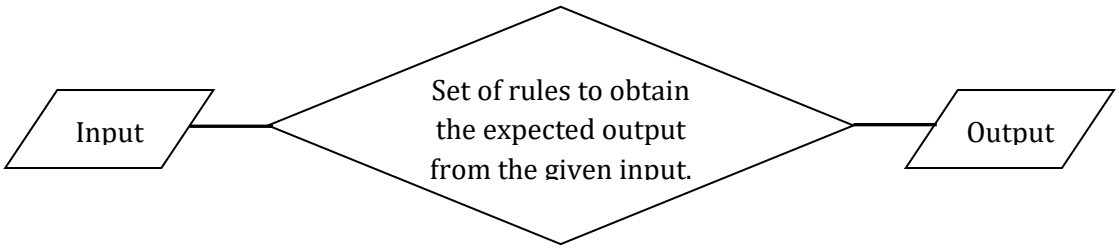
INTRODUCTION

An Algorithm is a finite set of steps to complete a task.

It is a blue print or, logic of a program.



Algorithm

Algorithm vs. Program

Given a problem to solve, the design phase produces an algorithm and the implementation phase then produces a program that expresses the designed algorithm.

Characteristics of an algorithm:

Every algorithm must be satisfied the following characteristics.

1. Input: Zero / more quantities are externally supplied.
2. Output: At least one quantity is produced.
3. Definiteness: Each step is clear and unambiguous. Each step must be clearly defined.
4. Finiteness: The algorithm must terminate after a finite time or, finite number of steps.
5. Correctness: Correct set of output values must be produced from the each set of inputs.
6. Effectiveness: Logic of an algorithm must be appropriate. One must be able to perform the steps of an algorithm without applying any intelligence.
7. Efficiency: An algorithm is efficient if it uses minimal running time and memory space as possible.
8. Feasibility: An algorithm must be simple, generic and practical so that it can be executed upon the available resources. It must not contain some future technology or anything.
9. Independent: It must be independent of language i.e. an algorithm should focus only on what are inputs, outputs and how to derive output.

Different Ways to Express an Algorithm:

1. Natural language
2. Flow charts
3. Pseudocode: A code that uses all the constructs of a programming language, but doesn't actually run anywhere.
4. Actual programming languages.

Design of Algorithm

Algorithm design refers to a method or process of solving a problem. There are two ways to design an algorithm. They are:

1. Top-down approach (Iterative algorithm): In the iterative algorithm, the function repeatedly runs until the condition is met or it fails.
2. Bottom-up approach (Recursive algorithm): In the recursive approach, the function calls itself until the condition is met.

For example, to find factorial of a given number n is given below:

1. Iterative :

```
Fact(n)
{
        for i = 1 to n
        fact = fact * i;
        return fact;
}
```

Here the factorial is calculated as $1 \times 2 \times 3 \times \ldots \times n$.

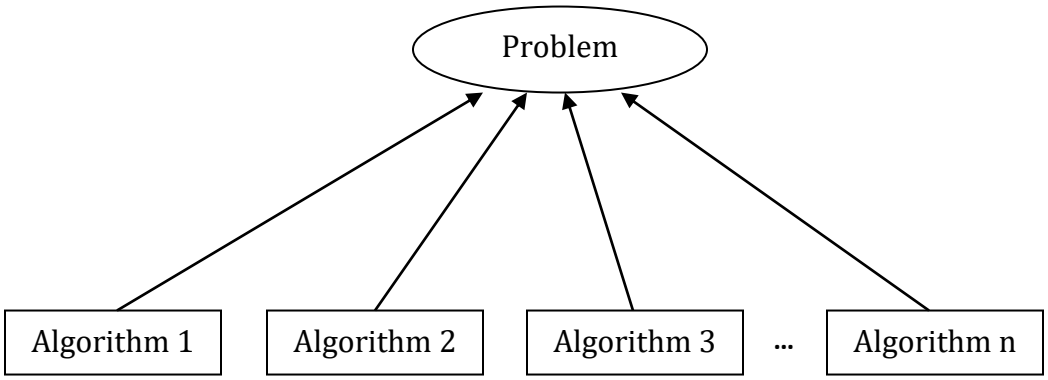
2. Recursive :

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

Here the factorial is calculated as $n \times (n - 1) \times \ldots \times 1$.

Analysis of Algorithm:

The analysis is a process of estimating the efficiency of an algorithm. An algorithm is efficient if it uses minimum memory and has less running time.



A problem can be solved in multiple ways. So we can say that we may have many algorithms to solve a particular problem. Performance analysis (Efficiency) helps us to select the best algorithm from many algorithms to solve a particular problem.

There are two basic parameters which are used to find the efficiency of an algorithm. They are:

- The amount of memory used (Space Complexity)
- The amount of compute time consumed on any CPU (Time Complexity)

Time complexity of an algorithm can be calculated by using two methods:

1. Priori Analysis: Analysis of an algorithm before its execution is called posterior analysis.
2. Posteriori Analysis: Analysis of an algorithm after its execution is called posterior analysis.

While analyzing the time complexity of an algorithm, we are usually concerned with priori analysis.

Difference between a priori analysis and posteriori analysis:

	Priori Analysis	***Posteriori Analysis***
1.	It is an absolute analysis	It is a relative analysis.
2.	It is done before execution of the algorithm	It is done after execution of the algorithm
3.	It gives approximate answer	It gives exact answer
4.	It is independent of language of compiler and hardware	It is dependent on language of compiler and type of hardware
5.	It uses asymptotic notations	It does not use asymptotic notations
6.	Time complexity is same for every system	Time complexity differ from system to system

Space Complexity & Time Complexity:

Space Complexity:

It measures the total amount of memory or storage space an algorithm needs to complete. It includes both auxiliary space and space used by the input. Auxiliary space is the temporary space or extra space used by an algorithm.

$$S(P) = C + S_P(I)$$

Where C is constant representing fixed space requirement

And $S_P(I)$ is variable space requirement which depends on instance characteristic I.

While analyzing the space complexity of an algorithm, we are usually concerned with only the variable space requirements.

Example:

```
Algorithm abcd(a, b, c, d)
{
        return a + b + b * c + (a + b - d) / (a + b)-d;
}
```

$C = 4;\ S_P(I) = 0;$ $S(abcd) = 4 + 0 = 4.$

Example:

Algorithm sum(n) // sum of n numbers

{

int i, sum=0;

for$(i = n;\ i >= 1;\ i--)$

$sum = sum + i;$

return sum;

}

$$C = 1;\ S_P(I) = 2; \qquad\qquad S(sum) = 1 + 2 = 3.$$

Time Complexity: It is the amount of time taken to run an algorithm.

Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input. Time complexity is not a measurement of how much time it takes to execute a particular algorithm because such factors as programming language, operating system, and processing power are also considered .Time complexity is a type of computational complexity that describes the time required to execute an algorithm.

Step Count Method:

This method is used to calculate Time and space complexity.

Procedure:

- There is no count for "{" and "}" .
- Each basic statement like 'assignment' and 'return' have a count of 1.
- If a basic statement is iterated, then multiply by the number of times the loop is run.
- The loop statement is iterated n times, it has a count of (n + 1). Here the loop runs n times for the true case and a check is performed for the loop exit (the false condition), hence the additional 1 in the count.

Example:

1. Sum of elements in an array

Statements	SC (Time Complexity)	SC (Space Complexity))
Algorithm Sum(a,n)	0	
{	0	
sum=0	1	1 word for sum
for i = 1 to n do	$n+1$	1 word each for i and n
sum = sum + a[i];	n	n words for the array a[]
return sum;	1	
}	0	
Total:	$2n+3$	$(n+3)$ words

2. Adding two matrices of order m and n

Statements	Step Count (Time Complexity)
Algorithm Add(a, b, c, m, n)	0
{	0
for i = 1 to m do	$m+1$
for j = 1 to n do	$m(n+1)$
c[i,j] = a[i,j] + b[i,j]	mn
}	0
Total:	$2mn + 2m + 1$

3. To find n^{th} number in Fibonacci series

Statements	Step Count (Time Complexity)
Algorithm Fibonacci (n)	0
{	0
if $n \leq 1$ *then*	1
write (n)	0
else	0
$f2 = 0;$	1
$f1 = 1;$	1
for $i = 2$ *to n do*	n
{	0
$f = f1 + f2;$	$n-1$
$f2 = f1;$	$n-1$
$f1 = f;$	$n-1$
}	0
write (f)	1
}	0
Total:	$4n + 1$

Rate of Growth:

Rate of growth is defined as the rate at which the running time of the algorithm is increased when the input size is increased. We will ignore the lower order terms, since the lower order terms are relatively insignificant for large input.

For example

Let us assume that you went to a shop to buy a car and a bicycle. If your friend sees you there and asks what you are buying then in general we say buying a car. This is because, cost of a car is too big compared to cost of cycle (approximating the cost of cycle to the cost of a car).

$$Total\ cost\ = cost\ of\ car\ +\ cost\ of\ bicycle$$

$$Total\ cost\ \approx cost\ of\ car$$

If time complexity of algorithm A: $100n\ + 1$

Time complexity of algorithm B: $n^2 + n + 1$ then

Input Size	Run Time of Algorithm A	Run Time of Algorithm B
10	1,001	111
100	10,001	10,101
1000	100,001	1,001,001
10000	1000,001	$> 10^{10}$

$$100n\ + 1 \approx n$$
$$n^2 + n + 1 \approx n^2$$

Hence, the growth rate of algorithm A is linear and Algorithm B is quadratic.

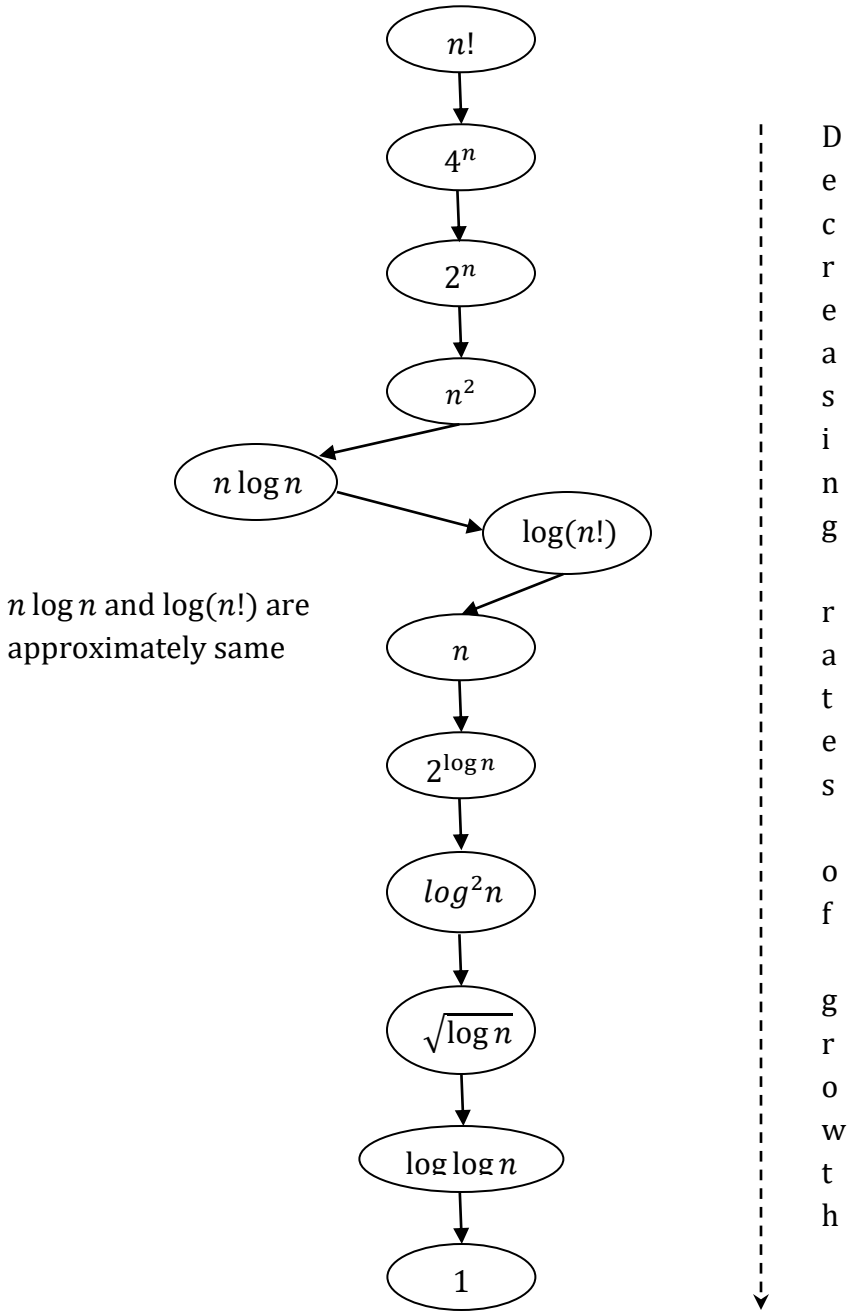
Time Complexity	**Name**	**Example**
1	Constant	Adding an element to the front of a linked list
log n	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in a unsorted array

