Analysis of Algorithms

Dr. Rupali Patil

Algorithms

Algorithm: a method or a process followed to solve a problem.

- An Experiment.

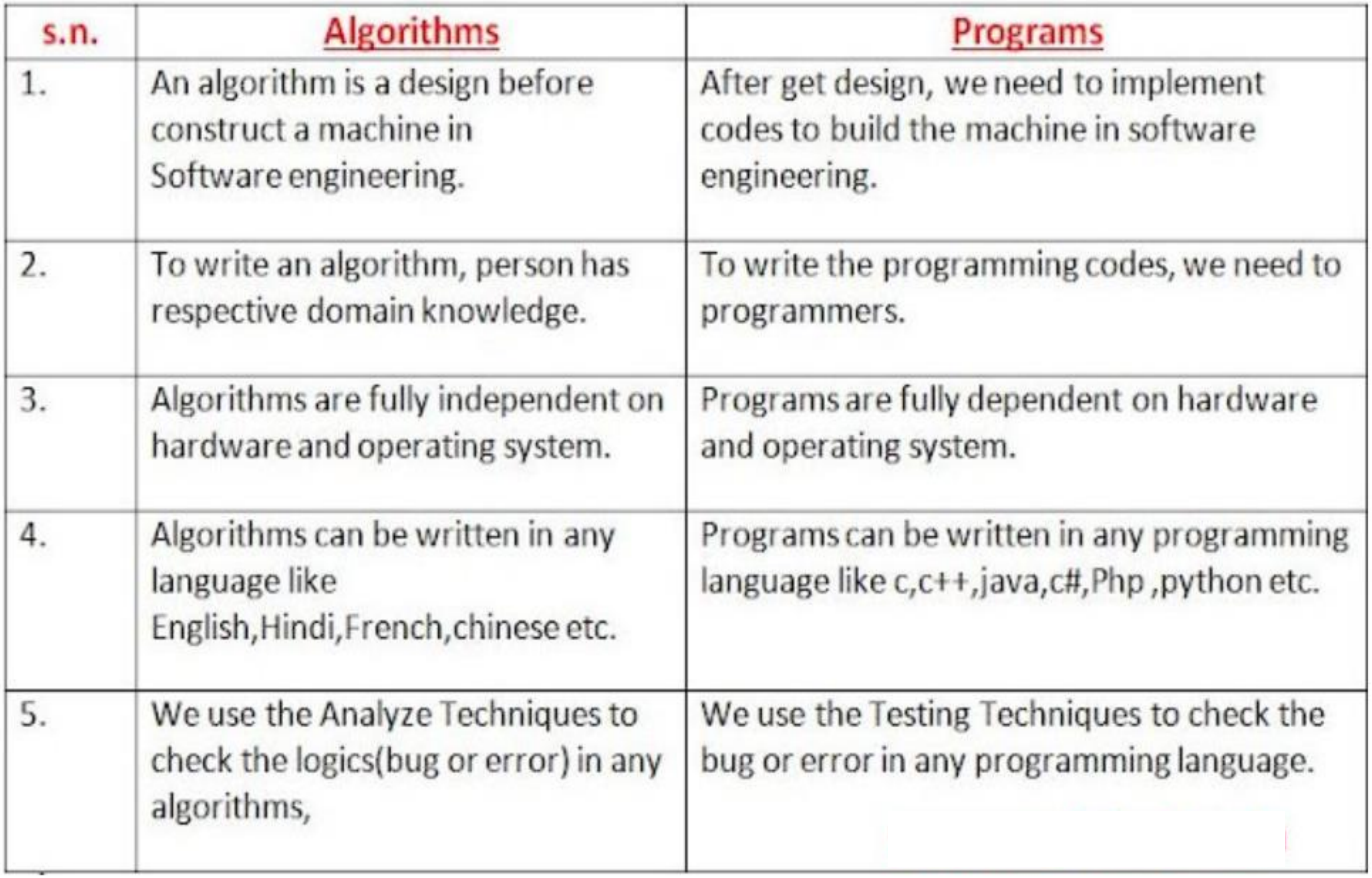
An algorithm takes the input to a problem (function) and transforms it to the output.

- A mapping of input to output.

A problem can have many algorithms.

- A recipe..

Algorithms and Programs



s.n.	Algorithms	Programs
1.	An algorithm is a design before construct a machine in Software engineering.	After get design, we need to implement codes to build the machine in software engineering.
2.	To write an algorithm, person has respective domain knowledge.	To write the programming codes, we need to programmers.
3.	Algorithms are fully independent on hardware and operating system.	Programs are fully dependent on hardware and operating system.
4.	Algorithms can be written in any language like English,Hindi,French,chinese etc.	Programs can be written in any programming language like c,c++,java,c#,Php ,python etc.
5.	We use the Analyze Techniques to check the logics(bug or error) in any algorithms,	We use the Testing Techniques to check the bug or error in any programming language.

Algorithm and its Properties

An algorithm is a finite set of instructions that, if followed, accomplishes particular task. In addition, all algorithms must satisfy the following criteria:

1. **Input**. Zero or more quantities are externally supplied.
2. **Output**. At least one quantity is produced.
3. **Definiteness**. It must be composed of a series of concrete steps. There can be no ambiguity as to which step will be performed next.
4. **Finiteness**. If we trace out the instructions of an algorithm, then for all cases, the algorithm **terminates after a finite number of steps.**
5. **Effectiveness**. Every instruction must be very basic so that it can be carried out, it also must be feasible. **It must be correct**

Algorithm Efficiency

There are often many approaches (algorithms) to solve a problem. How do we choose between them?

At the heart of computer program design are two (sometimes conflicting) goals.

