the values in $f(n) = \Omega(n^{\log_b a + \varepsilon})$ .... (Eq. 1)

In we insert all the value in (Eq. 1),1 we will get

$n_2 = \Omega(n^{1+\varepsilon})$ put $\varepsilon = 1$, then the equality will hold.

$n_2 = \Omega(n^{1+1}) = \Omega(n^2)$

Now we will also check the second condition:

$$2\left(\frac{n}{2}\right)^2 \le cn^2 \Rightarrow \frac{1}{2}n^2 \le cn^2$$

If we will choose c = ½, it is true:

$$\frac{1}{2}n^2 \le \frac{1}{2}n^2 \; \forall n \ge 1$$

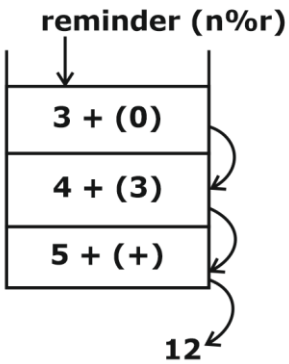
So it follows :T (n) = Θ ((f ((n))

$T(n) = \Theta(n^2)$

Ex:

Linked question

```
int too (int n, int r)
{
        if (n > 0)
        return ((n % r) + too (n/r, r))
        else
        return 0;
}
```



1. What is the return value of too (345, 10)

A. 345

B. 10

C. 12

D. 9

2. What is the return value of too (513, 2)

A. 2

B. 3

C. 6

D. 8

Ex :

```
int Do something ( int n)
    {
        (n ≤ 2)
        return 1 ;
        else
        return (Do some thing (floor (sq rt (n))) + n);
    }
```

1. Find time complexity :

A. O ($\log_2 n$)

B. O ($\log_2 \log_2 n$)

C. O ($n \log_2 n$)

D. O (n)

Case (i). Examples :

Ex :

$$T(n) = 16T\left(\frac{n}{4}\right) + n$$

$$= 16T\left(\frac{n}{4}\right) + \theta(n\log n)$$

a = 16, b = 4, k = 1, p = 0

From case (i) is $a > b^k$,

$16 > 4^1 \rightarrow$ yes

$\Rightarrow T(n) = \theta(n \log_b a) = \theta(n \log_4 16) = \theta(n^2)$

Ex :

$$T(n) = 4T\left(\frac{n}{2}\right) + \log n$$

$$= 4T\left(\frac{n}{2}\right) + \theta(n^{\circ}\log n)$$

a = 4, b = 2, k = 0, p = 1

Is $a > b^k$, $4 > 2^{\circ}$ (yes)

$T(n) = \theta(n \log_2 4) = \theta(n^2)$

Ex :

$$T(n) = \sqrt{2}T\left(\frac{n}{2}\right) + \log n$$

$$= \sqrt{2}T\left(\frac{n}{2}\right) + \theta(n^{\circ}\log n)$$

$a = \sqrt{2}$, b = 2, k = 0, p = 1

Is.

$a > b^k, \sqrt{2} > 2^{\circ}$ (yes)

$$\Rightarrow T(n) = \theta\left(n\log_2\sqrt{2}\right) = \theta\left(\sqrt{n}\right)$$

Ex. :

$$T(n) = 3T\left(\frac{n}{2}\right) + \frac{n}{2}$$

$$T(n) = 3T\left(\frac{n}{2}\right) + \theta(n\log^{\circ} n)$$

a = 3, b = 2, k = 1, p = 0

∴ Is $a > b^k$, 3 > 2 (yes)

⇒ $T(n) = \theta(n \log_2 3)$

Case (ii) examples :

Ex :

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

a = 2, b = 2, k = 2, p = 0

Is, $a < b^k$, $2 < 2^2$ → yes.

p = 0 (≥ 0) , case (ii) (a)

⇒ $T(n) = \theta(n^2)$

Ex :

$$T(n) = 6T\left(\frac{n}{3}\right) + n^2\log n$$

Is, $a < b^k$, $6 < 3^2$ → yes

p = 1 (≥ 0), case (ii) (a)

⇒ $T(n) = \theta(n^2 \log n)$

Ex :

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2$$

$$= 4T\left(\frac{n}{2}\right) + \theta\left(n^2\log^{\circ} r_1\right)$$

a = 4, b = 2, k = 2, p = 0

Is. a = b^k, 4 = 2^2 → yes

p = 0 (> - 1), case (iii) a

$\Rightarrow$ $T(n) = \theta\left(n^{\log_b a} \log^{p+1} n\right)$

$\Rightarrow$ $T(n) = \theta\left(n^{\log_2 4} \log^{0+1} n\right)$

= θ (n^2 log n)

Ex :

$$T(n) = 3T\left(\frac{n}{3}\right) + \frac{n}{2}$$

a = 3, b = 3, k = 1, p = 0

Is. a = b^k, 3 = $3^{'}$ → yes.

p = 0 (> - 1), case (iii) a

$\Rightarrow$ $T(n) = \theta\left(n^{\log_3 3} \log^{0+1} n\right)$

= θ (n log n)

Ex :

$$T(n) = 3T\left(\frac{n}{3}\right) + \frac{n}{\log n}$$

a = 3, b = 2, k = 1, p = - 1

Is. a = b^k, 3 = $3^{'}$ → yes.

P = - 1 (= -1) , case (iii) (b)

$\Rightarrow T(n) = \theta\left(n^{\log_b a} \log \log n\right)$

= θ (n log lon n)

Ex :

$$T(n) = \delta T\left(\frac{n}{2}\right) + \frac{n^3}{\log^2 n}\ d$$

a = δ, b = 2, k = 3, p = - 2

$$T(n) = \delta T\left(\frac{n}{2}\right) + n^3 \log^{-2} n$$

Is. a = b^k, δ = 2^3 → yes.

p = - 2 (< - 1) , case (iii) (c)

$\Rightarrow T(n) = \theta\left(n^{\log_b a}\right) = \theta\left(n^{\log_2 \delta}\right)$

= θ (n^3)

Ex:

$T(n) = 2T\left(\frac{n}{2}\right) + 0(n)$ which of the following FALSE.

A. T (n) = 0 (n^2)

B. T (n) = 0 (n log n)

C. T (n) θ (n log n)

D. T (n) = Ω (n^2)

SPECIAL CASES IN MASTER THEOREM :

1) $T(n) = 0.5 + \left(\frac{n}{2}\right) + n^2$

Since a = 0.5 (< 1)

So, we can't apply master theorem

2) $T(n) = 2^n T\left(\frac{n}{2}\right) + n^2$

Here, 'a' can't be a run^n.

So, we can't apply M.T.

3) $T(n) = 2T\left(\frac{n}{2}\right) + \boxed{-n^2}$

Negative fun^n. can't allow in M.T. So, we can't apply M.T.

4) $T(n) = 2T\left(\frac{n}{2}\right) + \boxed{2^n} \leftarrow$ exponential fun^n, then put is directly in answer.

$T(n) = 2T\left(\frac{n}{2}\right) + 2^n$

Ans : 0 (2^n)

5) $T(n) = 2T\left(\frac{n}{2}\right) + n!$

Ans : 0 (n !)

****