nlog n	Linear Logarithmic	Sorting n items by 'Divide and Conquer'
n^2	Quadratic	Shortest path between 2 nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

The execution time for six of the typical functions is given below:

n	$log_2 n$	$n.log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296

The diagram below shows the relationship between different rates of growth:



Best, Worst, and Average Case Complexity:

In analyzing algorithms, we consider three types of time complexity:

1. **Best-case complexity:** This represents the minimum time required for an algorithm to complete when given the optimal input. It denotes an algorithm operating at its peak efficiency under ideal circumstances.
2. **Worst-case complexity:** This denotes the maximum time an algorithm will take to finish for any given input. It represents the scenario where the algorithm encounters the most unfavourable input.
3. **Average-case complexity:** This estimates the typical running time of an algorithm when averaged over all possible inputs. It provides a more realistic evaluation of an algorithm's performance.

Asymptotic Notations:

Asymptotic notations are the way to express time and space complexity. It represents the running time of an algorithm. If we have more than one algorithm with alternative steps then to choose among them, the algorithm with lesser complexity should be selected. To represents these complexities, asymptotic notations are used.
There are five asymptotic notations:

- Big Oh − $O(n)$
- Big Theta − $\Theta(n)$
- Big Omega − $\Omega(n)$
- Little oh − $o(n)$
- Little omega − $\omega(n)$

Big-oh (O) notation:

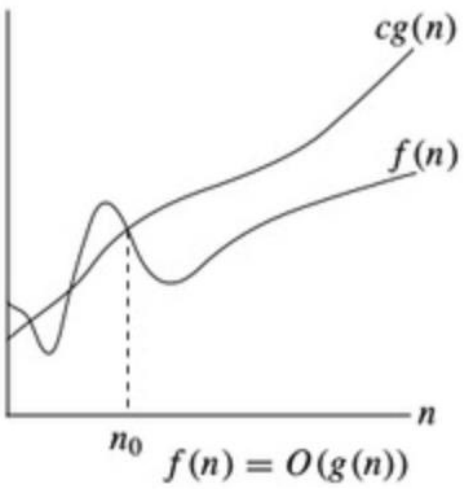
It is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time it could possibly take for the algorithm to complete.

Let $f(n)$ and $g(n)$ be asymptotically non-negative functions.

We say that $f(n)$ is in $O(g(n)$ if there exists a positive integer n_0 and a real positive constant $c > 0$, such that for all integers $n \geq n_0$,

$$0 \leq f(n) \leq cg(n)$$

$O(g(n)) = \{f(n)$: There exist positive constants c and n_0, such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$.



Guiding principles

Following are some principles that are normally followed while using big-oh (O) notations:

1. The coefficient of higher order terms should be ignored.
2. The lower order terms should be ignored.
3. The base should be changed by from one constant to another constant-i.e. the base value of the logarithm should be changed by only a constant factor.
   For example, if $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$ then,
   a) $T_1(n) + T_2(n) = \max\left[O\big(f(n)\big), O\big(g(n)\big)\right]$
   b) $T_1(n) \times T_2(n) = O(f(n) \times g(n))$
   c) $O\big(f(n)\big) \times C = O\big(f(n)\big)$ for any constant C.

Example 1:

$f(n) = 13$

$f(n) \leq 13 \times 1$

Here, $c = 13$ and $n_0 = 0$ and $g(n) = 1$

Hence, $f(n) = O(1)$

Example 2:

$f(n) = 3n + 5$

$3n + 5 \leq 3n + 5n$

$3n + 5 \leq 8n$

Here, $c = 8$ and $n_0 = 1$

Hence, $f(n) = O(n)$

Example 3:

$f(n) = 3n^2 + 5$

$3n^2 + 5 \leq 3n^2 + 5n$

$3n^2 + 5 \leq 3n^2 + 5n^2$

$3n^2 + 5 \leq 8n^2$

Here, $c = 8$ and $n_0 = 1$

Hence, $f(n) = O(n^2)$

Example 4:

$f(n) = 7n^2 + 5n$

$7n^2 + 5n \leq 7n^2 + 5n^2$

$7n^2 + 5n \leq 12n^2$

Here, $c = 12$ and $n_0 = 1$

Hence, $f(n) = O(n^2)$

Example 5:

$f(n) = 2^n + 6n^2 + 3n$

$2^n + 6n^2 + 3n \leq 2^n + 6n^2 + 3n^2 \leq 2^n + 6.2^n + 3.2^n \leq 10.2^n$

Here, $c = 10$ and $n_0 = 1$

Hence, $f(n) = O(2^n)$

Example 6:

Prove that $f(n) = n! = O(n^n)$

Proof:

$f(n) = n!$

$= n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$

$= (n^2 - n) \times (n-2) \times \ldots \times 2 \times 1$

$= (n^3 - 3n^2 + 2n) \times (n-3) \times \ldots 2 \times 1$

……

……

$\leq Cn^n$

$= O(n^n)$

Hence, $f(n) = O(n^n)$

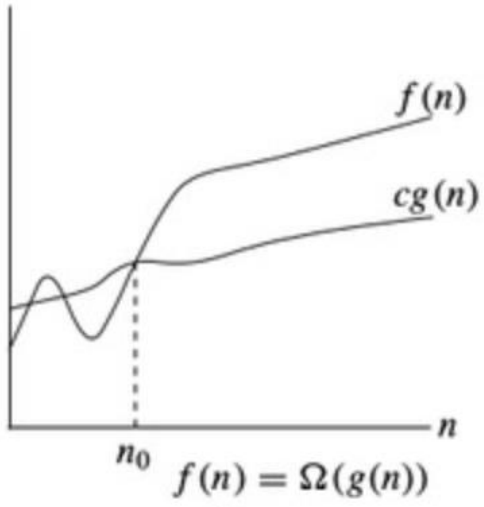
Big Omega (Ω) Notation:

Big Omega (Ω) is the method used for expressing the lower bound of an algorithm's running time. It is the measure of the smallest amount of time it could possibly take for the algorithm to complete.
Let $f\ (n)$ and $g(n)$ be asymptotically non-negative functions.

We say $that\ \ f\ (n)$ is Ω($g\ (n)$) if there exists a positive integer n_0 and a positive constant c such that for all integers $n \geq n_0$,

$$0 \leq cg(n) \leq f(n)$$

Ω($g\ (n)$) = {$f(n)$: There exist positive constants c and n_0, such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$}.



Example 1:

$f(n) = 13$
$f(n) \geq 12 \times 1$
where $c = 12$ and $n_0 = 0$
Hence, $f(n) = \Omega(1)$

Example 2:
$f(n) = 3n + 5$
$3n + 5 > 3n$
where $c = 3$ and $n_0 = 1$
Hence, , $f(n) = \Omega(n)$

Example 3:
$f(n) = 3n^2 + 5$
$3n^2 + 5 > 3n^2$
where $c = 3$ and $n_0 = 1$
Hence, $f(n) = \Omega(n^2)$

Example 4:
$f(n) = 7n^2 + 5n$
$7n^2 + 5n > 7n^2$
where $c = 7$ and $n_0 = 1$
Hence, $f(n) = \Omega(n^2)$

Example 5:
$f(n) = 2^n + 6n^2 + 3n$
$2^n + 6n^2 + 3n > 2^n$
where $c = 1$ and $n_0 = 1$
Hence, $f(n) = \Omega(2^n)$

Big Theta Notation:

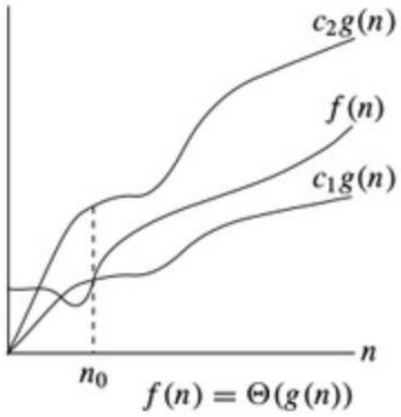
It is the method of expressing the tight bound of an algorithm's running time.

Let $f(n)$ and $g(n)$ be asymptotically non-negative functions.

We say *that* $f(n)$ is $\theta(g(n))$ if there exists a positive integer n_0 and positive constants c_1 and c_2 such that for all integers $n \geq n_0$,

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$\theta(\ g\ (\ n\ ))\ =\ \{f(n)$: There exist positive constants c_1 , c_2 and n_0, such that $0\ \leq c_1 g(n) \leq\ f(n)\ \leq c_2 g(n)$ for all $n\ \geq n_0\}$.