1. To design an algorithm that is **easy to understand, code, debug.**
2. To design an algorithm that makes **efficient use of the computer's resources.**

Specifications of good Algorithm

- Work Correctly for all case
- Steps are clear
- Effective Time utilization
- Effective Space utilization
- Give best solution

Analysis of Algorithm

- Apriori analysis
  - Time
  - Space
  - Cost (Software Engineering)
- Posterior analysis

A Priori analysis	**A Posteriori Testing**
It is done before execution of an **algorithm**.	It is done after execution of an algorithm. Or after writing the **program**
Priori analysis is an absolute analysis.	Posteriori analysis is a relative analysis.
It is independent of language of compiler and types of hardware/OS.	It is dependent on language of compiler and type of hardware/OS
It will give approximate answer.	It will give exact answer.
It uses the asymptotic notations to represent how much time the algorithm will take in order to complete its execution.	It doesn't use asymptotic notations to represent the time complexity of an algorithm.

Apriori analysis in general

- Buying a cellphone
  - Budget
  - User age group
  - Technical specification
  - User reviews

Apriori analysis in general

- Buying a home
  - Budget
  - Area (price to area ratio)
  - Location & Locality
  - Amenities in the apartment
  - Amenities around the place
  - other factors

Apriori analysis in general

- Preparing for examination
  - # of chapters
  - #of days/hours in hand
  - Weightage given to every topic
  - Difficulty level
  - Importance of examination score

Apriori analysis in general

- Admission for higher studies
  - ?
  - ?
  - ?
  - ?

Writing an Algorithm

- Comments begin with // and continue until the end of line.
- Blocks are indicated with matching braces: { and }.
- A compound statement (i.e., a collection of simple statements) can be represented as a block. The body of a procedure also forms a block.
- Statements are delimited by ;
- An identifier begins with a letter.
- Assignment of values to variables is done using the assignment statement **(variable) := (expression);**
- There are two Boolean values true and false.
- The logical operators and, or, and not
- The relational operators <,>,<=,>=,== and !=
- The following looping statements are employed: for, while, and repeat-until.
- Conditional statements are if,if else and case

Writing an Algorithm

- Algorithm consist of two parts: Head and Body

Algorithm Name ((parameter list)),where Name is the name of the procedure and ((parameter list)).

The body has one or more a listing of the procedure parameters.

```
Algorithm Max(A, n)
// A is an array of size n.
Result == A[1];
for i:= 2 to n do
if A[i] > Result then Result :=
A[i]; return Result;
```

How to analyze algorithms

- Time
- Space
- Performance Analysis
  - ❖ Best Case
  - ❖ Average Case
  - ❖ Worst Case

Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size "n" of the input data.

Approaches to calculate Time/Space Complexity:

1. Frequency count/Step count Method
2. Asymptotic Notations – (Order of)

Frequency count/Step count Method

Rules:

1. For comments, declaration

   count = 0

2. return and assignment statement

   count = 1

3. Ignore lower order exponents when higher order exponents are present

   Ex. Complexity of following algo is as follows:

   $f(n) = 6n^3 + 10n^2 + 15n + 3 \Longrightarrow 6n^3$

4. Ignore constant multipliers

   $6n^3 \Longrightarrow n^3$

   $f(n) = O(n^3)$

Example 1:sum of n values of an array

Algorithm sum (int a[], int n	Time Complexity	Space Complexity
`s = 0;`	→ 1 unit	a[] = n words
`for(i=0; i<n; i++)`	→ n+1	n= 1 word
`{`		s= 1 word
`s=s + a[i];`	→ n	i= 1 word
`}`		n+3
`return s;`	→ 1	
	→ 2n+3 f(n) = O(n)	Space complexity = O(n)

Example 2: Addition of two square Matrices of dimension $n \times n$

```
Algorithm addMat (int a[][], int b[][])
{ int c[][];
  for(i=0; i<n; i++) {
     for(j=0; j<n; j++) {
      c[i][j] = a[i][j] + b[i][j]
  }
 }
}
```

Time Complexity
$\rightarrow n+1$
$\rightarrow n\times(n+1)$
$\rightarrow n\times n$
$\rightarrow n+1+n^2+n+n^2$ $\rightarrow 2n^2 + 2n + 1$ $f(n) = O(n^2)$

Space Complexity
a[][] = n^2 words
b[][] = n^2 words
c[][] = n^2 words
i= 1 word
j= 1 word
n= 1 word
$\rightarrow 3n^2 + 3$ Space complexity = $O(n^2)$

Example 3:Multiplication of two Matrices of dimension $n \times n$

```
Algorithm matMul (int a[][],
int b[][])
{ int c[][];
  for(i=0; i<n; i++) {
     for(j=0; j<n; j++) {
      c[i][j] = 0;
      for(k=0; k<n; k++){
        c[i][j] = a[i][j] *
b[i][j]
     }
   }
  }
}
```

Time Complexity
$\rightarrow n+1$
$\rightarrow n\times(n+1)$
$\rightarrow n\times n$
$\rightarrow n\times n\times(n+1)$
$n\times n\times n$
$\rightarrow n+1+n^2+n+n^2+n^3+1+n^3+n^2$ $\rightarrow 2n^3+3n^2+2n+1$

Space Complexity
a[][] = n^2 words
b[][] = n^2 words
c[][] = n^2 words
i= 1 word
j= 1 word
k=1 word
n= 1 word
$\rightarrow 3n^2+4$ Space complexity = $O(n^2)$

Example: loops

1.

for(i=0; i<n; i++) {
statements;
}

Time Complexity
→ n+1
→ n
f(n) = 2n+1 f(n) = O(n)

2.

for(i=n; i>0; i--) {
statements;
}

Time Complexity
→ n+1
→ n
f(n) = 2n+1 f(n) = O(n)

Example: loops

3.

for(i=1; i<n; i=i+2) {
statements;
}

Time Complexity
$\rightarrow$ n+1
$\rightarrow$ n/2
$f(n) = 3n/2+1$ $f(n) = O(n)$

4.

for(i=0; i<n; i++) {
for(j=0; j<n; j++) {
statements;
}

Time Complexity
$\rightarrow$ n+1
$\rightarrow$ n(n+1)
$\rightarrow n \times n$
$f(n) = 2n^2+2n+1$ $f(n) = O(n^2)$

Example: loops (By tracing)