Example 1:
$f(n)\ =\ 123$
$122 \times 1\ \leq\ f(n)\ \leq 123 \times 1$
Here, $c_1 =\ 122, c_2\ =\ 123$ and $n_0\ =\ 0$
Hence, $f(n) =\ \theta(1)$

Example 2:
$f(n)\ =\ 3n\ +\ 5$
$3n\ <\ 3n\ +\ 5\ \leq\ 4n$
Here, $c_1\ =\ 3, c_2\ =\ 4$ and $n_0\ =\ 5$
Hence, $f(n)\ =\ \theta(n)$

Example 3:
$f(n)\ =\ 3n^2\ +\ 5$
$3n^2\ <\ 3n^2\ +\ 5 \leq 4n^2$
Here, $c_1\ =\ 3, c_2\ =\ 4$ and $n_0\ =\ 5$
Hence, $f(n)\ =\ \theta(n^2)$

Example 4:
$f(n)\ =\ 7n^2\ +\ 5n$
$7n^2\ <\ 7n^2 +\ 5n$ for all $n, c_1\ =\ 7$
Also, $7n^2\ +\ 5n\ \leq\ 8n^2$ for $n\ \geq\ n_0 =\ 5, c_2\ =\ 8$
Hence, $f(n)\ =\ \theta(n^2)$

Example 5:
$f(n)\ =\ 2^n\ +\ 6n^2\ +\ 3n$
$2^n\ <\ 2^n\ +\ 6n^2 +\ 3n\ <\ 2^n\ +\ 6n^2\ +\ 3n^2 <\ 2^n\ +\ 6.2^n\ +\ 3.2^n\ \leq\ 10.2^n$
Here, $c_1\ =\ 1, c_2\ =\ 10$ and $n_0\ =\ 1$
Hence, $f(n)\ =\ \theta(2^n\ )$

Example 7:

Prove that $\sum_{i=1}^{n} \log(i) = \theta(n \log n)$

Proof:

$$f(n) = \sum_{i=1}^{n} \log(i) = \log 1 + \log 2 + \cdots + \log n = 0 + \log 2 + \cdots + \log n$$

$$= \log(2 \times 3 \times 4 \times \ldots \times n) = \log n! = \theta(n \log n) \ (Since, \log n! = \theta(n \log n))$$

Example 8:

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

Proof:

$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \ for\ all\ n \geq 0$ ……(1)

$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \frac{1}{2}n\frac{1}{2}n \ for\ all\ n \geq 2$

$\frac{1}{2}n(n-1) \geq \frac{1}{2}n^2 - \frac{1}{4}n^2 \ for\ all\ n \geq 2$

$\frac{1}{2}n(n-1) \geq \frac{1}{4}n^2 \ for\ all\ n \geq 2$ ……(2)

$From\ 1\ and\ (2)\ \frac{1}{4}n^2 \leq \frac{1}{2}n(n-1) \leq \frac{1}{2}n^2 \ for\ all\ n \geq 2$

$Here\ c_1 = \frac{1}{4}, c_2 = \frac{1}{2}\ and\ n_0 = 2$

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

Example 9:

Prove that $\frac{1}{2}n^2 - 3n = \theta(n^2)$

Proof:

We need to find positive constants c_1 and c_2 such that

$$0 \leq c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

Dividing by n^2 we get

$$0 \leq c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

$c_1 \leq \frac{1}{2} - \frac{3}{n}$ holds for $n \geq 10$ and $c_1 = 1/5$

$\frac{1}{2} - \frac{3}{n} \leq c_2$ holds for $n \geq 10$ and $c_2 = 1$

Thus if $c_1 = 1/5, c_2 = 1$ and $n_0 = 10$,

$$0 \leq c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \text{for all } n \geq n_0$$

$\therefore \frac{1}{2}n^2 - 3n = \theta(n^2)$

Little-oh (o) Notation

The asymptotic upper bound provided by Big-oh(O) notation may or may not be asymptotically tight.

For Example, $2n^2 = O(n^2)$ is asymptotically tight but $2n = O(n^2)$ is not.

We use Little-Oh(o) notation to denote an upper bound that is not asymptotically tight.

$o(g(n)) = \{f(n)$: for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

For example, $2n = o(n^2)$ but $2n^2 \neq o(n^2)$

The main difference between Big-oh (O) notation and Little-oh (o) Notation is that in $f(n) = O(g(n))$, the bound $0 \leq f(n) \leq cg(n)$ holds for **som**e constant $c > 0$, but in $f(n) = o(g(n))$, the bound $0 \leq f(n) < cg(n)$ holds for **all** constant $c > 0$.

Little omega (ω) Notation:

The asymptotic lower bound provided by Big Omega (ω) notation may or may not be asymptotically tight. We use Little Omega (ω) notation to denote a lower bound that is not asymptotically tight.

$\omega(g(n)) = \{f(n)$: for any positive constant $c > 0$ there exists a constant $n_0 > 0$, such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.

For example,

$\frac{n^2}{2} = \omega(n)$, but $\frac{n^2}{2} \neq \omega(n^2)$

Example:

$an^2 + bn + c = \omega(1)$

$an^2 + bn + c = \omega(n)$

$an^2 + bn + c = \theta(n^2)$

$an^2 + bn + c = \Omega(1)$

$an^2 + bn + c = \Omega(n)$

$an^2 + bn + c = O(n^2)$

$an^2 + bn + c = O(n^6)$

$an^2 + bn + c = O(n^{50})$

$an^2 + bn + c = o(n^{19})$

Also,

$an^2 + bn + c \neq o(n^2)$

$an^2 + bn + c \neq \omega(n^{19})$

$an^2 + bn + c \neq O(n)$

Some more incorrect bounds are as follows:

$7n + 5 \neq O(1)$

$2n + 3 \neq O(1)$

$3n^2 + 16n + 2 \neq O(n)$

$5n^3 + n^2 + 3n + 2 \neq O(n^2)$

$7n + 5 \neq \Omega(n^2)$

$2n + 3 \neq \Omega(n^3)$

$10n^2 + 7 \neq \Omega(n^4)$

$7n + 5 \neq \theta(n^2)$

$2n^2 + 3 \neq \theta(n^3)$

Some more loose bounds are as follows:

$2n + 3 = O(n^2)$

$4n^2 + 5n + 6 = O(n^4)$

$5n^2 + 3 = \Omega(1)$

$2n^3 + 3n^2 + 2 = \Omega(n^2)$

Some correct bounds are as follows:

$2n + 8 = O(n)$

$2n + 8 = O(n^2)$

$2n + 8 = \theta(n)$

$2n + 8 = \Omega(n)$

$2n + 8 = o(n^2)$

$2n + 8 \neq o(n)$

$2n + 8 \neq \omega(n)$

$4n^2 + 3n + 9 = O(n^2)$

$4n^2 + 3n + 9 = \Omega(n^2)$

$4n^2 + 3n + 9 = \theta(n^2)$

$4n^2 + 3n + 9 = o(n^3)$

$4n^2 + 3n + 9 \neq o(n^2)$

$4n2 + 3n + 9 \neq \omega(n^2)$

Asymptotic comparison of functions:

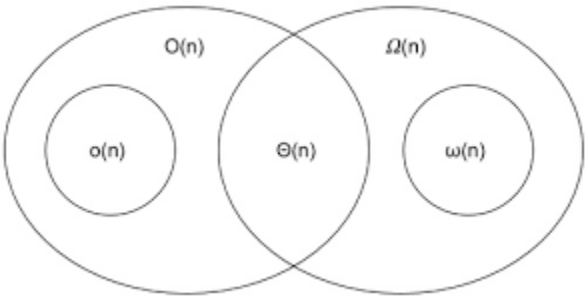
- $f(n) = O(g(n)) \approx f \leq g$
- $f(n) = \theta(g(n)) \approx f = g$
- $f(n) = \Omega(g(n)) \approx f \geq g$
- $f(n) = o(g(n)) \approx f < g$

- $f(n) = \omega(g(n)) \approx f > g$

N.B.:

- $f(n)$ is asymptotically smaller than $g(n)$ if $f(n) = o(g(n))$.
- $f(n)$ is asymptotically larger than $g(n)$ if $f(n) = \omega(g(n))$.

Properties of Asymptotic Notations



Reflexivity

- $f(n) = O(f(n))$
- $f(n) = \theta(f(n))$
- $f(n) = \Omega(f(n))$

Symmetry

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

Transitivity

- $f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- $f(n) = \theta(g(n))$ and $g(n) = \theta(h(n)) \Rightarrow f(n) = \theta(h(n))$
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
- $f(n) = o(g(n))$ and $g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$
- $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$

Transpose Symmetry

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

$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$

Comparing the growth rate of two functions using limits

Rules:

1. $f(n) = O(g(n))\ if\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = c < \infty\ where\ c \in \mathbb{R}$ (Can be zero)
2. $f(n) = \Omega(g(n))\ if\ \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$ (Can be ∞)
3. $f(n) = \theta(g(n))\ if\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = c, c \in \mathbb{R}^+$
4. $f(n) = o(g(n))\ if\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
5. $f(n) = \omega(g(n))\ if\ \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Example:

Compare the growth rate of two functions $f(n) = n^2$ and $g(n) = 2^n$using limits.

Solution:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^2}{2^n} = \lim_{n\to\infty} \frac{2n}{2^n \ln 2} = \lim_{n\to\infty} \frac{2}{2^n (\ln 2)^2} = 0$$

Since, the limit is equal to 0, the growth rate of $g(n) = 2^n$ is greater than the growth rate of $f(n) = n^2$.

Example:

Compare the growth rate of two functions $f(n) = 4n^3 + 2n + 4$ and $g(n) = 2n^3 - 100n$ using limits.

Solution:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{4n^3 + 2n + 4}{2n^3 - 100n} = \lim_{n\to\infty} \frac{\frac{4n^3 + 2n + 4}{n^3}}{\frac{2n^3 - 100n}{n^3}} = \lim_{n\to\infty} \frac{4 + \frac{2}{n^2} + \frac{4}{n^3}}{2 - \frac{100}{n^2}}$$

$$= \frac{4 + 0 + 0}{2 - 0} = 2$$

Since, the limit is equal to 2, the growth rate of $g(n) = 2n^3 - 100$ is 2 times the growth rate of $f(n) = 4n^3 + 2n + 4$ at sufficiently large n.

Comparing the growth rate of two functions using logarithms:

Rules:

1. $log\ ab = log\ a + log\ b$
2. $\log \frac{a}{b} = \log a - \log b$
3. $\log a^b = b \log a$
4. $a^{\log_c b} = b^{\log_c a}$
5. $a^b = n$ then $b = \log_a n$

Example:

Compare the growth rate of two functions $f(n) = n^2$ and $g(n) = n^3$using logarithms.

Solution:

$f(n) = n^2$ and $g(n) = n^3$

Taking Logarithm of both functions

f(n) $= \log n^2$ and $g(n) = \log n^3$

f(n) $= 2 \log n$ and $g(n) = 3 \log n$

$2 \log n < 3 \log n$

Hence, the growth rate of $g(n) = n^3$ is higher than the growth rate of $f(n) = n^2$

Example:

Compare the growth rate of two functions $f(n) = n^2 \log n$ and $g(n) = n\ (\log n)^{10}$using logarithms.

Solution:

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

Taking Logarithm of both functions

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

f(n) $= \log n^2 + \log\log n$ and $g(n) = \log n + \log(\log n)^{10}$

f(n) $= 2 \log n + \log\log n$ and $g(n) = \log n + 10 \log\log n$

$$n^2 \log n > n\ (\log n)^{10}$$

Hence, the growth rate of $f(n)$ is higher than the growth rate of $g(n)$

Example:

Compare the growth rate of two functions $f(n) = 3n^{\sqrt{n}}$ and $g(n) = 2^{\sqrt{n}\log n}$ using logarithms.

Solution:

$f(n) = 3n^{\sqrt{n}}$ and $g(n) = 2^{\sqrt{n}\log n}$

Taking Logarithm of both functions

$$\text{f(n)} = 3n^{\sqrt{n}} \quad and\ g(n) = 2^{\log n^{\sqrt{n}}} (Rule - 3)$$

$$\text{f(n)} = 3n^{\sqrt{n}} \quad and\ g(n) = \left(n^{\sqrt{n}}\right)^{\log 2} (Rule - 4)$$

$$\text{f(n)} = 3n^{\sqrt{n}} \quad and\ g(n) = n^{\sqrt{n}}$$

Hence, $3n^{\sqrt{n}} > 2^{\sqrt{n}\log n}$

Example:

Show that $\log x = o(x)$

Proof:

$$\lim_{n\to\infty} \frac{\log x}{x} = \lim_{n\to\infty} \frac{\log x}{x} = \frac{1}{\lim_{n\to\infty} \frac{x}{1}} = \lim_{n\to\infty} \frac{1}{x} = 0$$

$$\therefore \log x = o(x)$$

Example:

Arrange the following list of functions in ascending order of growth rate.

$$\log n, \quad n!, \quad n^2/\log n, \quad n.2^n, \quad (\log n)^{\log n}, \quad 3^n$$

Solution:

$\log n, \quad (\log n)^{\log n}, \quad n^2/\log n, \quad n.2^n, \quad 3^n, \quad n!$

Example:

Arrange the following list of functions in ascending order of growth rate,

$$2^{\sqrt{\log n}}, 2^n, n^{4/3}, n(\log n)^3, n^{\log n}, 2^{2^n}, 2^{n^2}$$

Solution:

If $f(n) = O(g(n))$, then $g(n)$ follows $f(n)$.

$$n(\log n)^3 = O(n^{4/3})$$

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

$$2^{n^2} = O\left(2^{2^n}\right)$$

$$2^{\sqrt{\log n}} = O\left(2^{\log n}\right) \Rightarrow 2^{\sqrt{\log n}} = O(n)$$

$$n^{4/3} = O\left(n^{\log n}\right) \ since\ 4/3 = O(\log n)$$

$$n^{\log n} = 2^{\log\left(n^{\log n}\right)} = 2^{\log n \log n} = 2^{(\log n)^2} = O(2^n)$$

Therefore the correct order is $2^{\sqrt{\log n}}, n(\log n)^3, n^{4/3}, n^{\log n}, 2^n, 2^{n^2}, 2^{2^n}$

RECURRENCES

A *recurrence relation* or simply recurrence for a sequence a_0, a_1, … is an equation that relates a_n to some of the terms a_0, a_1, …, a_{n-1}.

Initial conditions or, base condition for the sequence a_0, a_1, … are explicitly given values for a finite number of the terms of the sequence.

For example, to compute factorial of a number recursively we use the following algorithm:

```
factorial(n)
{
        if (n === 1)
                return 1;
        else
                return n * factorial(n - 1);
}
```

Time Complexity of the above recursive algorithm is

$T(n) = T(n-1) + 1$
$= [T(n-2) + 1] + 1 = T(n-2) + 2$
$= [T(n-3) + 1] + 2 = T(n-3) + 3$
$= \cdots \ldots \ldots \ldots \ldots \ldots$
$= T(n-n) + n = n$

Different techniques to solve recurrence relations:

1. Iterative Method
2. Substitution method
3. The recursion tree method
4. Master's theorem

Iterative Method:

This method is also called repeated substitution method. In this method, we keep substituting the smaller terms again and again until we reach the base condition

Example:

Solve the following recurrence equation:

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n-1) + 1, & n \geq 2 \end{cases}$$

Solution:

$\Rightarrow T(n) = 1 + 2 \cdot T(n-1)$

$\Rightarrow T(n) = 1 + 2\big(1 + 2T(n-2)\big)$

$\Rightarrow \mathrm{T}(n) = 1 + 2 + 4 \cdot T(n-2)$

$\Rightarrow \mathrm{T}(n) = 1 + 2 + 4 \cdot (1 + 2T(n-3))$

$\Rightarrow T(n) = 1 + 2 + 4 + 8 \cdot T(n-3)$

$\Rightarrow T(n) = 1 + 2 + 2^2 + 2^3 \cdot T(n-3)$

… … … … … … … … … … … … … … … …

… … … … … … … … … … … … … … … ….

$\Rightarrow T(n) = 1 + 2 + 2^2 + 2^3 + \cdots + 2^{n-1} \cdot T(1)$

$\Rightarrow T(n) = 1 + 2 + 2^2 + 2^3 + \cdots + 2^{n-1}$ (Since, $T(1) = 1$)

$\Rightarrow T(n) = \dfrac{2^n - 1}{2 - 1} (Geometric\ Sum\ Formula)$

$\Rightarrow T(n) = 2^n - 1$

Note: The repeated substitution method may not always be successful.

Example:

Solve the following recurrence relation:

$$T(n) = \begin{cases} 0, & n = 1 \\ 2T(n/2) + 1, & n \geq 2 \end{cases}$$

Solution:

$T(n) = 2T\left(\dfrac{n}{2}\right) + 1$

$= 1 + 2\left[1 + 2T\left({}^{n}/_{4}\right)\right]$

$= 1 + 2 + 4T\left({}^{n}/_{4}\right)$

$= 1 + 2 + 4 + 8\, T\left({}^{n}/_{8}\right)$

… … … … … … … … … … …

… … … … … … … … … … ….

$= 1 + 2 + 4 + \cdots + 2^{k-1} + 2^k T(\frac{n}{2^k})$ (where $\frac{n}{2^k} = 1 \Rightarrow n = 2^k$)

$= 1 + 2 + 4 + \cdots + 2^{k-1} + 2^k T(1)$

$= 1 + 2 + 4 + \cdots + 2^{k-1} + 0$

$= 1 + 2 + 4 + \cdots + 2^{k-1}$

$= \frac{2^k - 1}{2-1} = 2^k - 1 = n - 1$(Geometric Sum formula)

Example:

Solve the following recurrence relation to compute time complexity if Insertion Sort

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n - 1, & n \geq 2 \end{cases}$$

Solution:

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

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

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

… … … … … … … … … … …

… … … … … … … … … … …

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

$= (n-1) + (n-2) + (n-3) + \cdots + 1 + 0$ [Since, $T(1) = 0$]

$= \frac{(n-1)n}{2}$ (Arithmetic sum formula)

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

Example:

$$T(n) = \begin{cases} 1, & n = 1 \\ 3T\left(\frac{n}{4}\right) + n, & n \geq 2 \end{cases}$$

Solution:

$$T(n) = n + 3T\left(\frac{n}{4}\right) = 3n + \left(\frac{n}{4} + 3T\left(\frac{n}{16}\right)\right) + n = n + 3\left(\frac{n}{4}\right) + 9T\left(\frac{n}{16}\right)$$

$$= n + 3\left(\frac{n}{4}\right) + 9\left(\frac{n}{16}\right) + 27T\left(\frac{n}{64}\right) = \left(\frac{3}{4}\right)^0 n + \left(\frac{3}{4}\right)^1 n + \left(\frac{3}{4}\right)^2 n + \left(\frac{3}{4}\right)^3 n + \cdots$$

$$= \left[\left(\frac{3}{4}\right)^0 + \left(\frac{3}{4}\right)^1 + \left(\frac{3}{4}\right)^2 + \left(\frac{3}{4}\right)^3 + \cdots\right] n = \frac{1}{1 - 3/4} n = 4n$$

The substitution Method:

It involves guessing the form of the solution and then using mathematical induction find the constants and show that the guess is correct.

1. Guess the solution
2. Prove it correct by mathematical induction.

Example:

Solve the recurrence equation:

$$T(n) = \begin{cases} 1, & n = 1 \\ 2T(n-1) + 1, & n \geq 2 \end{cases}$$

Solution:

Suppose we guess that the solution to be exponential.

***Guess**: $T(n) = A\ 2^n + B$*

Induction Proof:

Basis of Induction:

LHS: $T(1) = 1$(*From the initial condition*)

RHS: $A2^1 + B = 2A + B$

we need $2A + B = 1$ ……..(1)

Induction Hypothesis:

Assume that $T(k) = A\ 2^k + B$ for $k \geq 1$

Induction Step:

To prove the solution is also correct for $n = k + 1$:

$i.e.$To Show $T(K+1) = A\ 2^{k+1} + B$

LHS: $T(k+1) = 2T(k) + 1$ $(from\ the\ recurrence\ equation)$

$= 2[A\ 2^k + B] + 1$ (*By Inductive hypothesis*)

$= A\ 2^{k+1} + (2B + 1)$

$= A\ 2^{k+1} + B$ … … … RHS

we need $2B + 1 = B$ … … … … … … … … (2)

Solving (1) and (2) $\begin{cases} 2A + B = 1 \\ 2B + 1 = B \end{cases}$

We get $B = -1$ and $A = 1$

Hence, $T(n) = A\ 2^n + B = 2^n - 1$

Example:

Solve the following recurrence relation:

$$T(n) = \begin{cases} 0, & n = 1 \\ T(n-1) + n - 1, & n \geq 2 \end{cases}$$

Solution:

Suppose we guess that the solution as $O(n^2)$

Guess: $T(n) = An^2 + Bn + C$

Basis of Induction:

LHS: $T(1) = 0$(*From the initial condition*)

RHS: $An^2 + Bn + C = A + B + C$

So we need $A + B + C = 0$ … … … … … … … … … … … (1).

Induction Hypothesis:

Assume that $T(k-1) = A\ (k-1)^2 + B(k-1) + C$ for $k \geq 1$

Induction Step:

To prove the solution is also correct for $n = k$

$i.e.$To Show $T(k) = A\,k^2 + Bk + C$

LHS.: $T(k) = T(k-1) + (k-1)$ (from the recurrence equation)

$$= A\,(k-1)^2 + B(k-1) + C + (k-1)$$

$$= A\,k^2 - 2Ak + A + Bk - B + C + k - 1$$

$$= A\,k^2 + (-2A + B + 1)k + (A - B + C - 1)$$

$$= A\,k^2 + Bk + C$$

So we need $-2A + B + 1 = B$ … … … … … … … … … (2)

And $A - B + C - 1 = C$ … … … … … … … … … … … … (3)

Solving (1), (2) and (3) $\begin{cases} A + B + C = 0 \\ -2A + B + 1 = B \\ A - B + C - 1 = C \end{cases}$

$$A = \frac{1}{2}, B = -\frac{1}{2}, C = 0.$$

Therefore, $T(n) = \frac{n^2}{2} - \frac{n}{2}$

Example:

Show that $T(n) = O(n \lg n)$ is the solution of the following recurrence relation by substitution method.

$$T(n) = \begin{cases} 0, & n = 1 \\ 2T(n/2) + n, & n > 2 \end{cases}$$

Proof:

We have to prove that $T(n) \leq cn \lg n$ for an appropriate choice of the constant $c > 0$.

Basis of Induction:

Here our guess does not hold for $n = 1$ because $1 \not\leq c.1\,log1$

Now for $n = 2$

LHS: $T(2) = 2T\left(\frac{n}{2}\right) + n = 2T(1) + 2 = 2 \times 0 + 2 = 2$(*From the initial condition*)

RHS: $c \times 2 \lg 2 = 2c$

Hence, $T(2) \leq c \times 2 \lg 2$ for $c \geq 1$

Induction Hypothesis:

Assume that $T(n) \leq cn \lg n$ for $2 \leq n < k$

Induction Step:

To prove the solution is also correct for $n = k$

LHS.: $T(k) = 2T(k/2) + k$(from the recurrence equation)

$$\leq 2c\ (k/2) \lg(k/2) + k$$

$$= c\ k \lg(k/2) + k$$

$$= c\ k \lg k - ck \lg 2 + k$$

$$= c\ k \lg k - k\ (\text{c} \lg 2 - 1)$$

$$= c\ k \lg k - k(c - 1)$$

$$= c\ k \lg k - ck + k$$

$$\leq ck \lg k \text{ for all } c \geq 1$$

Therefore, $T(n) = O(n \lg n)$

Example:

Show that $T(n) = O(\log n)$ is the solution of the following recurrence relation:

$$T(n) = T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + 1$$

Solution:

We have to prove that $T(n) \leq c \log n$ for an appropriate choice of the constant $c > 0$.