5.

```
for(i=0; i<n; i++) {
    for(j=0; j<i; j++) {
        statements;
    }
}
```

$$1 + 2 + 3 + 4 + \cdots + n = n(n+1)/2$$

$$T(n) = 0 + 1 + 2 + 3 + 4 + \cdots + n - 1 = \frac{(n-1)(n)}{2} = O(n^2)$$

Time Complexity		
i	j	statements
0	0	0
1	0	1
	1	
2	0	2
	1	
	2	
3	0	3
	1	
	2	
	3	
…	…	…
n	0 to n-1	n

Example 5: loops (By tracing)

6.

```
p=0 ;
  for(i=1; p<=n; i++) {
        p=p+i;
   }
}
```

=1+2+3+4+...+k>n

$= \; k(k+1)/2 > n$

$= \; k^2 + k/2 \; > n$

$\cong \; k^2 > n$

$$\boldsymbol{k = \sqrt{n} \; = \; O(\sqrt{n})}$$

Time Complexity		
i	**p**	**statements**
1	0+1	1
2	1+2	1
3	1+2+3	1
4	1+2+3+4	1
5	1+2+3+4+5	1
6	1+2+3+4+5 +6	1
k	1+2+3+4+… +k	???

Example: loops (By tracing)

6.

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

$$i >= n$$

$$i = 2^k$$

$$2^k >= n$$

$$2^k = n$$

$$\boldsymbol{k = log_2 n \ = \ O(log_2 n)}$$

Time Complexity	
I	statements
1*2	1
2*2	1
2*2*2	1
2*2*2*2	1
…	…
…	…
2^k	1

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Example: loops (By tracing)

7.

```
for(i=n; i>=1; i=i/2) {
      statements;
  }
}
```

$i<1$

$n/2^k = 1$

$n=2^k$

$$\boldsymbol{k = log_2 n \;=\; O(log_2 n)}$$

Time Complexity
i
n
$n/2$
$n/2^2$
$n/2^3$
$n/2^4$
…
$n/2^k$

Example: loops (By tracing)

8.

```
for(i=0; i*i<n; i++) {
     statements;
  }
}
```

$$k * k >= n$$

$$k^2 = n$$

$$k=\sqrt{n}$$

$$\boldsymbol{k = \sqrt{n} = O(\sqrt{n})}$$

Time Complexity	
i	statements
1	1
2	2^2
3	3^2
4	4^2
5	5^2
…	…
k	k^2

Example: loops (By tracing)

9.

```
for(i=1; i<n; i=i*2) {
        p++;
 }
for(j=1;j<p;j=j*2){
    statements;
}
```

$$p = \log_2 n$$

$$T(n) = \log_2 p$$

$$T(n) = \log_2 \log_2 n$$

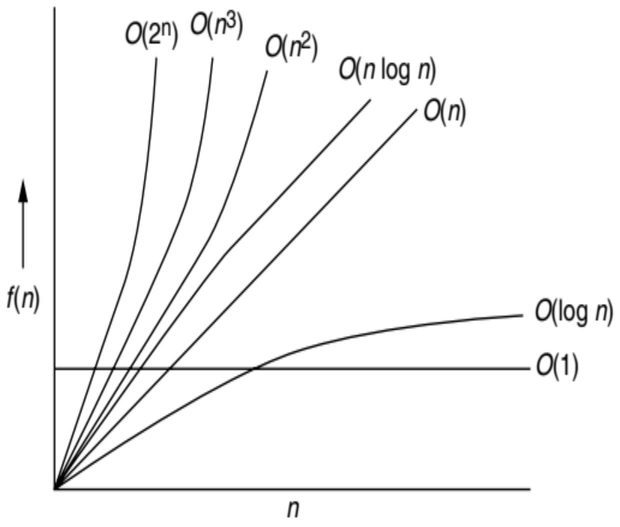
Rate of Growth

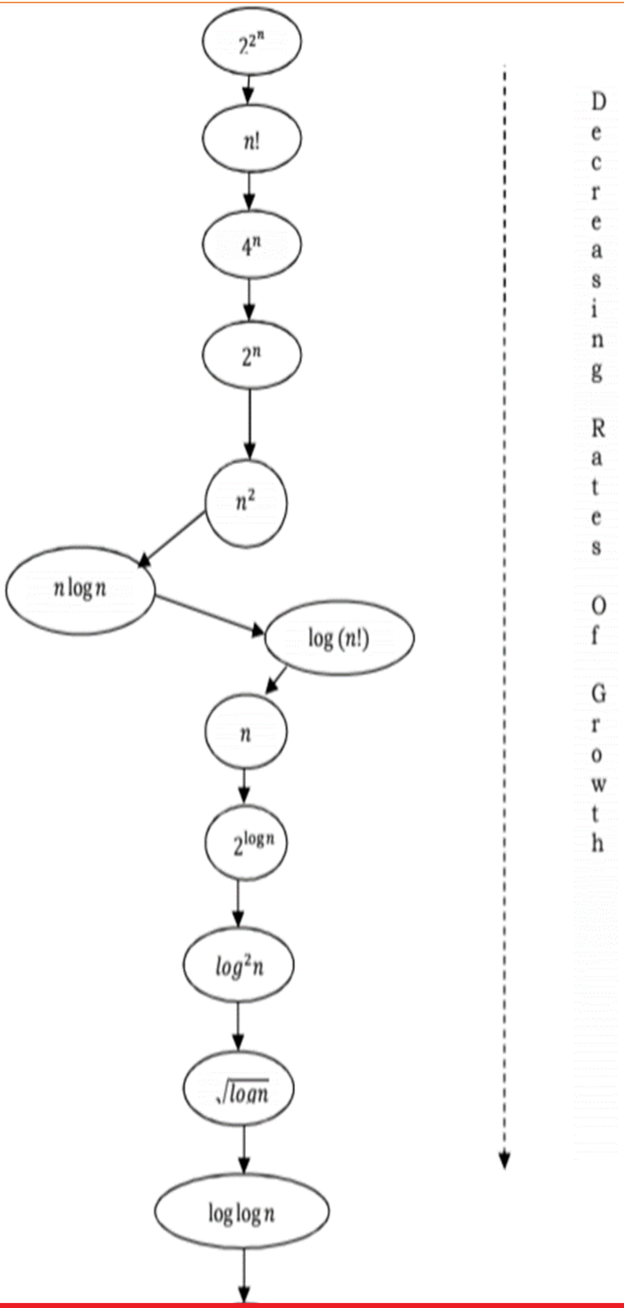
Rate at which the **running time increases as a function of input** is called Rate of Growth.

Example:

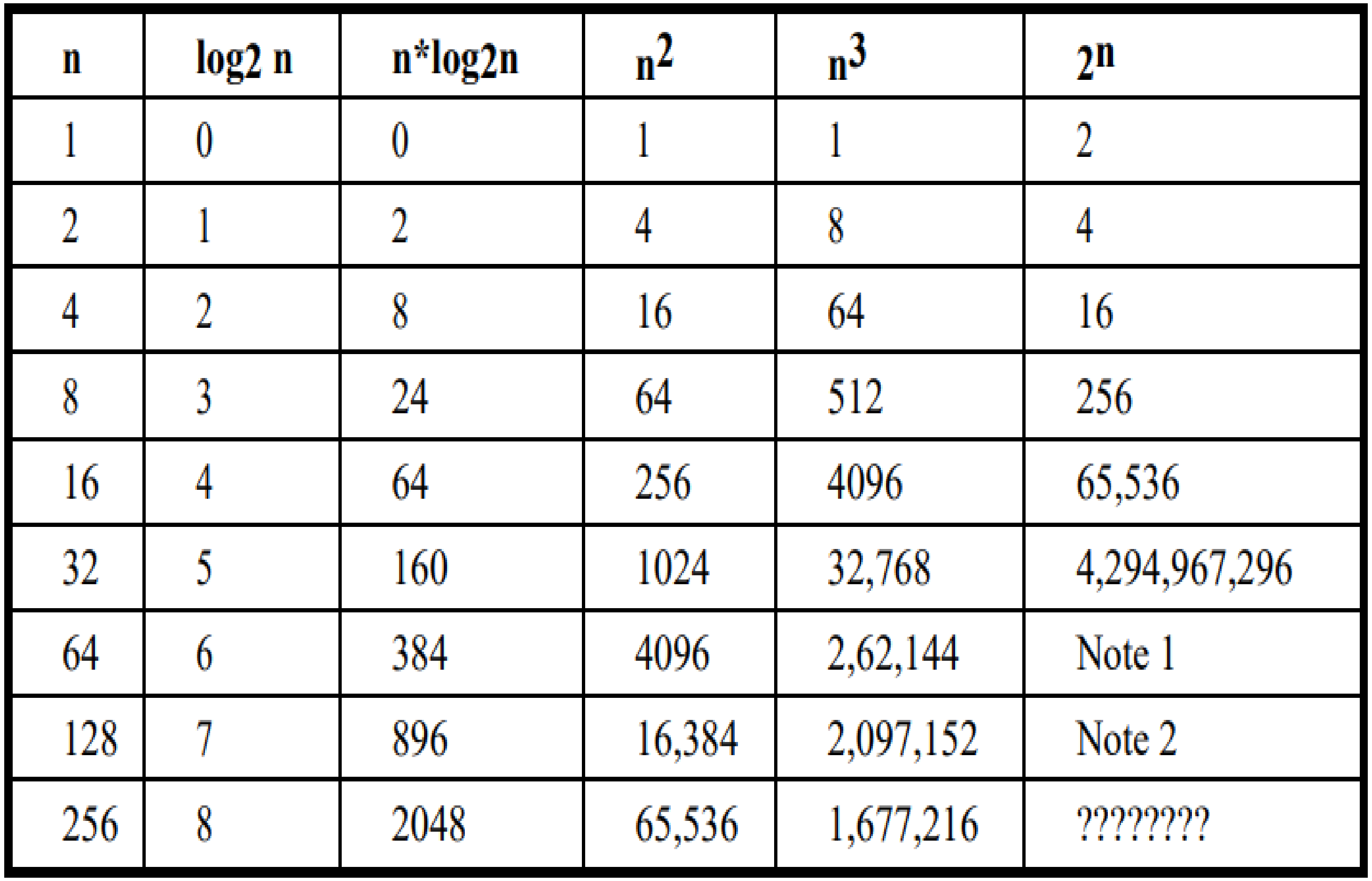
$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

n^4 , $2n^2$, $100n$ and 500 are individual cost of some functions and approximate to n^4 since n^4 is highest rate of growth.





Numerical Comparison of Different Algorithms



n	log2 n	n*log2n	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
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

Asymptotic Notations:

Asymptotic notations have been developed for analysis of algorithms.

By the word asymptotic means "for large values of n"

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

1. Big–OH(O)
2. Big–OMEGA(Ω),
3. Big–THETA (Θ)

Big O notation:

This notation gives the tight upper bound of the given function

Represented as:

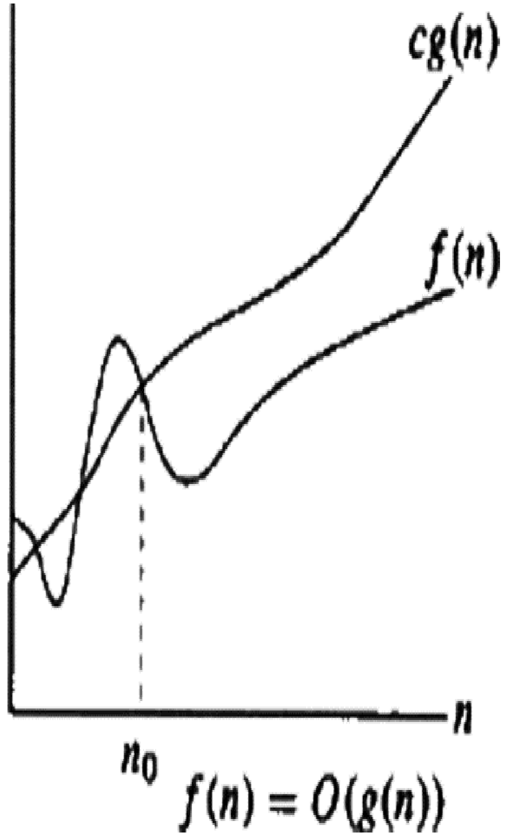
$$f(n) = O(g(n))$$

that means, at larger values of n, upper bound of *f(n)* is $g(n)$.