Inductive hypothesis:

Assume that $T(n) \leq c \lg n$ for $2 \leq n < k$

Inductive Step:

To prove the solution is correct for $n = k$

LHS.: $T(k) = T\left(\left\lfloor\frac{k}{2}\right\rfloor\right) + 1$ (By the recurrence relation)

$\leq T\left(\frac{k}{2}\right) + 1 \leq c\log\left(\frac{k}{2}\right) + 1$ (By Inductive Hypothesis)

$= c\log k - c\log_2 2 + 1 = c\log k - c + 1 \leq c\log k$ for $c \geq 1$

Therefore, $T(n) = O(\log n)$

Example:

Show that $T(n) = O(n\ \log n)$ is the solution of the following recurrence relation:

$$T(n) = 2T\left(\left\lfloor\frac{n}{2}\right\rfloor + 16\right) + n$$

Solution:

We have to prove that $T(n) \leq$ cn $\log n$ for an appropriate choice of the constant $c > 0$.

Inductive hypothesis:

Assume that $T(n) \leq cn \lg n$ for $2 \leq n < k$

Inductive Step:

To prove the solution is correct for $n = k$

LHS: $T(k) = 2T\left(\left\lfloor\frac{k}{2}\right\rfloor + 16\right) + k$ (By the recurrence relation)

$\leq 2\left[c\left(\left\lfloor\frac{k}{2}\right\rfloor + 16\right)\log\left(\left\lfloor\frac{k}{2}\right\rfloor + 16\right)\right] + k$(By inductive hypothesis)

$$\leq 2\left[c\left(\frac{k}{2} + 16\right)\log\left(\frac{k}{2} + 16\right)\right] + k = c(k+32)\log\left(\frac{k+32}{2}\right) + k \leq ck\log\left(\frac{k}{2}\right) \leq ck\log k$$

Recursion Tree Method:

It is a pictorial representation of a given recurrence relation, which shows how Recurrence is divided till Boundary conditions. It is useful when the divide & Conquer algorithm is used.

Basic Steps to solve recurrence relation using recursion tree method:

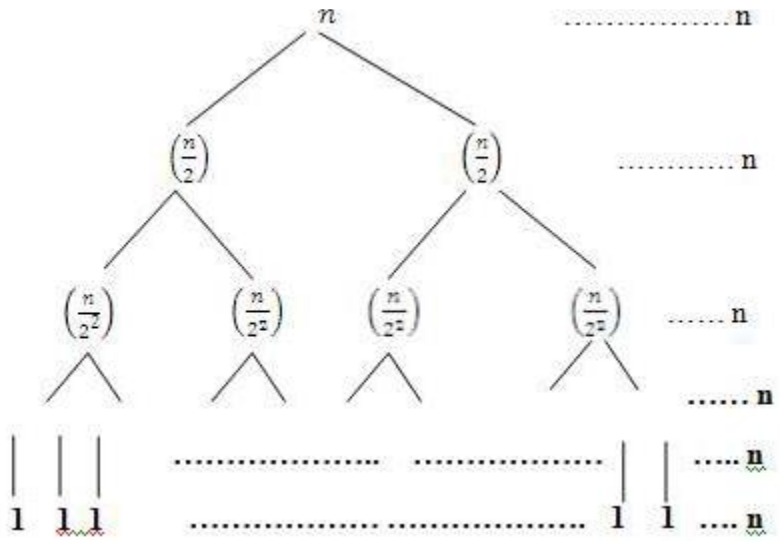
1. Draw a recursive tree for given recurrence relation. In general, we consider the second term in recurrence as root of the tree.
2. Calculate cost of each level
3. Determine the total number of levels in the recursion tree
4. Add cost of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation.

Example:

Solve the recurrence $T(n) = 2T(n/2) + n$ using recursion tree method.

Solution:

Step1: First you make a recursion tree of a given recurrence, where n is the root.



The given recurrence relation shows-

- A problem of size n will get divided into 2 sub-problems of size n/2.
- Then, each sub-problem of size n/2 will get divided into 2 sub-problems of size n/4 and so on.
- At the bottom most layer, the size of sub-problems will reduce to 1.

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-

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

Then-

$$\frac{n}{2^h} = 1$$

$$\Rightarrow n = 2^h$$

Taking log on both sides, we get $h\ log2 = logn$

$$\Rightarrow h = \log_2 n$$

$\therefore$ Total number of levels in the recursion tree $= \log_2 n + 1$

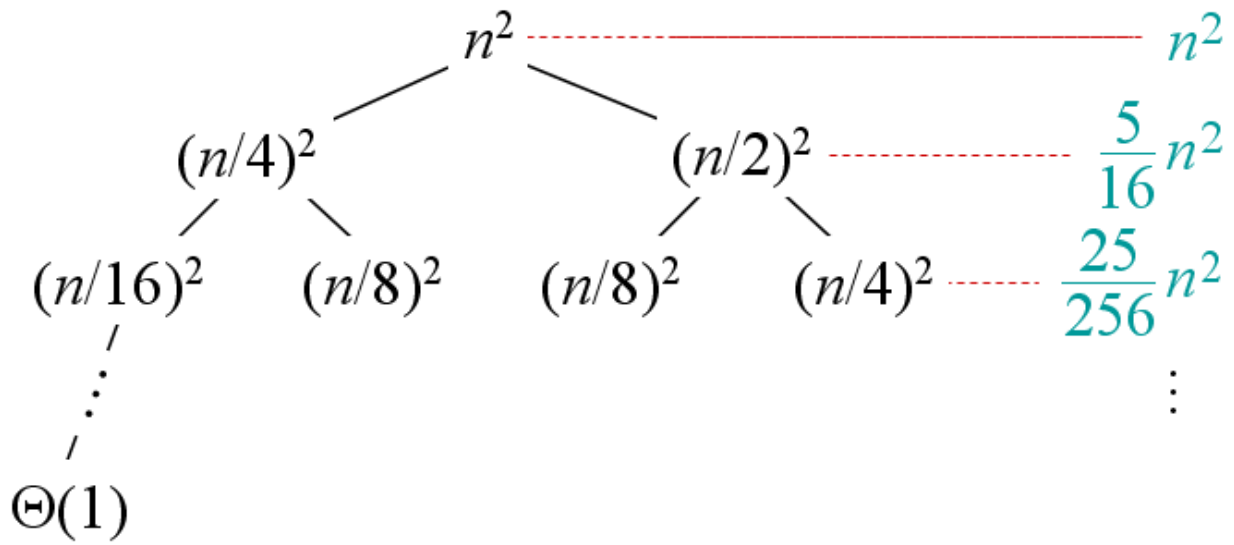
Step 6: Add costs of all the levels of the recursion tree and simplify the expression so obtained in terms of asymptotic notation-

$$T(n) = n + n + \cdots n = (\log_2 n + 1)\, n = \boldsymbol{\theta}\ (\boldsymbol{n} \log_2 n)$$

Example

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

Solution:



$$T(n) = n^2 + \frac{5}{16}n^2 + \left(\frac{5}{16}\right)^2 n^2 + \left(\frac{5}{16}\right)^3 + \cdots \ldots \ldots = n^2\left(1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \cdots ..\right)$$

By geometric series, $1 + x + x^2 \ldots . + x^n = \frac{1-x^{n+1}}{1-x}$ for $x \neq 1$

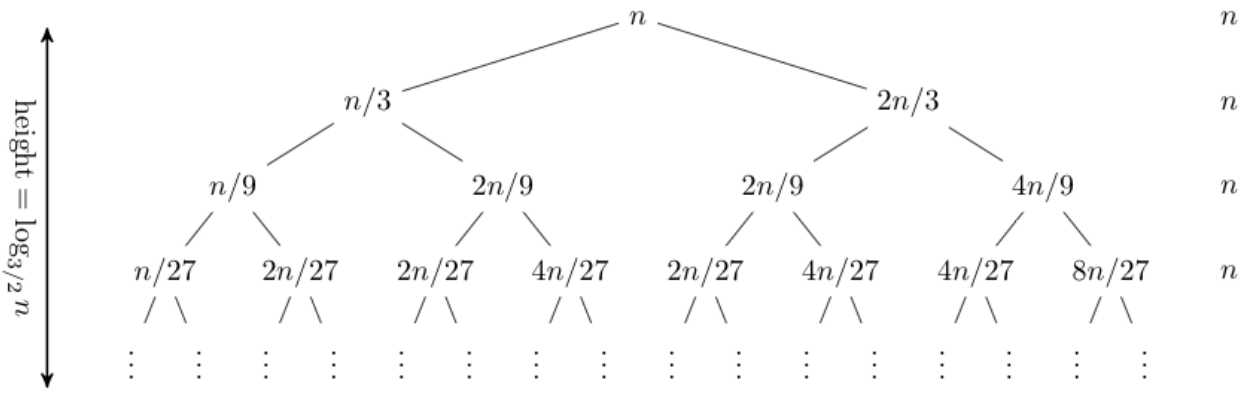
$$1 + x + x^2 + \cdots = \frac{1}{1-x} \qquad \text{for } |x| < 1$$

$$1 + \frac{5}{16} + \left(\frac{5}{16}\right)^2 + \cdots = \frac{1}{1 - 5/16} = \frac{1}{11/16} = \frac{16}{11}$$

Hence, $T(n) = \frac{16}{11}\boldsymbol{n^2} = \boldsymbol{\theta(n^2)}$

Example

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



The longest path is the rightmost one.

Let height of the tree $= h$

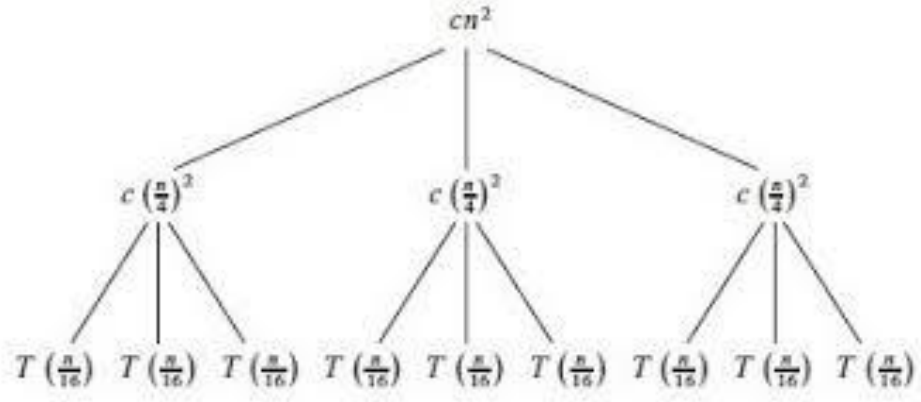
$\frac{n}{\left(\frac{3}{2}\right)^h} = 1 \quad \Rightarrow n = \left(\frac{3}{2}\right)^h \quad \Rightarrow \log n = h \log \frac{3}{2} \quad \Rightarrow h = \frac{\log n}{\log 3/2} \quad \Rightarrow \; h = \log_{3/2} n$

$$\therefore T(n) = n + n + n + \cdots = n \log_{3/2} n = O(n \log n)$$

Example

$T(n) = 3T\left(\frac{n}{4}\right) + cn^2$

Solution:



Let height of the tree $= h$

$$\frac{n}{4^h} = 1 \qquad \Rightarrow n = 4^h \qquad \Rightarrow \log n = h \log 4 \qquad \Rightarrow h = \log_4 n$$

Total Number of levels in the recursion tree $= \log_4 n + 1$

Cost of level-0 $= cn^2$

Cost of level-1 $= c(n/4)^2 + c(n/4)^2 + c(n/4)^2 = (3/16)cn^2$

Cost of level-2 $= c(n/16)^2 \times 9 = (3/16)^2 cn^2$ and so on.

$$T(n) = cn^2 + \left(\frac{3}{16}\right)cn^2 + (3/16)^2 cn^2 + \cdots$$

By geometric series, $1 + x + x^2 \ldots . + x^n = \frac{1-x^{n+1}}{1-x}$ for $x \neq 1$

$$1 + x + x^2 + \cdots = \frac{1}{1-x} \qquad \text{for } |x| < 1$$

$$1 + \frac{3}{16} + \left(\frac{3}{16}\right)^2 + \cdots = \frac{1}{1 - 3/16} = \frac{1}{13/16} = \frac{16}{13}$$

Hence, $T(n) = \frac{16}{13}\boldsymbol{n^2} = \boldsymbol{\theta(n^2)}$

The Master Theorem Method:

The master theorem method is the most useful method for solving recurrences of the form:

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

where $a \geq 1, b > 1$, and f is asymptotically positive and n/b is equal to either $\lfloor n/b \rfloor$ or, $\lceil n/b \rceil$.

The solution can be obtained by using any one of following three common rules:

1. If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some constant $\epsilon > 0$, then $T(n) = \theta\left(n^{\log_b a}\right)$
2. If $f(n) = O(n^{\log_b a})$, then $T(n) = \theta(n^{\log_b a} \lg n)$
3. If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ then $T(n) = \theta(f(n))$
   The condition $af(n/b) \leq cf(n)$ is called the regularity condition.

Example:

Find the solution of the following problem by using Master's theorem:

$T(n) = 9T(n/3) + n$

Solution:

Here, $a = 9, b = 3, f(n) = n$

$\log_b a = \log_3 9 = 2$

$n^{\log_b a} = n^2$

By rule 1, $f(n) = n = O(n^{2-1}) = O\left(n^{\log_3 9 - 1}\right)$ where $\epsilon = 1$

Hence, $T(n) = \theta\left(n^{\log_b a}\right) = \theta\left(n^{\log_3 9}\right) = \theta(n^2)$

Example:

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

Solution:

Here, $a = 4, b = 2, f(n) = n$

$\log_b a = \log_2 4 = 2$

$n^{\log_b a} = n^2$

By rule 1, $f(n) = n = O(n^{2-1}) = O\left(n^{\log_2 4-1}\right)$ where $\in = 1$

Hence, $T(n) = \theta\left(n^{\log_b a}\right) = \theta(n^2)$

Example:

$T(n) = T(2n/3) + 1$

Solution:

Here, $a = 1, b = 3/2, f(n) = 1$

$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$

By rule 2, $f(n) = 1 = O(1)$

Hence, $T(n) = \theta(1 \lg n) = \theta(\lg n)$

Example:

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

Solution:

Here, $a = 4, b = 2, f(n) = n^2$

$n^{\log_b a} = n^{\log_2 4} = n^2$

By rule 2, $f(n) = n^2 = O(n^2)$

Hence, $T(n) = \theta(n^2 \lg n)$

Example:

$T(n) = 4T(n/2) + n^3$

Solution:

Here, $a = 4, b = 2, f(n) = n^3$

$n^{\log_b a} = n^{\log_2 4} = n^2$

$f(n) = n^3 = \Omega(n^3) = \Omega(n^{2+1}) = \Omega\left(n^{\log_b a+\in}\right)$ where $\in = 1$

And also we have to check the 2nd condition,

$af\left(\frac{n}{b}\right) = 4f\left(\frac{n}{2}\right) = \frac{4n^3}{8} = \frac{n^3}{2} \leq cn^3$ where $c = 1/2$.

Hence, by rule 3, $T(n) = \theta(f(n)) = \theta(n^3)$

Example:

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

Solution:

Here, $a = 2, b = 2, f(n) = n^2$

$n^{\log_b a} = n^{\log_2 2} = n^1$

$f(n) = n^2 = \Omega(n^2) = \Omega(n^{1+1}) = \Omega\left(n^{\log_b a+\epsilon}\right)$ where $\epsilon = 1$

And $af\left(\frac{n}{b}\right) = 2f\left(\frac{n}{2}\right) = \frac{2n^2}{4} = \frac{n^2}{2} \leq cn^3$ where $c = 1/2$.

Hence, by rule 3, $T(n) = \theta(f(n)) = \theta(n^2)$

Example:

$T(n) = 3T\left(\frac{n}{4}\right) + n\log n$

Solution:

Here, $a = 3, b = 4, f(n) = n\log n$

$n^{\log_b a} = n^{\log_4 3} = n^{0.793}$

$f(n) = n\log n = \Omega(\mathrm{n}) = \Omega\left(n^{\log_4 3+0.2}\right) = \Omega\left(n^{\log_b a+\epsilon}\right)$ where $\epsilon = 0.2$

And $af\left(\frac{n}{b}\right) = 2f\left(\frac{n}{2}\right) = \frac{2n^2}{4} = \frac{n^2}{2} \leq cn^3$ where $c = 1/2$.

Now, for regularity condition of rule-3,

$af\left(\frac{n}{b}\right) = 3f\left(\frac{n}{4}\right) = 3\left(\frac{n}{4}\right)\log\left(\frac{n}{4}\right) \leq \frac{3}{4}\mathrm{n}\log n = cn\log n$ for $c = 3/4$

Therefore, the solution is $T(n) = \theta(f(n)) = \theta(n\log n)$

Example:

$$T(n) = T(\sqrt{n}) + \log n$$

Solution:

$$T(n) = T(n^{1/2}) + \log n$$

Let $n = 2^m \Rightarrow m = \log n$

$$T(2^m) = T(2^{m/2}) + \log 2^m$$

$$T(2^m) = T(2^{m/2}) + \text{m} \log 2$$

$$T(2^m) = T(2^{m/2}) + m$$

Let $T(2^m) = S(m)$

$$S(m) = S(m/2) + m$$

$$a = 1, b = 2, f(m) = m$$

$\log_b a = \log_2 1 = 0$

$f(m) = m = \Omega(m) = \Omega(m^{\log_b a + 1})$ where $\in = 1$

And $af\left(\frac{m}{b}\right) = 1f\left(\frac{m}{2}\right) = \frac{m}{2} \leq cm$ where $c = 1/2$

By rule 3, $S(m) = \theta\big(f(m)\big) = \theta(m)$

$\therefore T(n) = \theta(\log n)$ (Since $n = 2^m$)

Example:

$$T(n) = 4T(\sqrt{n}) + log^2 n$$

Solution:

Let $n = 2^m \Rightarrow m = \log n$

$$T(2^m) = 4T\big(2^{m/2}\big) + m^2$$

Let $T(2^m) = S(m)$

$$S(m) = 4S(m/2) + m^2$$

$$4 = 1, b = 2, f(m) = m^2$$

$\log_b a = \log_2 4 = 2$

$f(m) = m^2 = \text{O}(m^2) = \Omega(m^{\log_b a})$

By rule 2, $S(m) = \theta\left(m^{\log_b a} \log m\right) = \theta(m^2 \log m)$

$\therefore T(n) = \theta(\log^2 n \log(\log n))$ (Since $n = 2^m$)

Fourth condition of Master's theorem

If $f(n)$, the non-recursive cost is not a polynomial and it is a poly logarithmic function, then 4th condition of the master's theorem is applicable.

If $f(n) = \theta\left(n^{\log_b a} \log^k n\right)$ for some $k > 0$, then $T(n) = \theta\left(n^{\log_b a} \log^{k+1} n\right)$

Example:

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

Solution:

Here, $a = 2, b = 2, f(n) = n \log n$

$n^{\log_b a} = n^{\log_2 2} = n^1 = n$

$f(n) = \theta(n \log^1 n)$, $k = 1$

$T(n) = \theta\left(n^{\log_b a} \log^{k+1} n\right) = \theta(n \log^{1+1} n) = \theta(n \log^2 n)$

Example:

$$T(n) = T(\sqrt{n}) + \theta(\log \log n)$$

Solution:

Let $n = 2^m \Rightarrow m = \log n$

$$T(2^m) = T(2^{m/2}) + \theta(\log \log 2^m)$$

$$T(2^m) = T(2^{m/2}) + \theta(\log m \log 2)$$

$$T(2^m) = T(2^{m/2}) + \theta(\log m)$$

Let $T(2^m) = S(m)$

$$S(m) = S(m/2) + \theta(\log m)$$

Here, $a = 1, b = 2, f(m) = \theta(\log m)$

$m^{\log_b a} = m^{\log_2 1} = m^0 = 1$

$$f(m) = \theta\left(m^{\log_b a} \log^{k+1} m\right), k = 1$$

$$S(m) = \theta\left(m^{\log_b a} \log^{k+1} m\right) = \theta(\log^{1+1} m) = \theta(\log^2 m) = \theta(\log^2 \log n)$$

Advanced Master Theorem:

The advanced version of the Master Theorem provides a more general form of the theorem that can handle recurrence relations that are more complex than the basic form. The advanced version of the Master Theorem can handle recurrences with multiple terms and more complex functions.