Definition:

Big O notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that

$$0 <= f(n) <= c.g(n) for\ all\ n > n_0\}$$



Big Omega (Ω) notation:

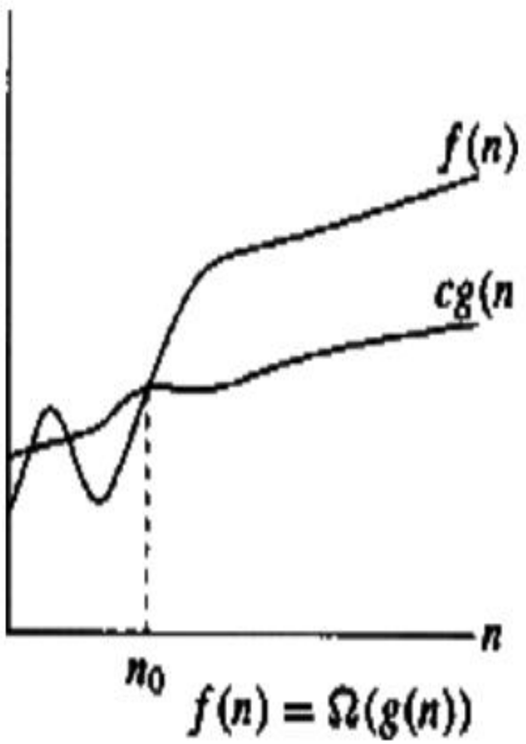
This notation gives the tight lower bound of the given function
Represented as:

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

that means, at larger values of n, lower bound of $f(n)$ is $g(n)$.

Definition:
Big Ω notation defined as $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that

$$0 <= c.g(n) \leq= f(n) \quad for\ all\ n > n_0\}$$



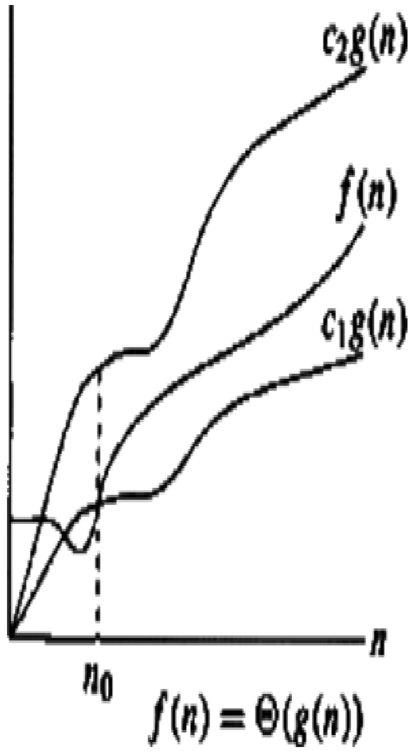
Big Theta (θ) Notation:

Average running time of an algorithm is always between lower bound and upper Bound

Represented as:

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

that means, at larger values of n, lower bound of $f(n)$ is $g(n)$.



Definition:

Bigθ notation defined as $\theta(g(n)) = \{f(n)$: there exist positive constants c_1 and c_2 and n_o such that

$$0 \leq c_1.g(n) \leq f(n) \leq c_1.g(n)$$
$$for\ all\ n > n_0\}$$

Properties of Asymptotic Notations:

1. Transitivity:

$f(n) = \theta(g(n))$ & $g(n) = \theta(h(n))$

$\rightarrow f(n) = \theta(h(n))$

Valid for O and Ω as well.

2. Reflexivity:

$f(n) = \theta(f(n))$

Valid for O and Ω as well.

3. Symmetry:

$f(n) = \theta(g(n))$, iff $g(n) = \theta(f(n))$

4. Transpose Symmetry:

$f(n) = \theta\big(g(n)\big)$ iff $g(n) = \Omega(f(n))$

Examples:

----

1. $f(n) = n$ & $g(n) = n^2$ & $h(n)=n^3$

   $n = O(n^2)$; $n2=O(n^3)$,
   then $n = O(n^3)$

----

2. $f(n) = n^3 = O(n^3) = \theta(n^3) = \Omega(n^3)$

----

3. $f(n) = n^2$ & $g(n) = n^2$
   then, $f(n) = \theta(n^2)$

----

4. $f(n) = n$ & $g(n) = \theta(n^2)$

Properties of Asymptotic Notations:

Observations:

1. If $f(n) = O(kg(n))$ for any constant $k>0$, then $f(n) = O(g(n))$
2. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $(f_1+ f_2)(n) = O(max(g_1(n), g_2(n))$
3. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $f_1(n) f_2(n) = O(g_1(n). g_2(n))$
4. If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then $f(n) = \theta(g(n))$

Recursion:

Recursion is an ability of an algorithm to repeatedly call itself until a certain condition is met.
Such condition is called the base condition.

The algorithm which calls itself is called a recursive algorithm.

The recursive algorithms must satisfy the following two conditions:

1. It must have the base case: The value of which algorithm does not call itself and can be evaluated without recursion.
2. Each recursive call must be to a case that eventually leads toward a base case.

Recursion:

Recurrence Relation:

An algorithm is said to be recursive if it can be defined in terms of itself.

The running time of recursive algorithm is expressed by means of recurrence relations.

A recurrence relation is an equation of inequality that describes a function in terms of its value on smaller inputs.

It is generally denoted by $\boldsymbol{T(n)}$ where $\boldsymbol{n}$ is the size of the input data of the problem.

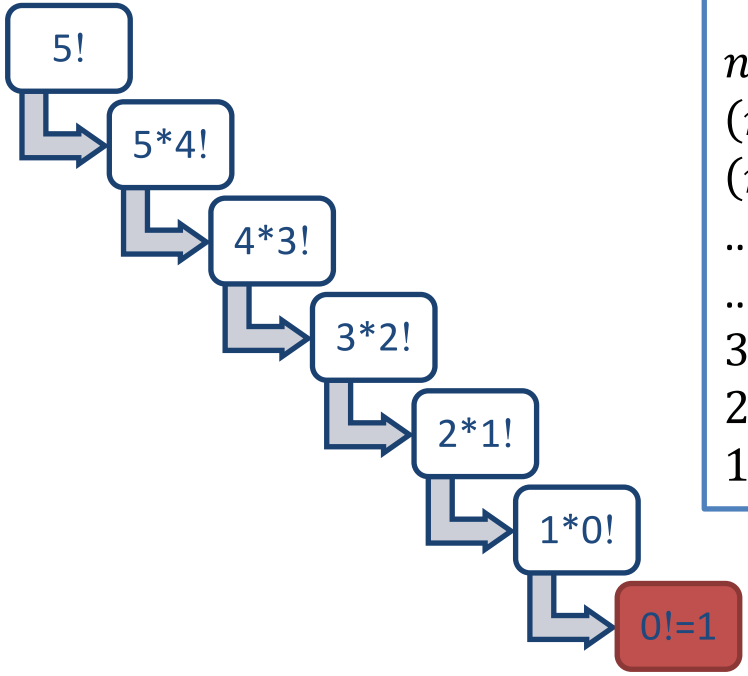
The recurrence relation satisfies both the conditions of recursion, that is, it has both the base case as well as the recursive case.

The portion of the recurrence relation that does not contain T is called the base case of the recurrence relation and

The portion of the recurrence relation that contains T is called the recursive case of the recurrence relation.

$$\boxed{\boldsymbol{T(n)} = \begin{cases} \boldsymbol{d} & ; \boldsymbol{n = 1} \\ \boldsymbol{T(n-1) + c} & ; \boldsymbol{n > 1} \end{cases}}$$

xample: Factorial



$$n! \ = \ n * (n-1) * (n-2) * (n-3) * \cdots 3 * 2 * 1$$

$$n! \ = \ n * (n-1)$$
$$(n-1)! = (n-1) * (n-2)\,!$$
$$(n-2)! = (n-2) * (n-3)\,!$$
…

…
$$3\,! \ = \ 3 * 2\,!$$
$$2\,! \ = \ 2 * 1\,!$$
$$1\,! \ = \ 1 * 0\,! \ = \ 1 * 1 \ = \ 1$$

→ **Base Condition**

Factorial Algorithm:

Recursive Method:

```
Algorithm fact(n)
If (n<0) then
return(“error”);
      else
If (n<2) then return(1);
      else
Return (n*fact(n-1);
End if
End if
End fact
```

Iterative Method:

```
Algorithm fact(n)
If (n<0) then return(“error”);
         else
If (n<2) then return(1);
         else
prod=1;
End if
End if
For(i=n down to 0)
         do prod=prod*i
         end for
Return prod
End fact
```

$$T(n) = \begin{cases} d & ; n = 1 \\ T(n-1) + c & ; n > 1 \end{cases}$$

Recursion:

Recurrence Relation:

There are various methods to solve recurrence:

1. Iterative Method / Substitution Method
2. Recurrence Tree
3. Master Method/ Master's Theorem

Recursion:

Recurrence Relation:

There are various methods to solve recurrence:

1. Iterative Method / Substitution Method
2. Recurrence Tree
3. Master Method/ Master's Theorem

Recursion:

Recurrence Relation:

There are various methods to solve recurrence:

1. Iterative Method / Substitution Method
2. Recurrence Tree
3. Master Method/ Master's Theorem

Recursion:

Master’s Theorem:

Dividing Functions

Decreasing Functions

1. Dividing functions:

Master’s method (for Dividing Functions) provides general method for solving recurrences of the form:

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & n > 1 \\ \theta(1) & n = 1 \end{cases}$$

Recursion:

1. Dividing functions:

Master’s method (for Dividing Functions) provides general method for solving recurrences of the form:

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & n > 1 \\ \theta(1) & n = 1 \end{cases}$$

Where,

$$f(n) = \Theta\left(n^k \log^p n\right)$$

and

$$a \geq 1 \quad ; \quad b > 1 \quad ; \quad k \geq 0$$

Recursion:

1. Dividing functions:

Master's method (for Dividing Functions) provides general method for solving recurrences of the form:

$$T(n) = \begin{cases} aT\left(\frac{n}{b}\right) + f(n) & n > 1 \\ \theta(1) & n = 1 \end{cases}$$