If the recurrence relation is of the form:

$T(n) = aT(n/b) + f(n)$ where $a \geq 1, b > 1$

And $f(n) = \theta(n^k log^p n)$, $k \geq 0$, and p is a real number then:

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

Example:

Solve the recurrence relation $T(n) = 8T\left(\frac{n}{2}\right) + n^2$

Solution:

In this problem, $a = 8, b = 2, f(n) = n^2 = \theta(n^2) = \theta(n^k log^p n)$, where $k = 2, p = 0$

$$a > b^k \; i.e. \; 8 > 2^2 \quad [Case\ 1]$$

Therefore, $T(n) = \theta\left(n^{\log_b a}\right) = T(n) = \theta\left(n^{\log_2 8}\right) = T(n) = \theta(n^3)$

Example:

Solve the recurrence relation

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

Solution:

In this problem, $a = 2, b = 2, f(n) = n\,log^2 n = \theta(n\,log^2 n) = \theta(n^k log^p n)$, where $k = 1, p = 2$

$$a = b^k\ i.e.\ 2 = 2^1, p > -1 \qquad [Case\ 2\ (a)]$$

Therefore, $T(n) = \theta\left(n^{\log_b a} \log^{p+1} n\right) = \theta\left(n^{\log_2 2} \log^{2+1} n\right) = \theta(n \log^3 n)$

Example:

Solve the recurrence relation

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

Solution:

In this problem, $a = 2, b = 2, f(n) = n/\log n = \theta(n\,log^{-1} n) = \theta(n^k log^p n)$, where $k = 1, p = -1$

$$a = b^k\ i.e.\ 2 = 2^1, p = -1 \qquad [Case\ 2\ (b)]$$

Therefore, $T(n) = \theta\left(n^{\log_b a} \log\log\ n\right) = \theta\left(n^{\log_2 2} \log\log\ n\right) = \theta(n\ \log\ (\log n))$

Example:

Solve the recurrence relation

$$T(n) = 16T\left(\frac{n}{4}\right) + n^2/\log^2 n$$

Solution:

In this problem, $a = 16, b = 4, f(n) = n^2/\log^2 n = \theta(n^2\ log^{-2} n) = \theta(n^k log^p n)$, where $k = 2, p = -2$

$$a = b^k\ i.e.\ 16 = 4^2, p < -1 \qquad [Case\ 2\ (c)]$$

Therefore, $T(n) = \theta\left(n^{\log_b a}\right) = \theta\left(n^{\log_4 16}\right) = \theta(n^2)$

Example:

Solve the recurrence relation

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

Solution:

In this problem, $a = 2, b = 2, f(n) = n^2 = \theta(n^2\ log^0 n) = \theta(n^k log^p n)$, where $k = 2, p = 0$

$$a = b^k\ i.e.\ 2 = 2^2, p \geq 0 \quad [Case\ 3(a)]$$

Therefore, $T(n) = \theta(n^k log^p n) = \theta(n^2 log^0 n) = \theta(n^2)$

Example:

Solve the recurrence relation

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

Solution:

In this problem, $a = 2, b = 2, f(n) = \frac{n^3}{\log n} = \theta(n^3\ log^{-1} n) = \theta(n^k log^p n)$, where $k = 3, p = -1$

$$a < b^k\ i.e.\ 2 < 2^3, p < 0 \quad [Case\ 3(b)]$$

Therefore, $T(n) = O(n^k) = O(n^3)$

Limitations of Master's Theorem:

The master theorem cannot be used for if:

- $T(n)$ is not monotone. eg. $T(n) = sin\ n$
- $f(n)$ is not a polynomial. eg. $f(n) = 2^n$
- a is not a constant. eg. $a = 2n$
- $a < 1$

Problem Set

1. $T(n) = 3T(n/2) + n^2$
   $T(n) = \theta(n^2)$ (Rule-3)

2. $T(n) = 4T(n/2) + n^2$
   $T(n) = \theta(n^2 \log n)$ (Rule-2)

3. $T(n) = T(n/2) + 2^n$
   $T(n) = \theta(2^n)$ (Rule 3)

4. $T(n) = 2^n T(n/2) + n^n$
   Master theorem is not applicable because 2^n is not a constant.

5. $T(n) = 16T(n/4) + n$
   $T(n) = \theta(n^2)$ (Rule-1)

6. $T(n) = 2T(n/4) + n^{0.51}$
   $T(n) = \theta(n^{0.51})$ (Rule-3)

7. $T(n) = 0.5T(n/2) + 1/n$
   Master theorem is not applicable because $a < 1$

8. $T(n) = 16T\left(\frac{n}{4}\right) + n!$
   $T(n) = \theta(n!)$ (Rule-3)

9. $T(n) = \sqrt{2}T(n/2) + log\, n$
   $T(n) = \theta(\sqrt{n})$ (Rule-1)

10. $T(n) = 3T(n/2) + n$
    $T(n) = \theta(n^{\log 3})$ (Rule-1)

11. T $(n) = 3T(n/3) + \sqrt{n}$
    $T(n) = \theta(n)$ (Rule 1)

12. $T(n) = 4T(n/2) + cn$
    $T(n) = \theta(n^2)$ (Rule-1)

13. $T(n) = 3T(n/4) + n\, log\, n$
    $T(n) = \theta(n \log n)$ (Rule-3)

14. $T\ (n)\ =\ 3T\ (n/3)\ +\ n/2$
    $T(n) = \theta(n \log n)$ (Rule-2)

15. $T\ (n)\ =\ 6T\ (n/3)\ +\ n^2\ log\ n$
    $T(n) = \theta(n^2\ log\ n)$ (Rule-3)
16. $T\ (n)\ =\ 4T\ (n/2)\ +\ n/\ log\ n$
    $T\ (n)\ =\ \theta(n^2)$ (Rule-1)

17. $T\ (n)\ =\ 64T\ (n/8)\ -\ n^2\ log\ n$
    Master theorem is not applicable because $f(n)$ is not positive.

18. $T\ (n) =\ 7T\ \left(\frac{n}{3}\right) +\ n^2$
    $T\ (n)\ =\ \theta(n^2)$ (Rule-3)

19. T (n) = 4T (n/2) + log n
    $T\ (n)\ =\theta(n^2)$ (Rule-1)

ALGORITHM DESIGN TECHNIQUES

Algorithmic strategy / technique / paradigm is a general approach by which many problems can be solved algorithmically.

Following are some popular design techniques:

a) Brute Force
b) Divide and Conquer
c) Dynamic Programming
d) Greedy Technique
e) Branch & Bound
f) Backtracking

BRUTE FORCE APPROACH

Brute force Approach is a type of problem solving approach wherein a problem solution is directly based on the problem definition that is provided. It is a top down approach. It is considered as the easiest approach to adopt and is also very useful when problem domain is not that much complex.

Example:

- Computing a factorial of a number
- Multiplication of matrices
- Searching and sorting.

Let's consider an example to compute x^n (where, $x > 0$ and n is any non-integer).

Based on the definition of exponentiation $x^n = x \times x \times \ldots \times x$

Implies using brute force to solve the above exponentiation problem it requires $(n-1)$ repetitive multiplications.

But to solve above problem using recursive approach the complexity of the problem is reduced to $O(log(n))$ because

$$x^n = x \times x^{n-1}$$

Divide and Conquer Method:

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide.

Divide and conquer algorithm consists of two parts:

1. Divide : Divide the problem into a number of sub problems. The sub problems are solved recursively.
2. Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Merge Sort

The merge sort algorithm that uses the divide and conquer technique. The algorithm comprises three steps, which are as follows:

1. Divides the n-element list, into two sub-lists of n/2 elements each, such that both the sub-lists hold half of the element in the list.
2. Recursively sort the sub-lists using merge sort.
3. Merge the sorted sub-lists to generate the sorted list.

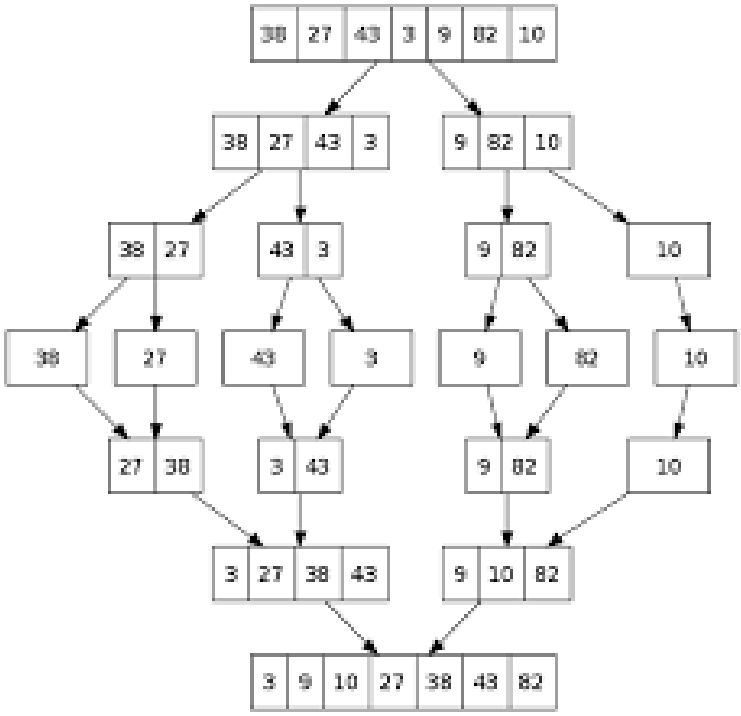
Algorithm

MERGE SORT (A, p, r)

if $p < r$ *then*
    $q \leftarrow (p+r)/2$ *// defines the current array in 2 parts .*
    MERGE-SORT (A, p, q) // sort the 1st part of array
    MERGE-SORT (A, q+1,r) // sort the 2nd part of array
    MERGE (A, p, q, r)

MERGE (A, p, q, r)

$n_1 \leftarrow q - p + 1$
$n_2 \leftarrow r - q$
create arrays $L[1 \ldots . n_1 + 1]$ *and* $R\ [\ 1 \ldots . n_2 + 1\ ]$
for $i \leftarrow 1$ *to* n_1 *do*
    $L[i] \leftarrow A\ [\ p + i - 1]$
for $j \leftarrow 1$ *to* n_2 *do*
    $R[j] \leftarrow A[\ q + j]$
$L\ [n_1 + 1] \leftarrow \infty$
$R\ [n_2 + 1] \leftarrow \infty$
$i \leftarrow 1$
$j \leftarrow 1$
for $k \leftarrow p$ *to* r *do*
    if $L\ [i] \leq R[j]$ *then*
        $A[k] \leftarrow L[\ i]$
        $i \leftarrow i + 1$
    else $A[k] \leftarrow R[j]$
        $j \leftarrow j+1$



Analysis of Merge Sort:

Let $T(n)$ represents the total time taken by merge sort algorithm to sort an array of size n.

When we have $n > 1$ elements, we break down the running time as follows:

1. *Divide*: The divide step just computes middle of the array, which takes constant time. Thus, it is equal to $\theta(1)$.
2. *Conquer*: Time taken by the algorithm to recursively sort the two halves of the array, each of size $n/2$ is $2T(n/2)$.
3. *Combine:* Time taken to merge the two sorted halves is $\theta(n)$.

The running time of a recursive procedure can be expressed as a recurrence relation:

$$T(n) = \begin{cases} solving\ trivial\ problem & if\ n = 1 \\ No.of\ peices\ \times T\left(\dfrac{n}{ReductionFactor}\right) + divide + Combine & if\ n\ > 1 \end{cases}$$

Hence, the recurrence relation of Merge sort is:

$$T(n) = \begin{cases} \theta(1) & if\ n = 1 \\ 2T\left(\dfrac{n}{2}\right) + \theta(n) & if\ n\ > 1 \end{cases}$$

Solution:

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

$= n + 2[n/2 + 2T(n/4)]$

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

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

$= 3n + 8\,T(n/8) = 3n + 2^3T(\frac{n}{2^3})$

… … … … … … … … … … ..

… … … … … … … … … … … ….

$= kn + 2^kT(\frac{n}{2^k})$ (where $\frac{n}{2^k} = 1 \Rightarrow n = 2^k \;\Rightarrow \log n = k \log 2 \Longrightarrow \log n = k$)

$= nk + 2^kT(1) = nlog\ n + 2^{\log n}$(Since T(1)=1)

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

Space Complexity

The space complexity of merge sort is $O(n)$, which means that it requires an auxiliary array to temporarily store the merged array. The auxiliary array must be the same size as the main input array.

Quick Sort

It is one of the fastest sorting algorithms based on the divide and conquer approach. This algorithm takes an array of values, chooses one of the values as the 'pivot' element, and divides the array in such a way that elements less than pivot are kept on the left side and the elements greater than pivot are on the right side of the pivot.