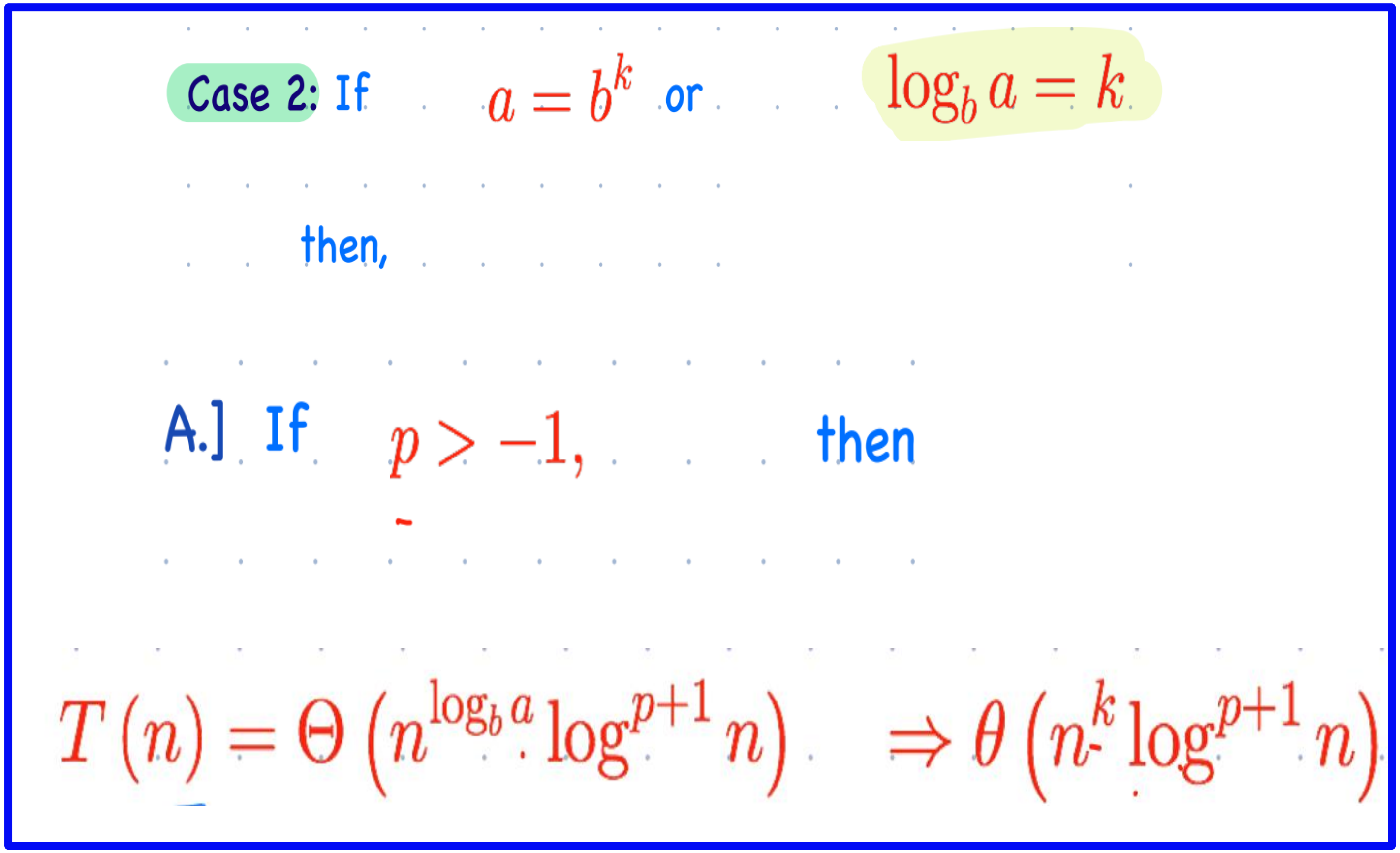
Case 1: If $a > b^k$ or $\log_b a > k$

then, $T(n) = \Theta\left(n^{\log_b a}\right)$

Recursion:

1. Dividing functions:

Master's method (for Dividing Functions) provides general method for solving recurrences of the form:



Case 2: If $a = b^k$ or $\log_b a = k$

then,

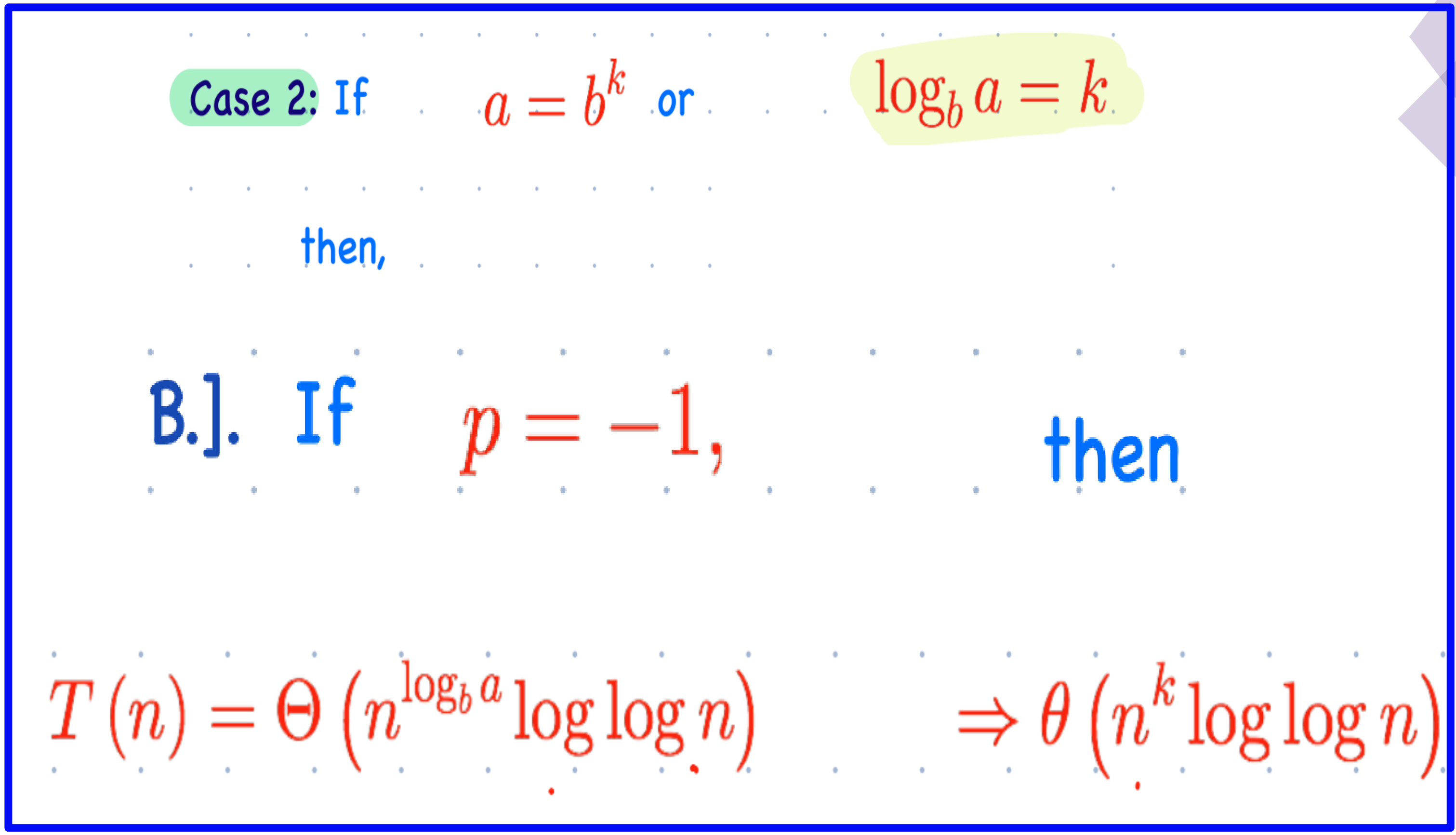
A.] If $p > -1$, then

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

Recursion:

1. Dividing functions:

Master's method (for Dividing Functions) provides general method for solving recurrences of the form:



Case 2: If $a = b^k$ or $\log_b a = k$

then,

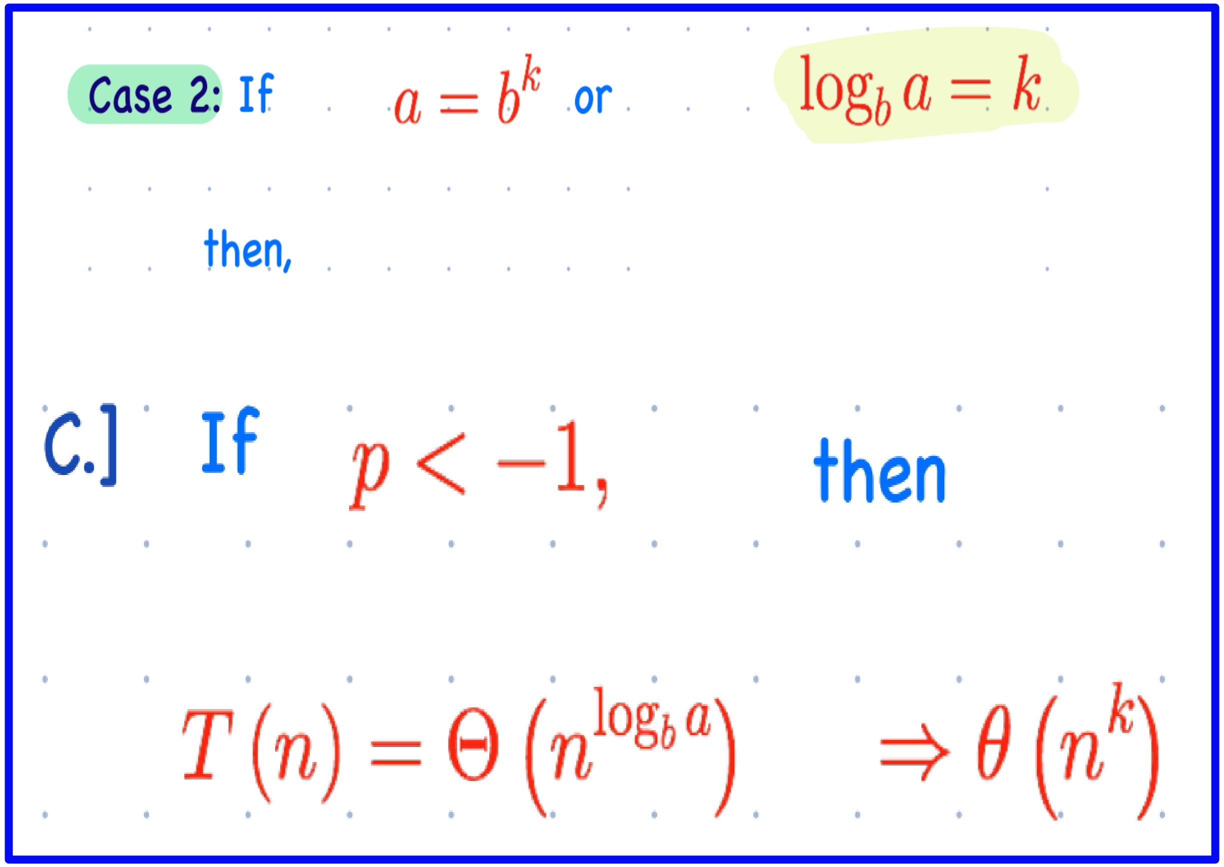
B.]. If $p = -1$, then

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

Recursion:

1. Dividing functions:

Master's method (for Dividing Functions) provides general method for solving recurrences of the form:



Case 2: If $a = b^k$ or $\log_b a = k$

then,

C.] If $p < -1,$ then

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

Recursion:

1. Dividing functions:

Master's method (for Dividing Functions) provides general method for solving recurrences of the form:

Case 3: If $a < b^k$ or $\log_b a < k$

A.] If $p \geq 0$ then

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

B.] If $p < 0$ then

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

Master’s Theorem:

1.

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

Exer

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

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

Master's Theorem (for Decreasing Function)

Let T(n) be a function defined on positive n for some constants c, where a>0, b>0, k >0, then

$$T(n) = \begin{cases} c & if\ n \leq 1 \\ aT(n-b) + f(n) & if\ n > 1 \end{cases}$$

$$f(n) = O(n^k)$$

$$T(n) = O(n^k) \quad if a < 1$$

$$= O(n^{k+1}) \checkmark \quad if a = 1$$

$$= O\left(n^k \cdot a^{\frac{n}{b}}\right) \quad if a > 1$$

Algorithm Classification

- Recursion
- Divide and Conquer Technique
- Greedy Technique
- Dynamic Programming Technique
- Backtracking Technique
- String Matching Algorithms
- Non-deterministic Polynomial Algorithms