Working Procedure:

1. Choose a value in the array to be the pivot element.
2. Order the rest of the array so that lower values than the pivot element are on the left, and higher values are on the right.
3. Swap the pivot element with the first element of the higher values so that the pivot element lands in between the lower and higher values.
4. Do the same operations (recursively) for the sub-arrays on the left and right side of the pivot element.

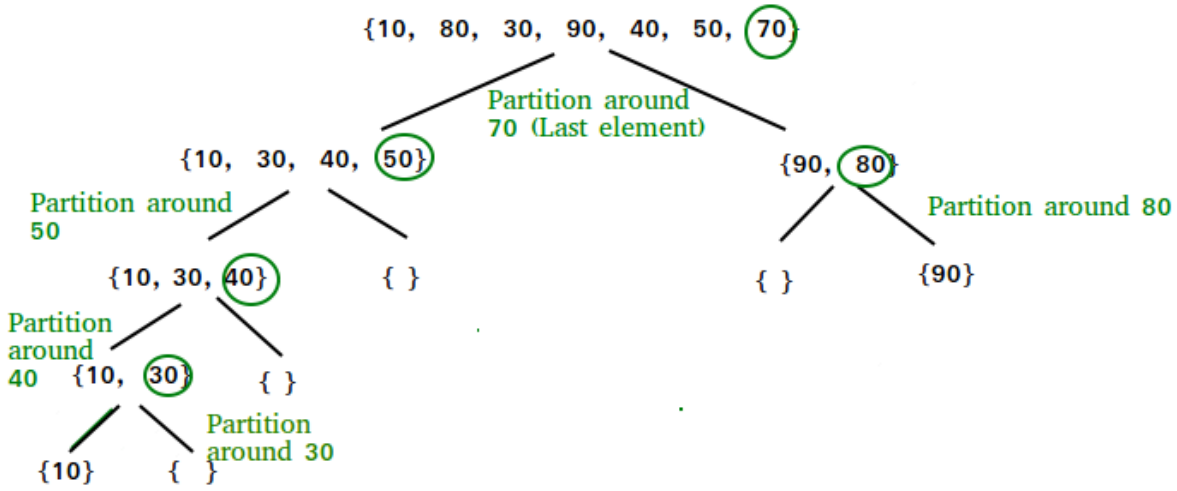
Algorithm:

Quick_sort*(A, p, r)*

```
if (p<r) then
    q←Partition(A, p, r);
    Quick_sort(A, p, q-1);
    Quick_sort(A, q+1, r);
```

Partition*(A , p ,r)*

```
x ← A[r];
i ← p − 1;
for j ← p to r − 1 do
    if A[j] <= x then
        i = i + 1;
        exchange A[i] ↔ A[j];
exchange A[i + 1] ↔ A[r];
return (i + 1);
```



Example:

	10	80	30	90	40	50	70
Index:	1	2	3	4	5	6	7

$p = 1, r = 7,$ pivot $x = a[r] = a[7] = 70, i = p - 1 = 0$
Now for $j = 1$ to 7

Pass 1:

$j = 1$ and $i = 0$

Test condition	Action	Value of the variables
$a[j] \leq x$, $10 \leq 70, Yes$	$i = i + 1$ Swap $(a[i], a[j])$, Swap (10, 10)	$i = 0 + 1 = 1$ $j = 1$

	10	80	30	90	40	50	70
Index:	1	2	3	4	5	6	7

Pass 2:

$j = 2$ and $i = 1$

Test condition	Action	Value of the variables
$a[j] \leq x$, $80 \leq 70, No$	No Action	$i = 1$ $j = 2$

	10	80	30	90	40	50	70
Index:	1	2	3	4	5	6	7

Pass 3:

$j = 3$ and $i = 1$

Test condition	Action	Value of the variables
$a[j] \leq x$, $30 \leq 70, Yes$	$i = i + 1$ Swap $(a[i], a[j])$, Swap (80, 30)	$i = 1 + 1 = 2$ $j = 3$

	10	30	80	90	40	50	70
Index:	1	2	3	4	5	6	7

Pass 4:

$j = 4$ and $i = 2$

Test condition	Action	Value of the variables
$a[j] \leq x$, $90 \leq 70, No$	No Action	$i = 2$ $j = 4$

	10	30	80	90	40	50	70
Index:	1	2	3	4	5	6	7

Pass 5:

$j = 5$ and $i = 2$

Test condition	Action	Value of the variables
$a[j] \leq x$, $40 \leq 70, Yes$	$i = i + 1$ Swap $(a[i], a[j])$, Swap (40, 80)	$i = 2 + 1 = 3$ $j = 5$

	10	30	40	90	80	50	70
Index:	1	2	3	4	5	6	7

Pass 6:

$j = 6$ and $i = 3$

Test condition	Action	Value of the variables
$a[j] \leq x$, $50 \leq 70, Yes$	$i = i + 1$ Swap $(a[i], a[j])$, Swap (90, 50)	$i = 3 + 1 = 4$ $j = 6$

	10	30	40	50	80	90	70
Index:	1	2	3	4	5	6	7

Before pass 7, j becomes 7, so we come out from the loop.
Now we have to swap $(a[i + 1],\ \ a[q])$ i.e. Swap (80, 70)

	10	30	40	50	70	90	80
Index:	1	2	3	4	5	6	7

Now, the element 70 is brought to its appropriate position by the partition function. Now the same procedure will be applied to the left part and right part of the element 70.

Thus, return $q = i + 1 = 5$
Now call $Quick\_sort(A, p, q - 1)$ and $Quick\_sort(A, q + 1, r)$,
i.e, $Quick\_sort(A, 1, 4)$ and $Quick\_sort(A, 6, 7)$.

Analysis of Quick Sort

Best Case:

The best-case scenario for quicksort occurs when the pivot chosen at the each step divides the array into roughly two equal halves. If the procedure partition produces two regions of

size $n/2$. Partition algorithm performs n comparisons (possibly n−1 or n+1, depending on the implementation).

The running time of Quick Sort can be expressed as a recurrence relation:

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

By using Master's method, the solution of the recurrence $2T\left(\frac{n}{2}\right) + n$ is $\theta(n \log n)$

Therefore Best Case Time Complexity of Quick Sort is $\theta(n \log n)$.

Worst Case:

The worst-case scenario for quick sort occurs when the array is already sorted and the pivot is always chosen as the smallest or largest element. In this case the pivot consistently results a highly unbalances partitions at each step.

Let $T(n)$ represents the total time taken by Quick Sort algorithm to sort an array of size n. Since part 1 contains $n-1$ elements and part 2 contains 1 element and partition procedure requires n comparisons then

$$T(n) = T(n-1) + T(1) + n = T(n-1) + n \qquad (\because T(1) = 0)$$

Solution:

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

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

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

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

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

$$= n + (n-1) + (n-2) + (n-3) + \ldots\ldots\ldots + 3 + 2 + 0 (\because T(1) = 0)$$

$$= (n + (n-1) + (n-2) + (n-3) + \ldots\ldots\ldots + 3 + 2 + 1) - 1$$

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

Therefore Worst Case Time Complexity of Quick Sort is $O(n^2)$.

Average Case:

Suppose that size of the array be n and in the first pass the pivot partitions the array into two sizes $k-1$ and $n-k$.

Then the running time is given by the recurrence:

$T(n) = n + T(k-1) + T(n-k)$ with $T(0) = 0$ and $T(1) = 0$

Assume that the pivot element will be a random element of the array to be partitioned.

That is, for $k = 1, 2, \ldots, n$, the probability that the pivot element is the k^{th} largest element of the array is $1/n$.

Since, all values of k are equally likely, we must average over all k.

$$T(n) = \frac{\sum_{k=1}^{n} n + T(k-1) + T(n-k)}{n}, T(0) = 0, T(1) = 0 \ldots . (1)$$

$$= n + \frac{\sum_{k=1}^{n} T(k-1) + T(n-k)}{n}$$

$$= n + \frac{\sum_{k=1}^{n} T(k-1)}{n} + \frac{\sum_{k=1}^{n} T(n-k)}{n}$$

By substituting $i = k-1$ in the above equation we get:

$$T(n) = n + \frac{\sum_{i=0}^{n-1} T(i)}{n} + \frac{\sum_{i=0}^{n-1} T(n-i-1)}{n}$$

Since, $\sum_{i=0}^{n-1} T(i) = \sum_{i=0}^{n-1} T(n-i-1)$, $T(n)$ can be written as

$$T(n) = n + \frac{\sum_{i=0}^{n-1} T(i)}{n} + \frac{\sum_{i=0}^{n-1} T(i)}{n} = n + \frac{2\sum_{i=0}^{n-1} T(i)}{n}$$

$$\Rightarrow nT(n) = n^2 + 2\sum_{i=0}^{n-1} T(i) \qquad \ldots (2)$$

Substituting n by $n-1$, we get

$$(n-1)T(n-1) = (n-1)^2 + 2\sum_{i=0}^{n-2} T(i) \quad \ldots (3)$$

By subtracting equation (3) from (2) we get,

$$nT(n) - (n-1)T(n-1) = n^2 + 2\sum_{i=0}^{n-1} T(i) - (n-1)^2 - 2\sum_{i=0}^{n-2} T(i)$$

$$\Rightarrow nT(n) - (n-1)T(n-1) = n^2 - (n-1)^2 + 2T(n-1)$$

Rearranging and simplifying the above equation we get

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

Dividing both sides by $n(n+1)$ we get

$$\frac{T(n)}{(n+1)} = \frac{T(n-1)}{n} + \frac{2n-1}{n(n+1)}$$

$$\Rightarrow \frac{T(n)}{(n+1)} \approx \frac{T(n-1)}{n} + \frac{2}{n}$$

Let $S(n) = \frac{T(n)}{(n+1)}$, then the recurrence relation becomes:

$$S(n) = S(n-1) + \frac{2}{n}, \qquad S(1) = 0$$

Now, by applying repeated substitution method,

$$S(n) = \frac{2}{n} + S(n-1)$$

$$= \frac{2}{n} + \frac{2}{n-1} + S(n-2) = \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + S(n-3)$$

$$= \cdots \ldots \ldots \ldots \ldots \ldots \ldots \ldots = \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \cdots + \frac{2}{3} + \frac{2}{2} + S(1)$$

$$= \frac{2}{n} + \frac{2}{n-1} + \frac{2}{n-2} + \cdots + \frac{2}{3} + \frac{2}{2} + 1 = 2\left[\frac{1}{1} + \frac{1}{2} + \cdots \frac{1}{n}\right] = 2\sum_{i=1}^{n} \frac{1}{i} \approx 2\log n$$

[Harmonic Series]

$$\text{So, } S(n) \approx 2\log n$$

$$\frac{T(n)}{(n+1)} \approx 2\log n$$

$$\Rightarrow T(n) = 2(n+1)\log n \approx 1.39 \text{ n} \log n$$

The expected case for quick sort is fairly close to the best case (only 39% more comparisons) and nothing like the worst case.

In most (not all) tests, quick sort turns out to be a bit faster than merge sort.

Quick sort performs 39 & more comparisons than merge sort, but much less movement (copying) of array elements.

Space Complexity:

In quick sort, the space complexity is calculated on the basis of space used by the recursion stack. In the worst case, the space complexity is $O(n)$ because in worst case, n recursive calls are made. And, the average space complexity of a quick sort algorithm is $O(\log n)$.